Introduction to Time Series Forecasting with Python

How to Prepare Data and Develop Models to Predict the Future

Jason Brownlee

MACHINE
LEARNING
MASTERY

Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions.
The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.
No part of this eBook may be reproduced or transmitted in any form or by any means, electronic or mechanical, recording or by any information storage and retrieval system, without written permission from the author.

Copyright

Introduction to Time Series Forecasting with Python

© Copyright 2018 Jason Brownlee. All Rights Reserved.

Edition: v1.5

Contents

Copyright i

Welcome iii

I Fundamentals 1

1 Python Environment 2
- 1.1 Why Python? . . . 2
- 1.2 Python Libraries for Time Series . . . 3
- 1.3 Python Ecosystem Installation . . . 5
- 1.4 Summary . . . 8

2 What is Time Series Forecasting? 9
- 2.1 Time Series . . . 9
- 2.2 Time Series Nomenclature . . . 10
- 2.3 Describing vs. Predicting . . . 10
- 2.4 Components of Time Series . . . 11
- 2.5 Concerns of Forecasting . . . 12
- 2.6 Examples of Time Series Forecasting . . . 13
- 2.7 Summary . . . 13

3 Time Series as Supervised Learning 14
- 3.1 Supervised Machine Learning . . . 14
- 3.2 Sliding Window . . . 15
- 3.3 Sliding Window With Multivariates . . . 16
- 3.4 Sliding Window With Multiple Steps . . . 18
- 3.5 Summary . . . 18

II Data Preparation 20

4 Load and Explore Time Series Data 21
- 4.1 Daily Female Births Dataset . . . 21
- 4.2 Load Time Series Data . . . 21
- 4.3 Exploring Time Series Data . . . 22
- 4.4 Summary . . . 25

5 Basic Feature Engineering **26**
5.1 Feature Engineering for Time Series . . . 26
5.2 Goal of Feature Engineering . . . 27
5.3 Minimum Daily Temperatures Dataset . . . 27
5.4 Date Time Features . . . 28
5.5 Lag Features . . . 29
5.6 Rolling Window Statistics . . . 31
5.7 Expanding Window Statistics . . . 33
5.8 Summary . . . 34

6 Data Visualization **36**
6.1 Time Series Visualization . . . 36
6.2 Minimum Daily Temperatures Dataset . . . 37
6.3 Line Plot . . . 37
6.4 Histogram and Density Plots . . . 40
6.5 Box and Whisker Plots by Interval . . . 42
6.6 Heat Maps . . . 45
6.7 Lag Scatter Plots . . . 47
6.8 Autocorrelation Plots . . . 50
6.9 Summary . . . 52

7 Resampling and Interpolation **53**
7.1 Resampling . . . 53
7.2 Shampoo Sales Dataset . . . 54
7.3 Upsampling Data . . . 54
7.4 Downsampling Data . . . 59
7.5 Summary . . . 62

8 Power Transforms **63**
8.1 Airline Passengers Dataset . . . 63
8.2 Square Root Transform . . . 64
8.3 Log Transform . . . 68
8.4 Box-Cox Transform . . . 72
8.5 Summary . . . 74

9 Moving Average Smoothing **76**
9.1 Moving Average Smoothing . . . 76
9.2 Data Expectations . . . 77
9.3 Daily Female Births Dataset . . . 77
9.4 Moving Average as Data Preparation . . . 77
9.5 Moving Average as Feature Engineering . . . 80
9.6 Moving Average as Prediction . . . 82
9.7 Summary . . . 84

III Temporal Structure 85

10 A Gentle Introduction to White Noise 86
10.1 What is a White Noise? . . . 86
10.2 Why Does it Matter? . . . 86
10.3 Is your Time Series White Noise? . . . 87
10.4 Example of White Noise Time Series . . . 87
10.5 Summary . . . 92

11 A Gentle Introduction to the Random Walk 93
11.1 Random Series . . . 93
11.2 Random Walk . . . 94
11.3 Random Walk and Autocorrelation . . . 96
11.4 Random Walk and Stationarity . . . 97
11.5 Predicting a Random Walk . . . 101
11.6 Is Your Time Series a Random Walk? . . . 103
11.7 Summary . . . 103

12 Decompose Time Series Data 105
12.1 Time Series Components . . . 105
12.2 Combining Time Series Components . . . 106
12.3 Decomposition as a Tool . . . 106
12.4 Automatic Time Series Decomposition . . . 107
12.5 Summary . . . 111

13 Use and Remove Trends 112
13.1 Trends in Time Series . . . 112
13.2 Shampoo Sales Dataset . . . 114
13.3 Detrend by Differencing . . . 114
13.4 Detrend by Model Fitting . . . 115
13.5 Summary . . . 118

14 Use and Remove Seasonality 119
14.1 Seasonality in Time Series . . . 119
14.2 Minimum Daily Temperatures Dataset . . . 120
14.3 Seasonal Adjustment with Differencing . . . 121
14.4 Seasonal Adjustment with Modeling . . . 126
14.5 Summary . . . 130

15 Stationarity in Time Series Data 131
15.1 Stationary Time Series . . . 131
15.2 Non-Stationary Time Series . . . 132
15.3 Types of Stationary Time Series . . . 133
15.4 Stationary Time Series and Forecasting . . . 134
15.5 Checks for Stationarity . . . 134
15.6 Summary Statistics . . . 134
15.7 Augmented Dickey-Fuller test . . . 140

15.8 Summary . . . 142

IV Evaluate Models 144

16 Backtest Forecast Models 145
16.1 Model Evaluation . . . 145
16.2 Monthly Sunspots Dataset . . . 146
16.3 Train-Test Split . . . 146
16.4 Multiple Train-Test Splits . . . 148
16.5 Walk Forward Validation . . . 151
16.6 Summary . . . 153

17 Forecasting Performance Measures 154
17.1 Forecast Error (or Residual Forecast Error) . . . 154
17.2 Mean Forecast Error (or Forecast Bias) . . . 155
17.3 Mean Absolute Error . . . 155
17.4 Mean Squared Error . . . 156
17.5 Root Mean Squared Error . . . 157
17.6 Summary . . . 157

18 Persistence Model for Forecasting 158
18.1 Forecast Performance Baseline . . . 158
18.2 Persistence Algorithm . . . 159
18.3 Shampoo Sales Dataset . . . 159
18.4 Persistence Algorithm Steps . . . 159
18.5 Summary . . . 163

19 Visualize Residual Forecast Errors 164
19.1 Residual Forecast Errors . . . 164
19.2 Daily Female Births Dataset . . . 165
19.3 Persistence Forecast Model . . . 165
19.4 Residual Line Plot . . . 166
19.5 Residual Summary Statistics . . . 168
19.6 Residual Histogram and Density Plots . . . 169
19.7 Residual Q-Q Plot . . . 171
19.8 Residual Autocorrelation Plot . . . 173
19.9 Summary . . . 175

20 Reframe Time Series Forecasting Problems 176
20.1 Benefits of Reframing Your Problem . . . 176
20.2 Minimum Daily Temperatures Dataset . . . 177
20.3 Naive Time Series Forecast . . . 177
20.4 Regression Framings . . . 177
20.5 Classification Framings . . . 178
20.6 Time Horizon Framings . . . 180
20.7 Summary . . . 181

V Forecast Models 182

21 A Gentle Introduction to the Box-Jenkins Method 183
21.1 Autoregressive Integrated Moving Average Model . . . 183
21.2 Box-Jenkins Method . . . 184
21.3 Identification . . . 184
21.4 Estimation . . . 185
21.5 Diagnostic Checking . . . 185
21.6 Summary . . . 186

22 Autoregression Models for Forecasting 187
22.1 Autoregression . . . 187
22.2 Autocorrelation . . . 188
22.3 Minimum Daily Temperatures Dataset . . . 188
22.4 Quick Check for Autocorrelation . . . 188
22.5 Autocorrelation Plots . . . 190
22.6 Persistence Model . . . 192
22.7 Autoregression Model . . . 194
22.8 Summary . . . 198

23 Moving Average Models for Forecasting 199
23.1 Model of Residual Errors . . . 199
23.2 Daily Female Births Dataset . . . 200
23.3 Persistence Forecast Model . . . 200
23.4 Autoregression of Residual Error . . . 202
23.5 Correct Predictions with a Model of Residuals . . . 205
23.6 Summary . . . 207

24 ARIMA Model for Forecasting 208
24.1 Autoregressive Integrated Moving Average Model . . . 208
24.2 Shampoo Sales Dataset . . . 209
24.3 ARIMA with Python . . . 211
24.4 Rolling Forecast ARIMA Model . . . 216
24.5 Summary . . . 218

25 Autocorrelation and Partial Autocorrelation 219
25.1 Minimum Daily Temperatures Dataset . . . 219
25.2 Correlation and Autocorrelation . . . 219
25.3 Partial Autocorrelation Function . . . 222
25.4 Intuition for ACF and PACF Plots . . . 223
25.5 Summary . . . 224

26 Grid Search ARIMA Model Hyperparameters 225
26.1 Grid Searching Method . . . 225
26.2 Evaluate ARIMA Model . . . 226
26.3 Iterate ARIMA Parameters . . . 227
26.4 Shampoo Sales Case Study . . . 228

26.5 Daily Female Births Case Study . . . 230
26.6 Extensions . . . 232
26.7 Summary . . . 233

27 Save Models and Make Predictions **234**
27.1 Process for Making a Prediction . . . 234
27.2 Daily Female Births Dataset . . . 235
27.3 Select Time Series Forecast Model . . . 235
27.4 Finalize and Save Time Series Forecast Model . . . 237
27.5 Make a Time Series Forecast . . . 240
27.6 Update Forecast Model . . . 241
27.7 Extensions . . . 242
27.8 Summary . . . 242

28 Forecast Confidence Intervals **244**
28.1 ARIMA Forecast . . . 244
28.2 Daily Female Births Dataset . . . 245
28.3 Forecast Confidence Interval . . . 245
28.4 Interpreting the Confidence Interval . . . 246
28.5 Plotting the Confidence Interval . . . 247
28.6 Summary . . . 248

VI Projects **249**

29 Time Series Forecast Projects **250**
29.1 5-Step Forecasting Task . . . 250
29.2 Iterative Forecast Development Process . . . 251
29.3 Suggestions and Tips . . . 253
29.4 Summary . . . 254

30 Project: Monthly Armed Robberies in Boston **256**
30.1 Overview . . . 256
30.2 Problem Description . . . 257
30.3 Test Harness . . . 257
30.4 Persistence . . . 260
30.5 Data Analysis . . . 261
30.6 ARIMA Models . . . 266
30.7 Model Validation . . . 279
30.8 Extensions . . . 284
30.9 Summary . . . 285

31 Project: Annual Water Usage in Baltimore **286**
31.1 Overview . . . 286
31.2 Problem Description . . . 287
31.3 Test Harness . . . 287
31.4 Persistence . . . 288

31.5 Data Analysis . . . 289
31.6 ARIMA Models . . . 294
31.7 Model Validation . . . 303
31.8 Summary . . . 307

32 Project: Monthly Sales of French Champagne 308
32.1 Overview . . . 308
32.2 Problem Description . . . 309
32.3 Test Harness . . . 309
32.4 Persistence . . . 310
32.5 Data Analysis . . . 311
32.6 ARIMA Models . . . 317
32.7 Model Validation . . . 331
32.8 Summary . . . 336

VII Conclusions 337

33 How Far You Have Come 338

34 Further Reading 339
34.1 Applied Time Series . . . 339
34.2 Time Series . . . 339
34.3 Machine Learning . . . 340
34.4 Getting Help . . . 340
34.5 Contact the Author . . . 340

VIII Appendix 341

A Standard Time Series Datasets 342
A.1 Shampoo Sales Dataset . . . 343
A.2 Minimum Daily Temperatures Dataset . . . 344
A.3 Monthly Sunspots Dataset . . . 345
A.4 Daily Female Births Dataset . . . 347
A.5 Airline Passengers Dataset . . . 348

B Workaround for Saving ARIMA Models 350
B.1 Daily Female Births Dataset . . . 350
B.2 Python Environment . . . 350
B.3 ARIMA Model Save Bug . . . 351
B.4 ARIMA Model Save Bug Workaround . . . 352
B.5 Summary . . . 353

Welcome

Welcome to the *Introduction to Time Series Forecasting with Python*. You are one of those rare people that have decided to invest in your education and in your future and I am honored that I can help.

This book will show you how to make predictions on univariate time series problems using the standard tools in the Python ecosystem. Time series is an important and under served topic in applied machine learning, Python is the growing platform for machine learning and predictive modeling, and this book unlocks time series for Python. But, you have to do the work. I will lay out all of the topics, the tools, the code and the templates, but it is up to you to put them into practice on your projects and get results.

Before we dive in, it is important to first dial in to the goals, limitations and structure of this book to ensure that you can get the most out of it, quickly, efficiently and also have a lot of fun doing it. Let's start with the goals of the book.

Goals

The goal of this book is to *show you how to get results on univariate time series forecasting problems using the Python ecosystem*. This goal cannot be achieved until you apply the lessons from this book on your own projects and get results. This is a guidebook or a cookbook designed for immediate use.

Principles

This book was developed using the following five principles:

- **Application**: The focus is on the application of forecasting rather than the theory.
- **Lessons**: The book is broken down into short lessons, each focused on a specific topic.
- **Value**: Lessons focus on the most used and most useful aspects of a forecasting project.
- **Results**: Each lesson provides a path to a usable and reproducible result.
- **Speed**: Each lesson is designed to provide the shortest path to a result.

These principles shape the structure and organization of the book.

Prerequisites

This book makes some assumptions of you the reader. They are:

- **You're a Developer**: This is a book for developers. You are a developer of some sort. You know how to read and write code. You know how to develop and debug a program.
- **You know Python**: This is a book for Python people. You know the Python programming language, or you're a skilled enough developer that you can pick it up as you go along.
- **You know some Machine Learning**: This is a book for novice machine learning practitioners. You know some basic practical machine learning, or you can figure it out quickly.

Don't panic if you're not a perfect match. I will provide resources to help out including help on setting up a Python environment and resources for learning more about machine learning.

How to Read

The book was designed to fit three different reading styles. They are:

- **End-to-End**: The material is ordered such that you may read the book end-to-end, cover-to-cover and discover how you may work through time series forecasting problems.
- **Piecewise**: The lessons are standalone so that you can dip in to the topics you require piecewise for your forecasting project.
- **Template**: The projects are structured to provide a template that can be copied and used to work through your own forecasting projects.

Pick the reading style that is right for you to get the most out of the book as quickly as possible.

Reading Outcomes

If you choose to work through all of the lessons and projects of this book, you can set some reasonable expectations on your new found capabilities. They are:

- **Time Series Foundations**: You will be able to identify time series forecasting problems as distinct from other predictive modeling problems and how time series can be framed as supervised learning.
- **Transform Data For Modeling**: You will be able to transform, rescale, smooth and engineer features from time series data in order to best expose the underlying inherent properties of the problem (the signal) to learning algorithms for forecasting.

- **Harness Temporal Structure**: You will be able to analyze time series data and understand the temporal structure inherent in it such as trends and seasonality and how these structures may be addressed, removed and harnessed when forecasting.
- **Evaluate Models**: You will be able to devise a model test harness for a univariate forecasting problem and estimate the baseline skill and expected model performance on unseen data with various performance measures.
- **Apply Classical Methods**: You will be able to select, apply and interpret the results from classical linear methods such as Autoregression, Moving Average and ARIMA models on univariate time series forecasting problems.

You will be a capable predictive modeler for univariate time series forecasting problems using the Python ecosystem.

Limitations

This book is not all things to all people. Specifically:

- **Time Series Analysis**: This is a book of time series forecasting, not time series analysis. The primary concern of this book is using historical data to predict the future, not to understand the past.
- **Advanced Time Series**: This is not a treatment of complex time series problems. It does not provide tutorials on advanced topics like multi-step sequence forecasts, multivariate time series problems or spatial-temporal prediction problems.
- **Algorithm Textbook**: This is a practical book full of rules of application and code recipes. It is not a textbook, it does not describe why algorithms work or the equations that govern them.
- **Python Language Book**: This is not a treatment of the Python programming language. It does not provide an introduction to the history, syntax or semantics of Python code.

It is important to calibrate your expectations to the goals and principles of what this book is designed to achieve.

Structure

This book is presented as a series of lessons and projects.

- **Lessons**: Lessons take a tutorial format and focus on teaching one specific topic with just enough background to provide a context for the topic with a focus on examples and working code to demonstrate the topic in practice.
- **Projects**: Projects take a tutorial format and focus on the application of a suite of topics presented in lessons focused on getting an outcome on a real world predictive modeling dataset.

This book is primarily comprised of tutorial lessons. Lessons are organized into 5 main topics or themes. These are as follows:

- **Fundamentals**: This part contains introductory material such as how to setup a SciPy environment, examples of time series problems and what differentiates them from other types of predictive modeling and important concepts like random walk and white noise.
- **Data Preparation**: This part contains lessons on how to prepare data for forecasting. This includes methods such as scaling, transforms and feature engineering designed to better expose the inherent structure of the problem to modeling algorithms.
- **Temporal Structure**: This part focuses on the temporal structure of forecasting problems. This includes understanding systematic changes in the data over time such as trends and seasonality and the concept of stationarity.
- **Evaluate Models**: This part focuses on how to evaluate the performance of forecasts. This includes methods to back-test an algorithm, evaluate the skill of a model, perform diagnostics on the predictions from a model and reframe a problem.
- **Forecast Models**: This part focuses on classical linear models for time series forecasting. This includes autoregression, moving average and autoregressive integrated moving average or ARIMA models.

The lessons of the book culminate in three projects. The projects are designed to demonstrate both a structure for working through a forecasting project and the key techniques demonstrated in the book. These projects include:

- **Project 1**: Monthly Armed Robberies in Boston.
- **Project 2**: Annual Water Usage in Baltimore.
- **Project 3**: Monthly Sales of French Champagne.

Summary

You now have an idea of how the book is organized. You also now know the principles behind why the lessons are laser focused on fast repeatable results to help you deliver value as quickly as possible on your on time series forecasting projects.

Next

Next begins the Fundamentals part and the lessons focused on how time series forecasting problems are structured and how they are different to other types of predictive modeling problems. The first lesson gives you some background on how to setup your Python development environment.

Part I

Fundamentals

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

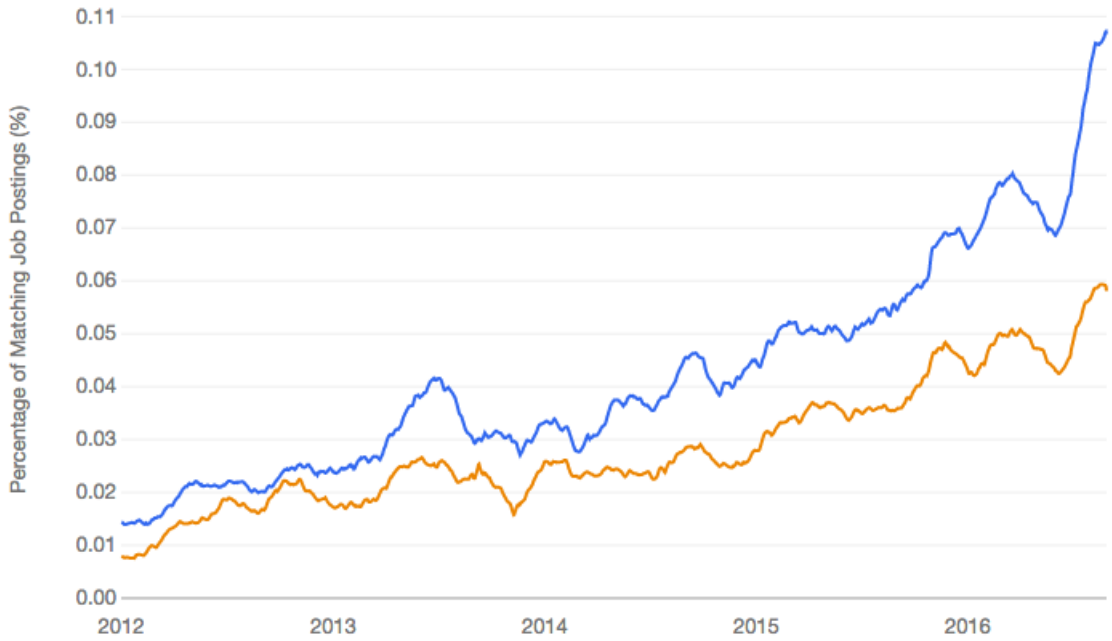

Figure 1.1: Plot of Python machine learning jobs (blue) compared to R machine learning jobs from indeed.com.

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: 1.0.0
numpy: 1.14.2
matplotlib: 2.1.1
pandas: 0.22.0
statsmodels: 0.8.0
```

```
sklearn: 0.19.1
```

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. 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, index_col=0, parse_dates=True,
    squeeze=True)
print(type(series))
print(series.head())
```

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

Listing 4.1: 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=True`: We give the function a hint that data in the first column contains dates that need to be parsed.
- `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
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 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.

```python
# summarize first few lines of a file
from pandas import read_csv
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
print(series.head(10))
```

Listing 4.3: 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 4.4: 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 read_csv
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
print(series.size)
```

Listing 4.5: 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 4.6: 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 read_csv
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
print(series['1959-01'])
```

Listing 4.7: 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 4.8: 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.

```python
# calculate descriptive statistics
from pandas import read_csv
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
print(series.describe())
```

Listing 4.9: 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 4.10: 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.

```python
# create date time features of a dataset
from pandas import read_csv
from pandas import DataFrame
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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

[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

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.

```
# create a lag feature
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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 read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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.

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

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

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

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, 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 read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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 read_csv
from matplotlib import pyplot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
series.plot()
pyplot.show()
```

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

Running the example creates a line plot.

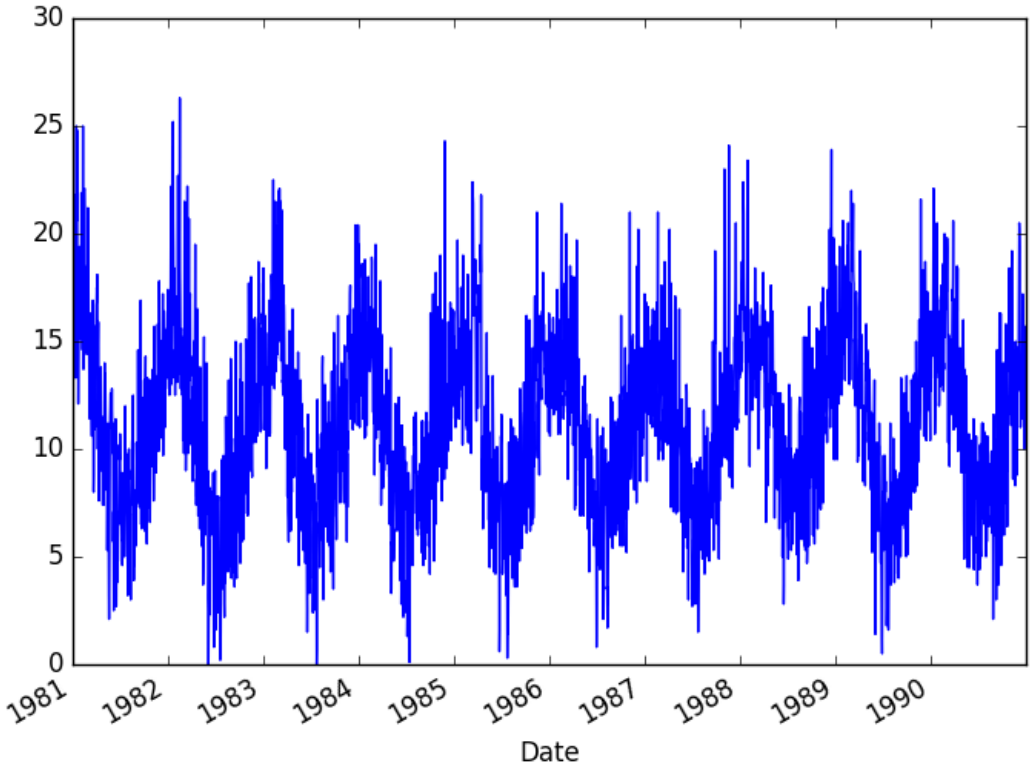

Figure 6.1: Line plot of the Minimum Daily Temperatures dataset.

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 read_csv
from matplotlib import pyplot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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.

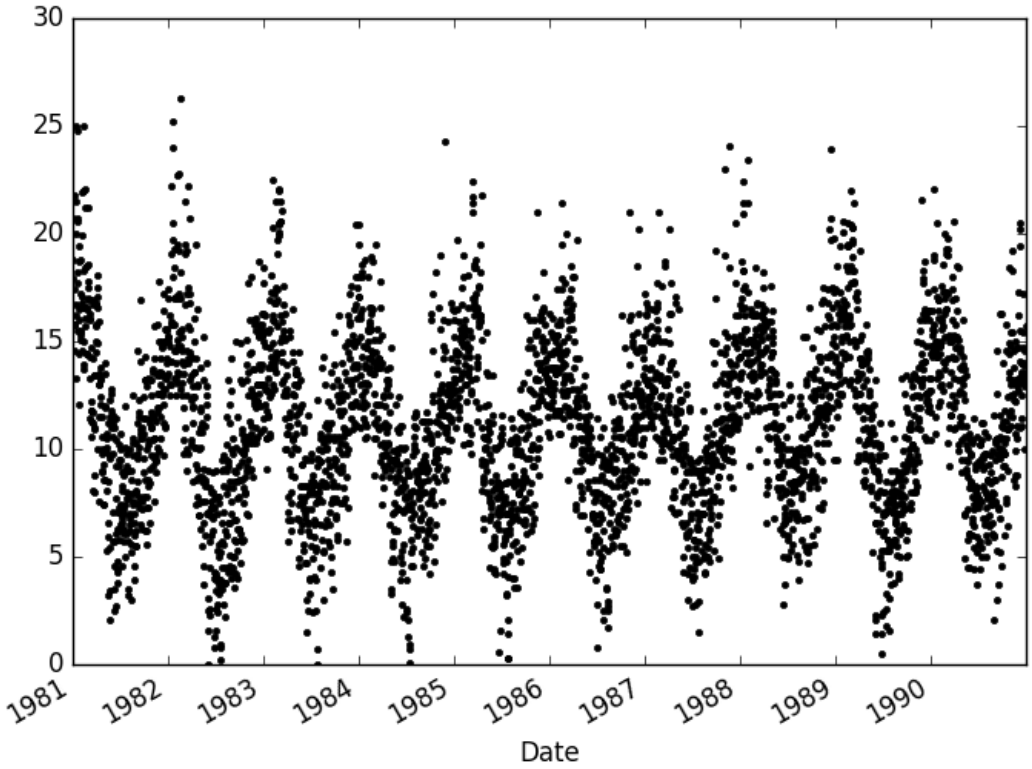

Figure 6.2: Dot line plot of the Minimum Daily Temperatures dataset.

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(Grouper(freq='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 read_csv
from pandas import DataFrame
from pandas import Grouper
from matplotlib import pyplot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
groups = series.groupby(Grouper(freq='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.

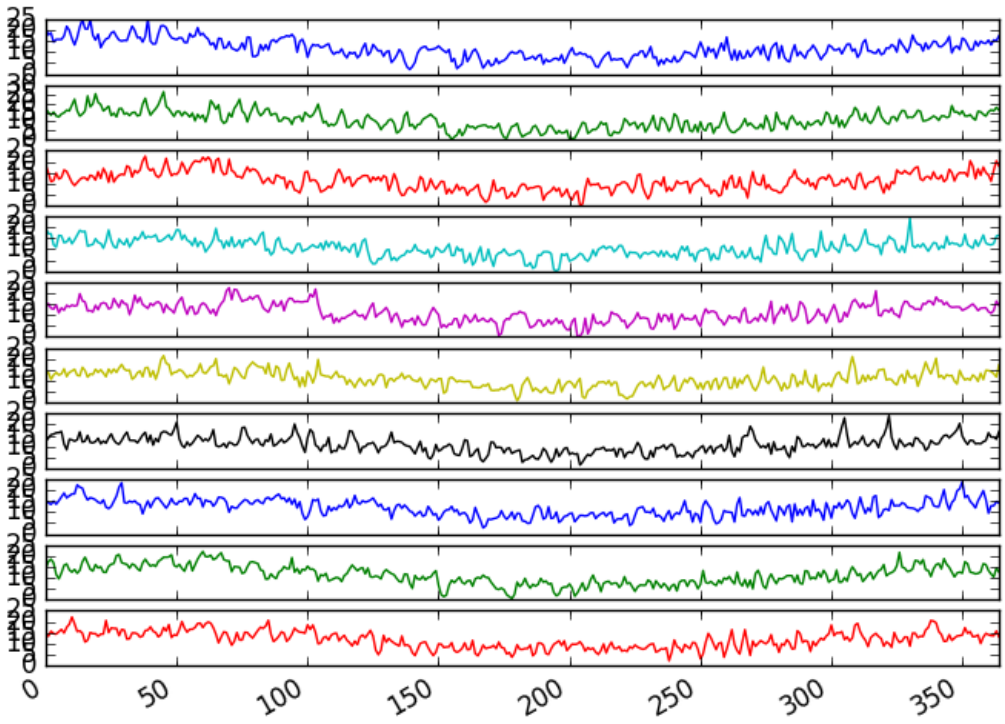

Figure 6.3: Stacked line plots of the Minimum Daily Temperatures dataset.

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 read_csv
```

```
from matplotlib import pyplot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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.

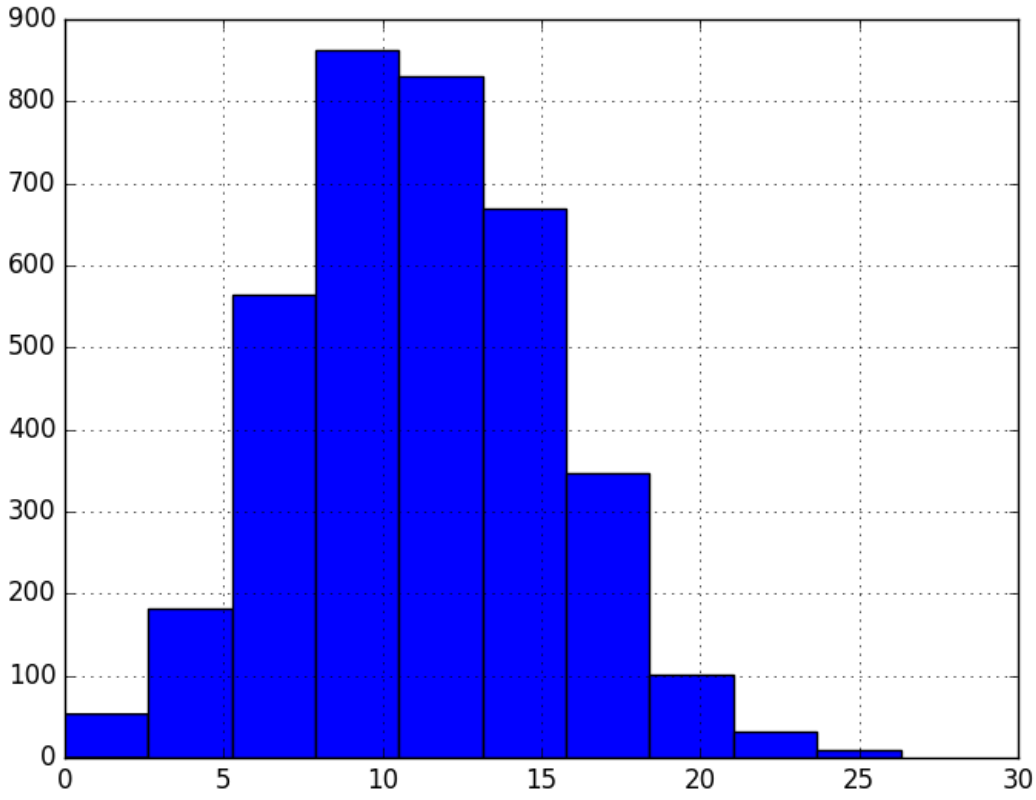

Figure 6.4: Histogram of the Minimum Daily Temperatures dataset.

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 read_csv
from matplotlib import pyplot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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.

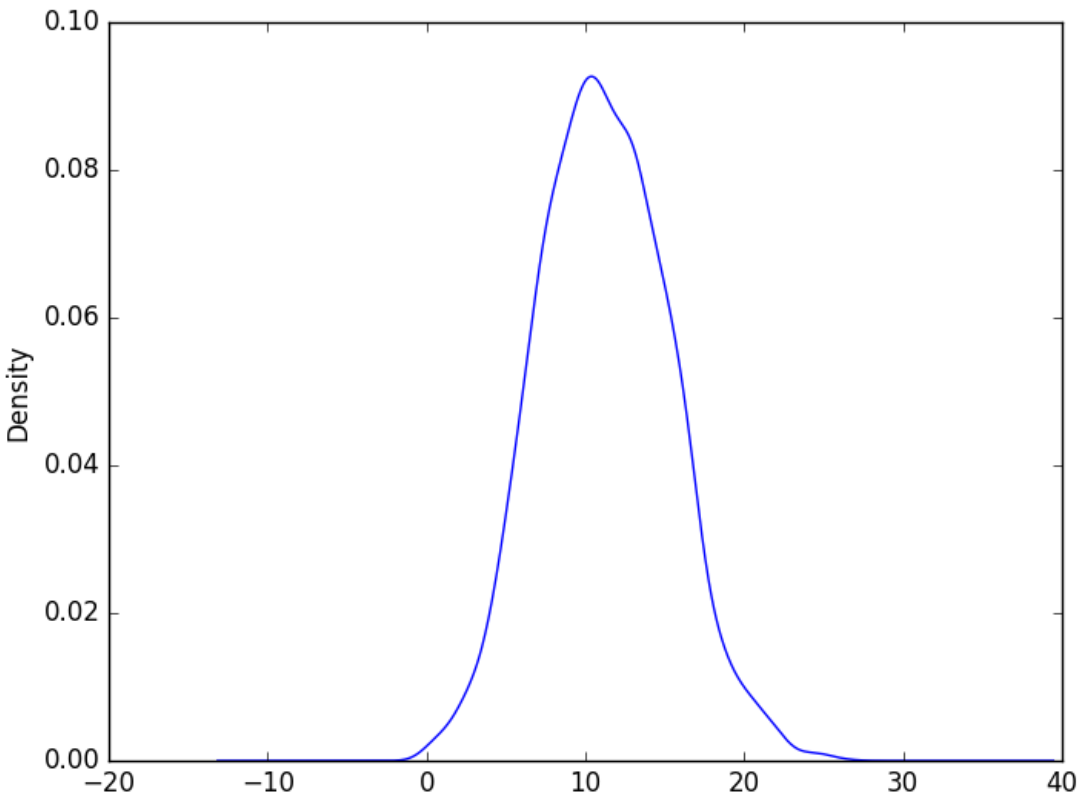

Figure 6.5: Density Plot of the Minimum Daily Temperatures dataset.

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 read_csv
from pandas import DataFrame
from pandas import Grouper
from matplotlib import pyplot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
groups = series.groupby(Grouper(freq='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.

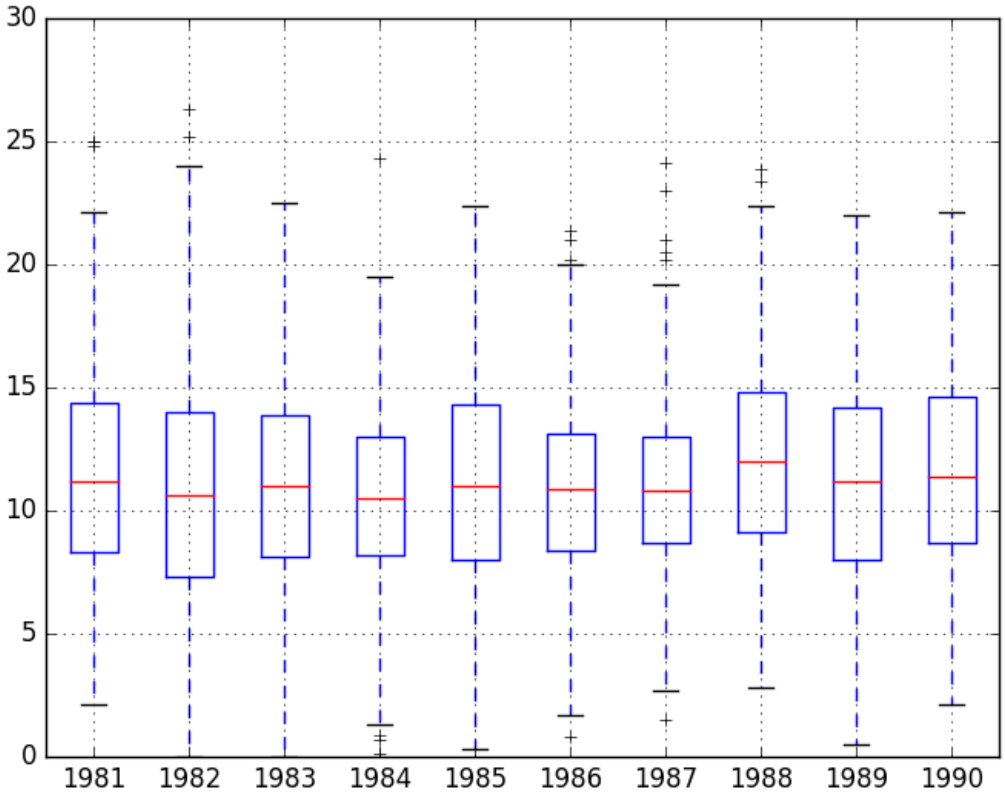

Figure 6.6: Yearly Box and Whisker Plots of the Minimum Daily Temperatures dataset.

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 read_csv
from pandas import DataFrame
from pandas import Grouper
from matplotlib import pyplot
from pandas import concat
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
one_year = series['1990']
groups = one_year.groupby(Grouper(freq='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.

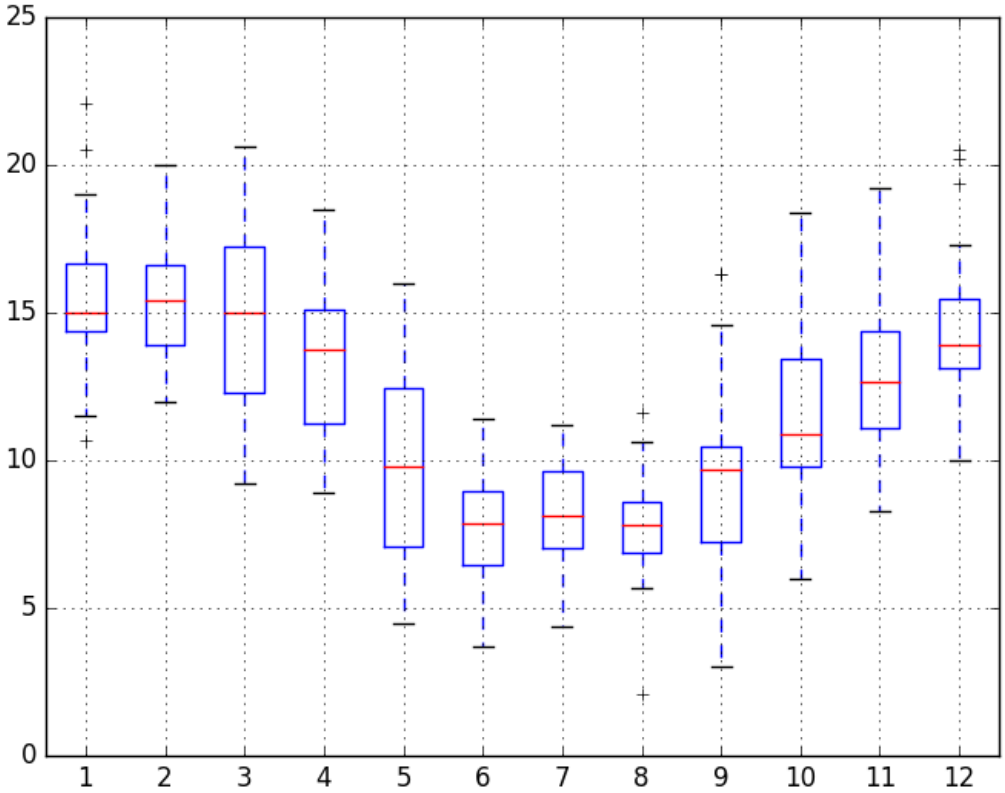

Figure 6.7: Monthly Box and Whisker Plots of the Minimum Daily Temperatures dataset.

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 read_csv
from pandas import DataFrame
from pandas import Grouper
from matplotlib import pyplot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
groups = series.groupby(Grouper(freq='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.

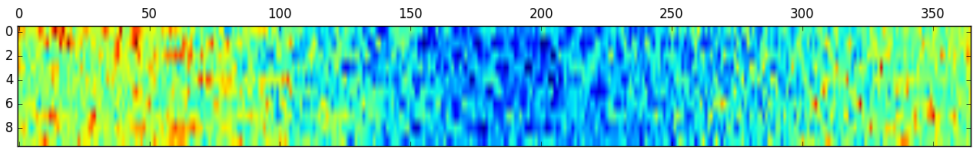

Figure 6.8: Yearly Heat Map Plot of the Minimum Daily Temperatures dataset.

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 read_csv
from pandas import DataFrame
from pandas import Grouper
from matplotlib import pyplot
from pandas import concat
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
one_year = series['1990']
groups = one_year.groupby(Grouper(freq='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.

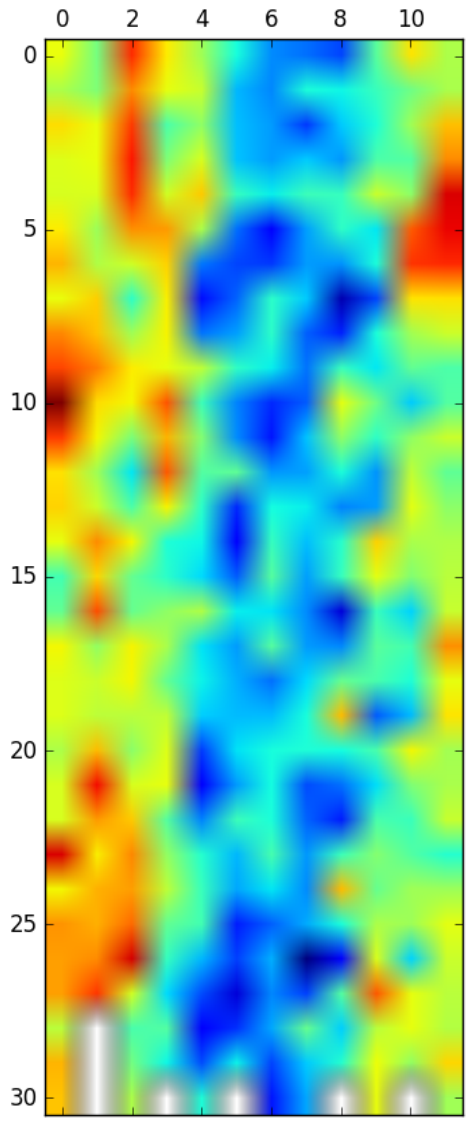

Figure 6.9: Monthly Heat Map Plot of the Minimum Daily Temperatures dataset.

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 observation at the next time step (`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 read_csv
from matplotlib import pyplot
from pandas.plotting import lag_plot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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.

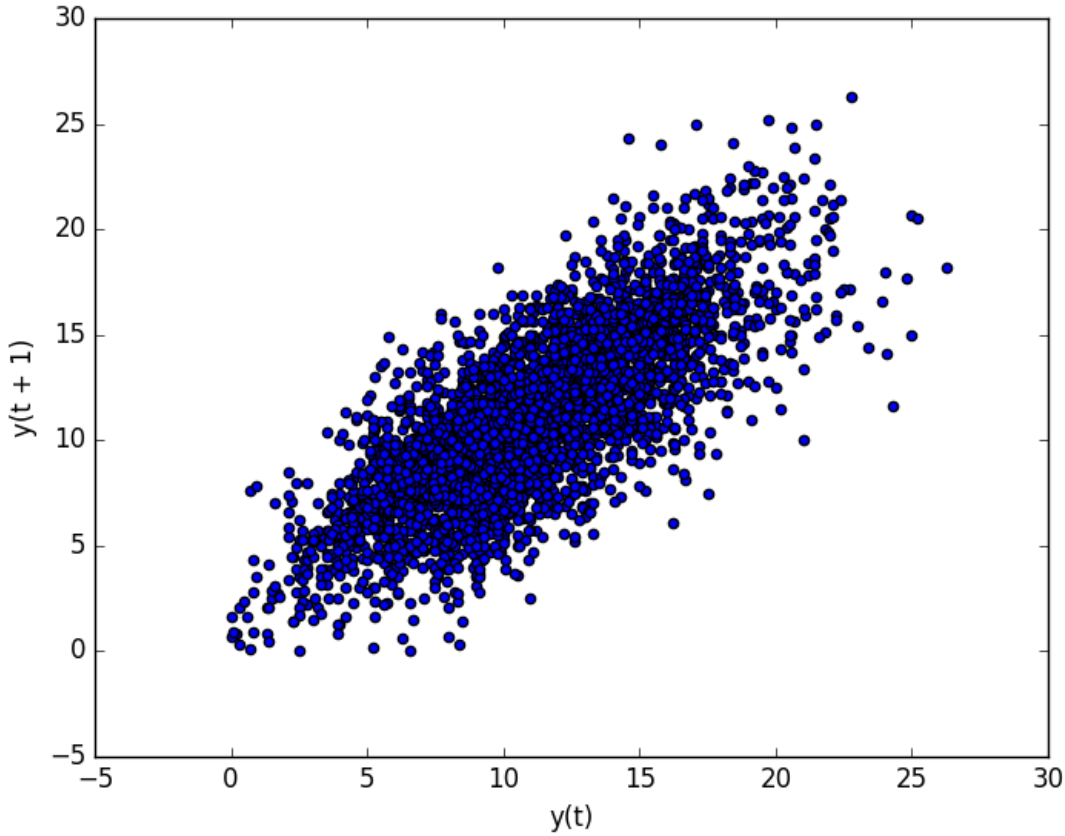

Figure 6.10: Lag scatter plot of the Minimum Daily Temperatures dataset.

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 read_csv
from pandas import DataFrame
from pandas import concat
from matplotlib import pyplot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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']
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 vs t-' + str(i))
  pyplot.scatter(x=dataframe['t'].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.

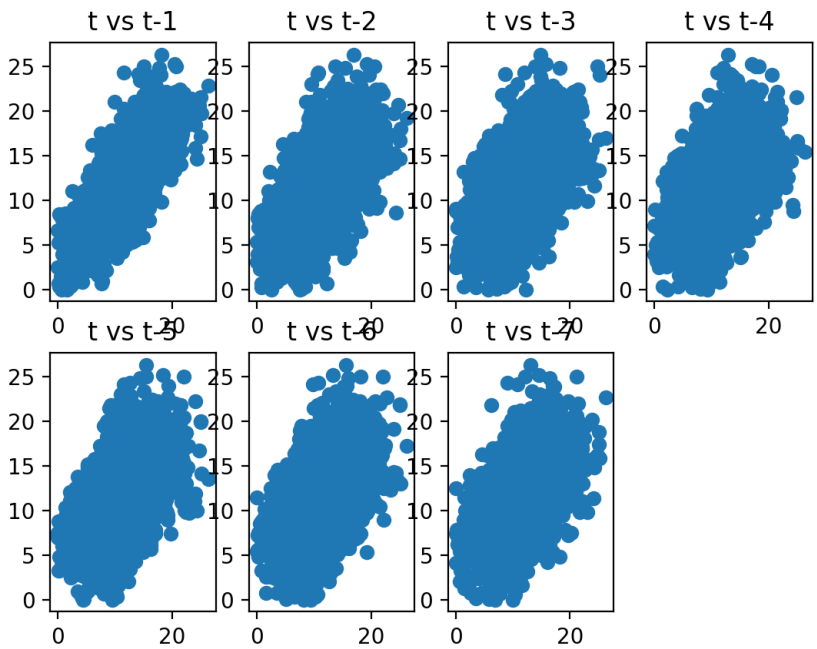

Figure 6.11: Multiple Lag scatter plots of the Minimum Daily Temperatures dataset.

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 read_csv
from matplotlib import pyplot
from pandas.plotting import autocorrelation_plot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
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.

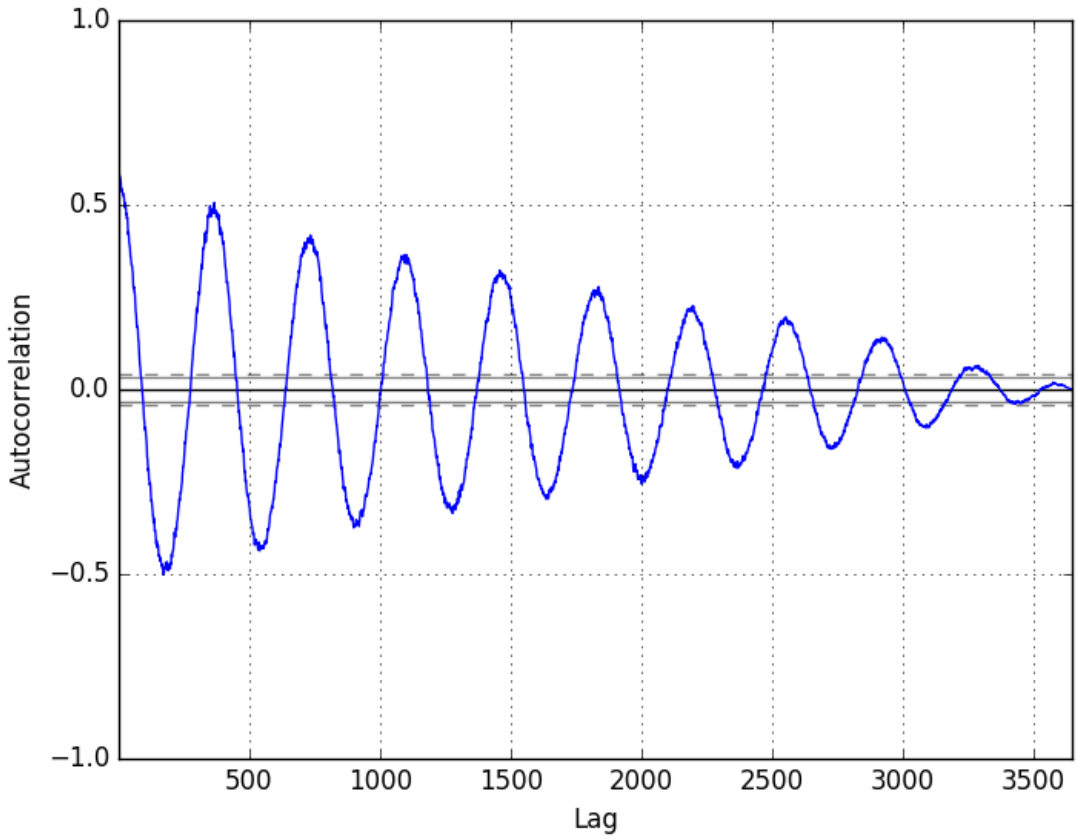

Figure 6.12: Autocorrelation Plot of the Minimum Daily Temperatures dataset.

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 7

Resampling and Interpolation

You may have observations at the wrong frequency. Maybe they are too granular or not granular enough. The Pandas library in Python provides the capability to change the frequency of your time series data. In this tutorial, you will discover how to use Pandas in Python to both increase and decrease the sampling frequency of time series data. After completing this tutorial, you will know:

- About time series resampling, the two types of resampling, and the 2 main reasons why you need to use them.
- How to use Pandas to upsample time series data to a higher frequency and interpolate the new observations.
- How to use Pandas to downsample time series data to a lower frequency and summarize the higher frequency observations.

Let's get started.

7.1 Resampling

Resampling involves changing the frequency of your time series observations. Two types of resampling are:

- **Upsampling**: Where you increase the frequency of the samples, such as from minutes to seconds.
- **Downsampling**: Where you decrease the frequency of the samples, such as from days to months.

In both cases, data must be invented. In the case of upsampling, care may be needed in determining how the fine-grained observations are calculated using interpolation. In the case of downsampling, care may be needed in selecting the summary statistics used to calculate the new aggregated values.

There are perhaps two main reasons why you may be interested in resampling your time series data:

- **Problem Framing**: Resampling may be required if your data is available at the same frequency that you want to make predictions.
- **Feature Engineering**: Resampling can also be used to provide additional structure or insight into the learning problem for supervised learning models.

There is a lot of overlap between these two cases. For example, you may have daily data and want to predict a monthly problem. You could use the daily data directly or you could downsample it to monthly data and develop your model. A feature engineering perspective may use observations and summaries of observations from both time scales and more in developing a model. Let's make resampling more concrete by looking at a real dataset and some examples.

7.2 Shampoo Sales Dataset

In this lesson, we will use the Shampoo Sales dataset as an example. This dataset describes the monthly number of sales of shampoo over a 3 year period. You can learn more about the dataset in Appendix A.1. Place the dataset in your current working directory with the filename `shampoo-sales.csv`.

The Shampoo Sales dataset only specifies year number and months (e.g. 1-01). A custom function is used to parse the dates and baseline the year number on an arbitrary year (1900) so that the date-times can be be interpreted correctly in the Pandas `Series`.

7.3 Upsampling Data

The observations in the Shampoo Sales are monthly. Imagine we wanted daily sales information. We would have to upsample the frequency from monthly to daily and use an interpolation scheme to fill in the new daily frequency. The Pandas library provides a function called `resample()` on the `Series` and `DataFrame` objects[1]. This can be used to group records when downsampling and making space for new observations when upsampling.

We can use this function to transform our monthly dataset into a daily dataset by calling resampling and specifying the preferred frequency of calendar day frequency or `D`. Pandas is clever and you could just as easily specify the frequency as `1D` or even something domain specific, such as `5D`. Pandas provides a table of built-in aliases[2].

```
# upsample to daily intervals
from pandas import read_csv
from pandas import datetime

def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
upsampled = series.resample('D').mean()
print(upsampled.head(32))
```

Listing 7.1: Example of Upsampling the Shampoo Sales dataset.

[1] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.resample.html
[2] http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases

Running this example prints the first 32 rows of the upsampled dataset, showing each day of January and the first day of February.

```
Month
1901-01-01   266.0
1901-01-02     NaN
1901-01-03     NaN
1901-01-04     NaN
1901-01-05     NaN
1901-01-06     NaN
1901-01-07     NaN
1901-01-08     NaN
1901-01-09     NaN
1901-01-10     NaN
1901-01-11     NaN
1901-01-12     NaN
1901-01-13     NaN
1901-01-14     NaN
1901-01-15     NaN
1901-01-16     NaN
1901-01-17     NaN
1901-01-18     NaN
1901-01-19     NaN
1901-01-20     NaN
1901-01-21     NaN
1901-01-22     NaN
1901-01-23     NaN
1901-01-24     NaN
1901-01-25     NaN
1901-01-26     NaN
1901-01-27     NaN
1901-01-28     NaN
1901-01-29     NaN
1901-01-30     NaN
1901-01-31     NaN
1901-02-01   145.9
Freq: D, Name: Sales, dtype: float64
```

Listing 7.2: Example output of upsampling the Shampoo Sales dataset.

We can see that the `resample()` function has created the rows by putting `NaN` values in the new values. We can see we still have the sales volume on the first of January and February from the original data. Next, we can interpolate the missing values at this new frequency. The Series Pandas object provides an `interpolate()` function to interpolate missing values, and there is a nice selection of simple and more complex interpolation functions. You may have domain knowledge to help choose how values are to be interpolated. A good starting point is to use a linear interpolation. This draws a straight line between available data, in this case on the first of the month, and fills in values at the chosen frequency from this line.

```
# upsample to daily intervals with linear interpolation
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot

def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
```

```
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
upsampled = series.resample('D').mean()
interpolated = upsampled.interpolate(method='linear')
print(interpolated.head(32))
interpolated.plot()
pyplot.show()
```

Listing 7.3: Example of upsampling and interpolating of the Shampoo Sales dataset.

Running this example, we can see interpolated values.

```
Month
1901-01-01   266.000000
1901-01-02   262.125806
1901-01-03   258.251613
1901-01-04   254.377419
1901-01-05   250.503226
1901-01-06   246.629032
1901-01-07   242.754839
1901-01-08   238.880645
1901-01-09   235.006452
1901-01-10   231.132258
1901-01-11   227.258065
1901-01-12   223.383871
1901-01-13   219.509677
1901-01-14   215.635484
1901-01-15   211.761290
1901-01-16   207.887097
1901-01-17   204.012903
1901-01-18   200.138710
1901-01-19   196.264516
1901-01-20   192.390323
1901-01-21   188.516129
1901-01-22   184.641935
1901-01-23   180.767742
1901-01-24   176.893548
1901-01-25   173.019355
1901-01-26   169.145161
1901-01-27   165.270968
1901-01-28   161.396774
1901-01-29   157.522581
1901-01-30   153.648387
1901-01-31   149.774194
1901-02-01   145.900000
Freq: D, Name: Sales, dtype: float64
```

Listing 7.4: Example output of upsampling and interpolating of the Shampoo Sales dataset.

Looking at a line plot, we see no difference from plotting the original data (see Appendix) as the plot already interpolated the values between points to draw the line.

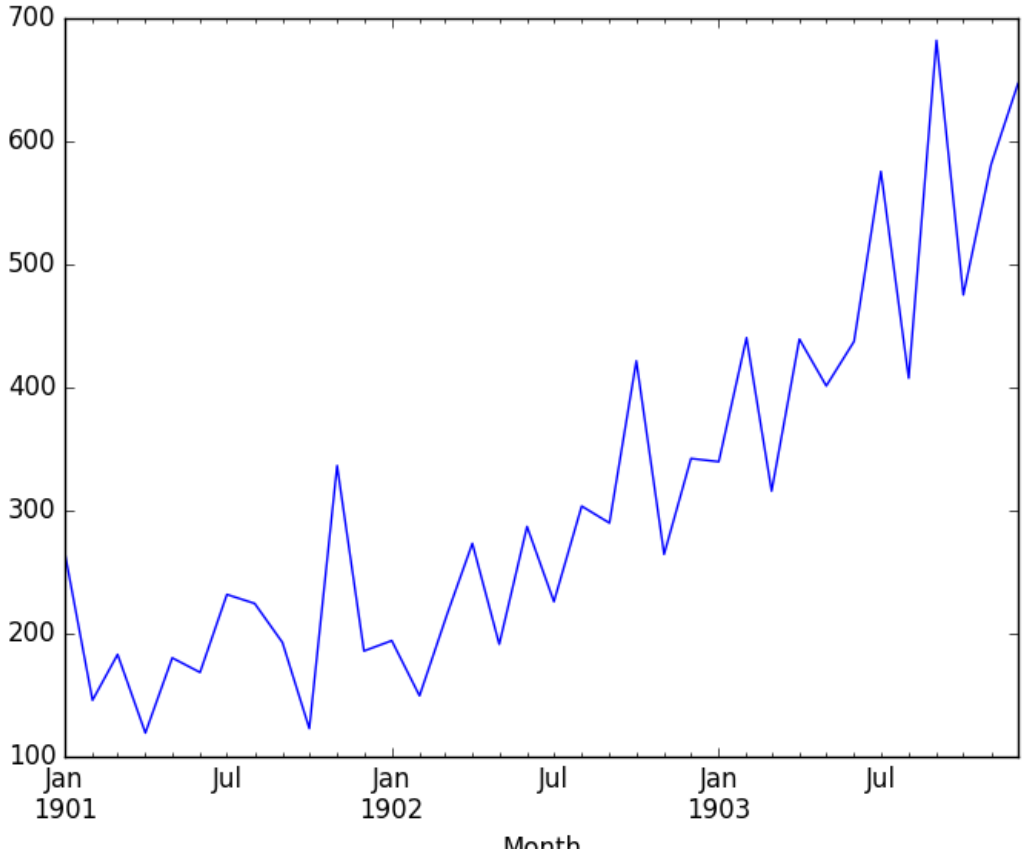

Figure 7.1: Line Plot of upsampled Shampoo Sales dataset with linear interpolation.

Another common interpolation method is to use a polynomial or a spline to connect the values. This creates more curves and can look more natural on many datasets. Using a spline interpolation requires you specify the order (number of terms in the polynomial); in this case, an order of 2 is just fine.

```
# upsample to daily intervals with spline interpolation
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot

def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
upsampled = series.resample('D').mean()
interpolated = upsampled.interpolate(method='spline', order=2)
print(interpolated.head(32))
interpolated.plot()
pyplot.show()
```

Listing 7.5: Example of upsampling and spline interpolation of the Shampoo Sales dataset.

Running the example, we can first review the raw interpolated values.

```
Month
1901-01-01   266.000000
1901-01-02   258.630160
1901-01-03   251.560886
1901-01-04   244.720748
1901-01-05   238.109746
1901-01-06   231.727880
1901-01-07   225.575149
1901-01-08   219.651553
1901-01-09   213.957094
1901-01-10   208.491770
1901-01-11   203.255582
1901-01-12   198.248529
1901-01-13   193.470612
1901-01-14   188.921831
1901-01-15   184.602185
1901-01-16   180.511676
1901-01-17   176.650301
1901-01-18   173.018063
1901-01-19   169.614960
1901-01-20   166.440993
1901-01-21   163.496161
1901-01-22   160.780465
1901-01-23   158.293905
1901-01-24   156.036481
1901-01-25   154.008192
1901-01-26   152.209039
1901-01-27   150.639021
1901-01-28   149.298139
1901-01-29   148.186393
1901-01-30   147.303783
1901-01-31   146.650308
1901-02-01   145.900000
Freq: D, Name: Sales, dtype: float64
```

Listing 7.6: Example output of upsampling and spline interpolation of the Shampoo Sales dataset.

Reviewing the line plot, we can see more natural curves on the interpolated values.

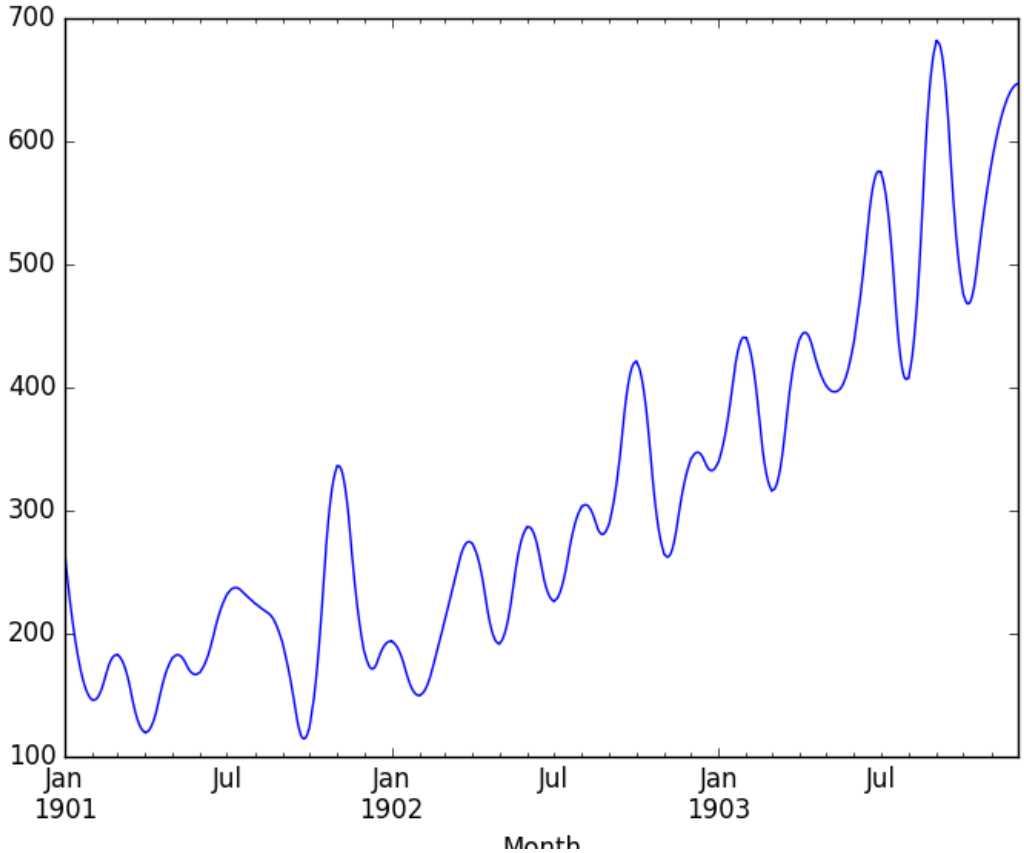

Figure 7.2: Line Plot of upsampled Shampoo Sales dataset with spline interpolation.

Generally, interpolation is a useful tool when you have missing observations. Next, we will consider resampling in the other direction and decreasing the frequency of observations.

7.4 Downsampling Data

The sales data is monthly, but perhaps we would prefer the data to be quarterly. The year can be divided into 4 business quarters, 3 months a piece. Instead of creating new rows between existing observations, the `resample()` function in Pandas will group all observations by the new frequency.

We could use an alias like `3M` to create groups of 3 months, but this might have trouble if our observations did not start in January, April, July, or October. Pandas does have a quarter-aware alias of `Q` that we can use for this purpose. We must now decide how to create a new quarterly value from each group of 3 records. A good starting point is to calculate the average monthly sales numbers for the quarter. For this, we can use the `mean()` function. Putting this all together, we get the following code example.

```
# downsample to quarterly intervals
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
```

```
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
resample = series.resample('Q')
quarterly_mean_sales = resample.mean()
print(quarterly_mean_sales.head())
quarterly_mean_sales.plot()
pyplot.show()
```

Listing 7.7: Example of downsampling of the Shampoo Sales dataset to quarterly means.

Running the example prints the first 5 rows of the quarterly data.

```
Month
1901-03-31   198.333333
1901-06-30   156.033333
1901-09-30   216.366667
1901-12-31   215.100000
1902-03-31   184.633333
Freq: Q-DEC, Name: Sales, dtype: float64
```

Listing 7.8: Example output of downsampling of the Shampoo Sales dataset to quarterly means.

We also plot the quarterly data, showing Q1-Q4 across the 3 years of original observations.

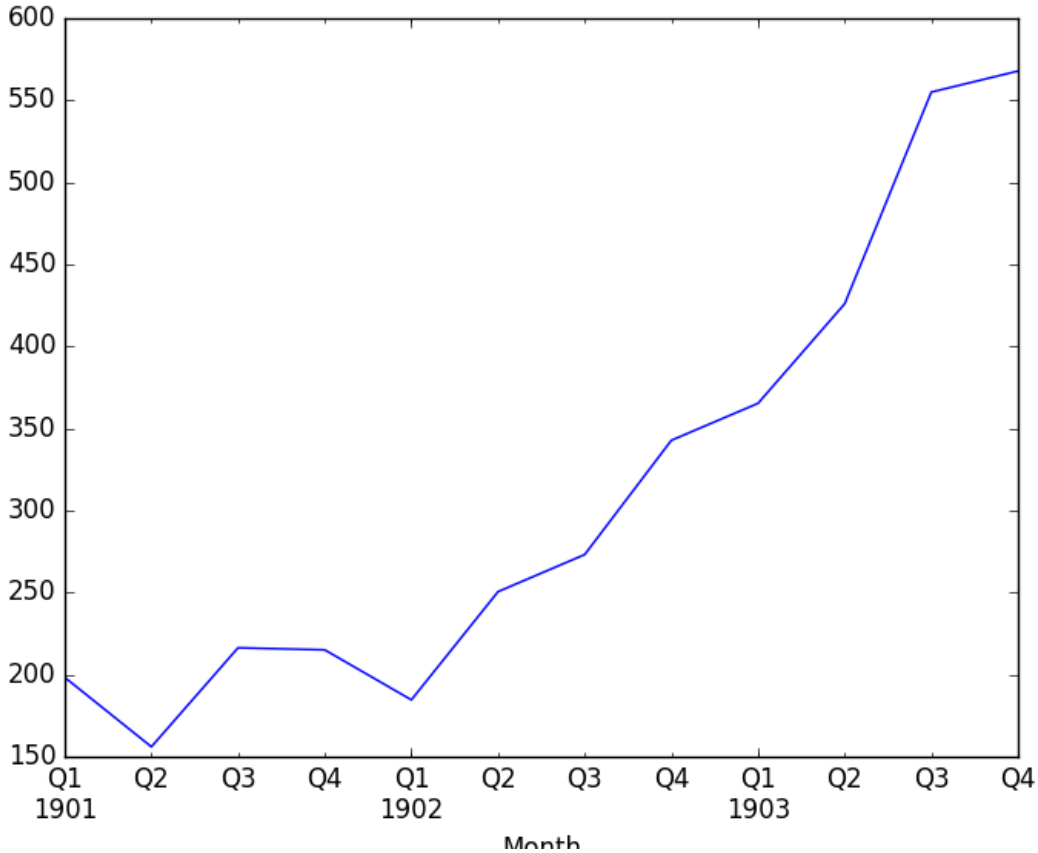

Figure 7.3: Line Plot of downsampling the Shampoo Sales dataset to quarterly mean values.

Perhaps we want to go further and turn the monthly data into yearly data, and perhaps later use that to model the following year. We can downsample the data using the alias `A` for year-end frequency and this time use sum to calculate the total sales each year.

```
# downsample to yearly intervals
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot

def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
resample = series.resample('A')
yearly_mean_sales = resample.sum()
print(yearly_mean_sales.head())
yearly_mean_sales.plot()
pyplot.show()
```

Listing 7.9: Example of downsampling of the Shampoo Sales dataset to yearly sum.

Running the example shows the 3 records for the 3 years of observations. We also get a plot, correctly showing the year along the x-axis and the total number of sales per year along the

y-axis.

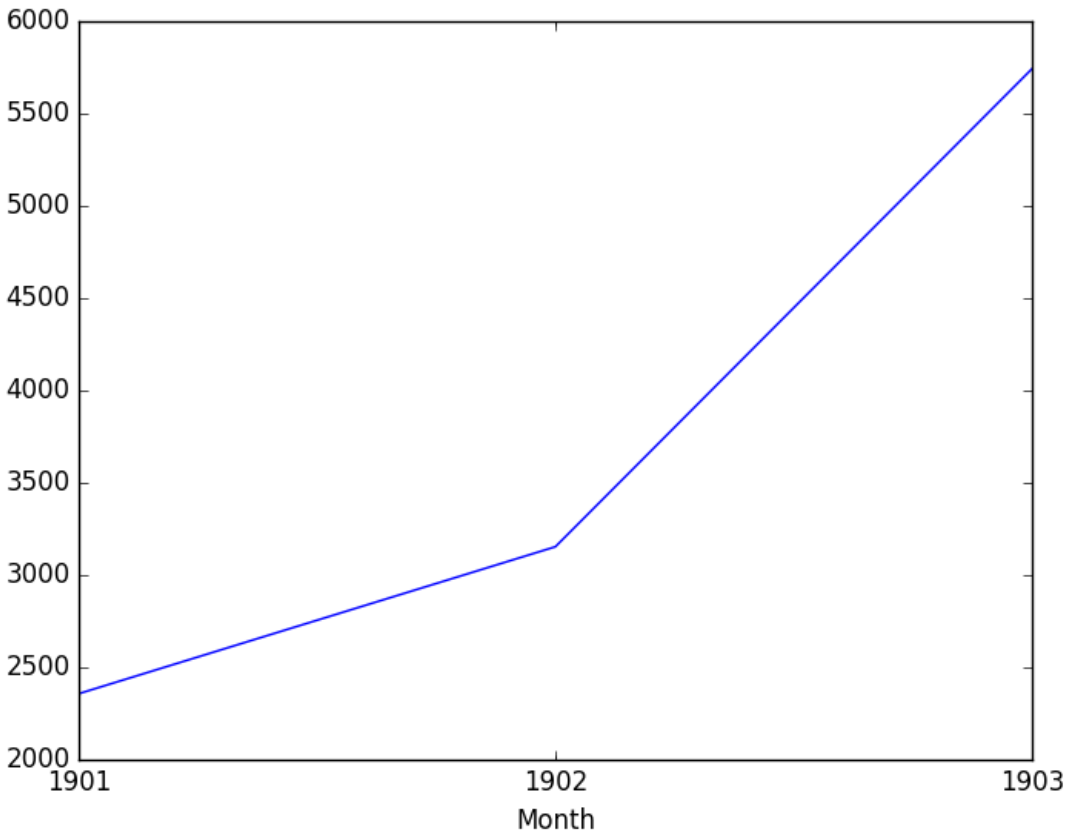

Figure 7.4: Line Plot of downsampling the Shampoo Sales dataset to yearly sum values.

7.5 Summary

In this tutorial, you discovered how to resample your time series data using Pandas in Python. Specifically, you learned:

- About time series resampling and the difference and reasons between downsampling and upsampling observation frequencies.
- How to upsample time series data using Pandas and how to use different interpolation schemes.
- How to downsample time series data using Pandas and how to summarize grouped data.

7.5.1 Next

In the next lesson you will discover how to change the variance of time series using power transforms.

Chapter 8

Power Transforms

Data transforms are intended to remove noise and improve the signal in time series forecasting. It can be very difficult to select a good, or even best, transform for a given prediction problem. There are many transforms to choose from and each has a different mathematical intuition. In this tutorial, you will discover how to explore different power-based transforms for time series forecasting with Python. After completing this tutorial, you will know:

- How to identify when to use and how to explore a square root transform.
- How to identify when to use and explore a log transform and the expectations on raw data.
- How to use the Box-Cox transform to perform square root, log, and automatically discover the best power transform for your dataset.

Let's get started.

8.1 Airline Passengers Dataset

In this lesson, we will use the Airline Passengers dataset as an example. This dataset describes the total number of airline passengers over time. You can learn more about the dataset in Appendix A.5. Place the dataset in your current working directory with the filename `airline-passengers.csv`. The example below loads the dataset and plots the data.

```
# load and plot a time series
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
pyplot.figure(1)
# line plot
pyplot.subplot(211)
pyplot.plot(series)
# histogram
pyplot.subplot(212)
pyplot.hist(series)
pyplot.show()
```

Listing 8.1: Load and Plot the Airline Passengers dataset.

Running the example creates two plots, the first showing the time series as a line plot and the second showing the observations as a histogram.

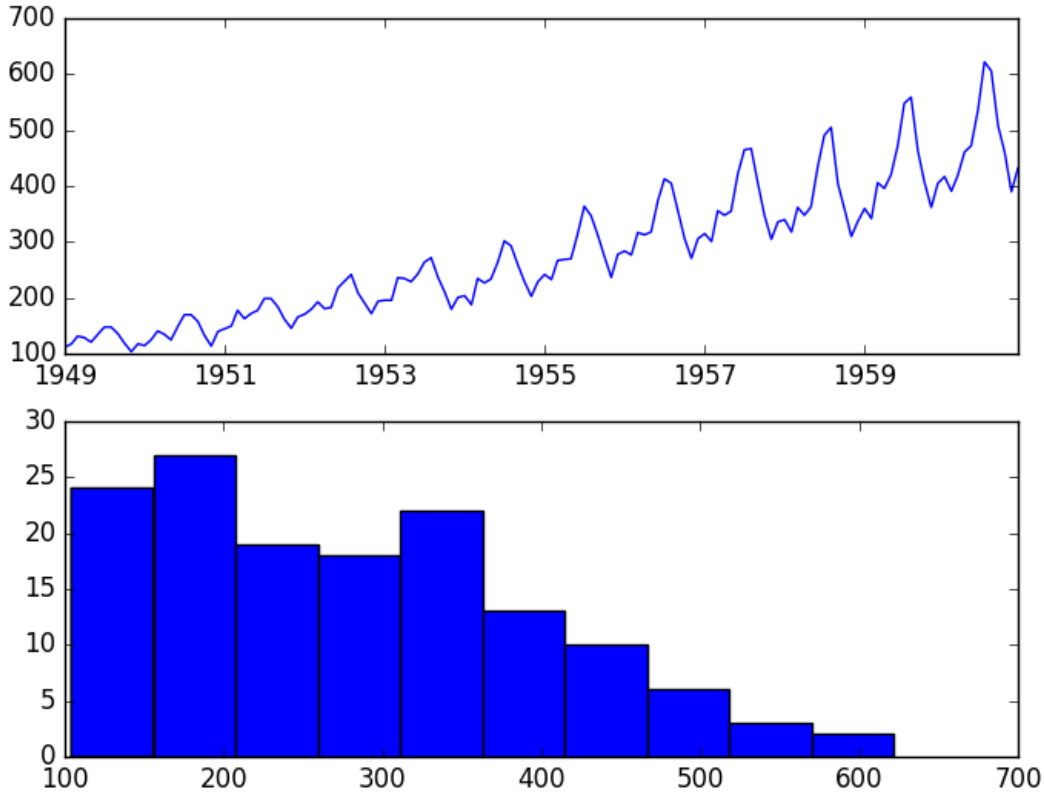

Figure 8.1: Line and density plots of the Airline Passengers dataset.

The dataset is non-stationary, meaning that the mean and the variance of the observations change over time. This makes it difficult to model by both classical statistical methods, like ARIMA, and more sophisticated machine learning methods, like neural networks. This is caused by what appears to be both an increasing trend and a seasonality component.

In addition, the amount of change, or the variance, is increasing with time. This is clear when you look at the size of the seasonal component and notice that from one cycle to the next, the amplitude (from bottom to top of the cycle) is increasing. In this tutorial, we will investigate transforms that we can use on time series datasets that exhibit this property.

8.2 Square Root Transform

A time series that has a quadratic growth trend can be made linear by taking the square root. Let's demonstrate this with a quick contrived example. Consider a series of the numbers 1 to 99 squared. The line plot of this series will show a quadratic growth trend and a histogram of the values will show an exponential distribution with a long trail. The snippet of code below creates and graphs this series.

```
# contrive a quadratic time series
from matplotlib import pyplot
series = [i**2 for i in range(1,100)]
pyplot.figure(1)
# line plot
pyplot.subplot(211)
pyplot.plot(series)
# histogram
pyplot.subplot(212)
pyplot.hist(series)
pyplot.show()
```

Listing 8.2: Contrived quadratic series.

Running the example plots the series both as a line plot over time and a histogram of observations.

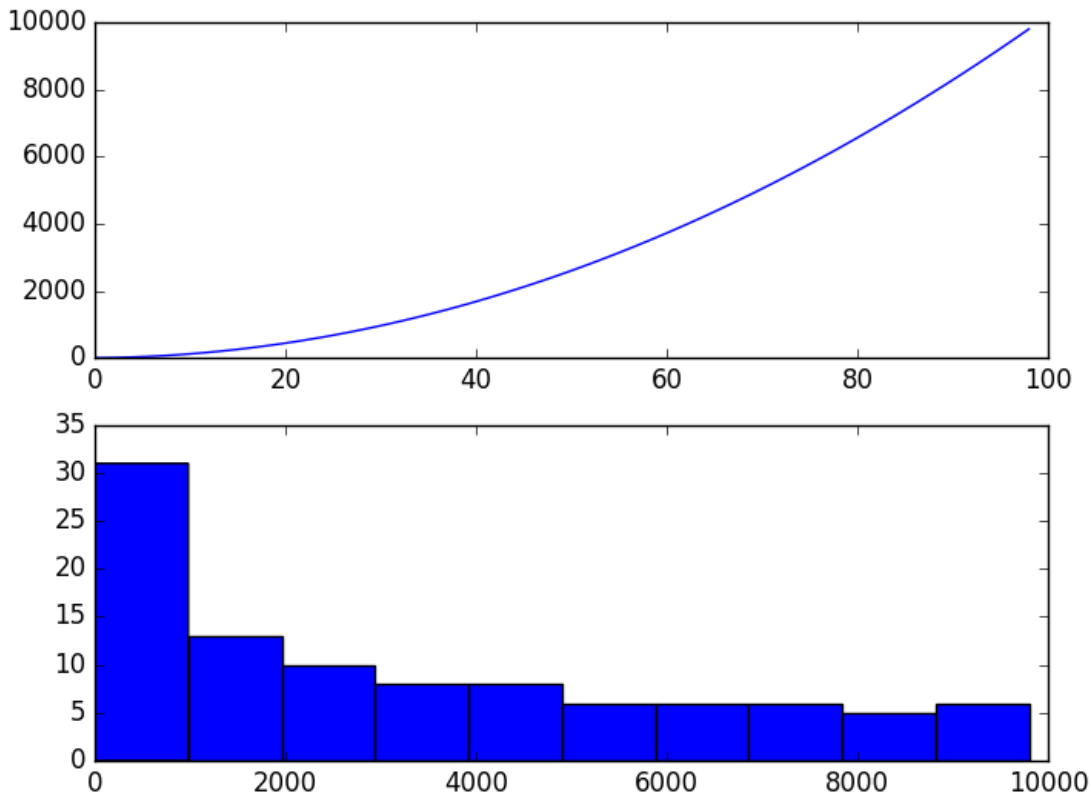

Figure 8.2: Line and density plots of the contrived quadratic series dataset.

If you see a structure like this in your own time series, you may have a quadratic growth trend. This can be removed or made linear by taking the inverse operation of the squaring procedure, which is the square root. Because the example is perfectly quadratic, we would expect the line plot of the transformed data to show a straight line. Because the source of the

squared series is linear, we would expect the histogram to show a uniform distribution. The example below performs a `sqrt()` transform on the time series and plots the result.

```
# square root transform a contrived quadratic time series
from matplotlib import pyplot
from numpy import sqrt
series = [i**2 for i in range(1,100)]
# sqrt transform
transform = series = sqrt(series)
pyplot.figure(1)
# line plot
pyplot.subplot(211)
pyplot.plot(transform)
# histogram
pyplot.subplot(212)
pyplot.hist(transform)
pyplot.show()
```

Listing 8.3: Square root transform of the quadratic series.

We can see that, as expected, the quadratic trend was made linear.

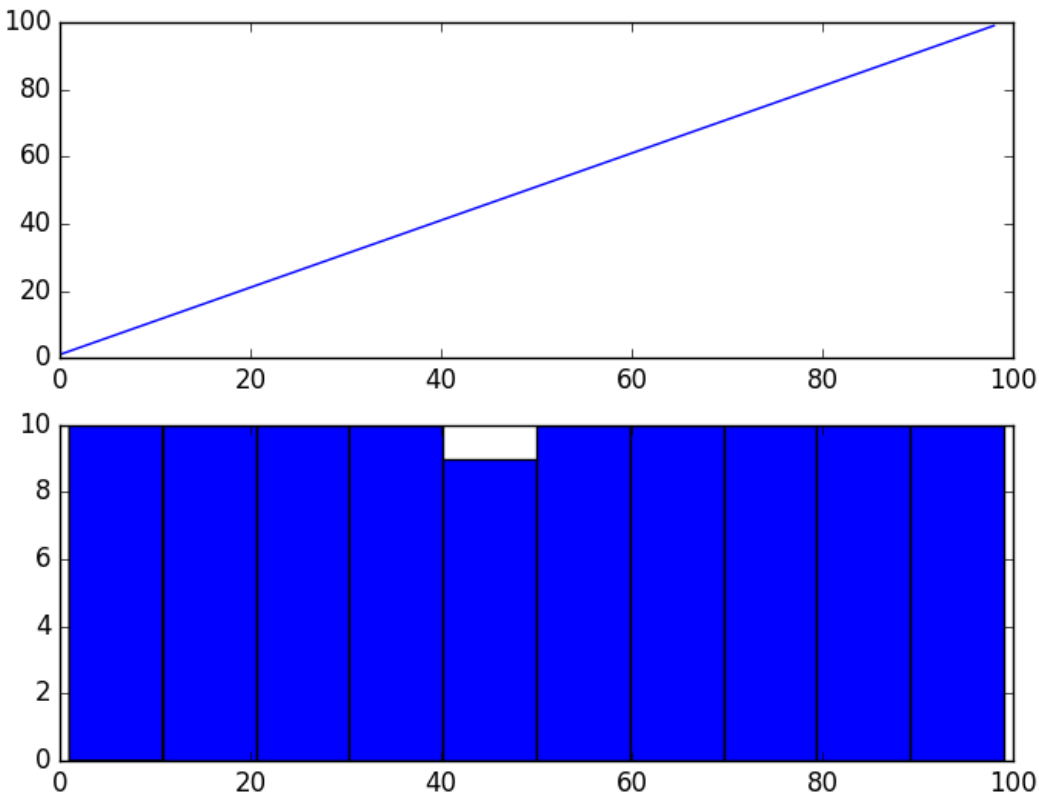

Figure 8.3: Line and density plots of the square root transformed quadratic series dataset.

It is possible that the Airline Passengers dataset shows a quadratic growth. If this is the case, then we could expect a square root transform to reduce the growth trend to be linear

and change the distribution of observations to be perhaps nearly Gaussian. The example below performs a square root of the dataset and plots the results.

```
# square root transform a time series
from pandas import read_csv
from pandas import DataFrame
from numpy import sqrt
from matplotlib import pyplot
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
dataframe = DataFrame(series.values)
dataframe.columns = ['passengers']
dataframe['passengers'] = sqrt(dataframe['passengers'])
pyplot.figure(1)
# line plot
pyplot.subplot(211)
pyplot.plot(dataframe['passengers'])
# histogram
pyplot.subplot(212)
pyplot.hist(dataframe['passengers'])
pyplot.show()
```

Listing 8.4: Square root transform of the Airline Passengers dataset.

We can see that the trend was reduced, but was not removed. The line plot still shows an increasing variance from cycle to cycle. The histogram still shows a long tail to the right of the distribution, suggesting an exponential or long-tail distribution.

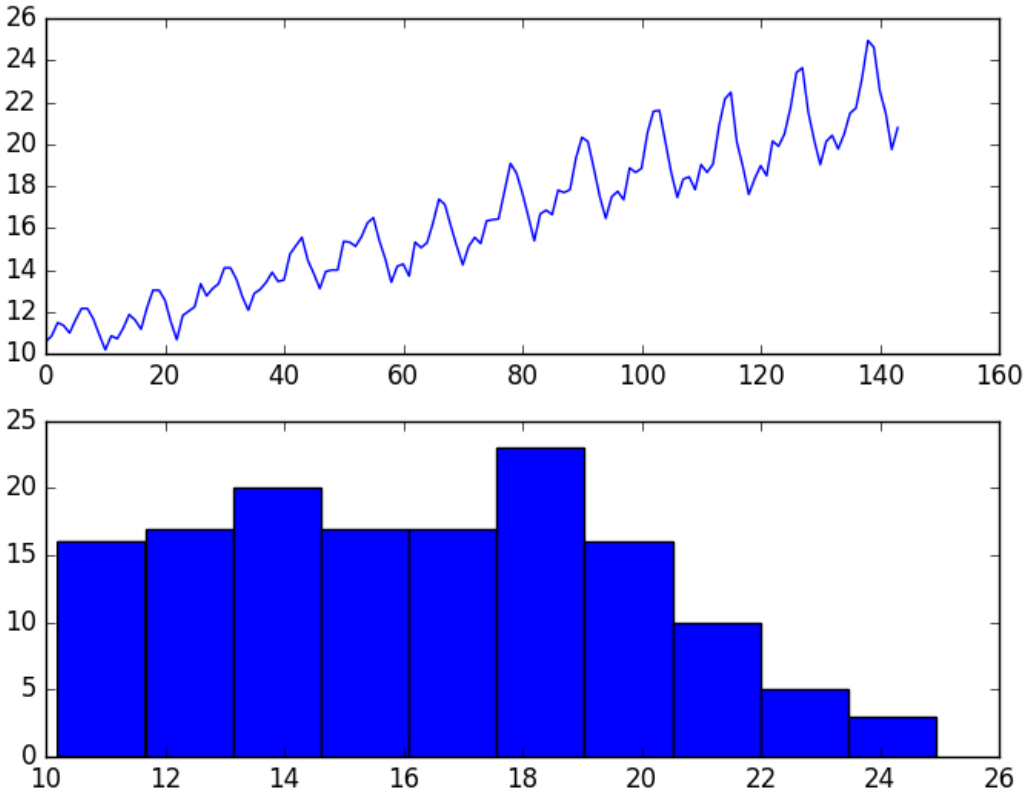

Figure 8.4: Line and density plots of the square root transformed Airline Passengers dataset.

8.3 Log Transform

A class of more extreme trends are exponential, often graphed as a hockey stick. Time series with an exponential distribution can be made linear by taking the logarithm of the values. This is called a log transform. As with the square and square root case above, we can demonstrate this with a quick example. The code below creates an exponential distribution by raising the numbers from 1 to 99 to the value e, which is the base of the natural logarithms or Euler's number (2.718...)[1].

```
# create and plot an exponential time series
from matplotlib import pyplot
from math import exp
series = [exp(i) for i in range(1,100)]
pyplot.figure(1)
# line plot
pyplot.subplot(211)
pyplot.plot(series)
# histogram
pyplot.subplot(212)
pyplot.hist(series)
```

[1] https://en.wikipedia.org/wiki/E_(mathematical_constant)

```
pyplot.show()
```

Listing 8.5: Contrived exponential series.

Running the example creates a line plot of the series and a histogram of the distribution of observations. We see an extreme increase on the line graph and an equally extreme long tail distribution on the histogram.

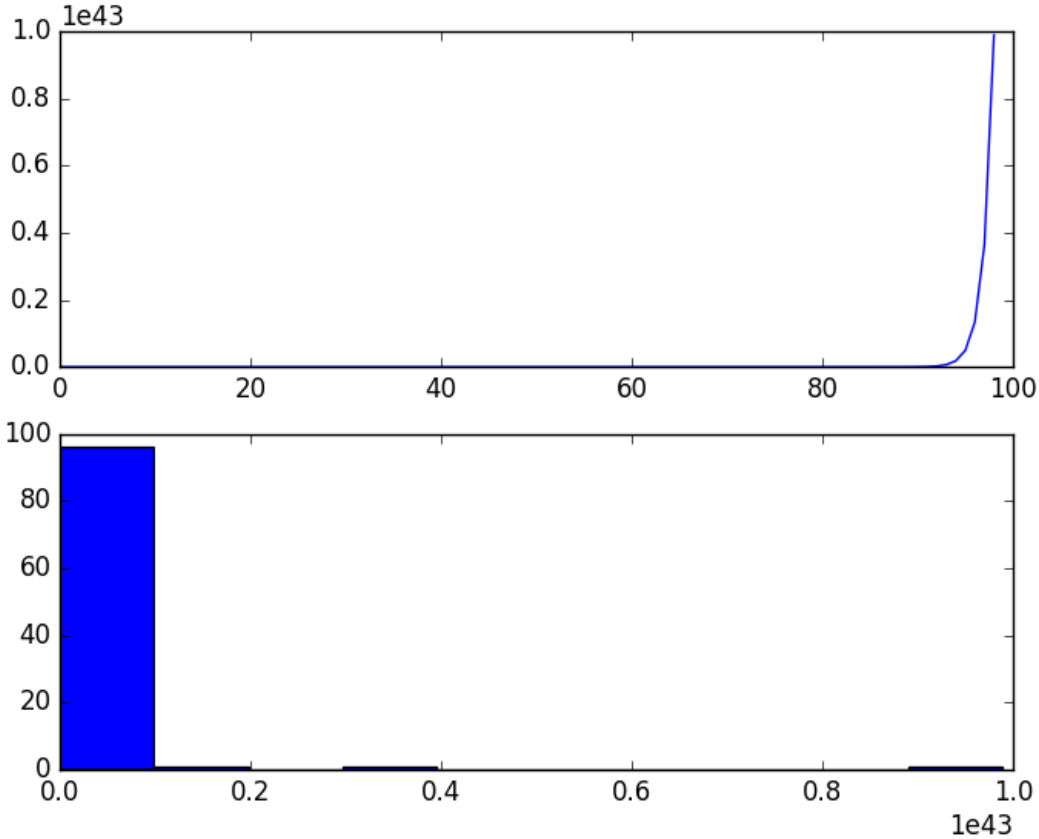

Figure 8.5: Line and density plots of the contrived exponential series dataset.

Again, we can transform this series back to linear by taking the natural logarithm of the values. This would make the series linear and the distribution uniform. The example below demonstrates this for completeness.

```
# log transform a contrived exponential time series
from matplotlib import pyplot
from math import exp
from numpy import log
series = [exp(i) for i in range(1,100)]
transform = log(series)
pyplot.figure(1)
# line plot
pyplot.subplot(211)
pyplot.plot(transform)
# histogram
```

```
pyplot.subplot(212)
pyplot.hist(transform)
pyplot.show()
```

Listing 8.6: Log transform of the exponential dataset.

Running the example creates plots, showing the expected linear result.

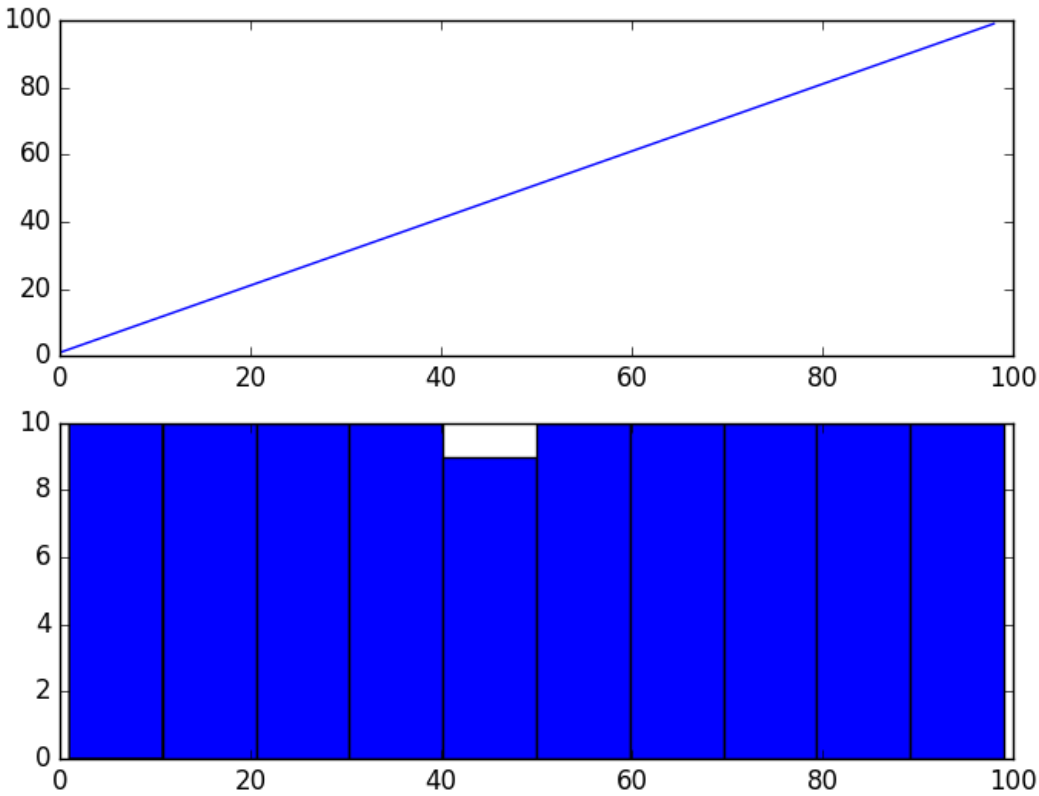

Figure 8.6: Line and density plots of the log transformed exponential dataset.

Our Airline Passengers dataset has a distribution of this form, but perhaps not this extreme. The example below demonstrates a log transform of the Airline Passengers dataset.

```
# log transform a time series
from pandas import read_csv
from pandas import DataFrame
from numpy import log
from matplotlib import pyplot
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
dataframe = DataFrame(series.values)
dataframe.columns = ['passengers']
dataframe['passengers'] = log(dataframe['passengers'])
pyplot.figure(1)
# line plot
pyplot.subplot(211)
```

```
pyplot.plot(dataframe['passengers'])
# histogram
pyplot.subplot(212)
pyplot.hist(dataframe['passengers'])
pyplot.show()
```

Listing 8.7: Log transform of the Airline Passengers dataset.

Running the example results in a trend that does look a lot more linear than the square root transform above. The line plot shows a seemingly linear growth and variance. The histogram also shows a more uniform or squashed Gaussian-like distribution of observations.

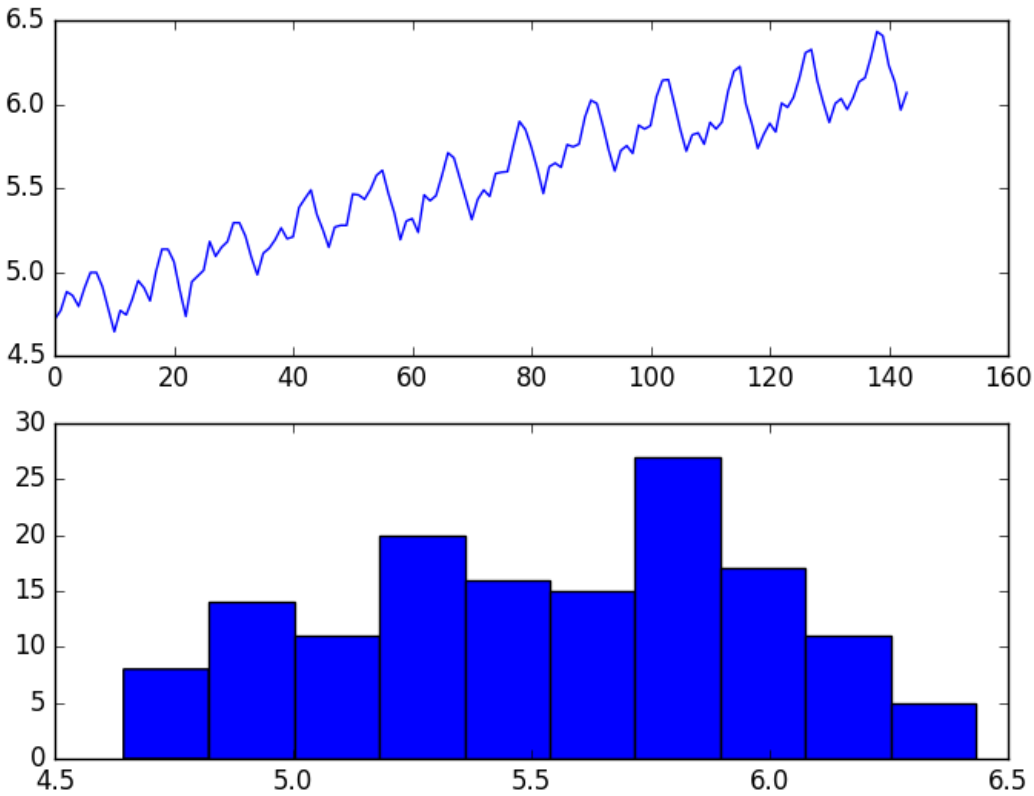

Figure 8.7: Line and density plots of the log transformed Airline Passengers dataset.

Log transforms are popular with time series data as they are effective at removing exponential variance. It is important to note that this operation assumes values are positive and non-zero. It is common to transform observations by adding a fixed constant to ensure all input values meet this requirement. For example:

$$transform = log(constant + x) \tag{8.1}$$

Where `log` is the natural logarithm, `transform` is the transformed series, `constant` is a fixed value that lifts all observations above zero, and `x` is the time series.

8.4 Box-Cox Transform

The square root transform and log transform belong to a class of transforms called power transforms. The Box-Cox transform[2] is a configurable data transform method that supports both square root and log transform, as well as a suite of related transforms.

More than that, it can be configured to evaluate a suite of transforms automatically and select a best fit. It can be thought of as a power tool to iron out power-based change in your time series. The resulting series may be more linear and the resulting distribution more Gaussian or Uniform, depending on the underlying process that generated it. The `scipy.stats` library provides an implementation of the Box-Cox transform. The `boxcox()` function[3] takes an argument, called `lambda`, that controls the type of transform to perform.

Below are some common values for `lambda`:

- `lambda` = -1. is a reciprocal transform.
- `lambda` = -0.5 is a reciprocal square root transform.
- `lambda` = 0.0 is a log transform.
- `lambda` = 0.5 is a square root transform.
- `lambda` = 1.0 is no transform.

For example, we can perform a log transform using the `boxcox()` function as follows:

```
# manually box-cox transform a time series
from pandas import read_csv
from pandas import DataFrame
from scipy.stats import boxcox
from matplotlib import pyplot
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
dataframe = DataFrame(series.values)
dataframe.columns = ['passengers']
dataframe['passengers'] = boxcox(dataframe['passengers'], lmbda=0.0)
pyplot.figure(1)
# line plot
pyplot.subplot(211)
pyplot.plot(dataframe['passengers'])
# histogram
pyplot.subplot(212)
pyplot.hist(dataframe['passengers'])
pyplot.show()
```

Listing 8.8: Box-Cox transform of the Airline Passengers dataset.

Running the example reproduces the log transform from the previous section.

[2] https://en.wikipedia.org/wiki/Power_transform#Box.E2.80.93Cox_transformation
[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.boxcox.html

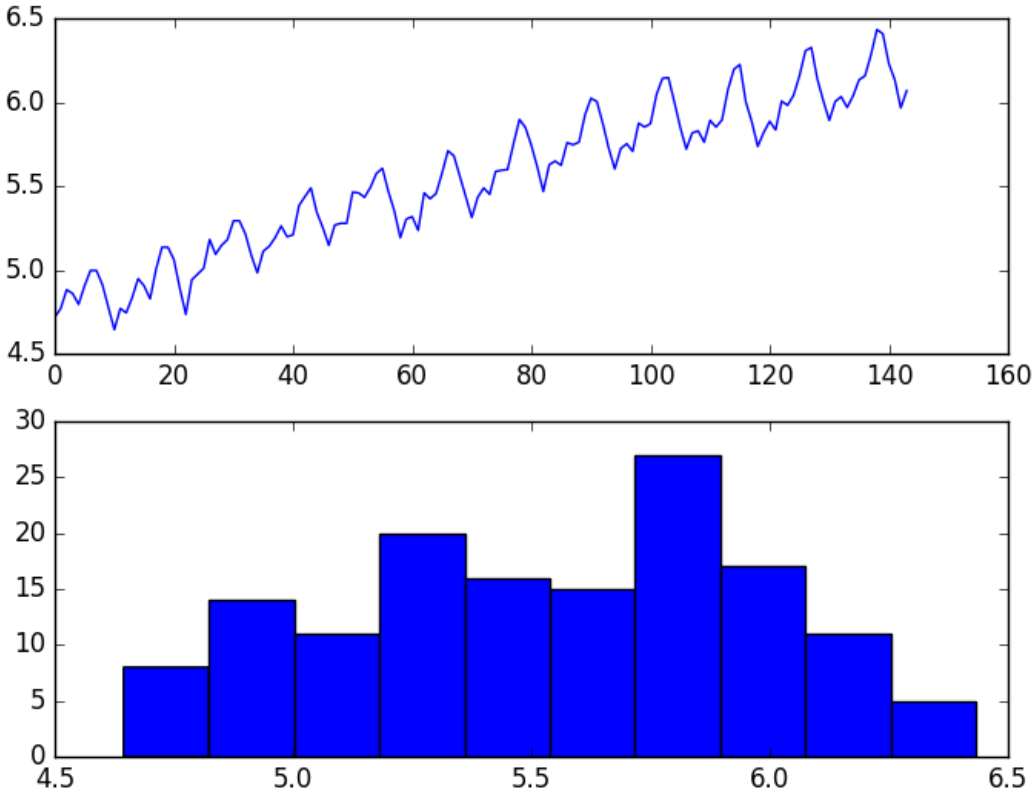

Figure 8.8: Line and density plots of the Box-Cox transformed Airline Passengers dataset.

We can set the `lambda` parameter to `None` (the default) and let the function find a statistically tuned value. The following example demonstrates this usage, returning both the transformed dataset and the chosen `lambda` value.

```
# automatically box-cox transform a time series
from pandas import read_csv
from pandas import DataFrame
from scipy.stats import boxcox
from matplotlib import pyplot
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
dataframe = DataFrame(series.values)
dataframe.columns = ['passengers']
dataframe['passengers'], lam = boxcox(dataframe['passengers'])
print('Lambda: %f' % lam)
pyplot.figure(1)
# line plot
pyplot.subplot(211)
pyplot.plot(dataframe['passengers'])
# histogram
pyplot.subplot(212)
pyplot.hist(dataframe['passengers'])
pyplot.show()
```

Listing 8.9: Optimized Box-Cox transform of the Airline Passengers dataset.

Running the example discovers the `lambda` value of 0.148023. We can see that this is very close to a `lambda` value of 0.0, resulting in a log transform and stronger (less than) than 0.5 for the square root transform.

```
Lambda: 0.148023
```

Listing 8.10: Example output the optimized Box-Cox transform on the Airline Passengers dataset.

The line and histogram plots are also very similar to those from the log transform.

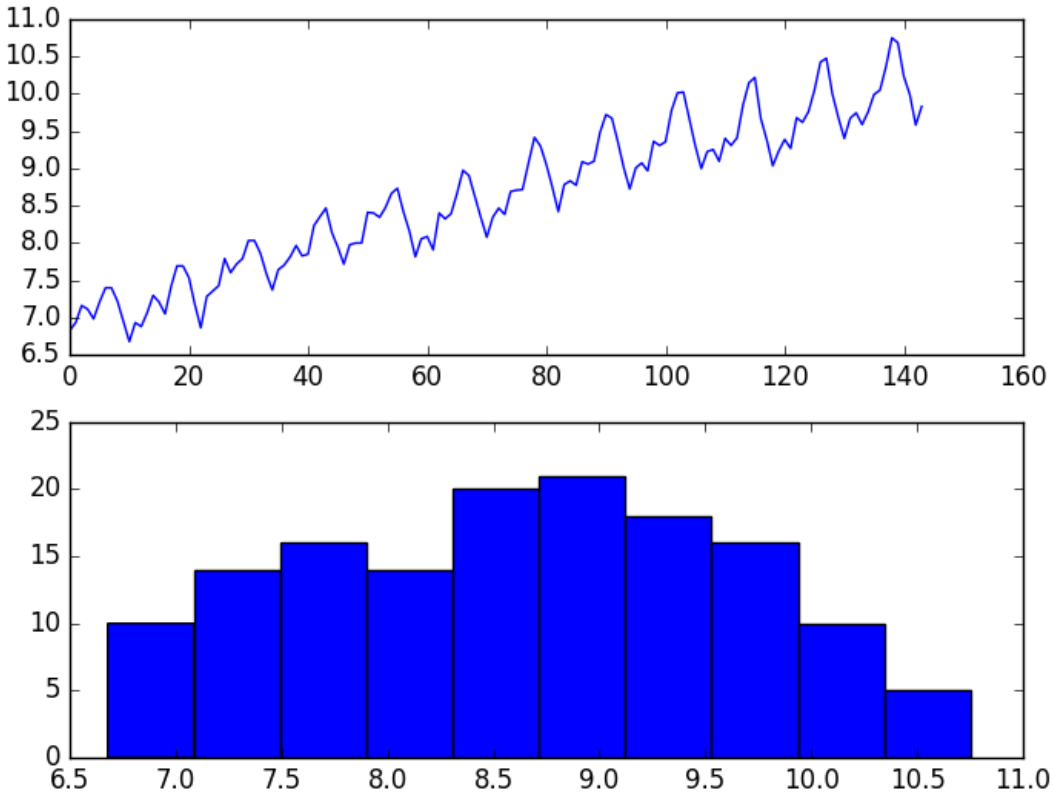

Figure 8.9: Line and density plots of the Optimized Box-Cox transformed Airline Passengers dataset.

8.5 Summary

In this tutorial, you discovered how to identify when to use and how to use different power transforms on time series data with Python. Specifically, you learned:

- How to identify a quadratic change and use the square root transform.

- How to identify an exponential change and how to use the log transform.
- How to use the Box-Cox transform to perform square root and log transforms and automatically optimize the transform for a dataset.

8.5.1 Next

In the next lesson you will discover how to use smoothing methods on a time series.

Chapter 9

Moving Average Smoothing

Moving average smoothing is a naive and effective technique in time series forecasting. It can be used for data preparation, feature engineering, and even directly for making predictions. In this tutorial, you will discover how to use moving average smoothing for time series forecasting with Python. After completing this tutorial, you will know:

- How moving average smoothing works and some expectations of your data before you can use it.
- How to use moving average smoothing for data preparation and feature engineering.
- How to use moving average smoothing to make predictions.

Let's get started.

9.1 Moving Average Smoothing

Smoothing is a technique applied to time series to remove the fine-grained variation between time steps. The hope of smoothing is to remove noise and better expose the signal of the underlying causal processes. Moving averages are a simple and common type of smoothing used in time series analysis and time series forecasting. Calculating a moving average involves creating a new series where the values are comprised of the average of raw observations in the original time series.

A moving average requires that you specify a window size called the window width. This defines the number of raw observations used to calculate the moving average value. The *moving* part in the moving average refers to the fact that the window defined by the window width is slid along the time series to calculate the average values in the new series. There are two main types of moving average that are used: Centered and Trailing Moving Average.

9.1.1 Centered Moving Average

The value at time (`t`) is calculated as the average of raw observations at, before, and after time (`t`). For example, a center moving average with a window of 3 would be calculated as:

$$center_ma(t) = mean(obs(t-1), obs(t), obs(t+1)) \tag{9.1}$$

This method requires knowledge of future values, and as such is used on time series analysis to better understand the dataset. A center moving average can be used as a general method to remove trend and seasonal components from a time series, a method that we often cannot use when forecasting.

9.1.2 Trailing Moving Average

The value at time (`t`) is calculated as the average of the raw observations at and before the time (`t`). For example, a trailing moving average with a window of 3 would be calculated as:

$$trail_ma(t) = mean(obs(t-2), obs(t-1), obs(t)) \tag{9.2}$$

Trailing moving average only uses historical observations and is used on time series forecasting. It is the type of moving average that we will focus on in this tutorial.

9.2 Data Expectations

Calculating a moving average of a time series makes some assumptions about your data. It is assumed that both trend and seasonal components have been removed from your time series. This means that your time series is stationary, or does not show obvious trends (long-term increasing or decreasing movement) or seasonality (consistent periodic structure).

There are many methods to remove trends and seasonality from a time series dataset when forecasting. Two good methods for each are to use the differencing method and to model the behavior and explicitly subtract it from the series.

Moving average values can be used in a number of ways when using machine learning algorithms on time series problems. In this tutorial, we will look at how we can calculate trailing moving average values for use as data preparation, feature engineering, and for directly making predictions. Before we dive into these examples, let's look at the Daily Female Births dataset that we will use in each example.

9.3 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`.

9.4 Moving Average as Data Preparation

Moving average can be used as a data preparation technique to create a smoothed version of the original dataset. Smoothing is useful as a data preparation technique as it can reduce the random variation in the observations and better expose the structure of the underlying causal processes.

The `rolling()` function[1] on the Series Pandas object will automatically group observations into a window. You can specify the window size, and by default a trailing window is created. Once the window is created, we can take the mean value, and this is our transformed dataset.

New observations in the future can be just as easily transformed by keeping the raw values for the last few observations and updating a new average value. To make this concrete, with a window size of 3, the transformed value at time (`t`) is calculated as the mean value for the previous 3 observations (`t-2`, `t-1`, `t`), as follows:

$$obs(t) = \frac{1}{3} \times (obs(t-2) + obs(t-1) + obs(t)) \tag{9.3}$$

For the Daily Female Births dataset, the first moving average would be on January 3rd, as follows:

$$\begin{aligned} obs(t) &= \frac{1}{3} \times (obs(t-2) + obs(t-1) + obs(t)) \\ &= \frac{1}{3} \times (35 + 32 + 30) \\ &= 32.333 \end{aligned} \tag{9.4}$$

Below is an example of transforming the Daily Female Births dataset into a moving average with a window size of 3 days, chosen arbitrarily.

```
# moving average smoothing as data preparation
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# tail-rolling average transform
rolling = series.rolling(window=3)
rolling_mean = rolling.mean()
print(rolling_mean.head(10))
# plot original and transformed dataset
series.plot()
rolling_mean.plot(color='red')
pyplot.show()
# zoomed plot original and transformed dataset
series[:100].plot()
rolling_mean[:100].plot(color='red')
pyplot.show()
```

Listing 9.1: Example of moving average as data preparation on the Daily Female Births dataset.

Running the example prints the first 10 observations from the transformed dataset. We can see that the first 2 observations will need to be discarded.

```
Date
1959-01-01        NaN
1959-01-02        NaN
1959-01-03  32.333333
1959-01-04  31.000000
1959-01-05  35.000000
1959-01-06  34.666667
```

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

```
1959-01-07   39.333333
1959-01-08   39.000000
1959-01-09   42.000000
1959-01-10   36.000000
Name: Births, dtype: float64
```

Listing 9.2: Example output of moving average as data preparation on the Daily Female Births dataset.

The raw observations are plotted with the moving average transform overlaid.

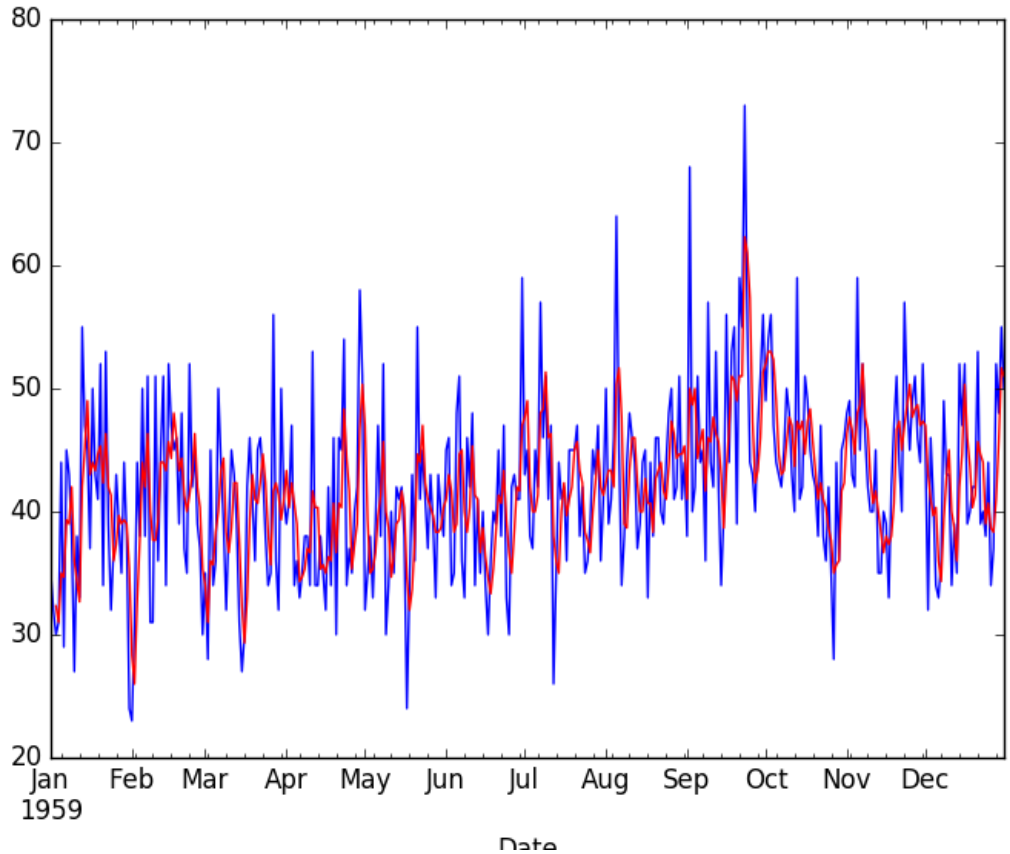

Figure 9.1: Line plot of the Daily Female Births dataset (blue) with a moving average (red).

To get a better idea of the effect of the transform, we can zoom in and plot the first 100 observations.

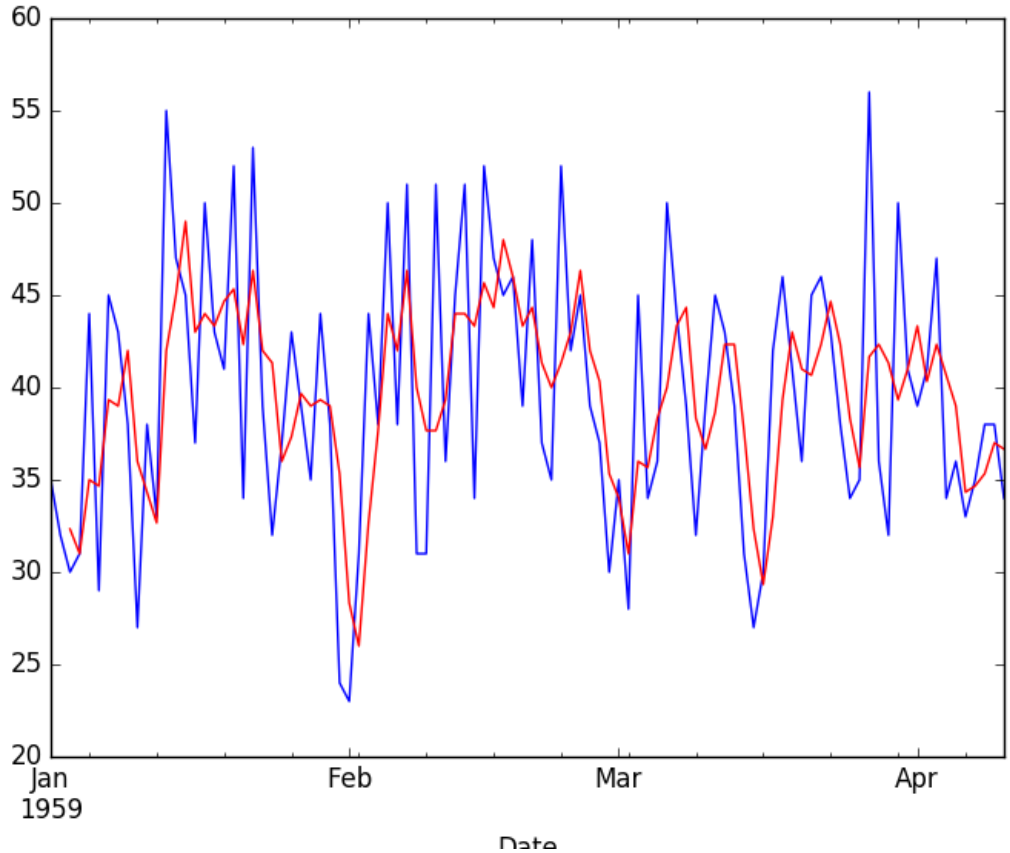

Figure 9.2: Line plot of the first 100 observations from the Daily Female Births dataset (blue) with a moving average (red).

Here, you can clearly see the lag in the transformed dataset. Next, let's take a look at using the moving average as a feature engineering method.

9.5 Moving Average as Feature Engineering

The moving average can be used as a source of new information when modeling a time series forecast as a supervised learning problem. In this case, the moving average is calculated and added as a new input feature used to predict the next time step. First, a copy of the series must be shifted forward by one time step. This will represent the input to our prediction problem, or a `lag=1` version of the series. This is a standard supervised learning view of the time series problem. For example:

```
X,     y
NaN,   obs1
obs1,  obs2
obs2,  obs3
```

Listing 9.3: Contrived example of a shifted time series with lag variables.

Next, a second copy of the series needs to be shifted forward by one, minus the window size. This is to ensure that the moving average summarizes the last few values and does not include the value to be predicted in the average, which would be an invalid framing of the problem as the input would contain knowledge of the future being predicted.

For example, with a window size of 3, we must shift the series forward by 2 time steps. This is because we want to include the previous two observations as well as the current observation in the moving average in order to predict the next value. We can then calculate the moving average from this shifted series. Below is an example of how the first 5 moving average values are calculated. Remember, the dataset is shifted forward 2 time steps and as we move along the time series, it takes at least 3 time steps before we even have enough data to calculate a `window=3` moving average.

```
Observations,   Mean
NaN          NaN
NaN, NaN       NaN
NaN, NaN, 35     NaN
NaN, 35, 32      NaN
30, 32, 35       32
```

Listing 9.4: Contrived example of a shifted time series with lag variables and a moving average.

Below is an example of including the moving average of the previous 3 values as a new feature, as well as the original `t` input feature for the Daily Female Births dataset.

```
# moving average smoothing as feature engineering
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
df = DataFrame(series.values)
width = 3
lag1 = df.shift(1)
lag3 = df.shift(width - 1)
window = lag3.rolling(window=width)
means = window.mean()
dataframe = concat([means, lag1, df], axis=1)
dataframe.columns = ['mean', 't', 't+1']
print(dataframe.head(10))
```

Listing 9.5: Example of moving average as feature engineering on the Daily Female Births dataset.

Running the example creates the new dataset and prints the first 10 rows. We can see that the first 3 rows cannot be used and must be discarded. The first row of the `lag=1` dataset cannot be used because there are no previous observations to predict the first observation, therefore a `NaN` value is used.

```
        mean   t    t+1
0        NaN   NaN   35
1        NaN  35.0   32
2        NaN  32.0   30
3        NaN  30.0   31
4  32.333333 31.0   44
5  31.000000 44.0   29
```

```
6  35.000000 29.0  45
7  34.666667 45.0  43
8  39.333333 43.0  38
9  39.000000 38.0  27
```

Listing 9.6: Example output of moving average as feature engineering on the Daily Female Births dataset.

The next section will look at how to use the moving average as a naive model to make predictions.

9.6 Moving Average as Prediction

The moving average value can also be used directly to make predictions. It is a naive model and assumes that the trend and seasonality components of the time series have already been removed or adjusted for. The moving average model for predictions can easily be used in a walk-forward manner. As new observations are made available (e.g. daily), the model can be updated and a prediction made for the next day. We can implement this manually in Python. Below is an example of the moving average model used in a walk-forward manner.

```
# moving average smoothing as a forecast model
from math import sqrt
from pandas import read_csv
from numpy import mean
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# prepare situation
X = series.values
window = 3
history = [X[i] for i in range(window)]
test = [X[i] for i in range(window, len(X))]
predictions = list()
# walk forward over time steps in test
for t in range(len(test)):
  length = len(history)
  yhat = mean([history[i] for i in range(length-window,length)])
  obs = test[t]
  predictions.append(yhat)
  history.append(obs)
  print('predicted=%f, expected=%f' % (yhat, obs))
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
# plot
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
# zoom plot
pyplot.plot(test[:100])
pyplot.plot(predictions[:100], color='red')
pyplot.show()
```

Listing 9.7: Example of moving average for prediction on the Daily Female Births dataset.

Running the example prints the predicted and expected value each time step moving forward, starting from time step 4 (1959-01-04). Finally, the root mean squared error (RMSE) is reported for all predictions made (RMSE is covered later in Chapter 17).

```
predicted=32.333333, expected=31.000000
predicted=31.000000, expected=44.000000
predicted=35.000000, expected=29.000000
...
predicted=38.666667, expected=37.000000
predicted=38.333333, expected=52.000000
predicted=41.000000, expected=48.000000
predicted=45.666667, expected=55.000000
predicted=51.666667, expected=50.000000
Test RMSE: 7.834
```

Listing 9.8: Example output of moving average for prediction on the Daily Female Births dataset.

The results show a RMSE of nearly 7.834 births per day. The example ends by plotting the expected test values compared to the predictions.

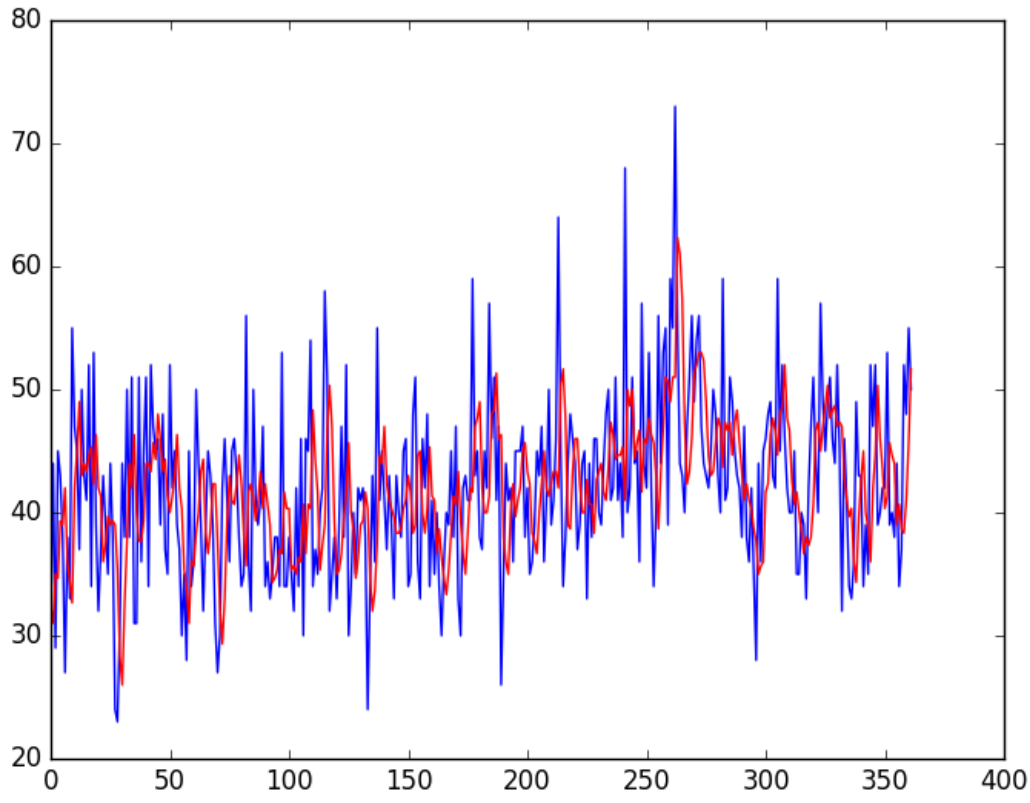

Figure 9.3: Line plot of the Daily Female Births dataset (blue) with a moving average predictions (red).

Again, zooming in on the first 100 predictions gives an idea of the skill of the 3-day moving

average predictions. Note the window width of 3 was chosen arbitrary and was not optimized.

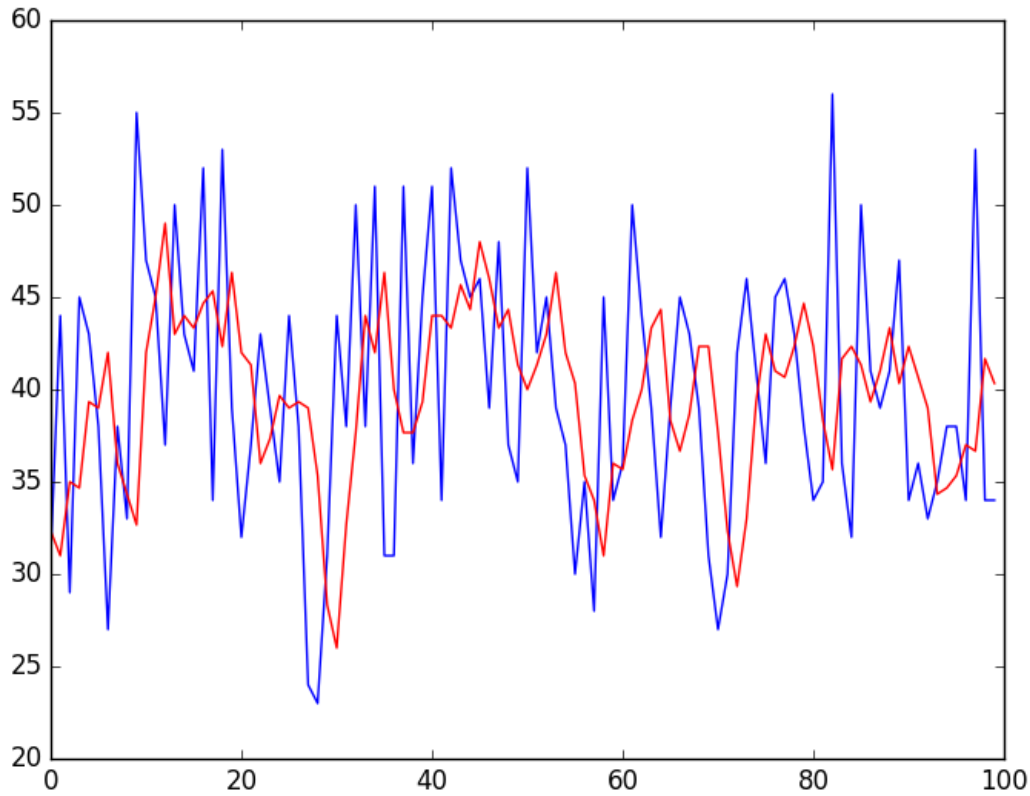

Figure 9.4: Line plot of the first 100 observations from the Daily Female Births dataset (blue) with a moving average predictions (red).

9.7 Summary

In this tutorial, you discovered how to use moving average smoothing for time series forecasting with Python. Specifically, you learned:

- How moving average smoothing works and the expectations of time series data before using it.
- How to use moving average smoothing for data preparation in Python.
- How to use moving average smoothing for feature engineering in Python.
- How to use moving average smoothing to make predictions in Python.

9.7.1 Next

This concludes Part II. Next, in Part III you will discover how to investigate the temporal structures in time series data, starting with the white noise model for time series.

Part III

Temporal Structure

Chapter 10

A Gentle Introduction to White Noise

White noise is an important concept in time series forecasting. If a time series is white noise, it is a sequence of random numbers and cannot be predicted. If the series of forecast errors are not white noise, it suggests improvements could be made to the predictive model. In this tutorial, you will discover white noise time series with Python. After completing this tutorial, you will know:

- The definition of a white noise time series and why it matters.
- How to check if your time series is white noise.
- Statistics and diagnostic plots to identify white noise in Python.

Let's get started.

10.1 What is a White Noise?

A time series may be white noise. A time series is white noise if the variables are independent and identically distributed with a mean of zero. This means that all variables have the same variance ($sigma^2$) and each value has a zero correlation with all other values in the series. If the variables in the series are drawn from a Gaussian distribution, the series is called Gaussian white noise.

10.2 Why Does it Matter?

White noise is an important concept in time series analysis and forecasting. It is important for two main reasons:

- **Predictability**: If your time series is white noise, then, by definition, it is random. You cannot reasonably model it and make predictions.
- **Model Diagnostics**: The series of errors from a time series forecast model should ideally be white noise.

Model Diagnostics is an important area of time series forecasting. Time series data are expected to contain some white noise component on top of the signal generated by the underlying process. For example:

$$y(t) = signal(t) + noise(t) \tag{10.1}$$

Once predictions have been made by a time series forecast model, they can be collected and analyzed. The series of forecast errors should ideally be white noise. When forecast errors are white noise, it means that all of the signal information in the time series has been harnessed by the model in order to make predictions. All that is left is the random fluctuations that cannot be modeled. A sign that model predictions are not white noise is an indication that further improvements to the forecast model may be possible.

10.3 Is your Time Series White Noise?

Your time series is not white noise if any of the following conditions are true:

- Does your series have a non-zero mean?
- Does the variance change over time?
- Do values correlate with lag values?

Some tools that you can use to check if your time series is white noise are:

- **Create a line plot**. Check for gross features like a changing mean, variance, or obvious relationship between lagged variables.
- **Calculate summary statistics**. Check the mean and variance of the whole series against the mean and variance of meaningful contiguous blocks of values in the series (e.g. days, months, or years).
- **Create an autocorrelation plot**. Check for gross correlation between lagged variables.

10.4 Example of White Noise Time Series

In this section, we will create a Gaussian white noise series in Python and perform some checks. It is helpful to create and review a white noise time series in practice. It will provide the frame of reference and example plots and statistical tests to use and compare on your own time series projects to check if they are white noise. Firstly, we can create a list of 1,000 random Gaussian variables using the `gauss()` function from the random module[1]. We will draw variables from a Gaussian distribution with a mean (`mu`) of 0.0 and a standard deviation (`sigma`) of 1.0. Once created, we can wrap the list in a Pandas `Series` for convenience.

[1] https://docs.python.org/3/library/random.html

```
from random import gauss
from random import seed
from pandas import Series
from pandas.plotting import autocorrelation_plot
# seed random number generator
seed(1)
# create white noise series
series = [gauss(0.0, 1.0) for i in range(1000)]
series = Series(series)
```

Listing 10.1: Example of creating a white noise series.

Next, we can calculate and print some summary statistics, including the mean and standard deviation of the series.

```
# summary stats
print(series.describe())
```

Listing 10.2: Calculate summary statistics of a white noise series.

Given that we defined the mean and standard deviation when drawing the random numbers, there should be no surprises.

```
count   1000.000000
mean      -0.013222
std        1.003685
min       -2.961214
25%       -0.684192
50%       -0.010934
75%        0.703915
max        2.737260
```

Listing 10.3: Example output of calculating summary statistics of a white noise series.

We can see that the mean is nearly 0.0 and the standard deviation is nearly 1.0. Some variance is expected given the small size of the sample. If we had more data, it might be more interesting to split the series in half and calculate and compare the summary statistics for each half. We would expect to see a similar mean and standard deviation for each sub-series. Now we can create some plots, starting with a line plot of the series.

```
# line plot
series.plot()
pyplot.show()
```

Listing 10.4: Create a line plot of the white noise series.

We can see that it does appear that the series is random.

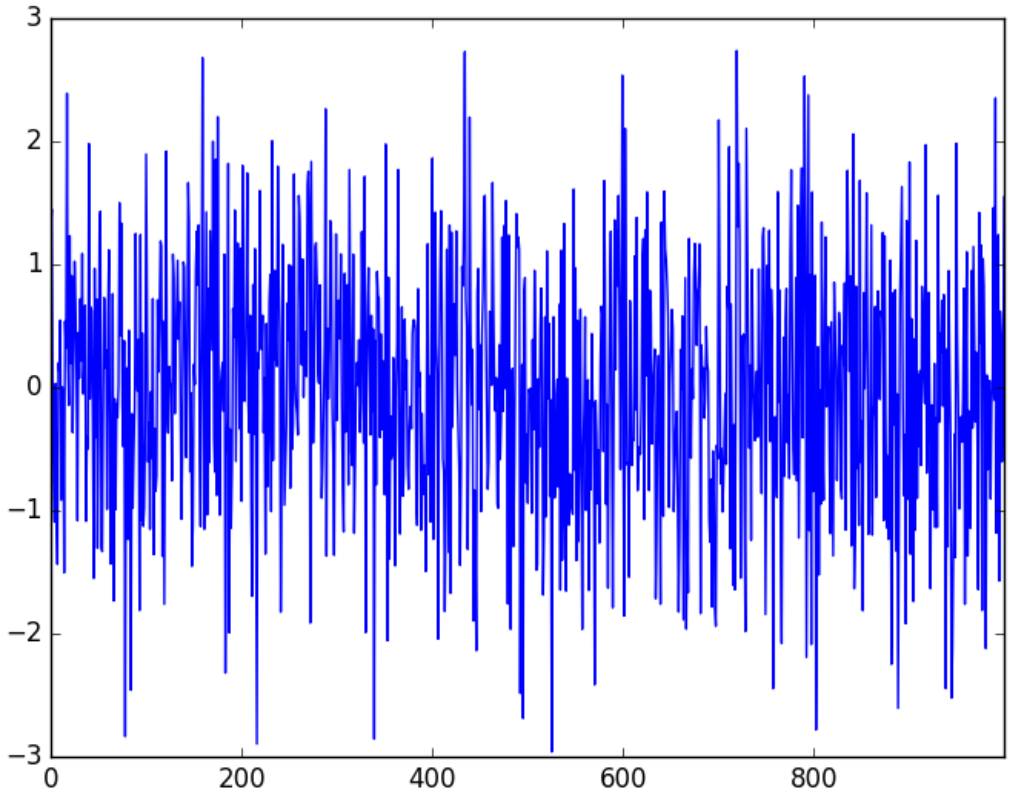

Figure 10.1: Line plot of white noise series.

We can also create a histogram and confirm the distribution is Gaussian.

```
# histogram plot
series.hist()
pyplot.show()
```

Listing 10.5: Create a histogram plot of the white noise series.

Indeed, the histogram shows the tell-tale bell-curve shape.

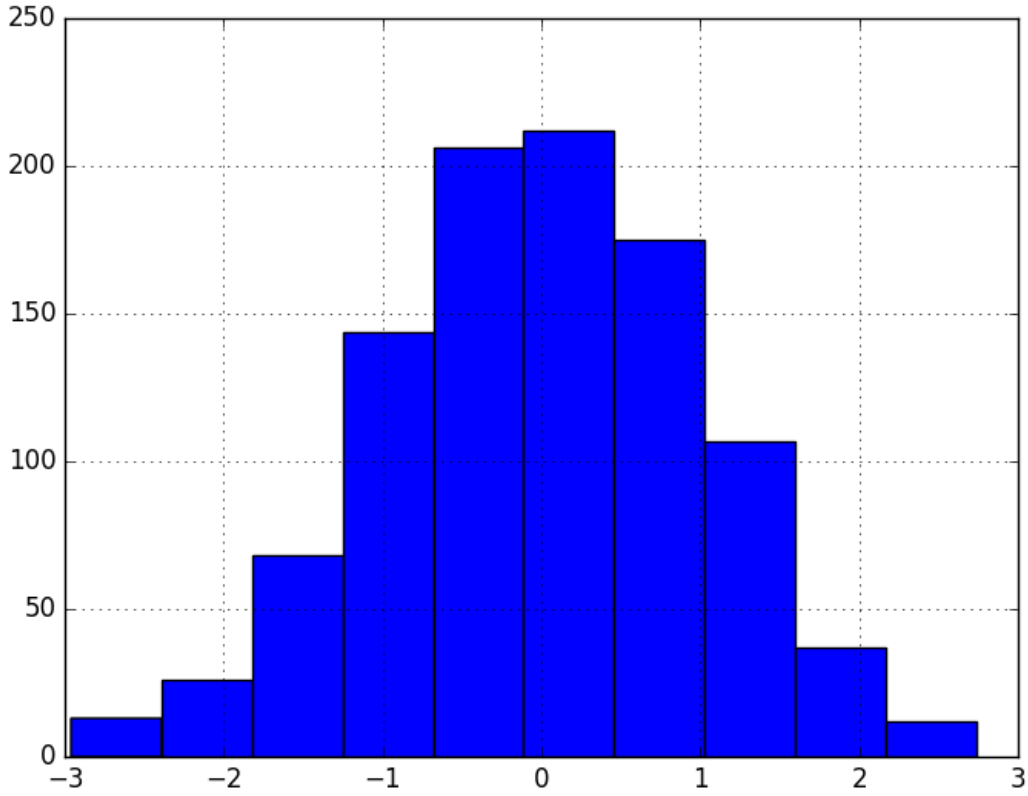

Figure 10.2: Histogram plot of white noise series.

Finally, we can create a correlogram and check for any autocorrelation with lag variables.

```
# autocorrelation
autocorrelation_plot(series)
pyplot.show()
```

Listing 10.6: Correlogram plot of the white noise series.

The correlogram does not show any obvious autocorrelation pattern. There are some spikes above the 95% and 99% confidence level, but these are a statistical fluke.

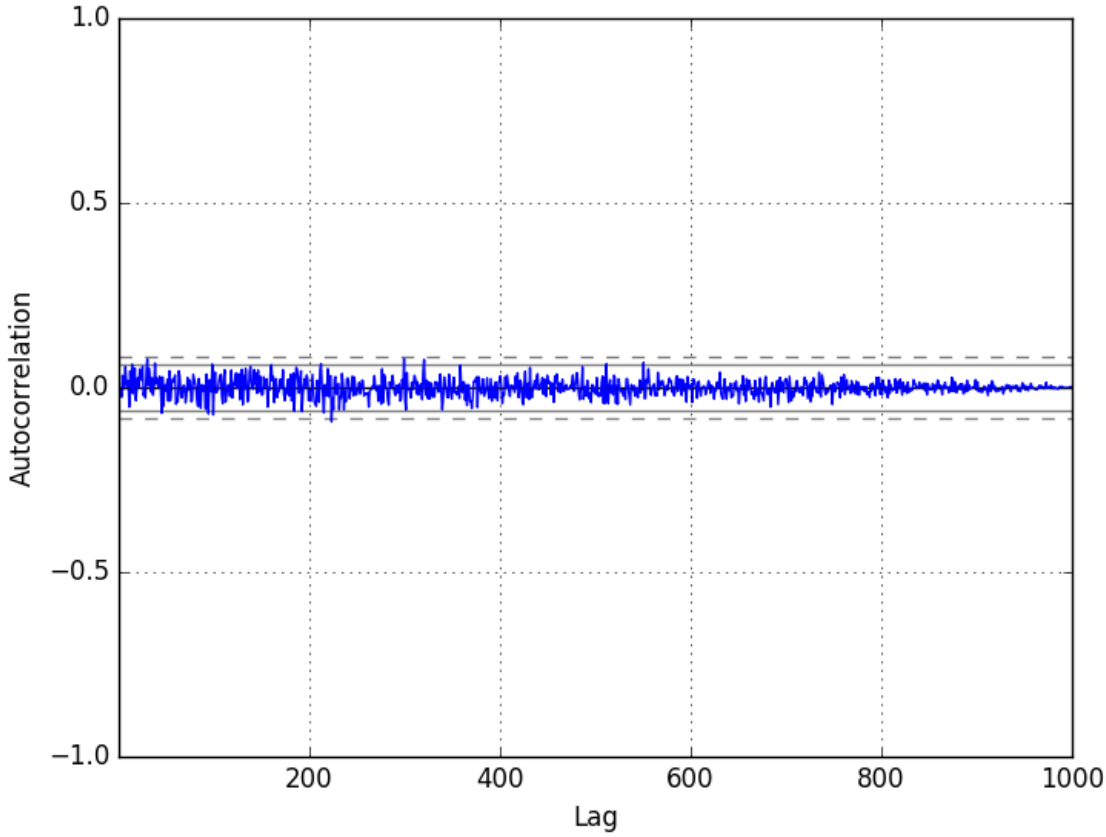

Figure 10.3: Correlogram plot of white noise series.

For completeness, the complete code listing is provided below.

```
# calculate and plot a white noise series
from random import gauss
from random import seed
from pandas import Series
from pandas.plotting import autocorrelation_plot
from matplotlib import pyplot
# seed random number generator
seed(1)
# create white noise series
series = [gauss(0.0, 1.0) for i in range(1000)]
series = Series(series)
# summary stats
print(series.describe())
# line plot
series.plot()
pyplot.show()
# histogram plot
series.hist()
pyplot.show()
# autocorrelation
autocorrelation_plot(series)
pyplot.show()
```

Listing 10.7: Example white noise series and diagnostics.

10.5 Summary

In this tutorial, you discovered white noise time series in Python. Specifically, you learned:

- White noise time series is defined by a zero mean, constant variance, and zero correlation.
- If your time series is white noise, it cannot be predicted, and if your forecast residuals are not white noise, you may be able to improve your model.
- The statistics and diagnostic plots you can use on your time series to check if it is white noise.

10.5.1 Next

In the next lesson you will discover about the predictability of time series problems and the random walk model.

Chapter 11

A Gentle Introduction to the Random Walk

How do you know your time series problem is predictable? This is a difficult question with time series forecasting. There is a tool called a random walk that can help you understand the predictability of your time series forecast problem. In this tutorial, you will discover the random walk and its properties in Python. After completing this tutorial, you will know:

- What the random walk is and how to create one from scratch in Python.
- How to analyze the properties of a random walk and recognize when a time series is and is not a random walk.
- How to make predictions for a random walk.

Let's get started.

11.1 Random Series

The Python standard library contains the `random` module[1] that provides access to a suite of functions for generating random numbers. The `randrange()` function[2] can be used to generate a random integer between 0 and an upper limit. We can use the `randrange()` function to generate a list of 1,000 random integers between 0 and 10. The example is listed below.

```
# create and plot a random series
from random import seed
from random import randrange
from matplotlib import pyplot
seed(1)
series = [randrange(10) for i in range(1000)]
pyplot.plot(series)
pyplot.show()
```

Listing 11.1: Example of a series of random numbers.

[1] https://docs.python.org/2.7/library/random.html
[2] https://docs.python.org/2.7/library/random.html#random.randrange

Running the example plots the sequence of random numbers. It's a real mess. It looks nothing like a time series.

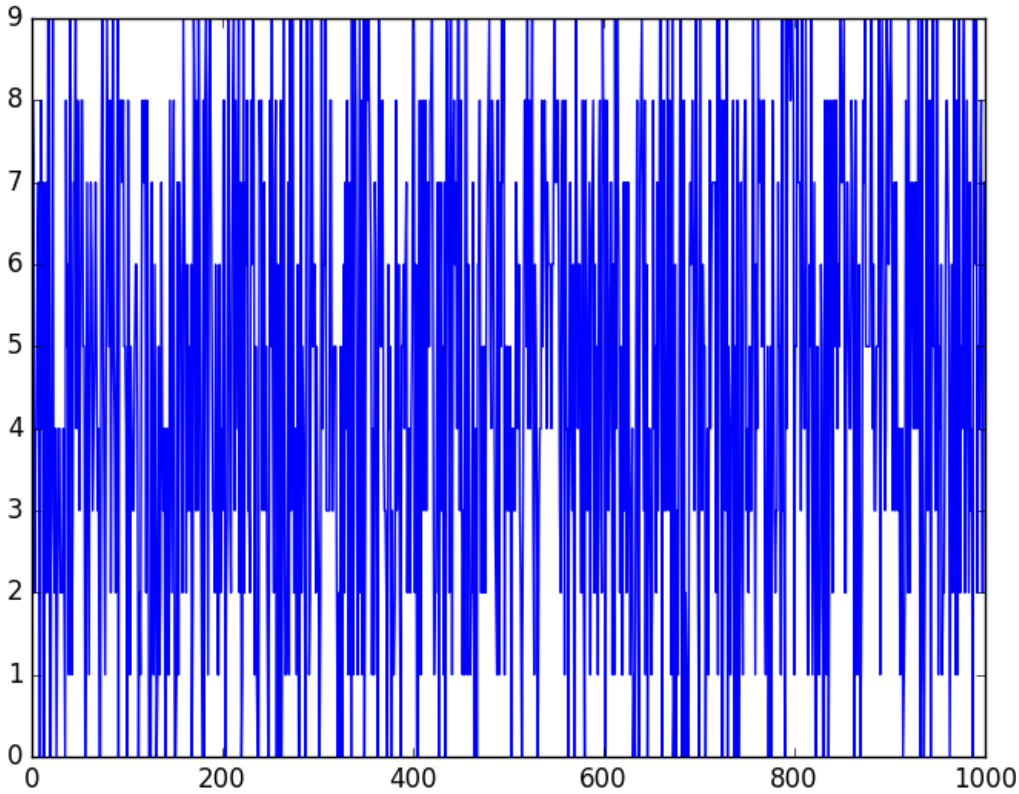

Figure 11.1: Plot of a Random Series.

This is not a random walk. It is just a sequence of random numbers also called white noise (see Chapter 10). A common mistake that beginners make is to think that a random walk is a list of random numbers, and this is not the case at all.

11.2 Random Walk

A random walk is different from a list of random numbers because the next value in the sequence is a modification of the previous value in the sequence. The process used to generate the series forces dependence from one-time step to the next. This dependence provides some consistency from step-to-step rather than the large jumps that a series of independent, random numbers provides. It is this dependency that gives the process its name as a *random walk* or a *drunkard's walk*. A simple model of a random walk is as follows:

1. Start with a random number of either -1 or 1.

2. Randomly select a -1 or 1 and add it to the observation from the previous time step.

3. Repeat step 2 for as long as you like.

More succinctly, we can describe this process as:

$$y(t) = B0 + B1 \times X(t-1) + e(t) \tag{11.1}$$

Where `y(t)` is the next value in the series. `B0` is a coefficient that if set to a value other than zero adds a constant drift to the random walk. `B1` is a coefficient to weight the previous time step and is set to 1.0. `X(t-1)` is the observation at the previous time step. `e(t)` is the white noise or random fluctuation at that time. We can implement this in Python by looping over this process and building up a list of 1,000 time steps for the random walk. The complete example is listed below.

```
# create and plot a random walk
from random import seed
from random import random
from matplotlib import pyplot
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
  movement = -1 if random() < 0.5 else 1
  value = random_walk[i-1] + movement
  random_walk.append(value)
pyplot.plot(random_walk)
pyplot.show()
```

Listing 11.2: Example of a random walk.

Running the example creates a line plot of the random walk. We can see that it looks very different from our above sequence of random numbers. In fact, the shape and movement looks like a realistic time series for the price of a security on the stock market.

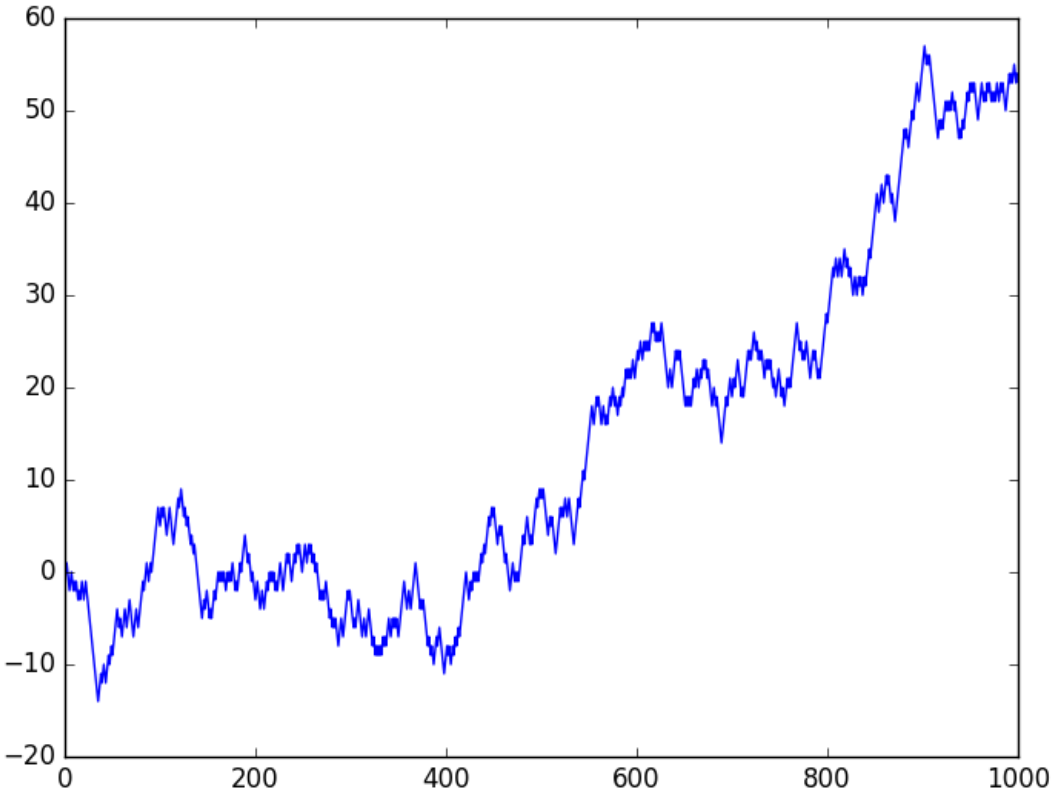

Figure 11.2: Plot of a Random Walk.

In the next sections, we will take a closer look at the properties of a random walk. This is helpful because it will give you context to help identify whether a time series you are analyzing in the future might be a random walk. Let's start by looking at the autocorrelation structure.

11.3 Random Walk and Autocorrelation

We can calculate the correlation between each observation and the observations at previous time steps. A plot of these correlations is called an autocorrelation plot or a correlogram[3]. Given the way that the random walk is constructed, we would expect a strong autocorrelation with the previous observation and a linear fall off from there with previous lag values. We can use the `autocorrelation_plot()` function in Pandas to plot the correlogram for the random walk.

The complete example is listed below. Note that in each example where we generate the random walk we use the same seed for the random number generator to ensure that we get the same sequence of random numbers, and in turn the same random walk.

```
# plot the autocorrelation of a random walk
from random import seed
from random import random
from matplotlib import pyplot
```

[3] https://en.wikipedia.org/wiki/Correlogram

```
from pandas.plotting import autocorrelation_plot
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
  movement = -1 if random() < 0.5 else 1
  value = random_walk[i-1] + movement
  random_walk.append(value)
autocorrelation_plot(random_walk)
pyplot.show()
```

Listing 11.3: Example of an autocorrelation plot of a random walk.

Running the example, we generally see the expected trend, in this case across the first few hundred lag observations.

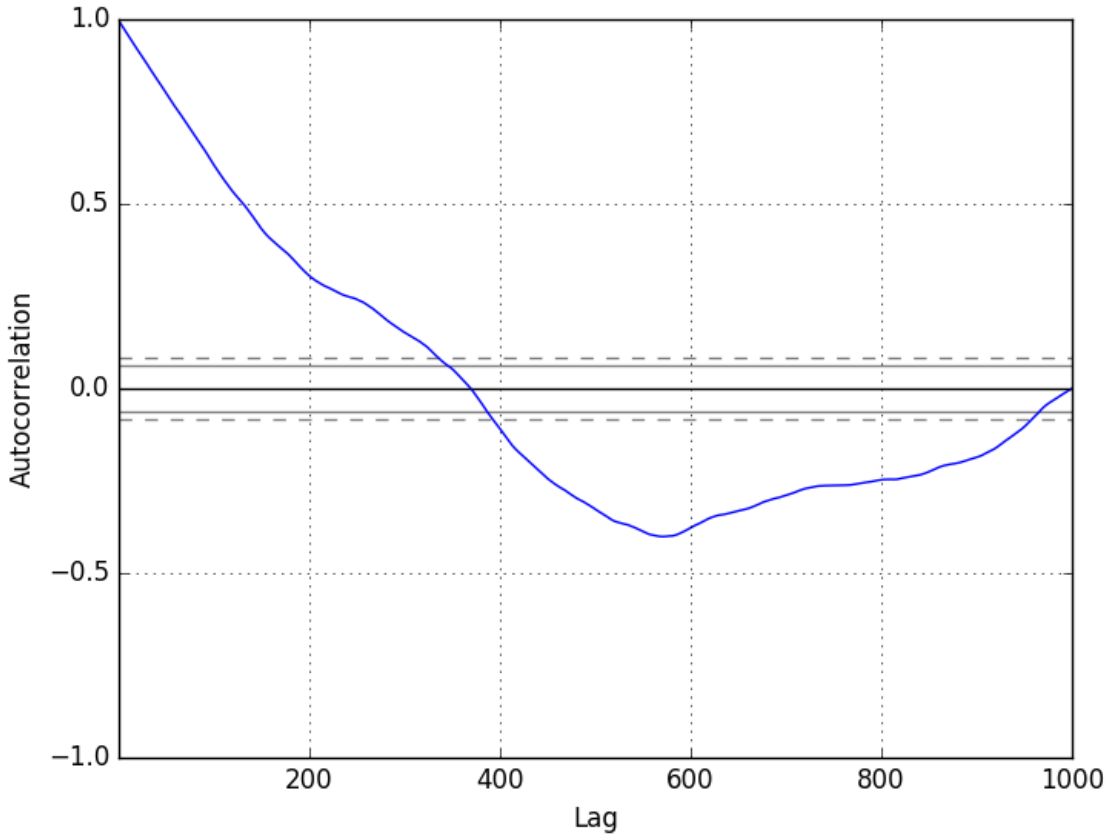

Figure 11.3: Plot of a Random Walk Correlogram.

11.4 Random Walk and Stationarity

A stationary time series is one where the values are not a function of time (stationarity is covered in more detail in Chapter 15). Given the way that the random walk is constructed and the results of reviewing the autocorrelation, we know that the observations in a random walk are dependent on time. The current observation is a random step from the previous observation.

Therefore we can expect a random walk to be non-stationary. In fact, all random walk processes are non-stationary. Note that not all non-stationary time series are random walks. Additionally, a non-stationary time series does not have a consistent mean and/or variance over time. A review of the random walk line plot might suggest this to be the case. We can confirm this using a statistical significance test, specifically the Augmented Dickey-Fuller test[4].

We can perform this test using the `adfuller()` function[5] in the Statsmodels library. The complete example is listed below.

```
# calculate the stationarity of a random walk
from random import seed
from random import random
from statsmodels.tsa.stattools import adfuller
# generate random walk
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
  movement = -1 if random() < 0.5 else 1
  value = random_walk[i-1] + movement
  random_walk.append(value)
# statistical test
result = adfuller(random_walk)
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 11.4: Example of calculating stationarity of a random walk.

The null hypothesis of the test is that the time series is non-stationary. Running the example, we can see that the test statistic value was 0.341605. This is larger than all of the critical values at the 1%, 5%, and 10% confidence levels. Therefore, we can say that the time series does appear to be non-stationary with a low likelihood of the result being a statistical fluke.

```
ADF Statistic: 0.341605
p-value: 0.979175
Critical Values:
  5%: -2.864
  1%: -3.437
  10%: -2.568
```

Listing 11.5: Output of calculating stationarity of a random walk.

We can make the random walk stationary by taking the first difference. That is replacing each observation as the difference between it and the previous value. Given the way that this random walk was constructed, we would expect this to result in a time series of -1 and 1 values. This is exactly what we see. The complete example is listed below.

```
# calculate and plot a differenced random walk
from random import seed
from random import random
```

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

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

```
from matplotlib import pyplot
# create random walk
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
  movement = -1 if random() < 0.5 else 1
  value = random_walk[i-1] + movement
  random_walk.append(value)
# take difference
diff = list()
for i in range(1, len(random_walk)):
  value = random_walk[i] - random_walk[i - 1]
  diff.append(value)
# line plot
pyplot.plot(diff)
pyplot.show()
```

Listing 11.6: Example of calculating the differenced series of a random walk.

Running the example produces a line plot showing 1,000 movements of -1 and 1, a real mess.

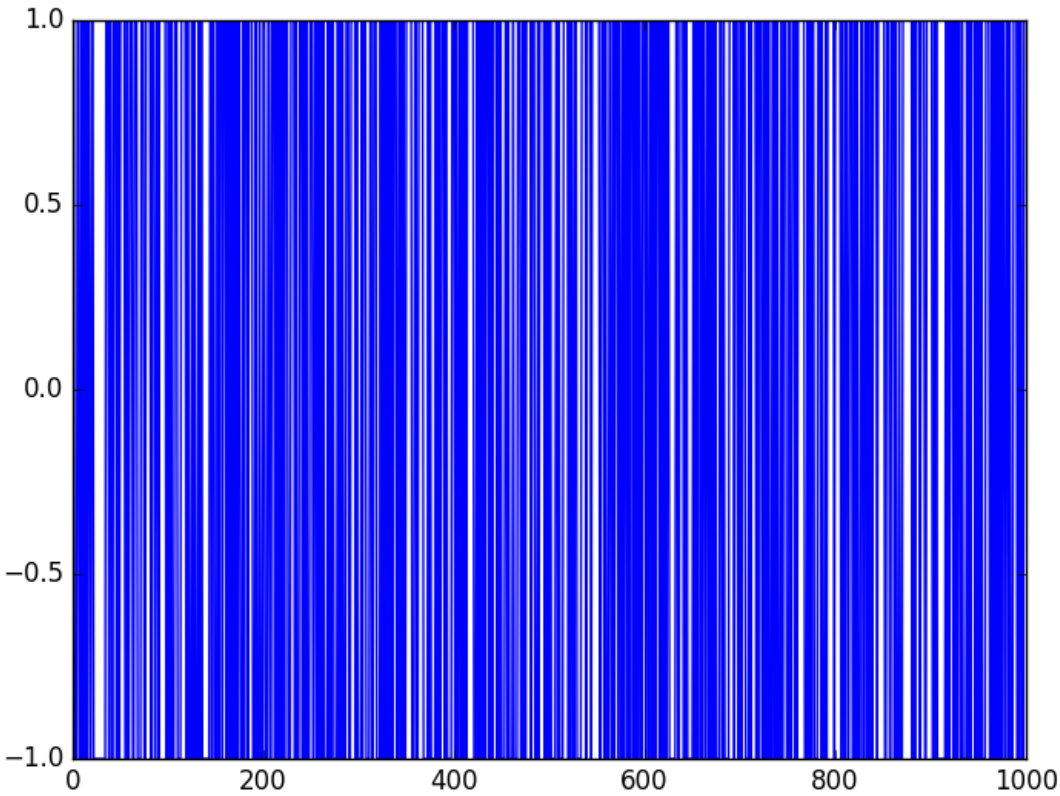

Figure 11.4: Plot of a Differenced Random Walk.

This difference graph also makes it clear that really we have no information to work with here other than a series of random moves. There is no structure to learn. Now that the time

series is stationary, we can recalculate the correlogram of the differenced series. The complete example is listed below.

```
# plot the autocorrelation of a differenced random walk
from random import seed
from random import random
from matplotlib import pyplot
from pandas.plotting import autocorrelation_plot
# create random walk
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
  movement = -1 if random() < 0.5 else 1
  value = random_walk[i-1] + movement
  random_walk.append(value)
# take difference
diff = list()
for i in range(1, len(random_walk)):
  value = random_walk[i] - random_walk[i - 1]
  diff.append(value)
# line plot
autocorrelation_plot(diff)
pyplot.show()
```

Listing 11.7: Example of calculating the correlogram of the differenced series of a random walk.

Running the example, we can see no significant relationship between the lagged observations, as we would expect from the way the random walk was generated. All correlations are small, close to zero and below the 95% and 99% confidence levels (beyond a few statistical flukes).

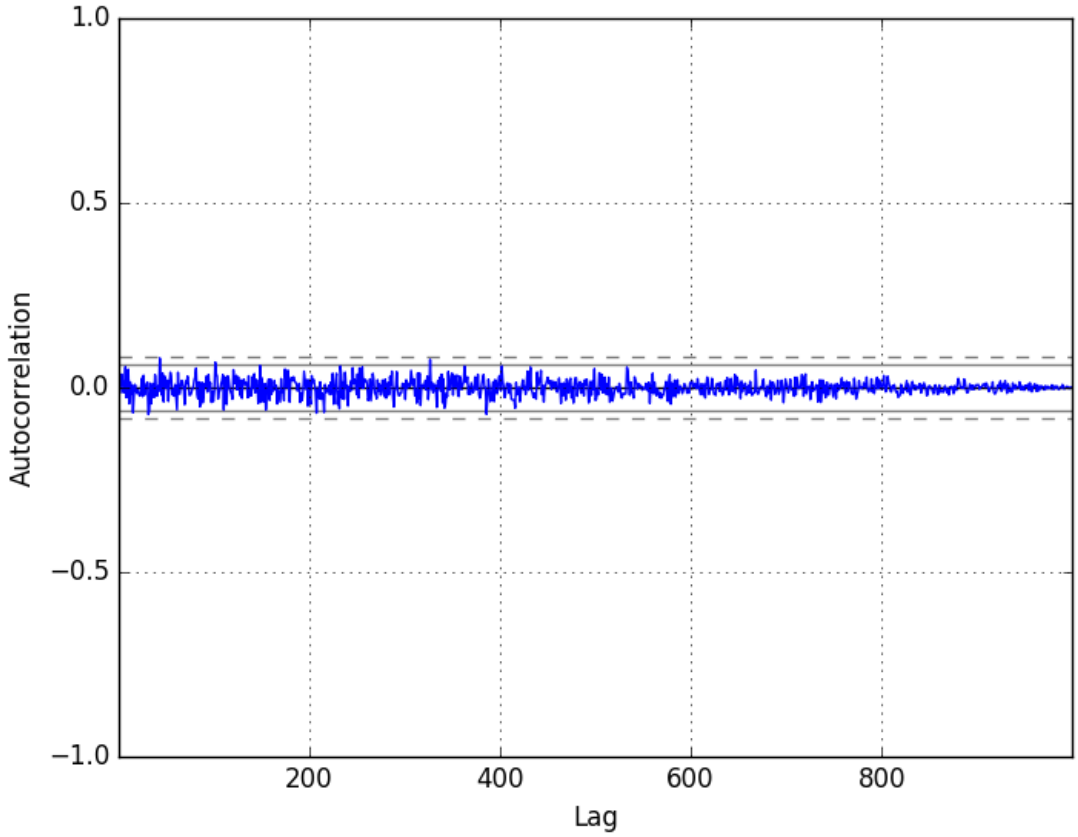

Figure 11.5: Plot of the Correlogram of the Differenced Random Walk.

11.5 Predicting a Random Walk

A random walk is unpredictable; it cannot reasonably be predicted. Given the way that the random walk is constructed, we can expect that the best prediction we could make would be to use the observation at the previous time step as what will happen in the next time step. Simply because we know that the next time step will be a function of the prior time step. This is often called the naive forecast, or a persistence model (covered in Chapter 18).

We can implement this in Python by first splitting the dataset into train and test sets, then using the persistence model to predict the outcome using a rolling forecast method. Once all predictions are collected for the test set, the root mean squared error (RMSE) is calculated.

```
# persistence forecasts for a random walk
from random import seed
from random import random
from sklearn.metrics import mean_squared_error
from math import sqrt
# generate the random walk
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
```

```
  movement = -1 if random() < 0.5 else 1
  value = random_walk[i-1] + movement
  random_walk.append(value)
# prepare dataset
train_size = int(len(random_walk) * 0.66)
train, test = random_walk[0:train_size], random_walk[train_size:]
# persistence
predictions = list()
history = train[-1]
for i in range(len(test)):
  yhat = history
  predictions.append(yhat)
  history = test[i]
rmse = sqrt(mean_squared_error(test, predictions))
print('Persistence RMSE: %.3f' % rmse)
```

Listing 11.8: Example of predicting a random walk with persistence.

Running the example estimates the RMSE of the model as 1. This too is expected, given that we know that the variation from one time step to the next is always going to be 1, either in the positive or negative direction.

```
Persistence RMSE: 1.000
```

Listing 11.9: Output of predicting a random walk with persistence.

Another error that beginners to the random walk make is to assume that if the range of error (variance) is known, then we can make predictions using a random walk generation type process. That is, if we know the error is either -1 or 1, then why not make predictions by adding a randomly selected -1 or 1 to the previous value. We can demonstrate this random prediction method in Python below.

```
# random predictions for a random walk
from random import seed
from random import random
from sklearn.metrics import mean_squared_error
from math import sqrt
# generate the random walk
seed(1)
random_walk = list()
random_walk.append(-1 if random() < 0.5 else 1)
for i in range(1, 1000):
  movement = -1 if random() < 0.5 else 1
  value = random_walk[i-1] + movement
  random_walk.append(value)
# prepare dataset
train_size = int(len(random_walk) * 0.66)
train, test = random_walk[0:train_size], random_walk[train_size:]
# random prediction
predictions = list()
history = train[-1]
for i in range(len(test)):
  yhat = history + (-1 if random() < 0.5 else 1)
  predictions.append(yhat)
  history = test[i]
rmse = sqrt(mean_squared_error(test, predictions))
```

```
print('Random RMSE: %.3f' % rmse)
```

Listing 11.10: Example of predicting a random walk with random chance.

Running the example, we can see that indeed the algorithm results in a worse performance than the persistence method, with a RMSE of 1.328.

```
Random RMSE: 1.328
```

Listing 11.11: Output of predicting a random walk with random chance.

Persistence, or the naive forecast, is the best prediction we can make for a random walk time series.

11.6 Is Your Time Series a Random Walk?

Your time series may be a random walk. Some ways to check if your time series is a random walk are as follows:

- The time series shows a strong temporal dependence that decays linearly or in a similar pattern.
- The time series is non-stationary and making it stationary shows no obviously learnable structure in the data.
- The persistence model provides the best source of reliable predictions.

This last point is key for time series forecasting. Baseline forecasts with the persistence model quickly flesh out whether you can do significantly better. If you can't, you're probably working with a random walk. Many time series are random walks, particularly those of security prices over time. The random walk hypothesis is a theory that stock market prices are a random walk and cannot be predicted[6].

> A random walk is one in which future steps or directions cannot be predicted on the basis of past history. When the term is applied to the stock market, it means that short-run changes in stock prices are unpredictable.
>
> — Page 26, *A Random Walk Down Wall Street*.

The human mind sees patterns everywhere and we must be vigilant that we are not fooling ourselves and wasting time by developing elaborate models for random walk processes.

11.7 Summary

In this tutorial, you discovered how to explore the random walk with Python. Specifically, you learned:

- How to create a random walk process in Python.
- How to explore the autocorrelation and non-stationary structure of a random walk.
- How to make predictions for a random walk time series.

[6] https://en.wikipedia.org/wiki/Random_walk_hypothesis

11.7.1 Next

In the next lesson you will discover how to decompose a time series into its constituent parts of level, trend and seasonality.

Chapter 12

Decompose Time Series Data

Time series decomposition involves thinking of a series as a combination of level, trend, seasonality, and noise components. Decomposition provides a useful abstract model for thinking about time series generally and for better understanding problems during time series analysis and forecasting. In this tutorial, you will discover time series decomposition and how to automatically split a time series into its components with Python. After completing this tutorial, you will know:

- The time series decomposition method of analysis and how it can help with forecasting.
- How to automatically decompose time series data in Python.
- How to decompose additive and multiplicative time series problems and plot the results.

Let's get started.

12.1 Time Series Components

A useful abstraction for selecting forecasting methods is to break a time series down into systematic and unsystematic components.

- **Systematic**: Components of the time series that have consistency or recurrence and can be described and modeled.
- **Non-Systematic**: Components of the time series that cannot be directly modeled.

A given time series is thought to consist of three systematic components including level, trend, seasonality, and one non-systematic component called noise. These components are defined as follows:

- **Level**: The average value in the series.
- **Trend**: The increasing or decreasing value in the series.
- **Seasonality**: The repeating short-term cycle in the series.
- **Noise**: The random variation in the series.

12.2 Combining Time Series Components

A series is thought to be an aggregate or combination of these four components. All series have a level and noise. The trend and seasonality components are optional. It is helpful to think of the components as combining either additively or multiplicatively.

12.2.1 Additive Model

An additive model suggests that the components are added together as follows:

$$y(t) = Level + Trend + Seasonality + Noise \tag{12.1}$$

An additive model is linear where changes over time are consistently made by the same amount. A linear trend is a straight line. A linear seasonality has the same frequency (width of cycles) and amplitude (height of cycles).

12.2.2 Multiplicative Model

A multiplicative model suggests that the components are multiplied together as follows:

$$y(t) = Level \times Trend \times Seasonality \times Noise \tag{12.2}$$

A multiplicative model is nonlinear, such as quadratic or exponential. Changes increase or decrease over time. A nonlinear trend is a curved line. A nonlinear seasonality has an increasing or decreasing frequency and/or amplitude over time.

12.3 Decomposition as a Tool

This is a useful abstraction. Decomposition is primarily used for time series analysis, and as an analysis tool it can be used to inform forecasting models on your problem. It provides a structured way of thinking about a time series forecasting problem, both generally in terms of modeling complexity and specifically in terms of how to best capture each of these components in a given model.

Each of these components are something you may need to think about and address during data preparation, model selection, and model tuning. You may address it explicitly in terms of modeling the trend and subtracting it from your data, or implicitly by providing enough history for an algorithm to model a trend if it may exist. You may or may not be able to cleanly or perfectly break down your specific time series as an additive or multiplicative model.

Real-world problems are messy and noisy. There may be additive and multiplicative components. There may be an increasing trend followed by a decreasing trend. There may be non-repeating cycles mixed in with the repeating seasonality components. Nevertheless, these abstract models provide a simple framework that you can use to analyze your data and explore ways to think about and forecast your problem.

12.4 Automatic Time Series Decomposition

There are methods to automatically decompose a time series. The Statsmodels library provides an implementation of the naive, or classical, decomposition method in a function called `seasonal_decompose()`[1]. It requires that you specify whether the model is additive or multiplicative.

Both will produce a result and you must be careful to be critical when interpreting the result. A review of a plot of the time series and some summary statistics can often be a good start to get an idea of whether your time series problem looks additive or multiplicative. The `seasonal_decompose()` function returns a result object. The result object contains arrays to access four pieces of data from the decomposition.

For example, the snippet below shows how to decompose a series into trend, seasonal, and residual components assuming an additive model. The result object provides access to the trend and seasonal series as arrays. It also provides access to the residuals, which are the time series after the trend, and seasonal components are removed. Finally, the original or observed data is also stored.

```
from statsmodels.tsa.seasonal import seasonal_decompose
series = ...
result = seasonal_decompose(series, model='additive')
print(result.trend)
print(result.seasonal)
print(result.resid)
print(result.observed)
```

Listing 12.1: Example of decomposing a time series.

These four time series can be plotted directly from the result object by calling the `plot()` function. For example:

```
from statsmodels.tsa.seasonal import seasonal_decompose
from matplotlib import pyplot
series = ...
result = seasonal_decompose(series, model='additive')
result.plot()
pyplot.show()
```

Listing 12.2: Example of plotting a decomposed time series.

Let's look at some examples.

12.4.1 Additive Decomposition

We can create a time series comprised of a linearly increasing trend from 1 to 99 and some random noise and decompose it as an additive model. Because the time series was contrived and was provided as an array of numbers, we must specify the frequency of the observations (the `freq=1` argument). If a Pandas `Series` object is provided, this argument is not required.

```
# additive decompose a contrived additive time series
from random import randrange
from matplotlib import pyplot
```

[1] http://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.seasonal_decompose.html

```
from statsmodels.tsa.seasonal import seasonal_decompose
series = [i+randrange(10) for i in range(1,100)]
result = seasonal_decompose(series, model='additive', freq=1)
result.plot()
pyplot.show()
```

Listing 12.3: Example of decomposing a contrived additive time series.

Running the example creates the series, performs the decomposition, and plots the 4 resulting series. We can see that the entire series was taken as the trend component and that there was no seasonality.

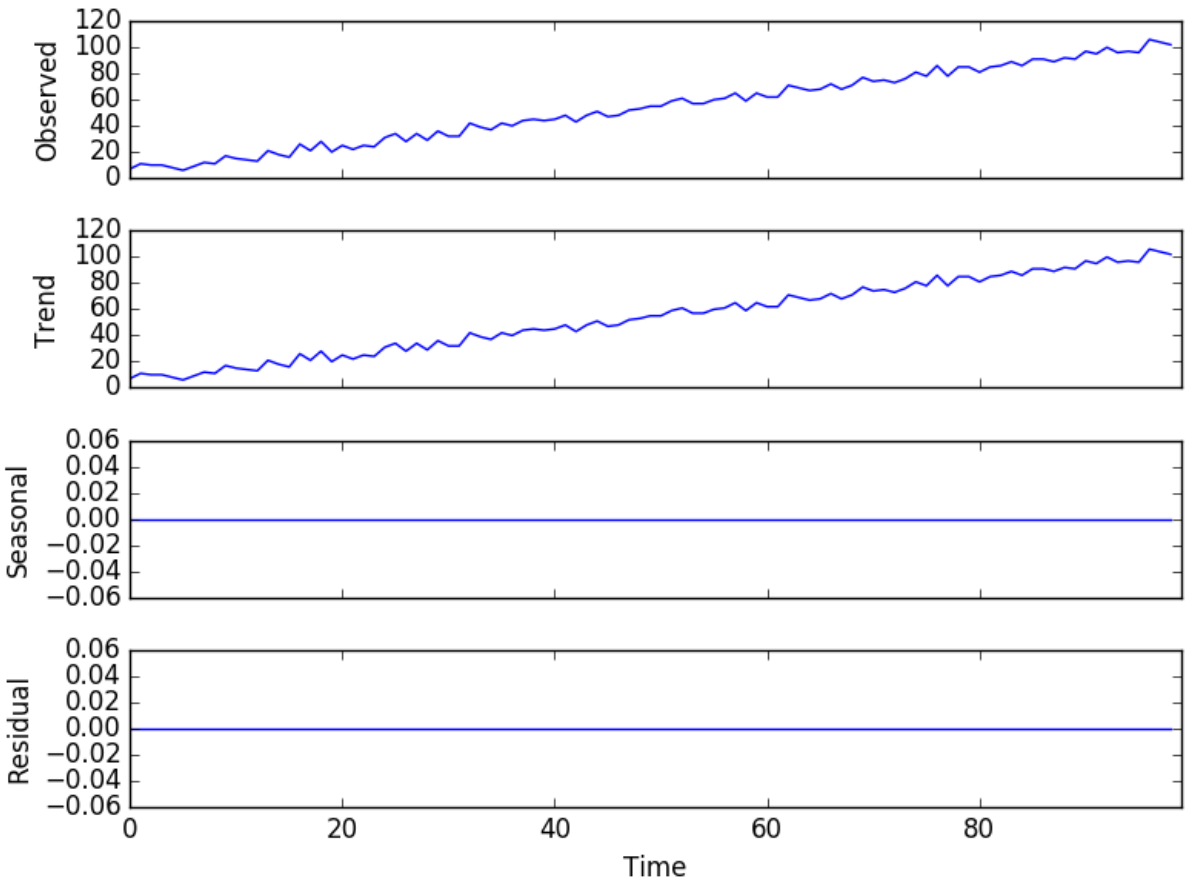

Figure 12.1: Line plots of a decomposed additive time series.

We can also see that the residual plot shows zero. This is a good example where the naive, or classical, decomposition was not able to separate the noise that we added from the linear trend. The naive decomposition method is a simple one, and there are more advanced decompositions available, like Seasonal and Trend decomposition using Loess or STL decomposition. Caution and healthy skepticism is needed when using automated decomposition methods.

12.4.2 Multiplicative Decomposition

We can contrive a quadratic time series as a square of the time step from 1 to 99, and then decompose it assuming a multiplicative model.

```
# multiplicative decompose a contrived multiplicative time series
from matplotlib import pyplot
from statsmodels.tsa.seasonal import seasonal_decompose
series = [i**2.0 for i in range(1,100)]
result = seasonal_decompose(series, model='multiplicative', freq=1)
result.plot()
pyplot.show()
```

Listing 12.4: Example of decomposing a contrived multiplicative time series.

Running the example, we can see that, as in the additive case, the trend is easily extracted and wholly characterizes the time series.

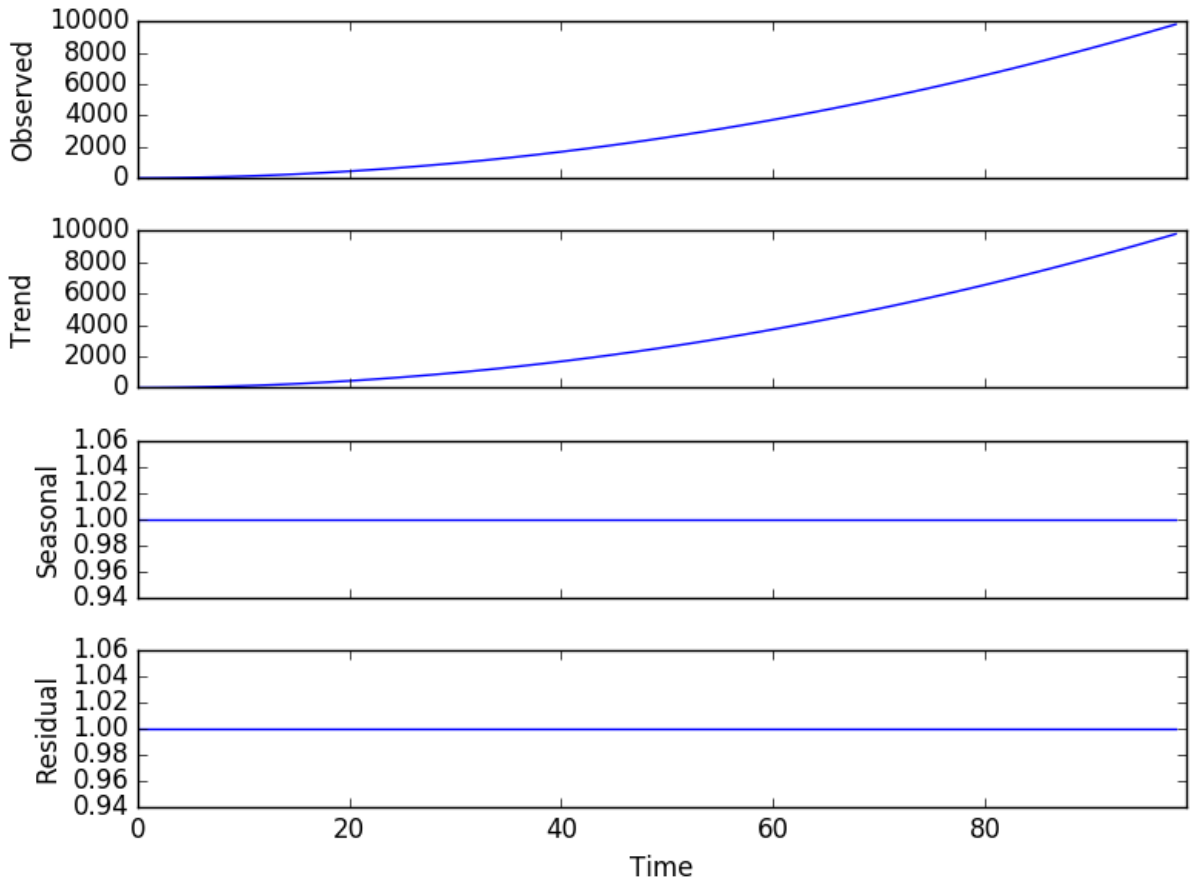

Figure 12.2: Line plots of a decomposed multiplicative time series.

Exponential changes can be made linear by data transforms. In this case, a quadratic trend can be made linear by taking the square root. An exponential growth in seasonality may be made linear by taking the natural logarithm.

Again, it is important to treat decomposition as a potentially useful analysis tool, but to consider exploring the many different ways it could be applied for your problem, such as on data after it has been transformed or on residual model errors. Let's look at a real world dataset.

12.4.3 Airline Passengers Dataset

In this lesson, we will use the Airline Passengers dataset as an example. This dataset describes the total number of airline passengers over time. You can learn more about the dataset in Appendix A.5. Place the dataset in your current working directory with the filename `airline-passengers.csv`. Reviewing a graph of the dataset in the Appendix, we will assume a multiplicative model. The example below decomposes the airline passengers dataset as a multiplicative model.

```
# multiplicative decompose time series
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.tsa.seasonal import seasonal_decompose
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
result = seasonal_decompose(series, model='multiplicative')
result.plot()
pyplot.show()
```

Listing 12.5: Multiplicative decomposition of the Airline Passengers dataset.

Running the example plots the observed, trend, seasonal, and residual time series. We can see that the trend and seasonality information extracted from the series does seem reasonable. The residuals are also interesting, showing periods of high variability in the early and later years of the series.

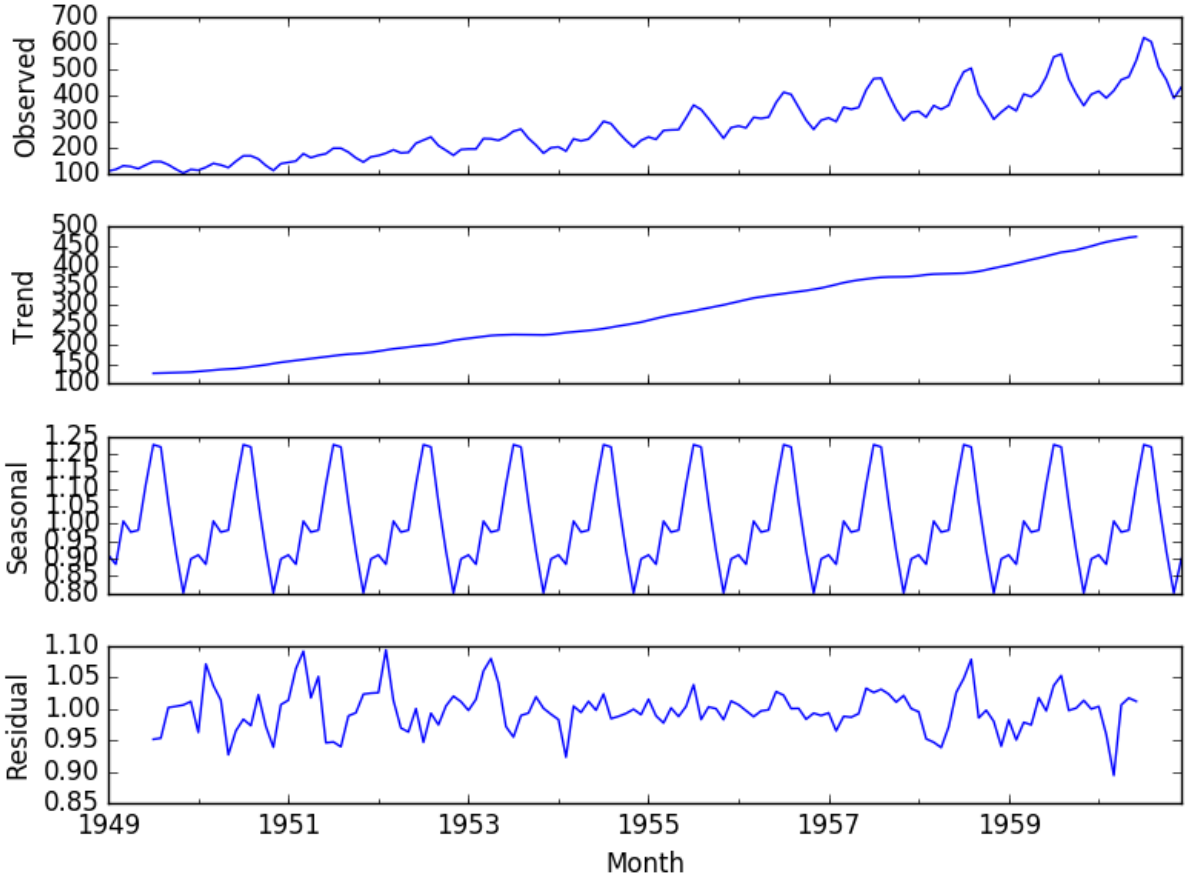

Figure 12.3: Line plots of a decomposed multiplicative time series.

12.5 Summary

In this tutorial, you discovered time series decomposition and how to decompose time series data with Python. Specifically, you learned:

- The structure of decomposing time series into level, trend, seasonality, and noise.
- How to automatically decompose a time series dataset with Python.
- How to decompose an additive or multiplicative model and plot the results.

12.5.1 Next

In the next lesson you will discover how to identify and model trends in time series data.

Chapter 13

Use and Remove Trends

Our time series dataset may contain a trend. A trend is a continued increase or decrease in the series over time. There can be benefit in identifying, modeling, and even removing trend information from your time series dataset. In this tutorial, you will discover how to model and remove trend information from time series data in Python. After completing this tutorial, you will know:

- The importance and types of trends that may exist in time series and how to identify them.
- How to use a simple differencing method to remove a trend.
- How to model a linear trend and remove it from a sales time series dataset.

Let's get started.

13.1 Trends in Time Series

A trend is a long-term increase or decrease in the level of the time series.

> In general, a systematic change in a time series that does not appear to be periodic is known as a trend.
>
> — Page 5, *Introductory Time Series with R.*

Identifying and understanding trend information can aid in improving model performance; below are a few reasons:

- **Faster Modeling**: Perhaps the knowledge of a trend or lack of a trend can suggest methods and make model selection and evaluation more efficient.
- **Simpler Problem**: Perhaps we can correct or remove the trend to simplify modeling and improve model performance.
- **More Data**: Perhaps we can use trend information, directly or as a summary, to provide additional information to the model and improve model performance.

13.1.1 Types of Trends

There are all kinds of trends. Two general classes that we may think about are:

- **Deterministic Trends**: These are trends that consistently increase or decrease.
- **Stochastic Trends**: These are trends that increase and decrease inconsistently.

In general, deterministic trends are easier to identify and remove, but the methods discussed in this tutorial can still be useful for stochastic trends. We can think about trends in terms of their scope of observations.

- **Global Trends**: These are trends that apply to the whole time series.
- **Local Trends**: These are trends that apply to parts or subsequences of a time series.

Generally, global trends are easier to identify and address.

13.1.2 Identifying a Trend

You can plot time series data to see if a trend is obvious or not. The difficulty is that in practice, identifying a trend in a time series can be a subjective process. As such, extracting or removing it from the time series can be just as subjective. Create line plots of your data and inspect the plots for obvious trends. Add linear and nonlinear trend lines to your plots and see if a trend is obvious.

13.1.3 Removing a Trend

A time series with a trend is called non-stationary. An identified trend can be modeled. Once modeled, it can be removed from the time series dataset. This is called detrending the time series. If a dataset does not have a trend or we successfully remove the trend, the dataset is said to be trend stationary.

13.1.4 Using Time Series Trends in Machine Learning

From a machine learning perspective, a trend in your data represents two opportunities:

- **Remove Information**: To remove systematic information that distorts the relationship between input and output variables.
- **Add Information**: To add systematic information to improve the relationship between input and output variables.

Specifically, a trend can be removed from your time series data (and data in the future) as a data preparation and cleaning exercise. This is common when using statistical methods for time series forecasting, but does not always improve results when using machine learning models. Alternately, a trend can be added, either directly or as a summary, as a new input variable to the supervised learning problem to predict the output variable.

One or both approaches may be relevant for your time series forecasting problem and may be worth investigating. Next, let's take a look at a dataset that has a trend

13.2 Shampoo Sales Dataset

In this lesson, we will use the Shampoo Sales dataset as an example. This dataset describes the monthly number of sales of shampoo over a 3 year period. You can learn more about the dataset in Appendix A.1. Place the dataset in your current working directory with the filename `shampoo-sales.csv`.

13.3 Detrend by Differencing

Perhaps the simplest method to detrend a time series is by differencing. Specifically, a new series is constructed where the value at the current time step is calculated as the difference between the original observation and the observation at the previous time step.

$$value(t) = observation(t) - observation(t-1) \tag{13.1}$$

This has the effect of removing a trend from a time series dataset. We can create a new difference dataset in Python by implementing this directly. A new list of observations can be created. Below is an example that creates the difference detrended version of the Shampoo Sales dataset.

```
# detrend a time series using differencing
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot

def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
X = series.values
diff = list()
for i in range(1, len(X)):
  value = X[i] - X[i - 1]
  diff.append(value)
pyplot.plot(diff)
pyplot.show()
```

Listing 13.1: Difference the Shampoo Sales dataset.

Running the example creates the new detrended dataset and then plots the time series. Because no difference value can be created for the first observation (there is nothing for it to be subtracted from), the new dataset contains one less record. We can see that indeed the trend does appear to have been removed (compared to original plot in the Appendix).

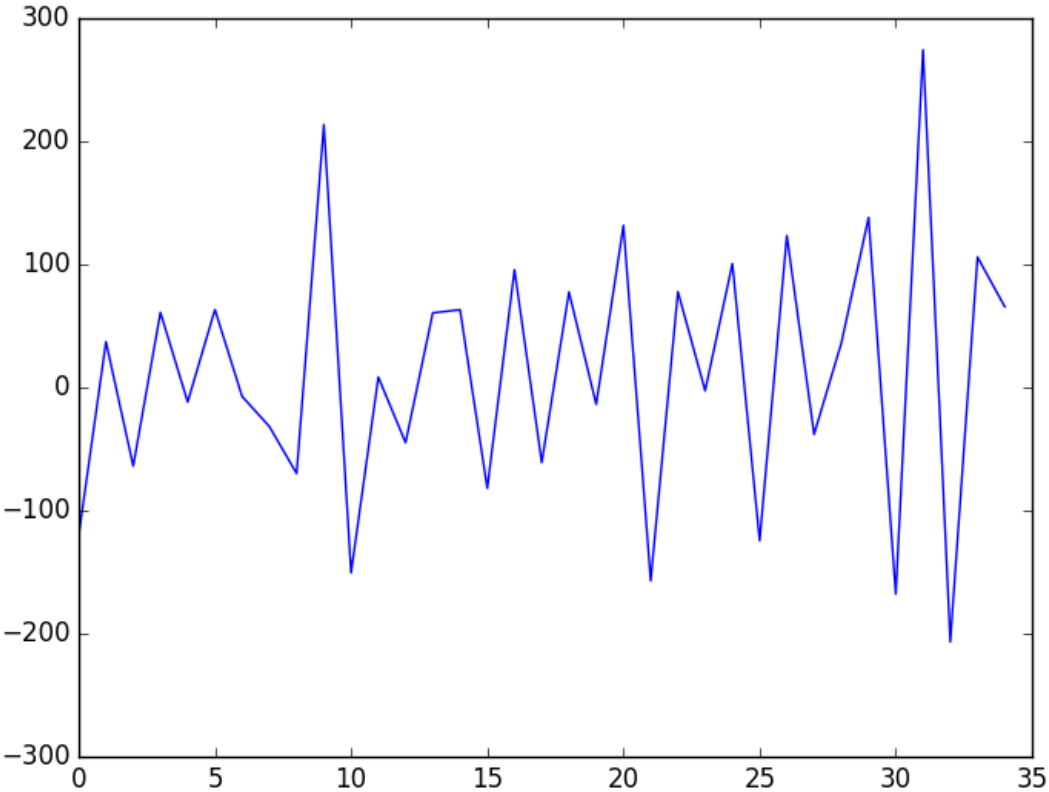

Figure 13.1: Line plots differenced Shampoo Sales dataset.

This approach works well for data with a linear trend. If the trend is quadratic (the change in the trend also increases or decreases), then a difference of the already-differenced dataset can be taken, a second level of differencing. This process can be further repeated if needed. Because differencing only requires the observation at the previous time step, it can easily be applied to unseen out-of-sample data to either pre-process or provide an additional input for supervised learning. Next, we will look at fitting a model to describe the trend.

13.4 Detrend by Model Fitting

A trend is often easily visualized as a line through the observations. Linear trends can be summarized by a linear model, and nonlinear trends may be best summarized using a polynomial or other curve-fitting method. Because of the subjective and domain-specific nature of identifying trends, this approach can help to identify whether a trend is present. Even fitting a linear model to a trend that is clearly super-linear or exponential can be helpful.

In addition to being used as a trend identification tool, these fit models can also be used to detrend a time series. For example, a linear model can be fit on the time index to predict the observation. This dataset would look as follows:

```
X,  y
1,  obs1
```

```
2,  obs2
3,  obs3
4,  obs4
5,  obs5
```

Listing 13.2: Dataset with time index as a feature.

The predictions from this model will form a straight line that can be taken as the trend line for the dataset. These predictions can also be subtracted from the original time series to provide a detrended version of the dataset.

$$value(t) = observation(t) - prediction(t) \tag{13.2}$$

The residuals from the fit of the model are a detrended form of the dataset. Polynomial curve fitting and other nonlinear models can also be used. We can implement this in Python by training a scikit-learn `LinearRegression` model[1] on the data.

```
# use a linear model to detrend a time series
from pandas import read_csv
from pandas import datetime
from sklearn.linear_model import LinearRegression
from matplotlib import pyplot
import numpy

def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
# fit linear model
X = [i for i in range(0, len(series))]
X = numpy.reshape(X, (len(X), 1))
y = series.values
model = LinearRegression()
model.fit(X, y)
# calculate trend
trend = model.predict(X)
# plot trend
pyplot.plot(y)
pyplot.plot(trend)
pyplot.show()
# detrend
detrended = [y[i]-trend[i] for i in range(0, len(series))]
# plot detrended
pyplot.plot(detrended)
pyplot.show()
```

Listing 13.3: Fit a linear model on the Shampoo Sales dataset.

Running the example first fits the linear model to the integer-indexed observations and plots the trend line over the original dataset.

[1] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

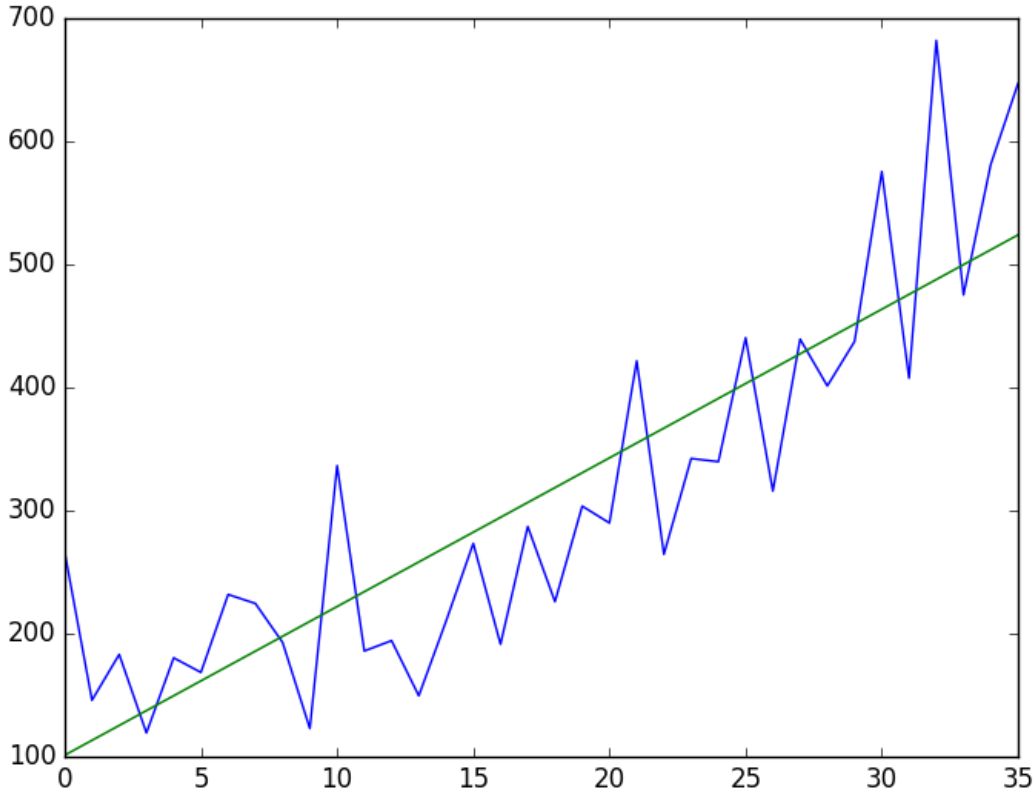

Figure 13.2: Line plot of the Shampoo Sales dataset (blue) and the linear fit (green).

Next, the trend is subtracted from the original dataset and the resulting detrended dataset is plotted.

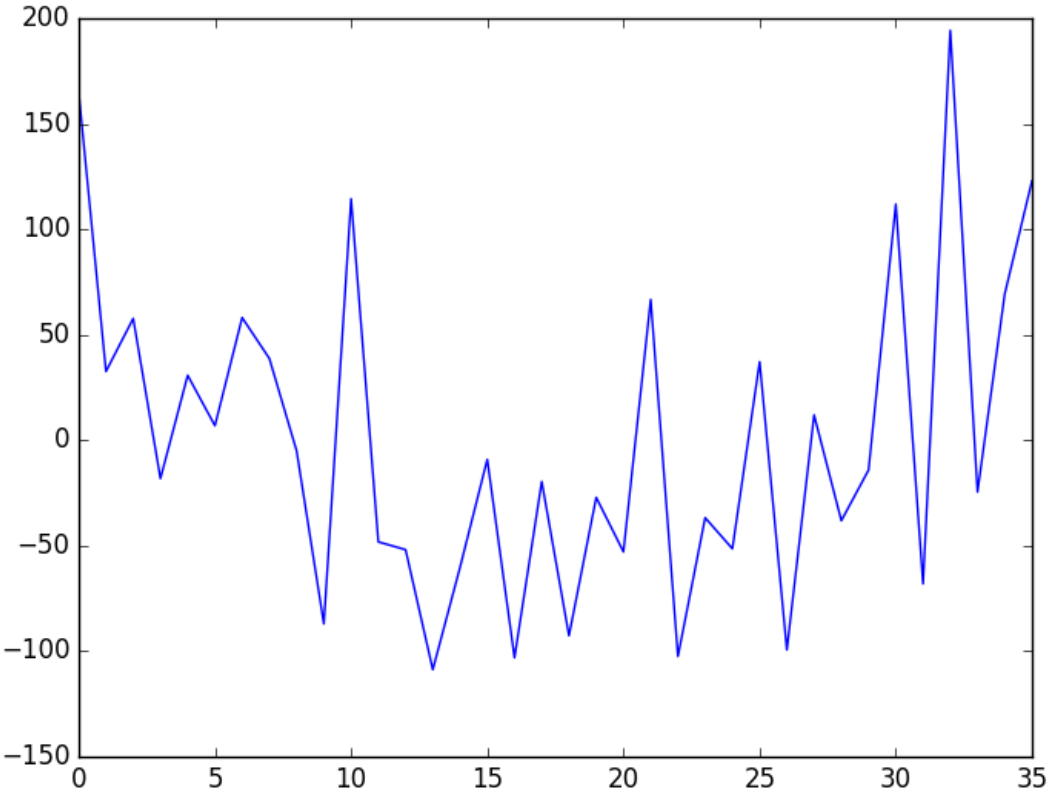

Figure 13.3: Line plot of the detrended Shampoo Sales dataset using the linear fit.

Again, we can see that this approach has effectively detrended the dataset. There may be a parabola in the residuals, suggesting that perhaps a polynomial fit may have done a better job. Because the trend model takes only the integer index of the observation as input, it can be used on new data to either detrend or provide a new input variable for the model.

13.5 Summary

In this tutorial, you discovered trends in time series data and how to remove them with Python. Specifically, you learned:

- About the importance of trend information in time series and how you may be able to use it in machine learning.
- How to use differencing to remove a trend from time series data.
- How to model a linear trend and remove it from time series data.

13.5.1 Next

In the next lesson you will discover how to identify and model seasonality in time series data.

Chapter 14

Use and Remove Seasonality

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

After completing this tutorial, you will know:

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

Let's get started.

14.1 Seasonality in Time Series

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

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

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

14.1.1 Benefits to Machine Learning

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

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

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

14.1.2 Types of Seasonality

There are many types of seasonality; for example:

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

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

14.1.3 Removing Seasonality

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

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

14.2 Minimum Daily Temperatures Dataset

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

14.3 Seasonal Adjustment with Differencing

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

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

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

Listing 14.1: Deseasonalize Minimum Daily Temperatures dataset using differencing.

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

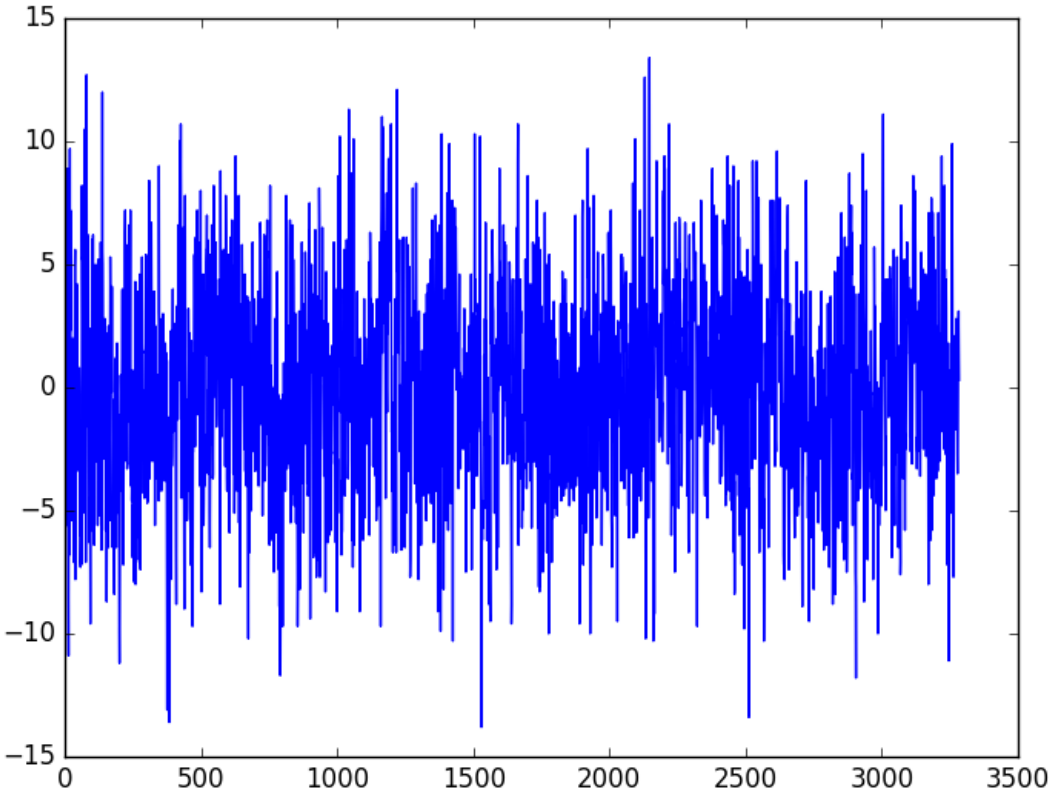

Figure 14.1: Line plot of the deseasonalized Minimum Daily Temperatures dataset using differencing.

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

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

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

Listing 14.2: Resampled Minimum Daily Temperatures dataset to monthly.

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

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

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

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

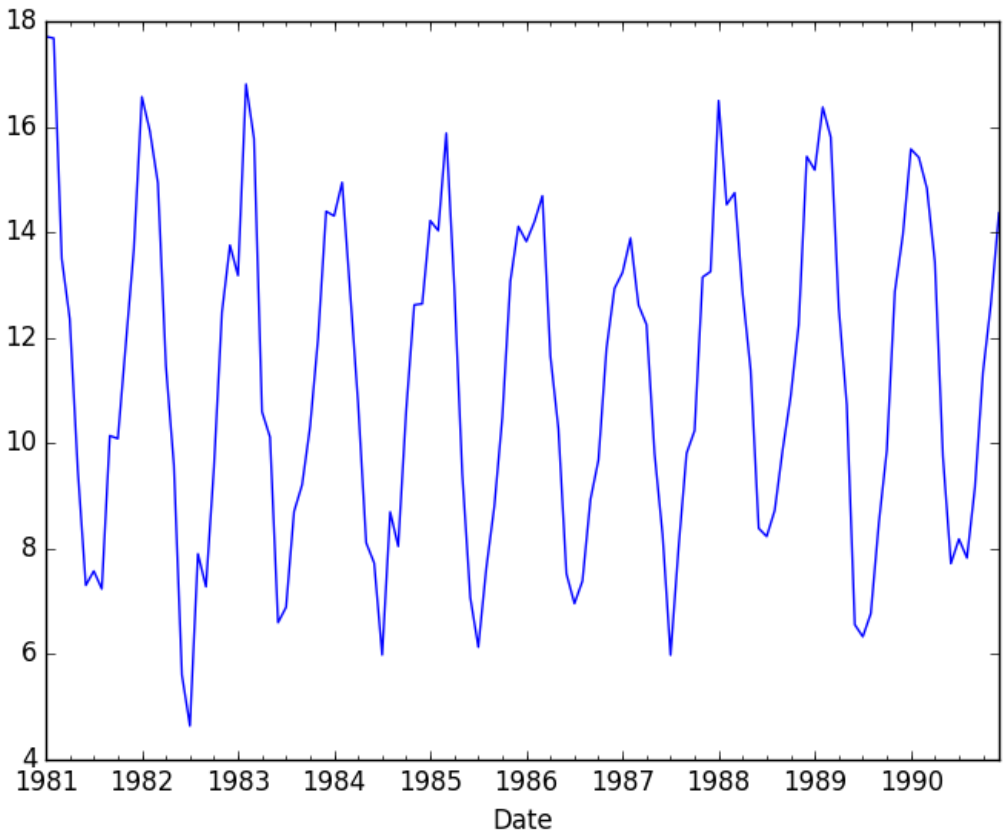

Figure 14.2: Line plot of the monthly Minimum Daily Temperatures dataset.

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

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

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

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

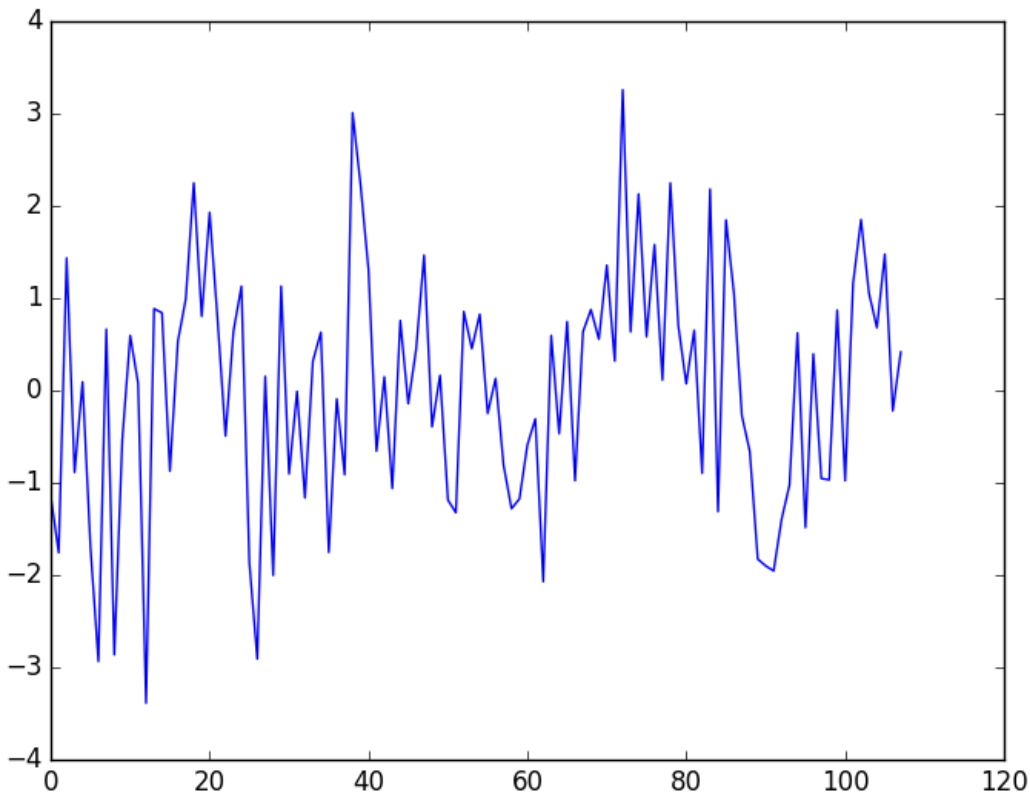

Figure 14.3: Line plot of the deseasonalized monthly Minimum Daily Temperatures dataset.

Next, we can use the monthly average minimum temperatures from the same month in the previous year to adjust the daily minimum temperature dataset. Again, we just skip the first year of data, but the correction using the monthly rather than the daily data may be a more stable approach.

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

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

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

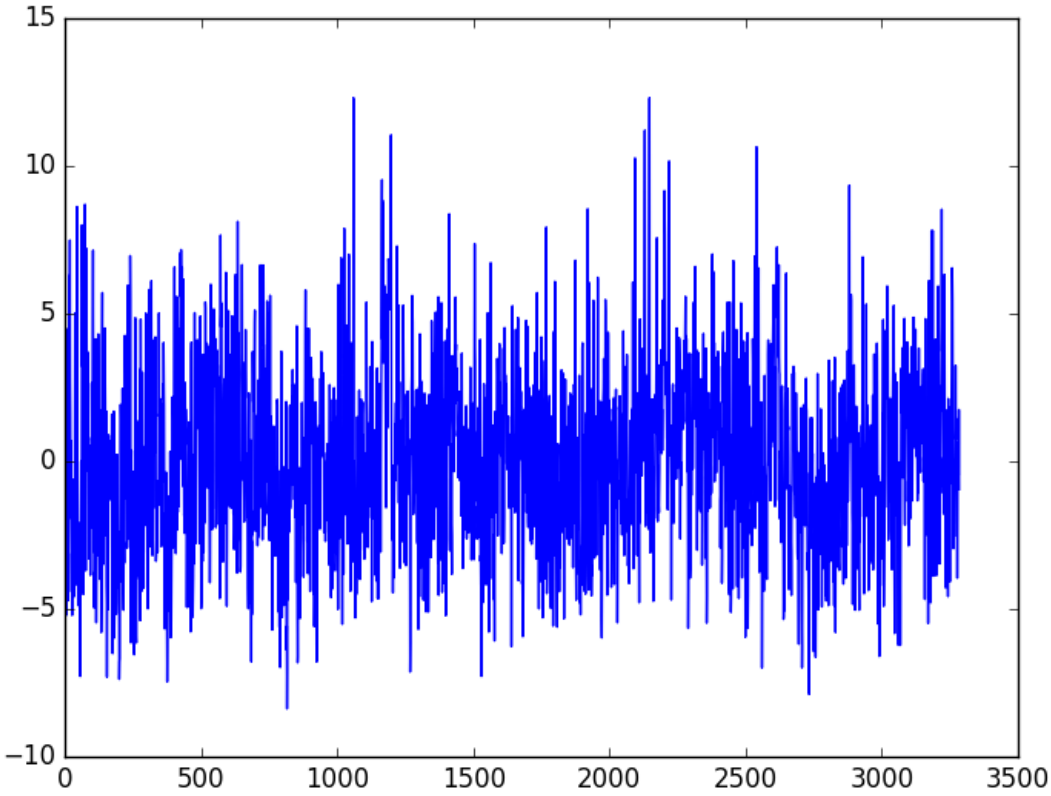

Figure 14.4: Line plot of the deseasonalized Minimum Daily Temperatures dataset using monthly data.

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

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

14.4 Seasonal Adjustment with Modeling

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

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

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

Listing 14.6: Example of time index as a feature.

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

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

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

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

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

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

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

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

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

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

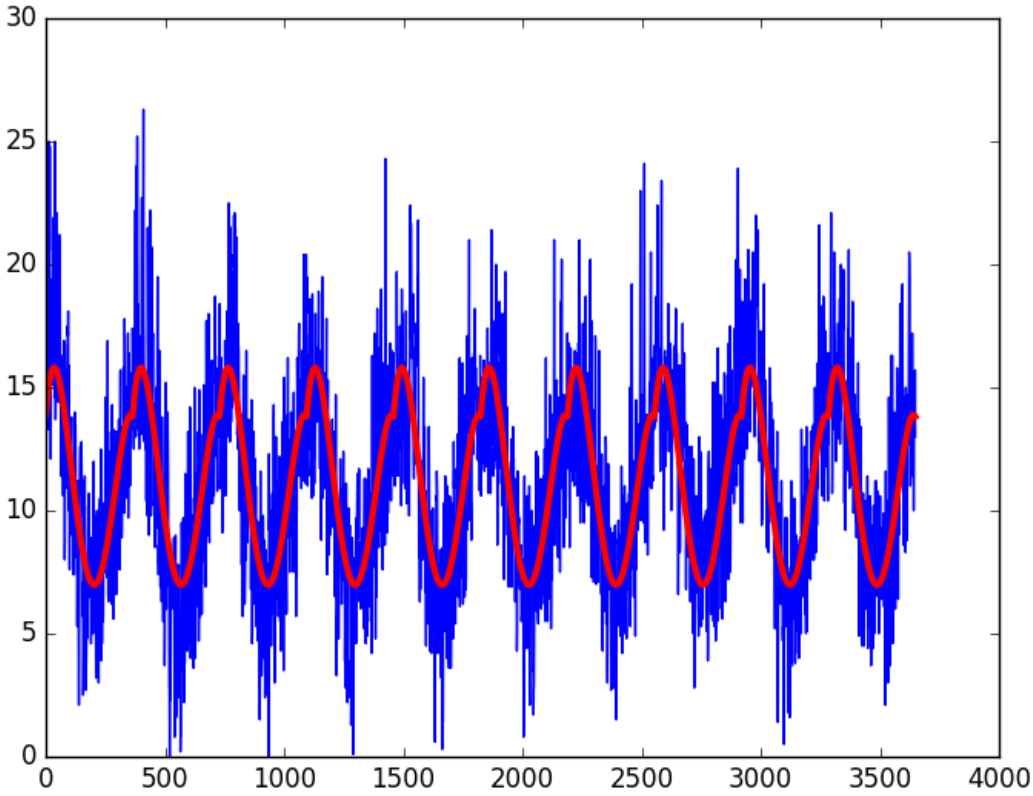

Figure 14.5: Line plot of the Minimum Daily Temperatures dataset (blue) and a nonlinear model of the seasonality (red).

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

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

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

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

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

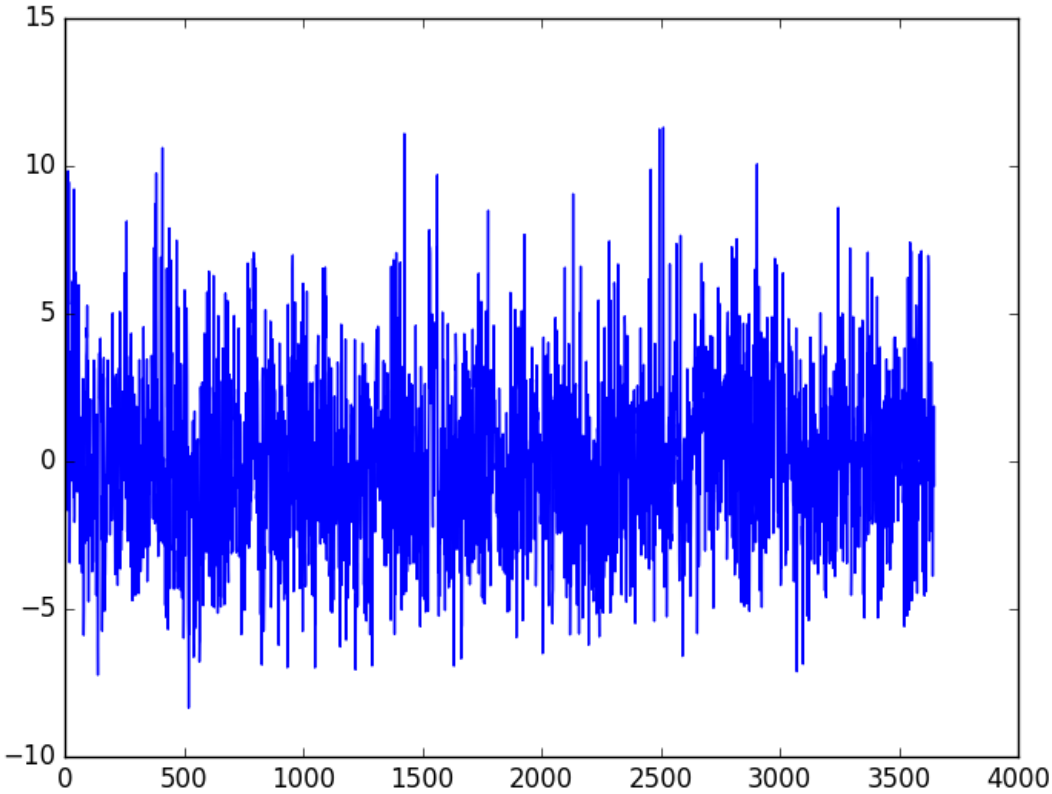

Figure 14.6: Line plot of the deseasonalized Minimum Daily Temperatures dataset using a nonlinear model.

14.5 Summary

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

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

14.5.1 Next

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

Chapter 15

Stationarity in Time Series Data

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

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

Let's get started.

15.1 Stationary Time Series

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

```
# load time series data
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
series.plot()
pyplot.show()
```

Listing 15.1: Load and plot Daily Female Births dataset.

Running the example creates the following plot.

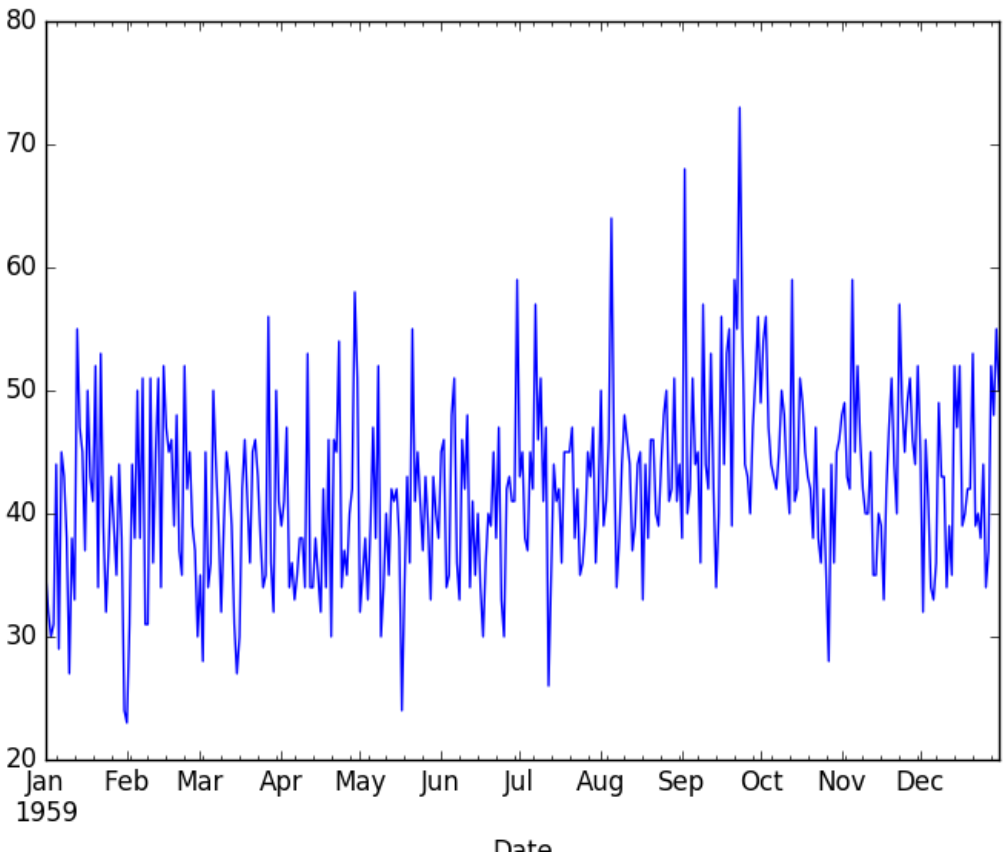

Figure 15.1: Line plot of the stationary Daily Female Births time series dataset.

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 seasonal 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 read_csv
from matplotlib import pyplot
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
series.plot()
pyplot.show()
```

Listing 15.2: Load and plot the Airline Passengers dataset.

Running the example creates the following plot.

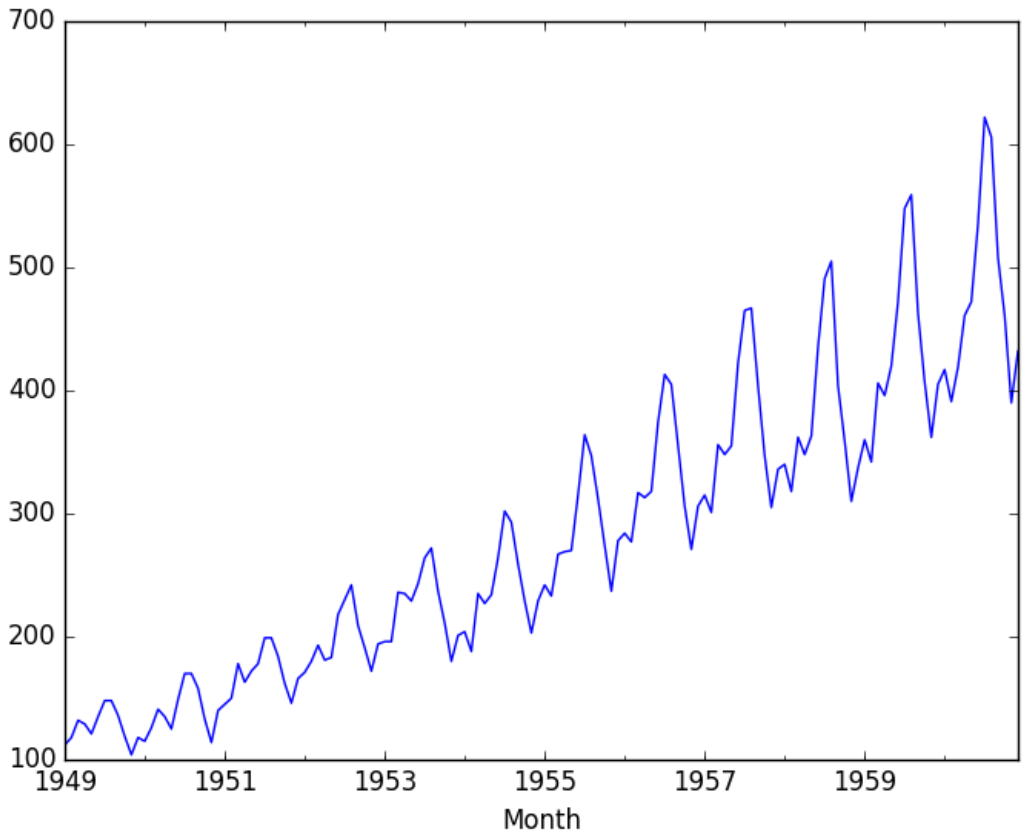

Figure 15.2: Line plot of the non-stationary Airline Passengers time series dataset.

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 read_csv
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
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.

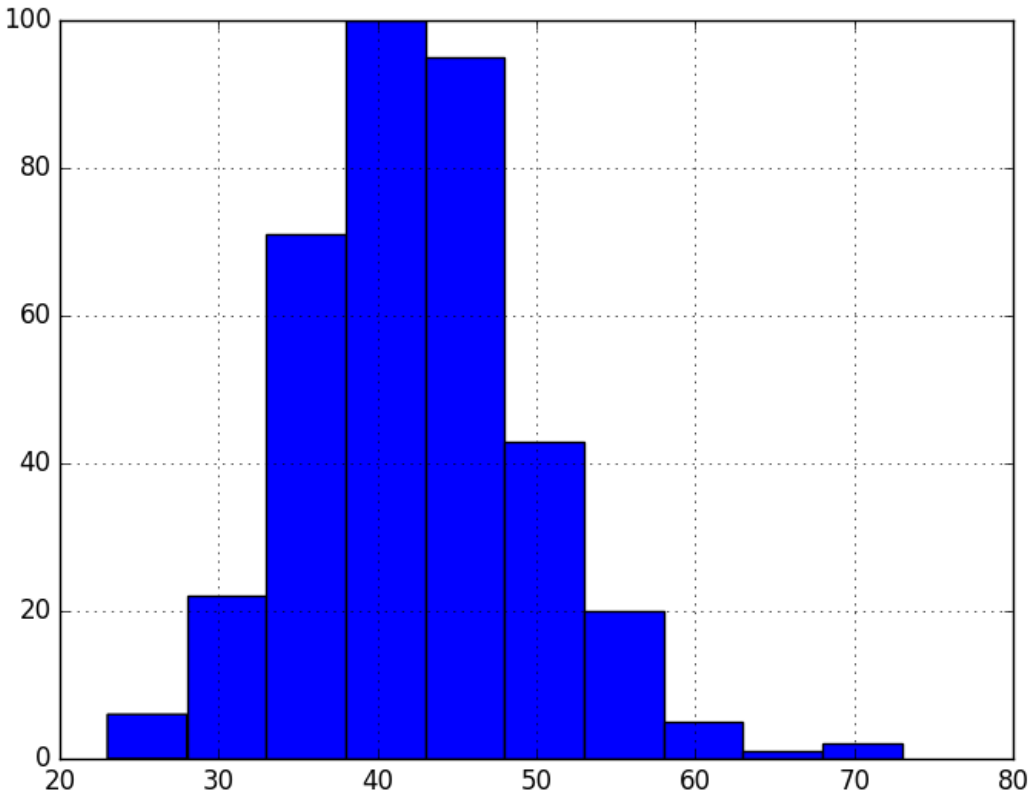

Figure 15.3: Histogram plot of the Daily Female Births dataset.

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 read_csv
```

```
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
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 read_csv
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
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 read_csv
from matplotlib import pyplot
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
```

```
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.

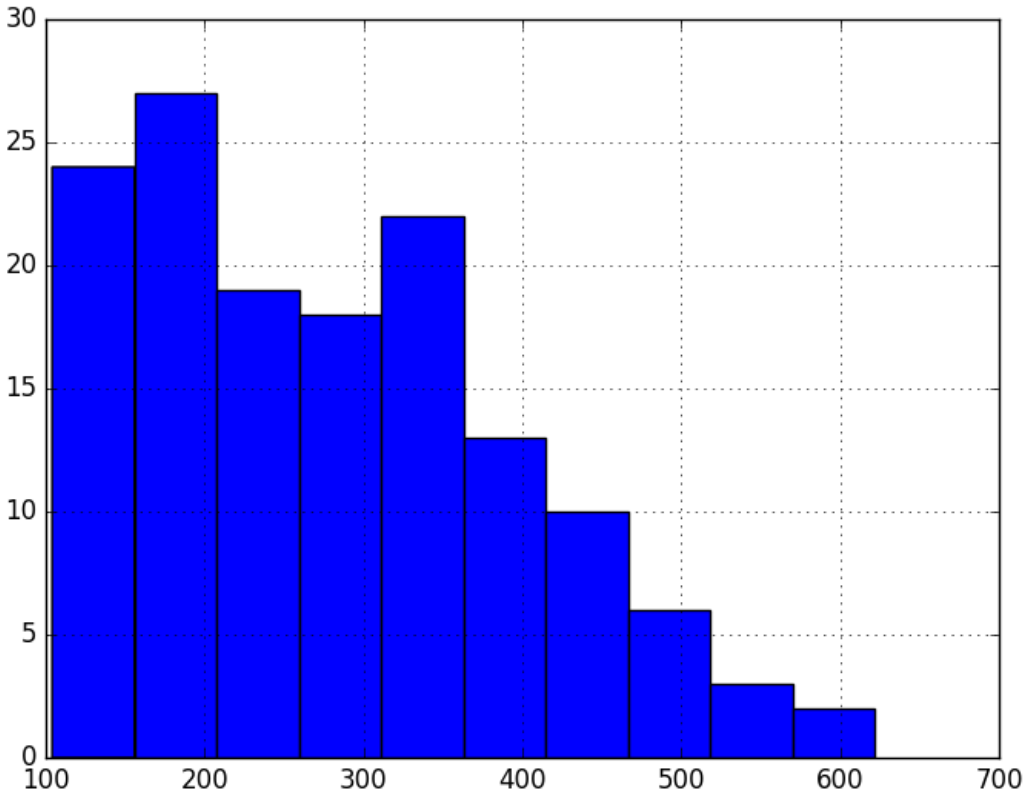

Figure 15.4: Histogram plot of the Airline Passengers dataset.

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 read_csv
from matplotlib import pyplot
from numpy import log
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
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.

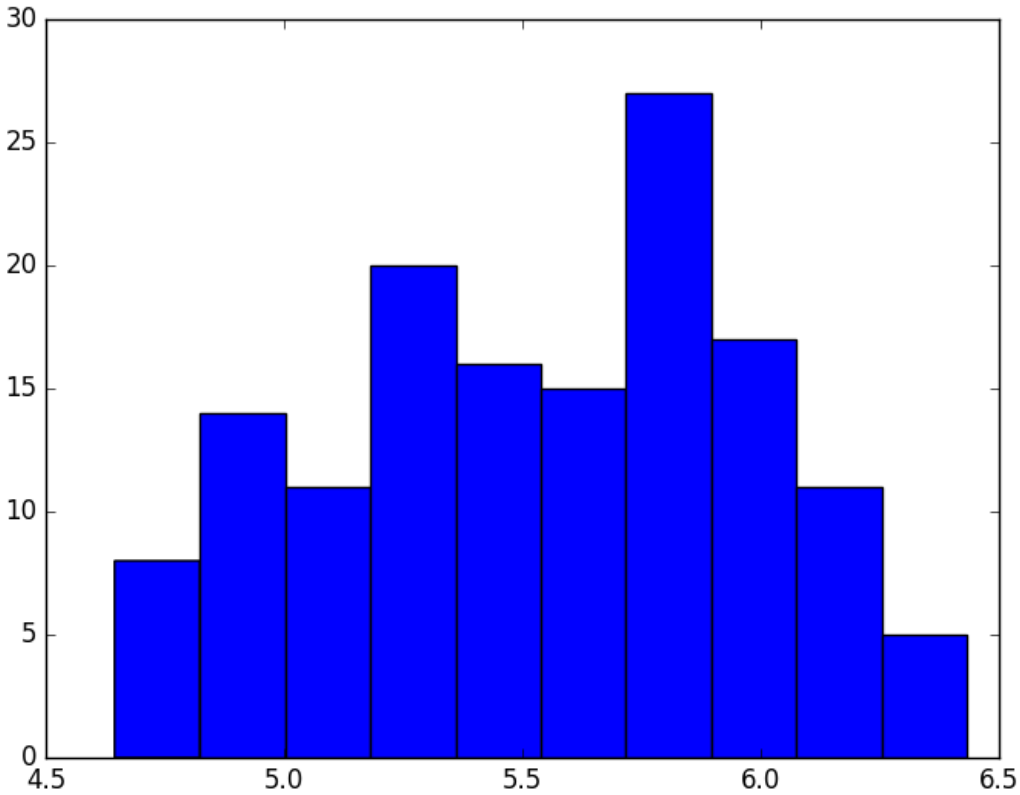

Figure 15.5: Histogram plot of the log-transformed Airline Passengers dataset.

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.

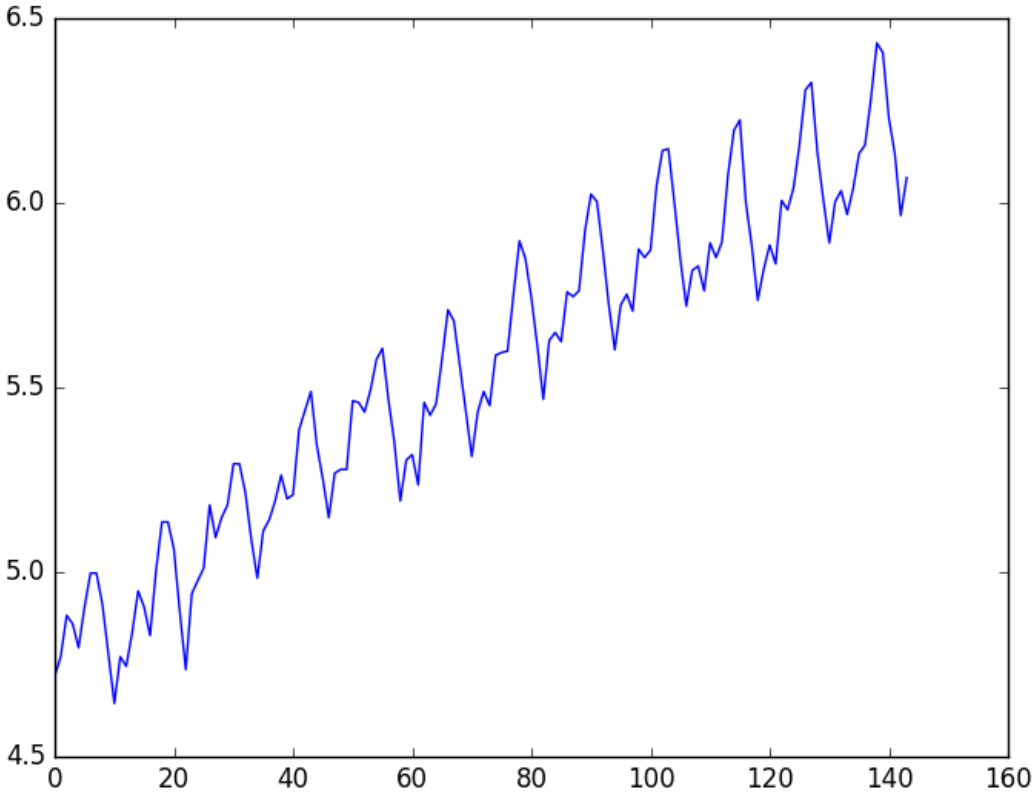

Figure 15.6: Line plot of the log-transformed Airline Passengers dataset.

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 read_csv
from numpy import log
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
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 read_csv
from statsmodels.tsa.stattools import adfuller
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
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 read_csv
from statsmodels.tsa.stattools import adfuller
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
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 read_csv
from statsmodels.tsa.stattools import adfuller
from numpy import log
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
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.

Part IV

Evaluate Models

Chapter 16

Backtest Forecast Models

The goal of time series forecasting is to make accurate predictions about the future. The fast and powerful methods that we rely on in machine learning, such as using train-test splits and k-fold cross-validation, do not work in the case of time series data. This is because they ignore the temporal components inherent in the problem. In this tutorial, you will discover how to evaluate machine learning models on time series data with Python. In the field of time series forecasting, this is called backtesting or hindcasting. After completing this tutorial, you will know:

- The limitations of traditional methods of model evaluation from machine learning and why evaluating models on out of sample data is required.
- How to create train-test splits and multiple train-test splits of time series data for model evaluation in Python.
- How walk-forward validation provides the most realistic evaluation of machine learning models on time series data.

Let's get started.

16.1 Model Evaluation

How do we know how good a given model is? We could evaluate it on the data used to train it. This would be invalid. It might provide insight into how the selected model works, and even how it may be improved. But, any estimate of performance on this data would be optimistic, and any decisions based on this performance would be biased. Why?

It is helpful to take it to an extreme: A model that remembered the timestamps and value for each observation would achieve perfect performance. All real models we prepare will report a pale version of this result.

When evaluating a model for time series forecasting, we are interested in the performance of the model on data that was not used to train it. In machine learning, we call this unseen or out of sample data. We can do this by splitting up the data that we do have available. We use some to prepare the model and we hold back some data and ask the model to make predictions for that period. The evaluation of these predictions will provide a good proxy for how the model will perform when we use it operationally.

In applied machine learning, we often split our data into a train and a test set: the training set used to prepare the model and the test set used to evaluate it. We may even use k-fold cross-validation that repeats this process by systematically splitting the data into k groups, each given a chance to be a held out model.

These methods cannot be directly used with time series data. This is because they assume that there is no relationship between the observations, that each observation is independent. This is not true of time series data, where the time dimension of observations means that we cannot randomly split them into groups. Instead, we must split data up and respect the temporal order in which values were observed.

In time series forecasting, this evaluation of models on historical data is called backtesting. In some time series domains, such as meteorology, this is called hindcasting, as opposed to forecasting. We will look at three different methods that you can use to backtest your machine learning models on time series problems. They are:

1. Train-Test split that respect temporal order of observations.
2. Multiple Train-Test splits that respect temporal order of observations.
3. Walk-Forward Validation where a model may be updated each time step new data is received.

First, let's take a look at a small, univariate time series data we will use as context to understand these three backtesting methods: the Sunspot dataset.

16.2 Monthly Sunspots Dataset

In this lesson, we will use the Monthly Sunspots dataset as an example. This dataset describes a monthly count of the number of observed sunspots for just over 230 years (1749-1983). You can learn more about the dataset in Appendix A.3. Place the dataset in your current working directory with the filename `sunspots.csv`.

16.3 Train-Test Split

You can split your dataset into training and testing subsets. Your model can be prepared on the training dataset and predictions can be made and evaluated for the test dataset. This can be done by selecting an arbitrary split point in the ordered list of observations and creating two new datasets. Depending on the amount of data you have available and the amount of data required, you can use splits of 50-50, 70-30 and 90-10. It is straightforward to split data in Python.

After loading the dataset as a Pandas `Series`, we can extract the NumPy array of data values. The split point can be calculated as a specific index in the array. All records up to the split point are taken as the training dataset and all records from the split point to the end of the list of observations are taken as the test set. Below is an example of this in Python using a split of 66-34.

```
# calculate a train-test split of a time series dataset
from pandas import read_csv
series = read_csv('sunspots.csv', header=0, index_col=0, parse_dates=True, squeeze=True)
X = series.values
train_size = int(len(X) * 0.66)
train, test = X[0:train_size], X[train_size:len(X)]
print('Observations: %d' % (len(X)))
print('Training Observations: %d' % (len(train)))
print('Testing Observations: %d' % (len(test)))
```

Listing 16.1: Example of a train-test split of the Monthly Sunspot dataset.

Running the example prints the size of the loaded dataset and the size of the train and test sets created from the split.

```
Observations: 2820
Training Observations: 1861
Testing Observations: 959
```

Listing 16.2: Example output of a train-test split of the Monthly Sunspot dataset.

We can make this visually by plotting the training and test sets using different colors.

```
# plot train-test split of time series data
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('sunspots.csv', header=0, index_col=0, parse_dates=True, squeeze=True)
X = series.values
train_size = int(len(X) * 0.66)
train, test = X[0:train_size], X[train_size:len(X)]
print('Observations: %d' % (len(X)))
print('Training Observations: %d' % (len(train)))
print('Testing Observations: %d' % (len(test)))
pyplot.plot(train)
pyplot.plot([None for i in train] + [x for x in test])
pyplot.show()
```

Listing 16.3: Example of a train-test split of the Monthly Sunspot dataset with plot.

Running the example plots the training dataset as blue and the test dataset as green.

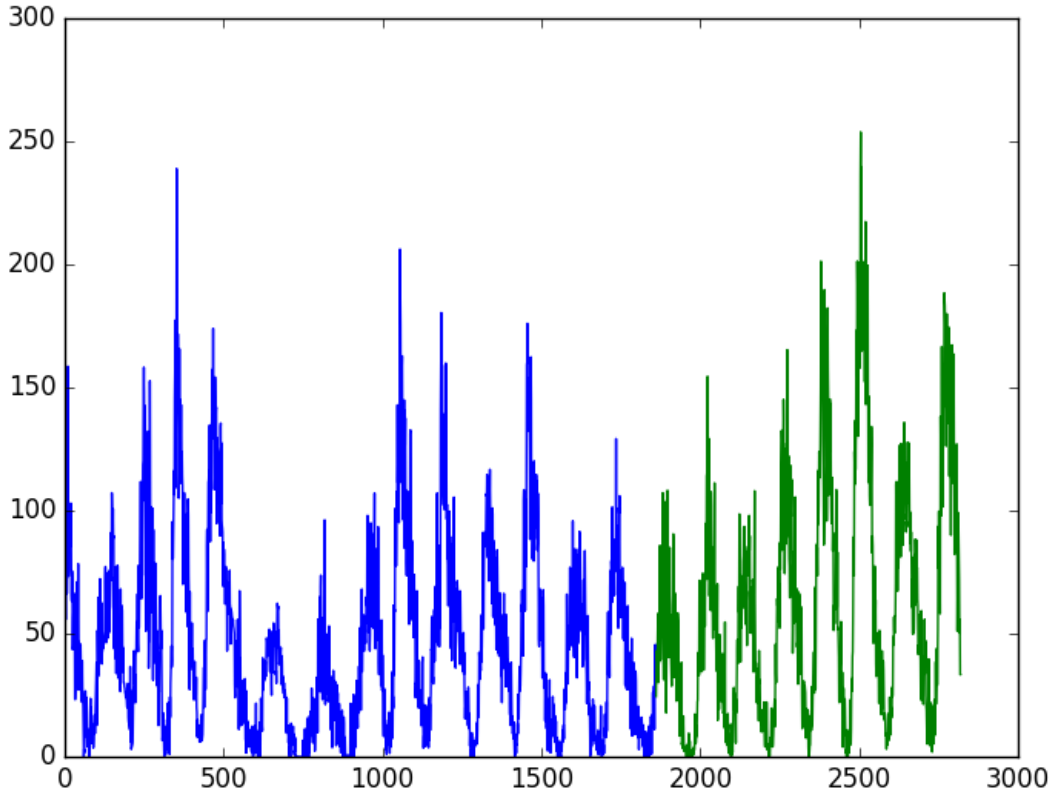

Figure 16.1: Line plot train (blue) and test (green) split of the Monthly Sunspot dataset.

Using a train-test split method to evaluate machine learning models is fast. Preparing the data is simple and intuitive and only one model is created and evaluated. It is useful when you have a large amount of data so that both training and tests sets are representative of the original problem. Next, we will look at repeating this process multiple times.

16.4 Multiple Train-Test Splits

We can repeat the process of splitting the time series into train and test sets multiple times. This will require multiple models to be trained and evaluated, but this additional computational expense will provide a more robust estimate of the expected performance of the chosen method and configuration on unseen data. We could do this manually by repeating the process described in the previous section with different split points.

Alternately, the scikit-learn library provides this capability for us in the `TimeSeriesSplit` object. You must specify the number of splits to create and the `TimeSeriesSplit` to return the indexes of the train and test observations for each requested split. The total number of training

and test observations are calculated each split iteration (`i`) as follows:

$$\begin{aligned} training_size &= i \times \frac{n_samples}{n_splits + 1} + n_samples \bmod (n_splits + 1) \\ test_size &= \frac{n_samples}{n_splits + 1} \end{aligned} \tag{16.1}$$

Where `n_samples` is the total number of observations and `n_splits` is the total number of splits. Let's make this concrete with an example. Assume we have 100 observations and we want to create 2 splits. For the first split, the train size would be calculated as:

$$\begin{aligned} train &= i \times \frac{n_samples}{n_splits + 1} + n_samples \bmod (n_splits + 1) \\ &= 1 \times \frac{100}{2 + 1} + 100 \bmod (2 + 1) \\ &= 33.3 \\ &= 33 \end{aligned} \tag{16.2}$$

The test set size for the first split would be calculated as:

$$\begin{aligned} test &= \frac{n_samples}{n_splits + 1} \\ &= \frac{100}{2 + 1} \\ &= 33.3 \\ &= 33 \end{aligned} \tag{16.3}$$

Or the first 33 records are used for training and the next 33 records are used for testing. The size of the train set on the second split is calculated as follows:

$$\begin{aligned} train &= i \times \frac{n_samples}{n_splits + 1} + n_samples \bmod (n_splits + 1) \\ &= 2 \times \frac{100}{2 + 1} + 100 \bmod (2 + 1) \\ &= 66.6 \\ &= 66 \end{aligned} \tag{16.4}$$

The test set size on the second split is calculated as follows:

$$\begin{aligned} test &= \frac{n_samples}{n_splits + 1} \\ &= \frac{100}{2 + 1} \\ &= 33.3 \\ &= 33 \end{aligned} \tag{16.5}$$

Or, the first 67 records are used for training and the remaining 33 records are used for testing. You can see that the test size stays consistent. This means that performance statistics calculated on the predictions of each trained model will be consistent and can be combined and compared. It provides an apples-to-apples comparison.

What differs is the number of records used to train the model each split, offering a larger and larger history to work with. This may make an interesting aspect of the analysis of results. Alternately, this too could be controlled by holding the number of observations used to train the model consistent and only using the same number of the most recent (last) observations in the training dataset each split to train the model, 33 in this contrived example.

Let's look at how we can apply the `TimeSeriesSplit` on our sunspot data. The dataset has 2,820 observations. Let's create 3 splits for the dataset. Using the same arithmetic above, we would expect the following train and test splits to be created:

- **Split 1**: 705 train, 705 test
- **Split 2**: 1,410 train, 705 test
- **Split 3**: 2,115 train, 705 test

As in the previous example, we will plot the train and test observations using separate colors. In this case, we will have 3 splits, so that will be 3 separate plots of the data.

```
# calculate repeated train-test splits of time series data
from pandas import read_csv
from sklearn.model_selection import TimeSeriesSplit
from matplotlib import pyplot
series = read_csv('sunspots.csv', header=0, index_col=0, parse_dates=True, squeeze=True)
X = series.values
splits = TimeSeriesSplit(n_splits=3)
pyplot.figure(1)
index = 1
for train_index, test_index in splits.split(X):
  train = X[train_index]
  test = X[test_index]
  print('Observations: %d' % (len(train) + len(test)))
  print('Training Observations: %d' % (len(train)))
  print('Testing Observations: %d' % (len(test)))
  pyplot.subplot(310 + index)
  pyplot.plot(train)
  pyplot.plot([None for i in train] + [x for x in test])
  index += 1
pyplot.show()
```

Listing 16.4: Example of multiple train-test split of the Monthly Sunspot dataset.

Running the example prints the number and size of the train and test sets for each split. We can see the number of observations in each of the train and test sets for each split match the expectations calculated using the simple arithmetic above.

```
Observations: 1410
Training Observations: 705
Testing Observations: 705

Observations: 2115
Training Observations: 1410
Testing Observations: 705

Observations: 2820
Training Observations: 2115
```

```
Testing Observations: 705
```

Listing 16.5: Example output multiple train-test split of the Monthly Sunspot dataset, spaced for readability.

The plot also shows the 3 splits and the growing number of total observations in each subsequent plot.

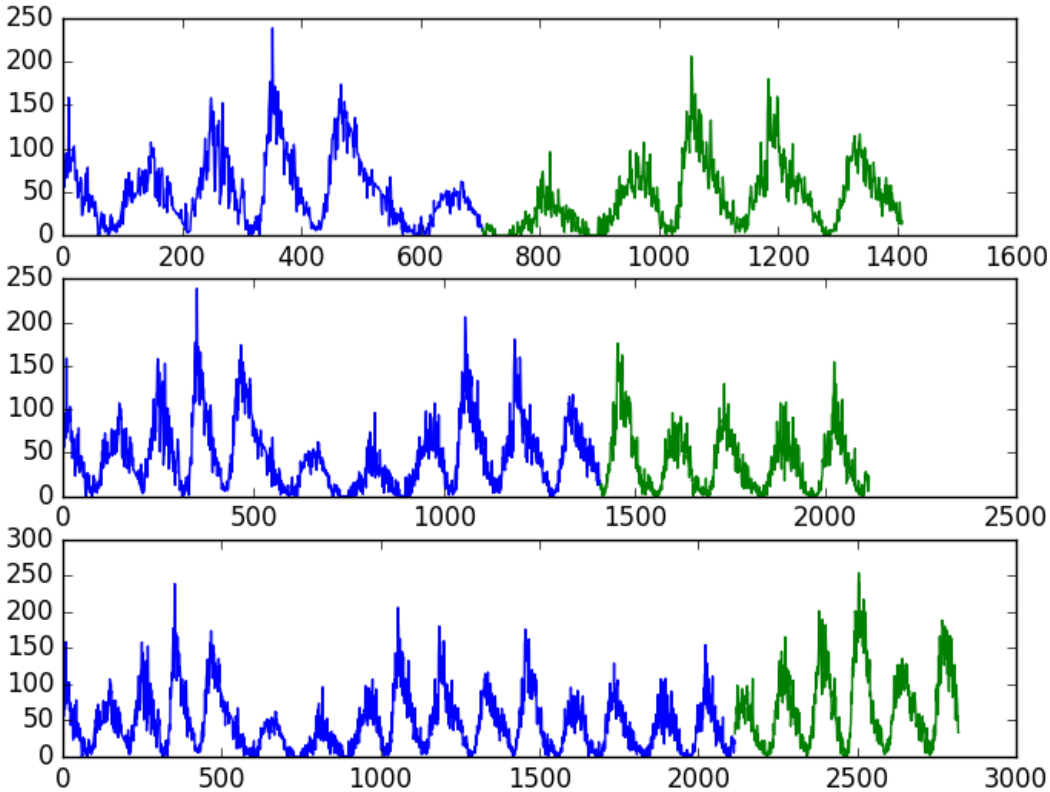

Figure 16.2: Line plots of repeated train (blue) and test (green) splits of the Monthly Sunspot dataset.

Using multiple train-test splits will result in more models being trained, and in turn, a more accurate estimate of the performance of the models on unseen data. A limitation of the train-test split approach is that the trained models remain fixed as they are evaluated on each evaluation in the test set. This may not be realistic as models can be retrained as new daily or monthly observations are made available. This concern is addressed in the next section.

16.5 Walk Forward Validation

In practice, we very likely will retrain our model as new data becomes available. This would give the model the best opportunity to make good forecasts at each time step. We can evaluate our machine learning models under this assumption. There are few decisions to make:

1. **Minimum Number of Observations**. First, we must select the minimum number of observations required to train the model. This may be thought of as the window width if a sliding window is used (see next point).

2. **Sliding or Expanding Window**. Next, we need to decide whether the model will be trained on all data it has available or only on the most recent observations. This determines whether a sliding or expanding window will be used.

After a sensible configuration is chosen for your test-setup, models can be trained and evaluated.

1. Starting at the beginning of the time series, the minimum number of samples in the window is used to train a model.

2. The model makes a prediction for the next time step.

3. The prediction is stored or evaluated against the known value.

4. The window is expanded to include the known value and the process is repeated (go to step 1.)

Because this methodology involves moving along the time series one-time step at a time, it is often called Walk Forward Testing or Walk Forward Validation. Additionally, because a sliding or expanding window is used to train a model, this method is also referred to as Rolling Window Analysis or a Rolling Forecast. This capability is currently not available in scikit-learn, although you could contrive the same effect with a carefully configured `TimeSeriesSplit`. Below is an example of how to split data into train and test sets using the Walk Forward Validation method.

```
# walk forward evaluation model for time series data
from pandas import read_csv
series = read_csv('sunspots.csv', header=0, index_col=0, parse_dates=True, squeeze=True)
X = series.values
n_train = 500
n_records = len(X)
for i in range(n_train, n_records):
  train, test = X[0:i], X[i:i+1]
  print('train=%d, test=%d' % (len(train), len(test)))
```

Listing 16.6: Example of walk-forward validation of the Monthly Sunspot dataset.

Running the example simply prints the size of the training and test sets created. We can see the train set expanding each time step and the test set fixed at one time step ahead. Within the loop is where you would train and evaluate your model.

```
train=500, test=1
train=501, test=1
train=502, test=1
...
train=2815, test=1
train=2816, test=1
train=2817, test=1
train=2818, test=1
train=2819, test=1
```

Listing 16.7: Example output walk-forward validation of the Monthly Sunspot dataset, trimmed for brevity.

You can see that many more models are created. This has the benefit again of providing a much more robust estimation of how the chosen modeling method and parameters will perform in practice. This improved estimate comes at the computational cost of creating so many models. This is not expensive if the modeling method is simple or dataset is small (as in this example), but could be an issue at scale. In the above case, 2,820 models would be created and evaluated.

As such, careful attention needs to be paid to the window width and window type. These could be adjusted to contrive a test harness on your problem that is significantly less computationally expensive. Walk-forward validation is the gold standard of model evaluation. It is the k-fold cross-validation of the time series world and is recommended for your own projects.

16.6 Summary

In this tutorial, you discovered how to backtest machine learning models on time series data with Python. Specifically, you learned:

- About the importance of evaluating the performance of models on unseen or out-of-sample data.
- How to create train-test splits of time series data, and how to create multiple such splits automatically.
- How to use walk-forward validation to provide the most realistic test harness for evaluating your models.

16.6.1 Next

In the next lesson you will discover performance measures for evaluating the skill of forecast models.

Chapter 17

Forecasting Performance Measures

Time series prediction performance measures provide a summary of the skill and capability of the forecast model that made the predictions. There are many different performance measures to choose from. It can be confusing to know which measure to use and how to interpret the results. In this tutorial, you will discover performance measures for evaluating time series forecasts with Python. Time series generally focus on the prediction of real values, called regression problems. Therefore the performance measures in this tutorial will focus on methods for evaluating real-valued predictions. After completing this tutorial, you will know:

- Basic measures of forecast performance, including residual forecast error and forecast bias.
- Time series forecast error calculations that have the same units as the expected outcomes such as mean absolute error.
- Widely used error calculations that punish large errors, such as mean squared error and root mean squared error.

Let's get started.

17.1 Forecast Error (or Residual Forecast Error)

The forecast error is calculated as the expected value minus the predicted value. This is called the residual error of the prediction.

$$forecast_error = expected_value - predicted_value \tag{17.1}$$

The forecast error can be calculated for each prediction, providing a time series of forecast errors. The example below demonstrates how the forecast error can be calculated for a series of 5 predictions compared to 5 expected values. The example was contrived for demonstration purposes.

```
# calculate forecast error
expected = [0.0, 0.5, 0.0, 0.5, 0.0]
predictions = [0.2, 0.4, 0.1, 0.6, 0.2]
forecast_errors = [expected[i]-predictions[i] for i in range(len(expected))]
print('Forecast Errors: %s' % forecast_errors)
```

Listing 17.1: Example of calculating forecast error.

Running the example calculates the forecast error for each of the 5 predictions. The list of forecast errors is then printed.

```
Forecast Errors: [-0.2, 0.09999999999999998, -0.1, -0.09999999999999998, -0.2]
```

Listing 17.2: Example output of calculating forecast error.

The units of the forecast error are the same as the units of the prediction. A forecast error of zero indicates no error, or perfect skill for that forecast.

17.2 Mean Forecast Error (or Forecast Bias)

Mean forecast error is calculated as the average of the forecast error values.

$$mean_forecast_error = mean(forecast_error) \tag{17.2}$$

Forecast errors can be positive and negative. This means that when the average of these values is calculated, an ideal mean forecast error would be zero. A mean forecast error value other than zero suggests a tendency of the model to over forecast (positive error) or under forecast (negative error). As such, the mean forecast error is also called the forecast bias. The forecast error can be calculated directly as the mean of the forecast values. The example below demonstrates how the mean of the forecast errors can be calculated manually.

```
# calculate mean forecast error
expected = [0.0, 0.5, 0.0, 0.5, 0.0]
predictions = [0.2, 0.4, 0.1, 0.6, 0.2]
forecast_errors = [expected[i]-predictions[i] for i in range(len(expected))]
bias = sum(forecast_errors) * 1.0/len(expected)
print('Bias: %f' % bias)
```

Listing 17.3: Example of calculating mean forecast error.

Running the example prints the mean forecast error, also known as the forecast bias.

```
Bias: -0.100000
```

Listing 17.4: Example output of calculating mean forecast error.

The units of the forecast bias are the same as the units of the predictions. A forecast bias of zero, or a very small number near zero, shows an unbiased model.

17.3 Mean Absolute Error

The mean absolute error, or MAE, is calculated as the average of the forecast error values, where all of the forecast values are forced to be positive. Forcing values to be positive is called making them absolute. This is signified by the absolute function `abs()` or shown mathematically as two pipe characters around the value: `|value|`.

$$mean_absolute_error = mean(abs(forecast_error)) \tag{17.3}$$

Where `abs()` makes values positive, `forecast_error` is one or a sequence of forecast errors, and `mean()` calculates the average value. We can use the `mean_absolute_error()` function[1] from the scikit-learn library to calculate the mean absolute error for a list of predictions. The example below demonstrates this function.

```
# calculate mean absolute error
from sklearn.metrics import mean_absolute_error
expected = [0.0, 0.5, 0.0, 0.5, 0.0]
predictions = [0.2, 0.4, 0.1, 0.6, 0.2]
mae = mean_absolute_error(expected, predictions)
print('MAE: %f' % mae)
```

Listing 17.5: Example of calculating mean absolute error.

Running the example calculates and prints the mean absolute error for a list of 5 expected and predicted values.

```
MAE: 0.140000
```

Listing 17.6: Example output of calculating mean absolute error.

These error values are in the original units of the predicted values. A mean absolute error of zero indicates no error.

17.4 Mean Squared Error

The mean squared error, or MSE, is calculated as the average of the squared forecast error values. Squaring the forecast error values forces them to be positive; it also has the effect of putting more weight on large errors. Very large or outlier forecast errors are squared, which in turn has the effect of dragging the mean of the squared forecast errors out resulting in a larger mean squared error score. In effect, the score gives worse performance to those models that make large wrong forecasts.

$$mean_squared_error = mean(forecast_error^2) \tag{17.4}$$

We can use the `mean_squared_error()` function[2] from scikit-learn to calculate the mean squared error for a list of predictions. The example below demonstrates this function.

```
# calculate mean squared error
from sklearn.metrics import mean_squared_error
expected = [0.0, 0.5, 0.0, 0.5, 0.0]
predictions = [0.2, 0.4, 0.1, 0.6, 0.2]
mse = mean_squared_error(expected, predictions)
print('MSE: %f' % mse)
```

Listing 17.7: Example of calculating mean squared error.

Running the example calculates and prints the mean squared error for a list of expected and predicted values.

[1] http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html

[2] http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html

```
MSE: 0.022000
```

Listing 17.8: Example output of calculating mean squared error.

The error values are in squared units of the predicted values. A mean squared error of zero indicates perfect skill, or no error.

17.5 Root Mean Squared Error

The mean squared error described above is in the squared units of the predictions. It can be transformed back into the original units of the predictions by taking the square root of the mean squared error score. This is called the root mean squared error, or RMSE.

$$rmse = \sqrt{mean_squared_error} \tag{17.5}$$

This can be calculated by using the `sqrt()` math function on the mean squared error calculated using the `mean_squared_error()` scikit-learn function.

```
# calculate root mean squared error
from sklearn.metrics import mean_squared_error
from math import sqrt
expected = [0.0, 0.5, 0.0, 0.5, 0.0]
predictions = [0.2, 0.4, 0.1, 0.6, 0.2]
mse = mean_squared_error(expected, predictions)
rmse = sqrt(mse)
print('RMSE: %f' % rmse)
```

Listing 17.9: Example of calculating root mean squared error.

Running the example calculates the root mean squared error.

```
RMSE: 0.148324
```

Listing 17.10: Example output of calculating root mean squared error.

The RMES error values are in the same units as the predictions. As with the mean squared error, an RMSE of zero indicates no error.

17.6 Summary

In this tutorial, you discovered a suite of 5 standard time series performance measures in Python. Specifically, you learned:

- How to calculate forecast residual error and how to estimate the bias in a list of forecasts.
- How to calculate mean absolute forecast error to describe error in the same units as the predictions.
- How to calculate the widely used mean squared error and root mean squared error for forecasts.

17.6.1 Next

In the next lesson you will discover the naive forecast called the persistence model.

Chapter 18

Persistence Model for Forecasting

Establishing a baseline is essential on any time series forecasting problem. A baseline in performance gives you an idea of how well all other models will actually perform on your problem. In this tutorial, you will discover how to develop a persistence forecast that you can use to calculate a baseline level of performance on a time series dataset with Python. After completing this tutorial, you will know:

- The importance of calculating a baseline of performance on time series forecast problems.
- How to develop a persistence model from scratch in Python.
- How to evaluate the forecast from a persistence model and use it to establish a baseline in performance.

Let's get started.

18.1 Forecast Performance Baseline

A baseline in forecast performance provides a point of comparison. It is a point of reference for all other modeling techniques on your problem. If a model achieves performance at or below the baseline, the technique should be fixed or abandoned. The technique used to generate a forecast to calculate the baseline performance must be easy to implement and naive of problem-specific details. Before you can establish a performance baseline on your forecast problem, you must develop a test harness. This is comprised of:

1. The dataset you intend to use to train and evaluate models.
2. The resampling technique you intend to use to estimate the performance of the technique (e.g. train/test split).
3. The performance measure you intend to use to evaluate forecasts (e.g. root mean squared error).

Once prepared, you then need to select a naive technique that you can use to make a forecast and calculate the baseline performance. The goal is to get a baseline performance on your time series forecast problem as quickly as possible so that you can get to work better understanding the dataset and developing more advanced models. Three properties of a good technique for making a baseline forecast are:

- **Simple**: A method that requires little or no training or intelligence.
- **Fast**: A method that is fast to implement and computationally trivial to make a prediction.
- **Repeatable**: A method that is deterministic, meaning that it produces an expected output given the same input.

A common algorithm used in establishing a baseline performance is the persistence algorithm.

18.2 Persistence Algorithm

The most common baseline method for supervised machine learning is the Zero Rule algorithm. This algorithm predicts the majority class in the case of classification, or the average outcome in the case of regression. This could be used for time series, but does not respect the serial correlation structure in time series datasets. The equivalent technique for use with time series dataset is the persistence algorithm.

The persistence algorithm uses the value at the current time step (`t`) to predict the expected outcome at the next time step (`t+1`). This satisfies the three above conditions for a baseline forecast. To make this concrete, we will look at how to develop a persistence model and use it to establish a baseline performance for a simple univariate time series problem. First, let's review the Shampoo Sales dataset.

18.3 Shampoo Sales Dataset

In this lesson, we will use the Shampoo Sales dataset as an example. This dataset describes the monthly number of sales of shampoo over a 3 year period. You can learn more about the dataset in Appendix A.1. Place the dataset in your current working directory with the filename `shampoo-sales.csv`.

18.4 Persistence Algorithm Steps

A persistence model can be implemented easily in Python. We will break this tutorial down into 4 steps:

1. Transform the univariate dataset into a supervised learning problem.
2. Establish the train and test datasets for the test harness.
3. Define the persistence model.
4. Make a forecast and establish a baseline performance.
5. Review the complete example and plot the output.

Let's dive in.

18.4.1 Step 1: Define the Supervised Learning Problem

The first step is to load the dataset and create a lagged representation. That is, given the observation at `t`, predict the observation at `t+1`.

```
# Create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
print(dataframe.head(5))
```

Listing 18.1: Create the lagged dataset.

This snippet creates the dataset and prints the first 5 rows of the new dataset. We can see that the first row (index 0) will have to be discarded as there was no observation prior to the first observation to use to make the prediction. From a supervised learning perspective, the `t` column is the input variable, or `X`, and the `t+1` column is the output variable, or `y`.

```
     t     t+1
0   NaN  266.0
1 266.0  145.9
2 145.9  183.1
3 183.1  119.3
4 119.3  180.3
```

Listing 18.2: Example output of creating the lagged dataset.

18.4.2 Step 2: Train and Test Sets

The next step is to separate the dataset into train and test sets. We will keep the first 66% of the observations for training and the remaining 34% for evaluation. During the split, we are careful to exclude the first row of data with the `NaN` value. No training is required in this case; it's just habit. Each of the train and test sets are then split into the input and output variables.

```
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
```

Listing 18.3: Split the lagged dataset.

18.4.3 Step 3: Persistence Algorithm

We can define our persistence model as a function that returns the value provided as input. For example, if the `t` value of 266.0 was provided, then this is returned as the prediction, whereas the actual real or expected value happens to be 145.9 (taken from the first usable row in our lagged dataset).

```
# persistence model
def model_persistence(x):
  return x
```

Listing 18.4: Define persistence prediction.

18.4.4 Step 4: Make and Evaluate Forecast

Now we can evaluate this model on the test dataset. We do this using the walk-forward validation method. No model training or retraining is required, so in essence, we step through the test dataset time step by time step and get predictions. Once predictions are made for each time step in the test dataset, they are compared to the expected values and a Root Mean Squared Error (RMSE) score is calculated.

```
# walk-forward validation
predictions = list()
for x in test_X:
  yhat = model_persistence(x)
  predictions.append(yhat)
rmse = sqrt(mean_squared_error(test_y, predictions))
print('Test RMSE: %.3f' % rmse)
```

Listing 18.5: Walk-forward validation with persistence prediction.

In this case, the RMSE is more than 133 shampoo sales per month over the test dataset.

```
Test RMSE: 133.156
```

Listing 18.6: Example output of walk-forward validation.

18.4.5 Step 5: Complete Example

Finally, a plot is made to show the training dataset and the diverging predictions from the expected values from the test dataset. From the plot of the persistence model predictions, it is clear that the model is one-step behind reality. There is a rising trend and month-to-month noise in the sales figures, which highlights the limitations of the persistence technique.

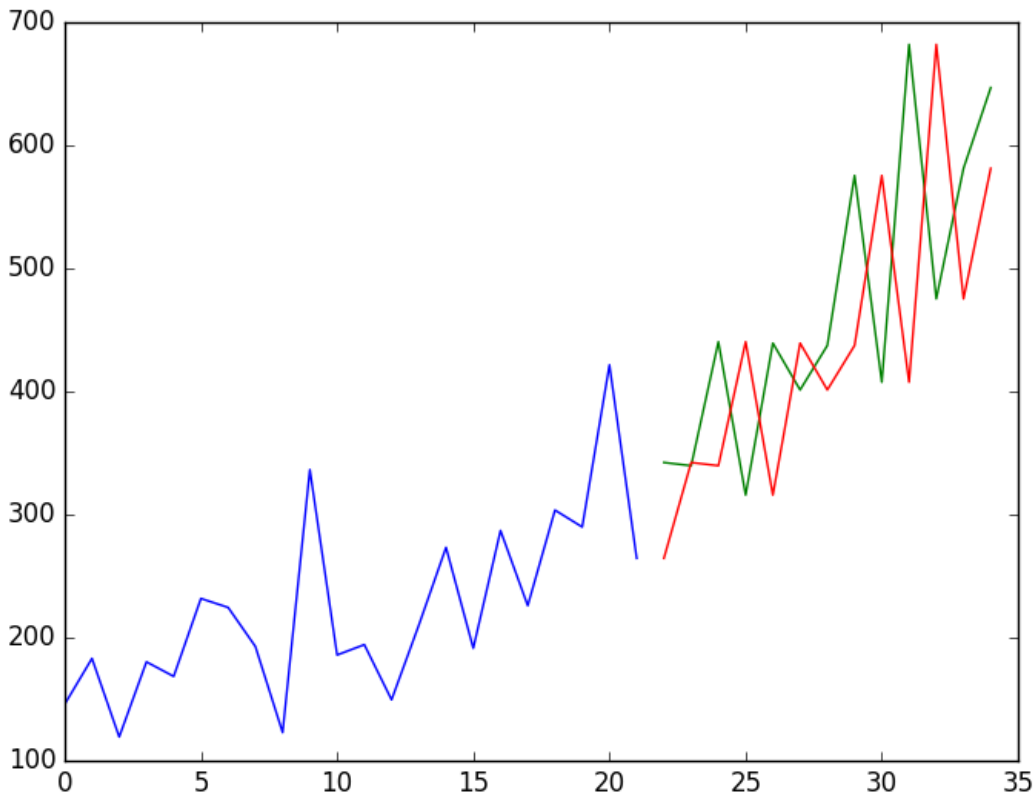

Figure 18.1: Line plot of the persistence forecast for the Shampoo Sales dataset showing the training set (blue), test set (green) and predictions (red).

The complete example is listed below.

```
# evaluate a persistence forecast model
from pandas import read_csv
from pandas import datetime
from pandas import DataFrame
from pandas import concat
from matplotlib import pyplot
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
print(dataframe.head(5))
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
```

```
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model
def model_persistence(x):
  return x
# walk-forward validation
predictions = list()
for x in test_X:
  yhat = model_persistence(x)
  predictions.append(yhat)
rmse = sqrt(mean_squared_error(test_y, predictions))
print('Test RMSE: %.3f' % rmse)
# plot predictions and expected results
pyplot.plot(train_y)
pyplot.plot([None for i in train_y] + [x for x in test_y])
pyplot.plot([None for i in train_y] + [x for x in predictions])
pyplot.show()
```

Listing 18.7: Complete example of a persistence forecast on the Shampoo Sales dataset.

We have seen an example of the persistence model developed from scratch for the Shampoo Sales problem. The persistence algorithm is naive. It is often called the naive forecast. It assumes nothing about the specifics of the time series problem to which it is applied. This is what makes it so easy to understand and so quick to implement and evaluate. As a machine learning practitioner, it can also spark a large number of improvements. Write them down. This is useful because these ideas can become input features in a feature engineering effort or simple models that may be combined in an ensembling effort later.

18.5 Summary

In this tutorial, you discovered how to establish a baseline performance on time series forecast problems with Python. Specifically, you learned:

- The importance of establishing a baseline and the persistence algorithm that you can use.
- How to implement the persistence algorithm in Python from scratch.
- How to evaluate the forecasts of the persistence algorithm and use them as a baseline.

18.5.1 Next

In the next lesson you will discover how to visualize residual errors from forecast models.

Chapter 19

Visualize Residual Forecast Errors

Forecast errors on time series regression problems are called residuals or residual errors. Careful exploration of residual errors on your time series prediction problem can tell you a lot about your forecast model and even suggest improvements. In this tutorial, you will discover how to visualize residual errors from time series forecasts. After completing this tutorial, you will know:

- How to create and review line plots of residual errors over time.
- How to review summary statistics and plots of the distribution of residual plots.
- How to explore the correlation structure of residual errors.

Let's get started.

19.1 Residual Forecast Errors

Forecast errors on a time series forecasting problem are called residual errors or residuals. A residual error is calculated as the expected outcome minus the forecast, for example:

$$residual_error = expected - forecast \tag{19.1}$$

Or, more succinctly and using standard terms as:

$$e = y - yhat \tag{19.2}$$

We often stop there and summarize the skill of a model as a summary of this error. Instead, we can collect these individual residual errors across all forecasts and use them to better understand the forecast model. Generally, when exploring residual errors we are looking for patterns or structure. A sign of a pattern suggests that the errors are not random.

We expect the residual errors to be random, because it means that the model has captured all of the structure and the only error left is the random fluctuations in the time series that cannot be modeled. A sign of a pattern or structure suggests that there is more information that a model could capture and use to make better predictions.

Before we start exploring the different ways to look for patterns in residual errors, we need context. In the next section, we will look at a dataset and a simple forecast method that we will use to generate residual errors to explore in this tutorial.

19.2 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`.

19.3 Persistence Forecast Model

The simplest forecast that we can make is to forecast that what happened in the previous time step will be the same as what will happen in the next time step. This is called the *naive forecast* or the persistence forecast model. We can implement the persistence model in Python.

After the dataset is loaded, it is phrased as a supervised learning problem. A lagged version of the dataset is created where the prior time step (`t`) is used as the input variable and the next time step (`t+1`) is taken as the output variable.

```
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
```

Listing 19.1: Create the lagged dataset.

Next, the dataset is split into training and test sets. A total of 66% of the data is kept for training and the remaining 34% is held for the test set. No training is required for the persistence model; this is just a standard test harness approach. Once split, the train and test sets are separated into their input and output components.

```
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
```

Listing 19.2: Split the lagged dataset.

The persistence model is applied by predicting the output value (`y`) as a copy of the input value (`X`).

```
# persistence model
predictions = [x for x in test_X]
```

Listing 19.3: Make the persistence forecast.

The residual errors are then calculated as the difference between the expected outcome (`test_y`) and the prediction (`predictions`).

```
# calculate residuals
residuals = [test_y[i]-predictions[i] for i in range(len(predictions))]
```

Listing 19.4: Calculate forecast errors for the persistence forecast.

The example puts this all together and gives us a set of residual forecast errors that we can explore in this tutorial.

```
# calculate residuals from a persistence forecast
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model
predictions = [x for x in test_X]
# calculate residuals
residuals = [test_y[i]-predictions[i] for i in range(len(predictions))]
residuals = DataFrame(residuals)
print(residuals.head())
```

Listing 19.5: Complete example of a persistence forecast on the Daily Female Births dataset.

Running the example prints the first 5 rows of the forecast residuals.

```
0   9.0
1 -10.0
2   3.0
3  -6.0
4  30.0
```

Listing 19.6: Example output of the first five forecast errors for the persistence forecast on the Daily Female Births dataset.

19.4 Residual Line Plot

The first plot is to look at the residual forecast errors over time as a line plot. We would expect the plot to be random around the value of 0 and not show any trend or cyclic structure. The array of residual errors can be wrapped in a Pandas `DataFrame` and plotted directly. The code below provides an example.

```
# line plot of residual errors
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
```

```
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model
predictions = [x for x in test_X]
# calculate residuals
residuals = [test_y[i]-predictions[i] for i in range(len(predictions))]
residuals = DataFrame(residuals)
# plot residuals
residuals.plot()
pyplot.show()
```

Listing 19.7: Create a line plot of the forecast residual errors for the Daily Female Births dataset.

Running the example shows a seemingly random plot of the residual time series. If we did see trend, seasonal or cyclic structure, we could go back to our model and attempt to capture those elements directly.

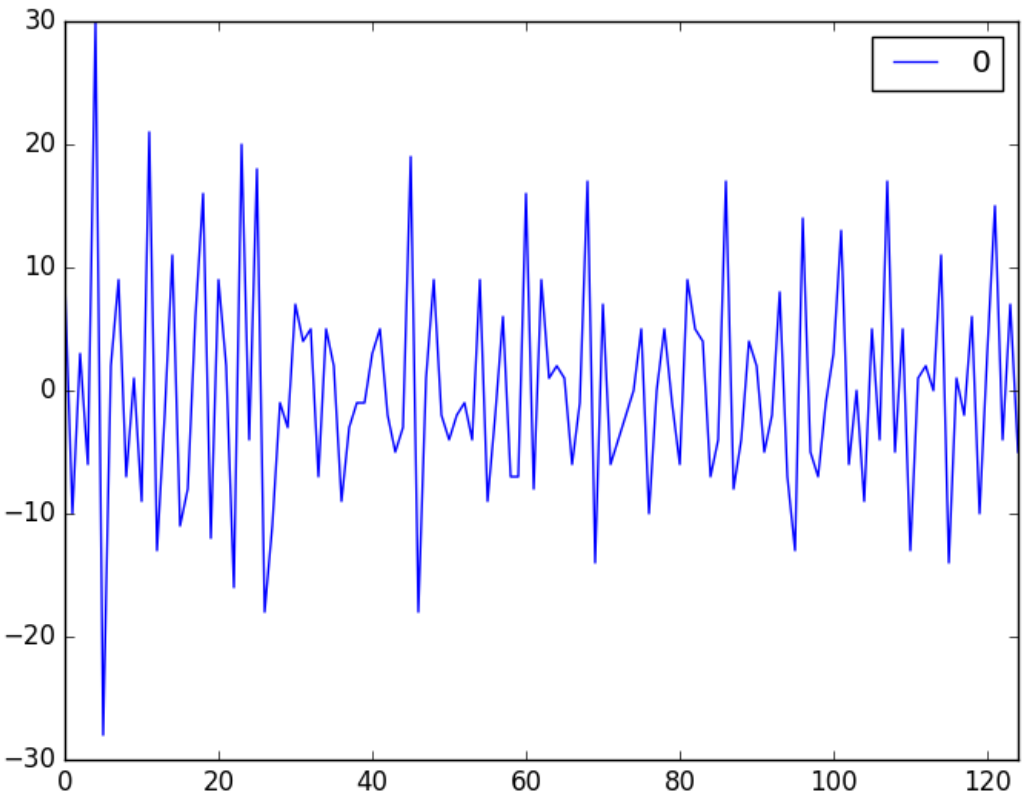

Figure 19.1: Line plot of the forecast residual errors for the Daily Female Births dataset.

Next, we look at summary statistics that we can use to see how the errors are spread around zero.

19.5 Residual Summary Statistics

We can calculate summary statistics on the residual errors. Primarily, we are interested in the mean value of the residual errors. A value close to zero suggests no bias in the forecasts, whereas positive and negative values suggest a positive or negative bias in the forecasts made. It is useful to know about a bias in the forecasts as it can be directly corrected in forecasts prior to their use or evaluation.

Below is an example of calculating summary statistics of the distribution of residual errors. This includes the mean and standard deviation of the distribution, as well as percentiles and the minimum and maximum errors observed.

```
# summary statistics of residual errors
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model
predictions = [x for x in test_X]
# calculate residuals
residuals = [test_y[i]-predictions[i] for i in range(len(predictions))]
residuals = DataFrame(residuals)
# summary statistics
print(residuals.describe())
```

Listing 19.8: Summary statistics of the residual errors for the Daily Female Births dataset.

Running the example shows a mean error value close to zero, but perhaps not close enough. It suggests that there may be some bias and that we may be able to further improve the model by performing a bias correction. This could be done by adding the mean residual error (0.064000) to forecasts. This may work in this case, but it is a naive form of bias correction and there are more sophisticated methods available.

```
count  125.000000
mean     0.064000
std      9.187776
min    -28.000000
25%     -6.000000
50%     -1.000000
75%      5.000000
max     30.000000
```

Listing 19.9: Example output of summary statistics of the residual errors for the Daily Female Births dataset.

Next, we go beyond summary statistics and look at methods to visualize the distribution of the residual errors.

19.6 Residual Histogram and Density Plots

Plots can be used to better understand the distribution of errors beyond summary statistics. We would expect the forecast errors to be normally distributed around a zero mean. Plots can help discover skews in this distribution. We can use both histograms and density plots to better understand the distribution of residual errors. Below is an example of creating one of each plot.

```
# density plots of residual errors
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model
predictions = [x for x in test_X]
# calculate residuals
residuals = [test_y[i]-predictions[i] for i in range(len(predictions))]
residuals = DataFrame(residuals)
# histogram plot
residuals.hist()
pyplot.show()
# density plot
residuals.plot(kind='kde')
pyplot.show()
```

Listing 19.10: Create density plots of the residual errors for the Daily Female Births dataset.

We can see that the distribution does have a Gaussian look, but is perhaps more pointy, showing an exponential distribution with some asymmetry. If the plot showed a distribution that was distinctly non-Gaussian, it would suggest that assumptions made by the modeling process were perhaps incorrect and that a different modeling method may be required. A large skew may suggest the opportunity for performing a transform to the data prior to modeling, such as taking the log or square root.

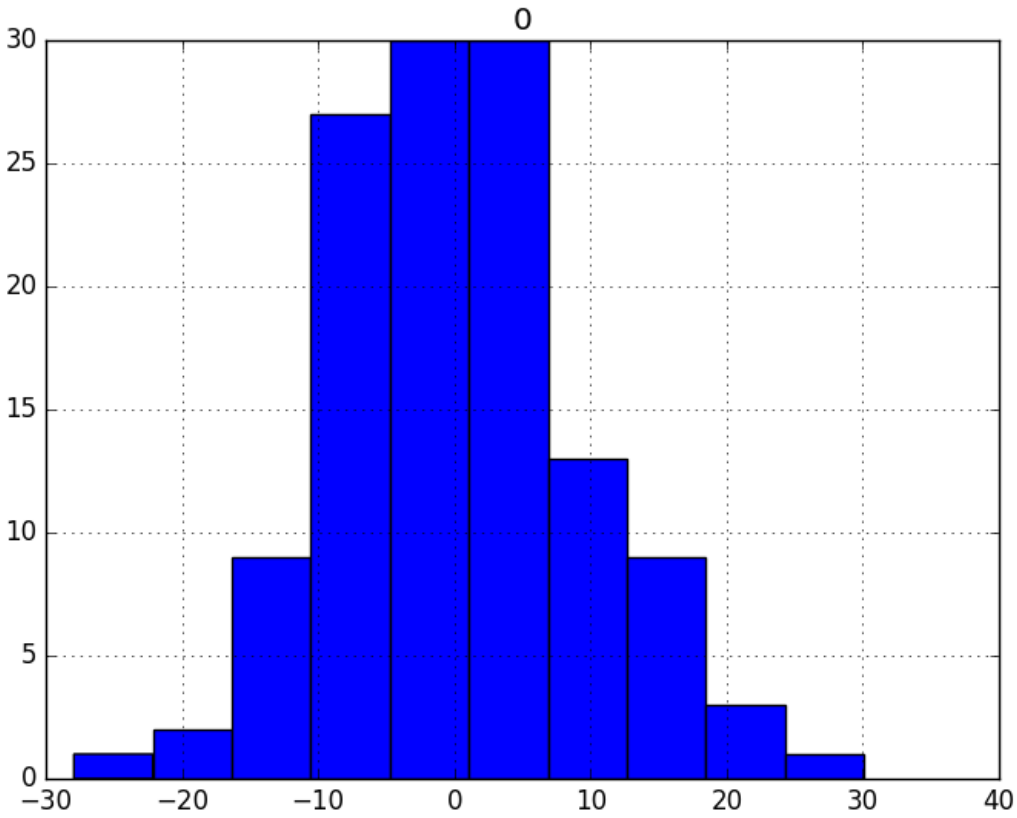

Figure 19.2: Histogram plot of the forecast residual errors for the Daily Female Births dataset.

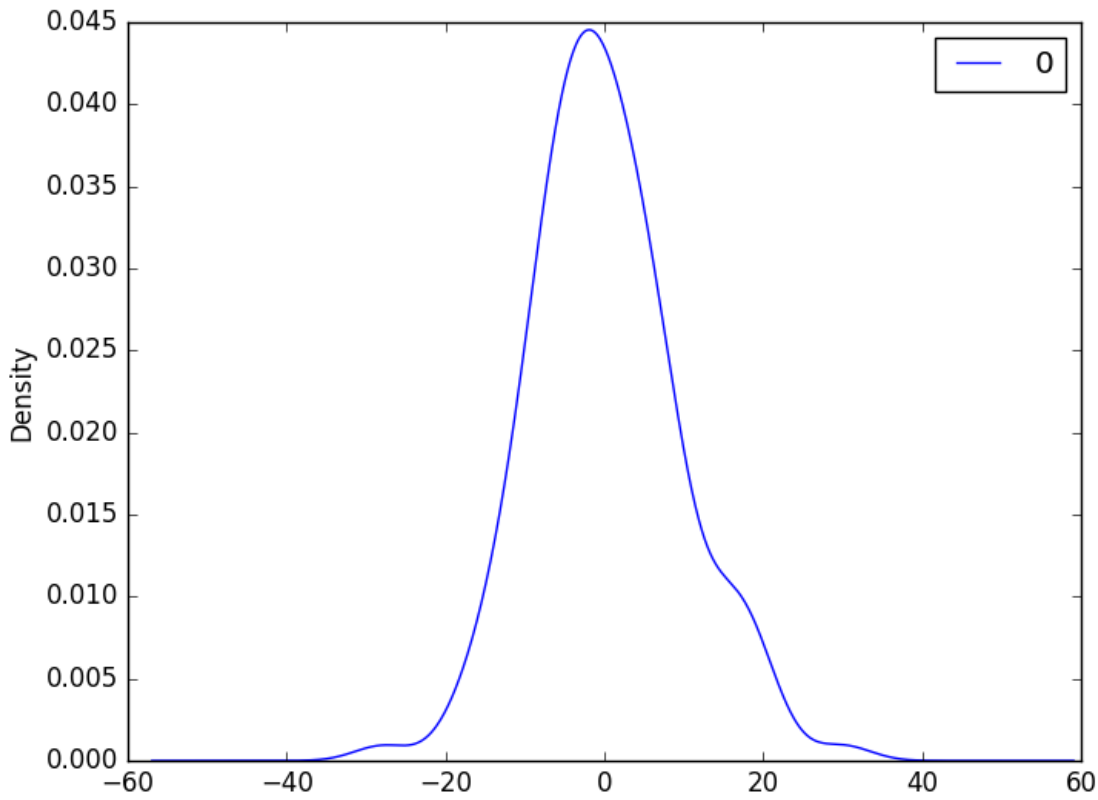

Figure 19.3: Density plot of the forecast residual errors for the Daily Female Births dataset.

Next, we will look at another quick, and perhaps more reliable, way to check if the distribution of residuals is Gaussian.

19.7 Residual Q-Q Plot

A Q-Q plot, or quantile plot, compares two distributions and can be used to see how similar or different they happen to be. We can create a Q-Q plot using the `qqplot()` function[1] in the Statsmodels library.

The Q-Q plot can be used to quickly check the normality of the distribution of residual errors. The values are ordered and compared to an idealized Gaussian distribution. The comparison is shown as a scatter plot (theoretical on the x-axis and observed on the y-axis) where a match between the two distributions is shown as a diagonal line from the bottom left to the top-right of the plot.

The plot is helpful to spot obvious departures from this expectation. Below is an example of a Q-Q plot of the residual errors. The x-axis shows the theoretical quantiles and the y-axis shows the sample quantiles.

[1] http://statsmodels.sourceforge.net/devel/generated/statsmodels.graphics.gofplots.qqplot.html

```
# qq plot of residual errors
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from matplotlib import pyplot
import numpy
from statsmodels.graphics.gofplots import qqplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model
predictions = [x for x in test_X]
# calculate residuals
residuals = [test_y[i]-predictions[i] for i in range(len(predictions))]
residuals = numpy.array(residuals)
qqplot(residuals, line='r')
pyplot.show()
```

Listing 19.11: Create a Q-Q plot of the residual errors for the Daily Female Births dataset.

Running the example shows a Q-Q plot that the distribution is seemingly normal with a few bumps and outliers.

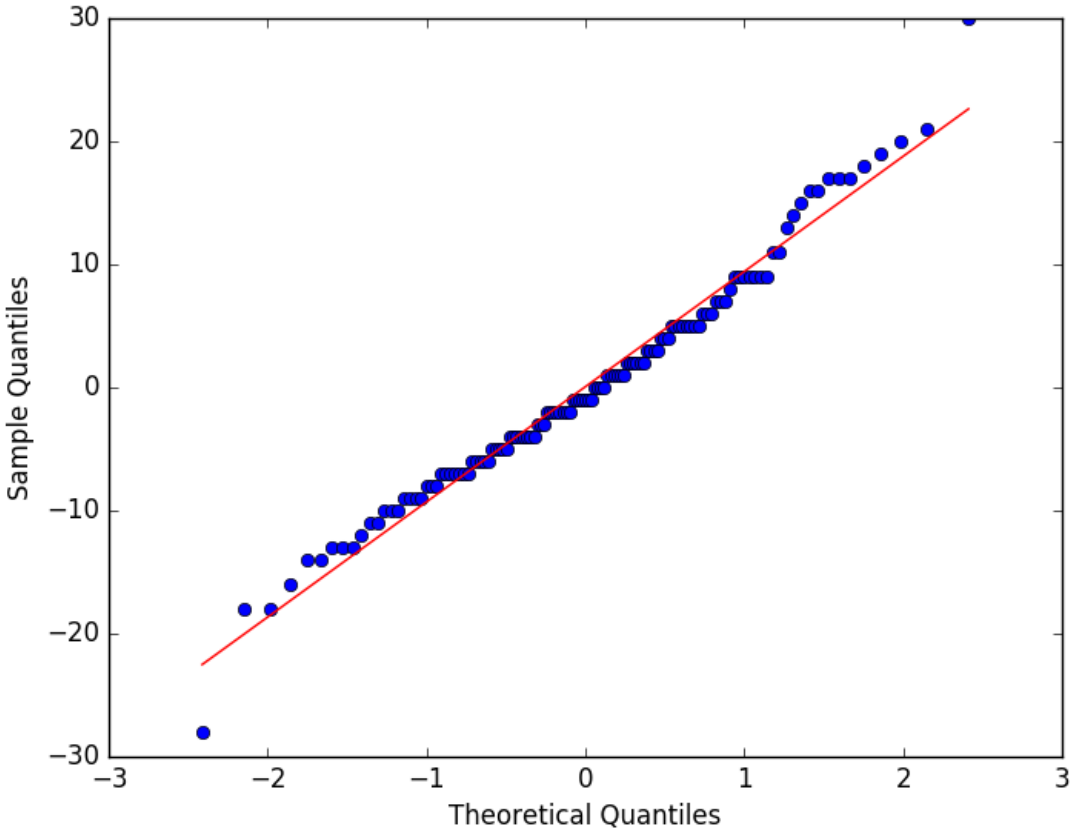

Figure 19.4: Q-Q plot of the forecast residual errors for the Daily Female Births dataset.

Next, we can check for correlations between the errors over time.

19.8 Residual Autocorrelation Plot

Autocorrelation calculates the strength of the relationship between an observation and observations at prior time steps. We can calculate the autocorrelation of the residual error time series and plot the results. This is called an autocorrelation plot. We would not expect there to be any correlation between the residuals. This would be shown by autocorrelation scores being below the threshold of significance (dashed and dotted horizontal lines on the plot).

A significant autocorrelation in the residual plot suggests that the model could be doing a better job of incorporating the relationship between observations and lagged observations, called autoregression. Pandas provides a built-in function for calculating an autocorrelation plot, called `autocorrelation_plot()`.

Below is an example of visualizing the autocorrelation for the residual errors. The x-axis shows the lag and the y-axis shows the correlation between an observation and the lag variable, where correlation values are between -1 and 1 for negative and positive correlations respectively.

```
# autoregression plot of residual errors
from pandas import read_csv
from pandas import DataFrame
```

```
from pandas import concat
from matplotlib import pyplot
from pandas.plotting import autocorrelation_plot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model
predictions = [x for x in test_X]
# calculate residuals
residuals = [test_y[i]-predictions[i] for i in range(len(predictions))]
residuals = DataFrame(residuals)
autocorrelation_plot(residuals)
pyplot.show()
```

Listing 19.12: Create an ACF plot of the residual errors for the Daily Female Births dataset.

Running the example creates an autoregression plot of other residual errors. We do not see an obvious autocorrelation trend across the plot. There may be some positive autocorrelation worthy of further investigation at lag 7 that seems significant.

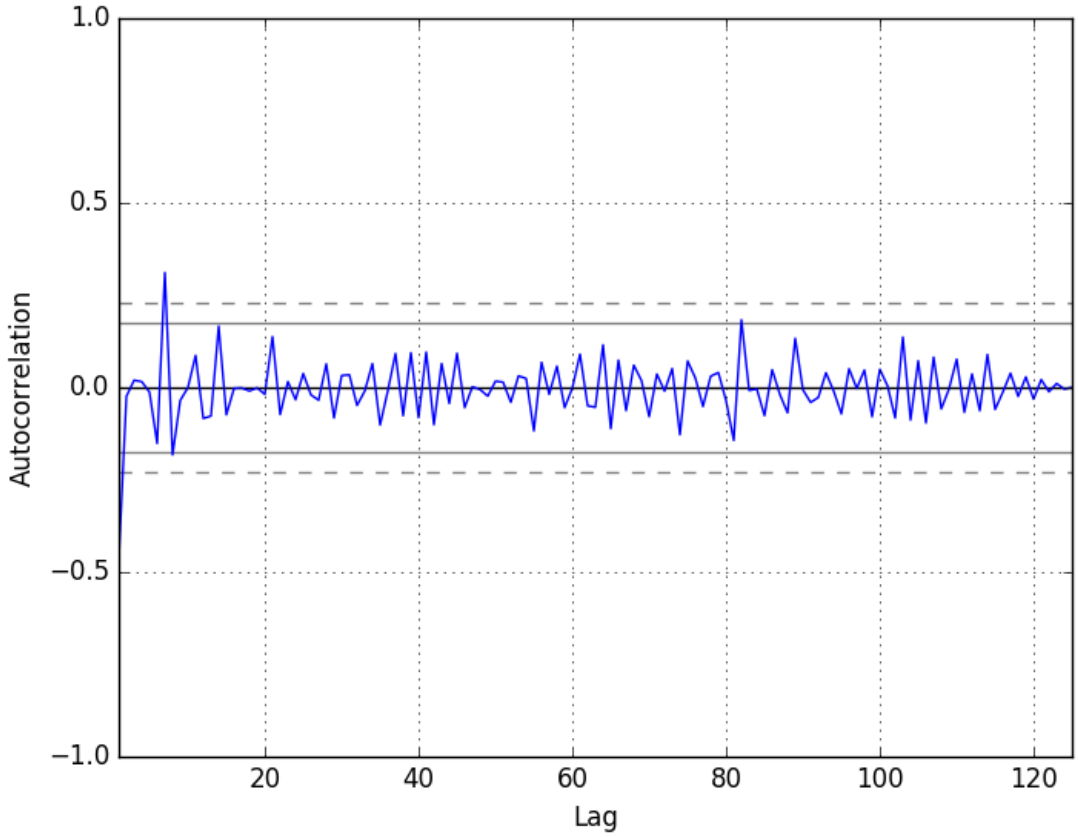

Figure 19.5: ACF plot of the forecast residual errors for the Daily Female Births dataset.

19.9 Summary

In this tutorial, you discovered how to explore the time series of residual forecast errors with Python. Specifically, you learned:

- How to plot the time series of forecast residual errors as a line plot.
- How to explore the distribution of residual errors using statistics, density plots, and Q-Q plots.
- How to check the residual time series for autocorrelation.

19.9.1 Next

In the next lesson you will discover how to reframe a time series forecast problem as classification or regression.

Chapter 20

Reframe Time Series Forecasting Problems

You do not have to model your time series forecast problem as-is. There are many ways to reframe your forecast problem that can both simplify the prediction problem and potentially expose more or different information to be modeled. A reframing can ultimately result in better and/or more robust forecasts. In this tutorial, you will discover how to reframe your time series forecast problem with Python. After completing this tutorial, you will know:

- How to reframe your time series forecast problem as an alternate regression problem.
- How to reframe your time series forecast problem as a classification prediction problem.
- How to reframe your time series forecast problem with an alternate time horizon.

Let's get started.

20.1 Benefits of Reframing Your Problem

Reframing your problem is the idea of exploring alternate perspectives on what is to be predicted. There are two potential benefits to exploring alternate framings of your time series forecast problem:

1. Simplify your problem.
2. Provide the basis for an ensemble forecast.

Both benefits ultimately have the result of leading to more skillful and/or more robust forecasts.

20.1.1 Simplify Your Problem

Perhaps the largest wins on a forecasting project can come from a reframing of the problem. This is because the structure and type of prediction problem has so much more impact than the choice of data transforms, choice of model, or the choice of model hyperparameters. It is the biggest lever in a project and must be carefully considered.

20.1.2 Ensemble Forecast

In addition to changing the problem you are working on, reframing plays another role: it can provide you with a suite of different, but highly related problems that you can model. The benefit of this is that the framings may be different enough to require differences in data preparation and modeling methods.

Models of differing perspectives on the same problem may capture different information from the input and in turn result in predictions that are skillful, but in different ways. These predictions may be combined in an ensemble to result in a more skillful or more robust forecast. In this tutorial, we will explore three different ways you may consider reframing your time series forecast problem. Before we dive in, let's look at a simple univariate time series problem of forecasting the minimum daily temperature to use as context for the discussion.

20.2 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`.

20.3 Naive Time Series Forecast

The naive approach is to predict the problem as-is. For reference, we will call this the naive time series forecast. In this case, the seasonal information can be removed to make the series seasonal stationary. The time series can then be modeled based on some function of the lagged observations. For example:

$$Temp(t+1) = B0 + (B1 \times Temp(t)) + (B2 \times Temp(t-1)) \ldots (Bn \times Temp(t-n)) \quad (20.1)$$

Where `Temp(t+1)` is the next temperature in the series to predict, `B0` to `Bn` are coefficients learned from training data and `Temp(t)` to `Temp(t-n)` are the current and lagged observations. This may be fine or even required by many problems. The risk is that a preconceived idea of how to frame the problem has influenced data collection, and in turn perhaps limited the results.

20.4 Regression Framings

Most time series prediction problems are regression problems, requiring the prediction of a real-valued output. Below are 5 different ways that this prediction problem could be re-phrased as an alternate regression problem:

- Forecast the change in the minimum temperature compared to the previous day.
- Forecast the minimum temperature relative to the average from the past 14 days.
- Forecast the minimum temperature relative to the average the same month last year.

- Forecast the minimum temperature rounded to the nearest 5 degrees Celsius.
- Forecast the average minimum temperature for the next 7 days.

Making the temperature relative is a linear transform and may not make the problem simpler and easier to predict, but it may shake loose new ideas or even new sources of data that you may consider. It also may help you think more clearly about how exactly the forecast will be used and what the hard requirements on that forecasted value actually are.

Transforming the granularity of a prediction problem does change the difficulty of the problem and can be very useful if the requirements of the problem permit such redefinitions. Below is an example of reframing the Minimum Daily Temperatures forecast problem to predict the daily temperature rounded to the nearest 5 degrees.

```
# reframe precision of regression forecast
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
# load data
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# round forecast to nearest 5
for i in range(len(dataframe['t+1'])):
  dataframe['t+1'][i] = int(dataframe['t+1'][i] / 5) * 5.0
print(dataframe.head(5))
```

Listing 20.1: Regression reframing of the Minimum Daily Temperatures dataset.

Running the example prints the first 5 rows of the reframed problem. The problem is defined as given the minimum temperature the day before in degrees Celsius, the minimum to the nearest 5 degrees.

```
    t    t+1
0   NaN  20.0
1  20.7  15.0
2  17.9  15.0
3  18.8  10.0
4  14.6  15.0
```

Listing 20.2: Example output of the regression reframing of the Minimum Daily Temperatures dataset.

20.5 Classification Framings

Classification involves predicting categorical or label outputs (like `hot` and `cold`). Below are 5 different ways that this prediction problem can be rephrased as a classification problem:

- Forecast whether a minimum temperature will be cold, moderate, or warm.
- Forecast whether a change in minimum temperature will be small or large.

- Forecast whether the minimum temperature will be a monthly minimum or not.
- Forecast whether the minimum will be higher or lower than the minimum in the previous year.
- Forecast whether the minimum temperature will rise or fall over the next 7 days.

A move to classification can simplify the prediction problem. This approach opens up ideas on labels as well as binary classification framings.

The native regression representation of the output variable means that most classification framings are likely to keep the ordinal structure (e.g. cold, moderate, hot). Meaning that there is an ordered relationship between the classes being predicted, which may not be the case when predicting labels like `dog` and `cat`.

The ordinal relationship permits both a hard classification problem as well as an integer prediction problem that can be post-hoc rounded into a specific category. Below is an example of transforming the Minimum Daily Temperatures forecast problem to a classification problem where each temperature value is an ordinal value of cold, moderate, or hot. These labels are mapped to integer values, defined as:

- `0 (cold)`: $<$ 10 degrees Celsius.
- `1 (moderate)`: $\geq$ 10 and $<$ 25 degrees Celsius.
- `2 (hot)`: $\geq$ 25 degrees Celsius.

```
# reframe regression as classification
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
# load data
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
# Create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# make discrete
for i in range(len(dataframe['t+1'])):
  value = dataframe['t+1'][i]
  if value < 10.0:
    dataframe['t+1'][i] = 0
  elif value >= 25.0:
    dataframe['t+1'][i] = 2
  else:
    dataframe['t+1'][i] = 1
print(dataframe.head(5))
```

Listing 20.3: Classification reframing of the Minimum Daily Temperatures dataset.

Running the example prints the first 5 rows of the reframed problem. Given the minimum temperature the day before in degrees Celsius, the goal is to predict the temperature as either cold, moderate, or hot (0, 1, 2 respectively).

```
    t    t+1
0   NaN  1.0
1  20.7  1.0
2  17.9  1.0
3  18.8  1.0
4  14.6  1.0
```

Listing 20.4: Example output of the classification reframing of the Minimum Daily Temperatures dataset.

20.6 Time Horizon Framings

Another axis that can be varied is the time horizon. The time horizon is the number of time steps in the future that are being predicted. Below are 5 different ways that this prediction problem can be re-phrased as a different time horizon:

- Forecast the minimum temperature for the next 7 days.
- Forecast the minimum temperature in 30 days time.
- Forecast the average minimum temperature next month.
- Forecast the day in the next week that will have the lowest minimum temperature.
- Forecast one year of minimum temperature values.

It is easy to get caught up in the idea that you require one-step forecasts. Focusing on reframings of the problem around time horizon forces you to think about point versus multi-step forecasts and how far in the future to consider.

You may be able to forecast far into the future, but the skill is likely going to vary, degrading further into the future you project. When thinking through the horizon of the forecast, also consider the minimum acceptable performance of forecasts. The example below transforms the Minimum Daily Temperatures forecast problem to predict the minimum temperature for the next 7 days.

```
# reframe time horizon of forecast
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
# load data
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values, values.shift(-1),
  values.shift(-2), values.shift(-3), values.shift(-4), values.shift(-5),
  values.shift(-6)], axis=1)
dataframe.columns = ['t', 't+1', 't+2', 't+3', 't+4', 't+5', 't+6', 't+7']
print(dataframe.head(14))
```

Listing 20.5: Time Horizon reframing of the Minimum Daily Temperatures dataset.

Running the example prints the first 14 records of the transformed dataset. The problem is defined as: given the minimum daily temperature from the day before in degrees Celsius, forecast the minimum daily temperature for the next 7 days.

```
     t    t+1  t+2  t+3  t+4  t+5  t+6  t+7
0   NaN  20.7  17.9  18.8  14.6  15.8  15.8  15.8
1   20.7  17.9  18.8  14.6  15.8  15.8  15.8  17.4
2   17.9  18.8  14.6  15.8  15.8  15.8  17.4  21.8
3   18.8  14.6  15.8  15.8  15.8  17.4  21.8  20.0
4   14.6  15.8  15.8  15.8  17.4  21.8  20.0  16.2
5   15.8  15.8  15.8  17.4  21.8  20.0  16.2  13.3
6   15.8  15.8  17.4  21.8  20.0  16.2  13.3  16.7
7   15.8  17.4  21.8  20.0  16.2  13.3  16.7  21.5
8   17.4  21.8  20.0  16.2  13.3  16.7  21.5  25.0
9   21.8  20.0  16.2  13.3  16.7  21.5  25.0  20.7
10  20.0  16.2  13.3  16.7  21.5  25.0  20.7  20.6
11  16.2  13.3  16.7  21.5  25.0  20.7  20.6  24.8
12  13.3  16.7  21.5  25.0  20.7  20.6  24.8  17.7
13  16.7  21.5  25.0  20.7  20.6  24.8  17.7  15.5
```

Listing 20.6: Example output of the time horizon reframing of the Minimum Daily Temperatures dataset.

20.7 Summary

In this tutorial, you discovered how to reframe your time series forecasting problem with Python. Specifically, you learned:

- How to devise alternate regression representations of your time series problem.
- How to frame your prediction problem as a classification problem.
- How to devise alternate time horizons for your prediction problem.

20.7.1 Next

This concludes Part IV. Next in Part V you will discover how to develop forecast models for univariate time series problems starting with the Box-Jenkins method.

Part V

Forecast Models

Chapter 21

A Gentle Introduction to the Box-Jenkins Method

The Autoregressive Integrated Moving Average Model, or ARIMA for short is a standard statistical model for time series forecast and analysis. Along with its development, the authors Box and Jenkins also suggest a process for identifying, estimating, and checking models for a specific time series dataset. This process is now referred to as the Box-Jenkins Method. In this lesson, you will discover the Box-Jenkins Method and tips for using it on your time series forecasting problem. Specifically, you will learn:

- About the ARIMA process and how the 3 steps of the Box-Jenkins Method.
- Best practice heuristics for selecting the `q`, `d`, and `p` model configuration for an ARIMA model.
- Evaluating models by looking for overfitting and residual errors as a diagnostic process.

Let's get started.

21.1 Autoregressive Integrated Moving Average Model

An ARIMA model is a class of statistical model for analyzing and forecasting time series data. ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration. This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- **AR**: *Autoregression*. A model that uses the dependent relationship between an observation and some number of lagged observations.
- **I**: *Integrated*. The use of differencing of raw observations (i.e. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- **MA**: *Moving Average*. A model that uses the dependency between an observation and residual errors from a moving average model applied to lagged observations.

Each of these components are explicitly specified in the model as a parameter. A standard notation is used of `ARIMA(p,d,q)` where the parameters are substituted with integer values to quickly indicate the specific ARIMA model being used. The parameters of the ARIMA model are defined as follows:

- `p`: The number of lag observations included in the model, also called the lag order.
- `d`: The number of times that the raw observations are differenced, also called the degree of differencing.
- `q`: The size of the moving average window, also called the order of moving average.

21.2 Box-Jenkins Method

The Box-Jenkins method was proposed by George Box and Gwilym Jenkins in their seminal 1970 textbook Time Series Analysis: Forecasting and Control[1]. The approach starts with the assumption that the process that generated the time series can be approximated using an ARMA model if it is stationary or an ARIMA model if it is non-stationary. The 2016 5th edition of the textbook (Part Two, page 177) refers to the process as a stochastic model building and that it is an iterative approach that consists of the following 3 steps:

1. **Identification**. Use the data and all related information to help select a sub-class of model that may best summarize the data.
2. **Estimation**. Use the data to train the parameters of the model (i.e. the coefficients).
3. **Diagnostic Checking**. Evaluate the fitted model in the context of the available data and check for areas where the model may be improved.

It is an iterative process, so that as new information is gained during diagnostics, you can circle back to step 1 and incorporate that into new model classes. Let's take a look at these steps in more detail.

21.3 Identification

The identification step is further broken down into: Assess whether the time series is stationary, and if not, how many differences are required to make it stationary. Identify the parameters of an ARMA model for the data.

21.3.1 Differencing

Below are some tips during identification.

- **Unit Root Tests**. Use unit root statistical tests on the time series to determine whether or not it is stationary. Repeat after each round of differencing.
- **Avoid over differencing**. Differencing the time series more than is required can result in the addition of extra serial correlation and additional complexity.

[1] http://www.amazon.com/dp/1118675029?tag=inspiredalgor-20

21.3.2 Configuring AR and MA

Two diagnostic plots can be used to help choose the `p` and `q` parameters of the ARMA or ARIMA. They are:

- **Autocorrelation Function (ACF)**. The plot summarizes the correlation of an observation with lag values. The x-axis shows the lag and the y-axis shows the correlation coefficient between -1 and 1 for negative and positive correlation.
- **Partial Autocorrelation Function (PACF)**. The plot summarizes the correlations for an observation with lag values that is not accounted for by prior lagged observations.

Both plots are drawn as bar charts showing the 95% and 99% confidence intervals as horizontal lines. Bars that cross these confidence intervals are therefore more significant and worth noting. Some useful patterns you may observe on these plots are:

- The model is AR if the ACF trails off after a lag and has a hard cut-off in the PACF after a lag. This lag is taken as the value for `p`.
- The model is MA if the PACF trails off after a lag and has a hard cut-off in the ACF after the lag. This lag value is taken as the value for `q`.
- The model is a mix of AR and MA if both the ACF and PACF trail off.

21.4 Estimation

Estimation involves using numerical methods to minimize a loss or error term. We will not go into the details of estimating model parameters as these details are handled by the chosen library or tool. I would recommend referring to a textbook for a deeper understanding of the optimization problem to be solved by ARMA and ARIMA models and optimization methods like Limited-memory BFGS used to solve it.

21.5 Diagnostic Checking

The idea of diagnostic checking is to look for evidence that the model is not a good fit for the data. Two useful areas to investigate diagnostics are:

1. Overfitting.
2. Residual Errors.

21.5.1 Overfitting

The first check is to check whether the model overfits the data. Generally, this means that the model is more complex than it needs to be and captures random noise in the training data. This is a problem for time series forecasting because it negatively impacts the ability of the model to generalize, resulting in poor forecast performance on out of sample data. Careful attention must be paid to both in-sample and out-of-sample performance and this requires the careful design of a robust test harness for evaluating models.

21.5.2 Residual Errors

Forecast residuals provide a great opportunity for diagnostics. A review of the distribution of errors can help tease out bias in the model. The errors from an ideal model would resemble white noise, that is a Gaussian distribution with a mean of zero and a symmetrical variance. For this, you may use density plots, histograms, and Q-Q plots that compare the distribution of errors to the expected distribution. A non-Gaussian distribution may suggest an opportunity for data pre-processing. A skew in the distribution or a non-zero mean may suggest a bias in forecasts that may be correct.

Additionally, an ideal model would leave no temporal structure in the time series of forecast residuals. These can be checked by creating ACF and PACF plots of the residual error time series. The presence of serial correlation in the residual errors suggests further opportunity for using this information in the model.

21.6 Summary

In this lesson, you discovered the Box-Jenkins Method for time series analysis and forecasting. Specifically, you learned:

- About the ARIMA model and the 3 steps of the general Box-Jenkins Method.
- How to use ACF and PACF plots to choose the `p` and `q` parameters for an ARIMA model.
- How to use overfitting and residual errors to diagnose a fit ARIMA model.

21.6.1 Next

In the next lesson you will discover how to develop an autoregressive model.

Chapter 22

Autoregression Models for Forecasting

Autoregression is a time series model that uses observations from previous time steps as input to a regression equation to predict the value at the next time step. It is a very simple idea that can result in accurate forecasts on a range of time series problems. In this tutorial, you will discover how to implement an autoregressive model for time series forecasting with Python. After completing this tutorial, you will know:

- How to explore your time series data for autocorrelation.
- How to develop an autocorrelation model and use it to make predictions.
- How to use a developed autocorrelation model to make rolling predictions.

Let's get started.

22.1 Autoregression

A regression model, such as linear regression, models an output value based on a linear combination of input values. For example:

$$yhat = b0 + (b1 \times X1) \tag{22.1}$$

Where `yhat` is the prediction, `b0` and `b1` are coefficients found by optimizing the model on training data, and `X` is an input value. This technique can be used on time series where input variables are taken as observations at previous time steps, called lag variables. For example, we can predict the value for the next time step (`t+1`) given the observations at the current (`t` and previous `t-1`). As a regression model, this would look as follows:

$$X(t+1) = b0 + (b1 \times X(t)) + (b2 \times X(t-1)) \tag{22.2}$$

Because the regression model uses data from the same input variable at previous time steps, it is referred to as an autoregression (regression of self).

22.2 Autocorrelation

An autoregression model makes an assumption that the observations at current and previous time steps are useful to predict the value at the next time step. This relationship between variables is called correlation. If both variables change in the same direction (e.g. go up together or down together), this is called a positive correlation. If the variables move in opposite directions as values change (e.g. one goes up and one goes down), then this is called negative correlation.

We can use statistical measures to calculate the correlation between the output variable and values at previous time steps at various different lags. The stronger the correlation between the output variable and a specific lagged variable, the more weight that autoregression model can put on that variable when modeling. Again, because the correlation is calculated between the variable and itself at previous time steps, it is called an autocorrelation. It is also called serial correlation because of the sequenced structure of time series data.

The correlation statistics can also help to choose which lag variables will be useful in a model and which will not. Interestingly, if all lag variables show low or no correlation with the output variable, then it suggests that the time series problem may not be predictable. This can be very useful when getting started on a new dataset.

In this tutorial, we will investigate the autocorrelation of a univariate time series then develop an autoregression model and use it to make predictions. Before we do that, let's first review the Minimum Daily Temperatures data that will be used in the examples.

22.3 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`.

22.4 Quick Check for Autocorrelation

There is a quick, visual check that we can do to see if there is an autocorrelation in our time series dataset. We can plot the observation at the current time step (`t`) with the observation at the previous time step (`t-1`) as a scatter plot. This could be done manually by first creating a lag version of the time series dataset and using a built-in scatter plot function in the Pandas library. But there is an easier way.

Pandas provides a built-in plot to do exactly this, called the `lag_plot()` function[1]. Below is an example of creating a lag plot of the Minimum Daily Temperatures dataset.

```
# lag plot of time series
from pandas import read_csv
from matplotlib import pyplot
from pandas.plotting import lag_plot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
lag_plot(series)
```

[1] http://pandas.pydata.org/pandas-docs/stable/visualization.html#lag-plot

```
pyplot.show()
```

Listing 22.1: Create a lag plot of the Minimum Daily Temperatures dataset.

Running the example plots the temperature data (`t`) on the x-axis against the temperature on the previous day (`t-1`) on the y-axis.

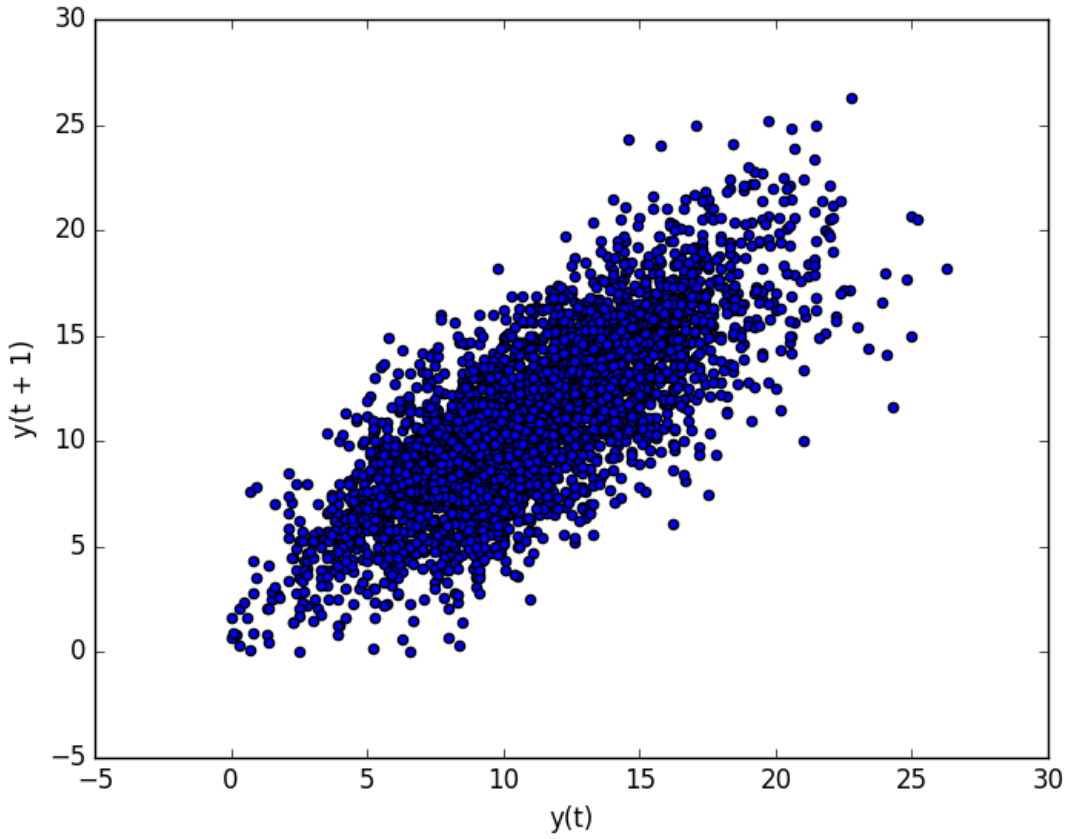

Figure 22.1: Lag plot of the Minimum Daily Temperatures dataset.

We can see a large ball of observations along a diagonal line of the plot. It clearly shows a relationship or some correlation. This process could be repeated for any other lagged observation, such as if we wanted to review the relationship with the last 7 days or with the same day last month or last year. Another quick check that we can do is to directly calculate the correlation between the observation and the lag variable.

We can use a statistical test like the Pearson correlation coefficient. This produces a number to summarize how correlated two variables are between -1 (negatively correlated) and +1 (positively correlated) with small values close to zero indicating low correlation and high values above 0.5 or below -0.5 showing high correlation.

Correlation can be calculated easily using the `corr()` function[2] on the `DataFrame` of the lagged dataset. The example below creates a lagged version of the Minimum Daily Temperatures dataset and calculates a correlation matrix of each column with other columns, including itself.

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

```
# correlation of lag=1
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
result = dataframe.corr()
print(result)
```

Listing 22.2: Calculate correlation for `lag=1` of the Minimum Daily Temperatures dataset.

This is a good confirmation for the plot above. It shows a strong positive correlation (0.77) between the observation and the `lag=1` value.

```
        t       t+1
t    1.00000 0.77487
t+1  0.77487 1.00000
```

Listing 22.3: Example output of the correlation for `lag=1` of the Minimum Daily Temperatures dataset.

This is good for one-off checks, but tedious if we want to check a large number of lag variables in our time series. Next, we will look at a scaled-up version of this approach.

22.5 Autocorrelation Plots

We can plot the correlation coefficient for each lag variable. This can very quickly give an idea of which lag variables may be good candidates for use in a predictive model and how the relationship between the observation and its historic values changes over time. We could manually calculate the correlation values for each lag variable and plot the result. Thankfully, Pandas provides a built-in plot called the `autocorrelation_plot()` function[3].

The plot provides the lag number along the x-axis and the correlation coefficient value between -1 and 1 on the y-axis. The plot also includes solid and dashed lines that indicate the 95% and 99% confidence interval for the correlation values. Correlation values above these lines are more significant than those below the line, providing a threshold or cutoff for selecting more relevant lag values.

```
# autocorrelation plot of time series
from pandas import read_csv
from matplotlib import pyplot
from pandas.plotting import autocorrelation_plot
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
autocorrelation_plot(series)
pyplot.show()
```

Listing 22.4: Create an autocorrelation plot of the Minimum Daily Temperatures dataset with pandas.

[3] http://pandas.pydata.org/pandas-docs/stable/visualization.html#autocorrelation-plot

Running the example shows the swing in positive and negative correlation as the temperature values change across summer and winter seasons each previous year.

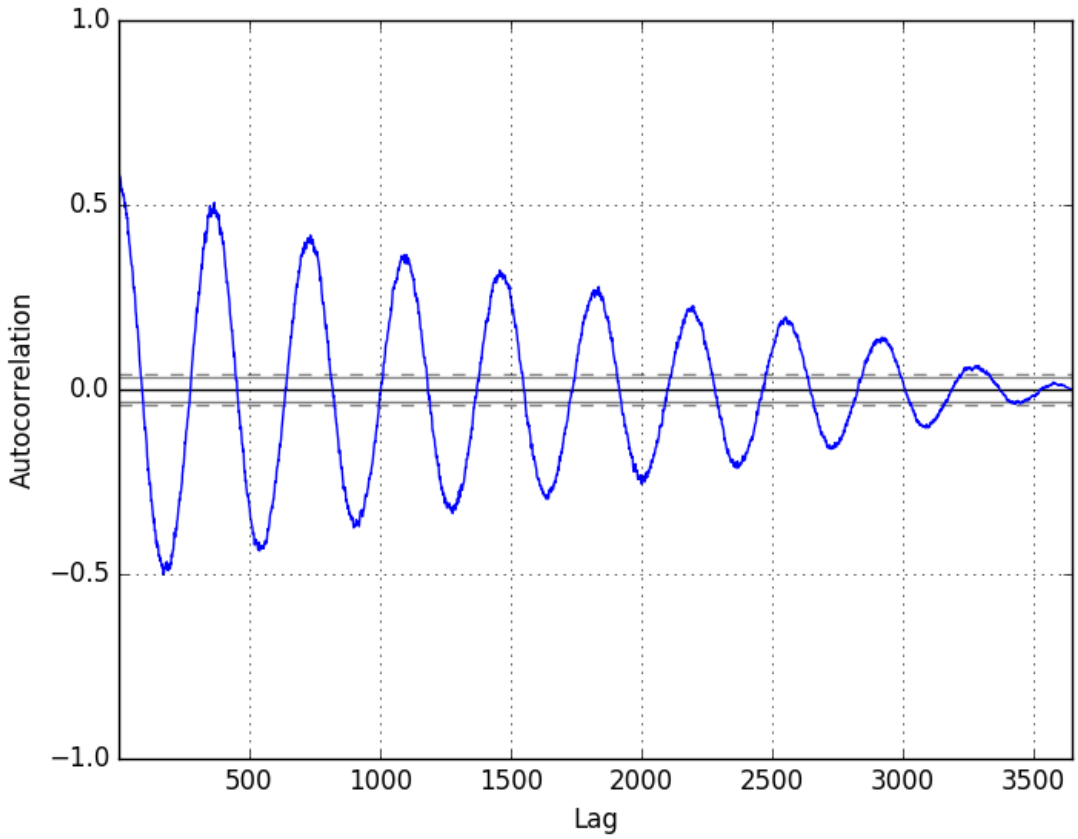

Figure 22.2: Autocorrelation plot of the Minimum Daily Temperatures dataset with pandas.

The Statsmodels library also provides a version of the plot in the `plot_acf()` function[4] as a line plot.

```
# autocorrelation plot of time series
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.graphics.tsaplots import plot_acf
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
plot_acf(series, lags=31)
pyplot.show()
```

Listing 22.5: Create an autocorrelation plot of the Minimum Daily Temperatures dataset with Statsmodels.

In this example, we limit the lag variables evaluated to 31 for readability.

[4] http://statsmodels.sourceforge.net/devel/generated/statsmodels.graphics.tsaplots.plot_acf.html

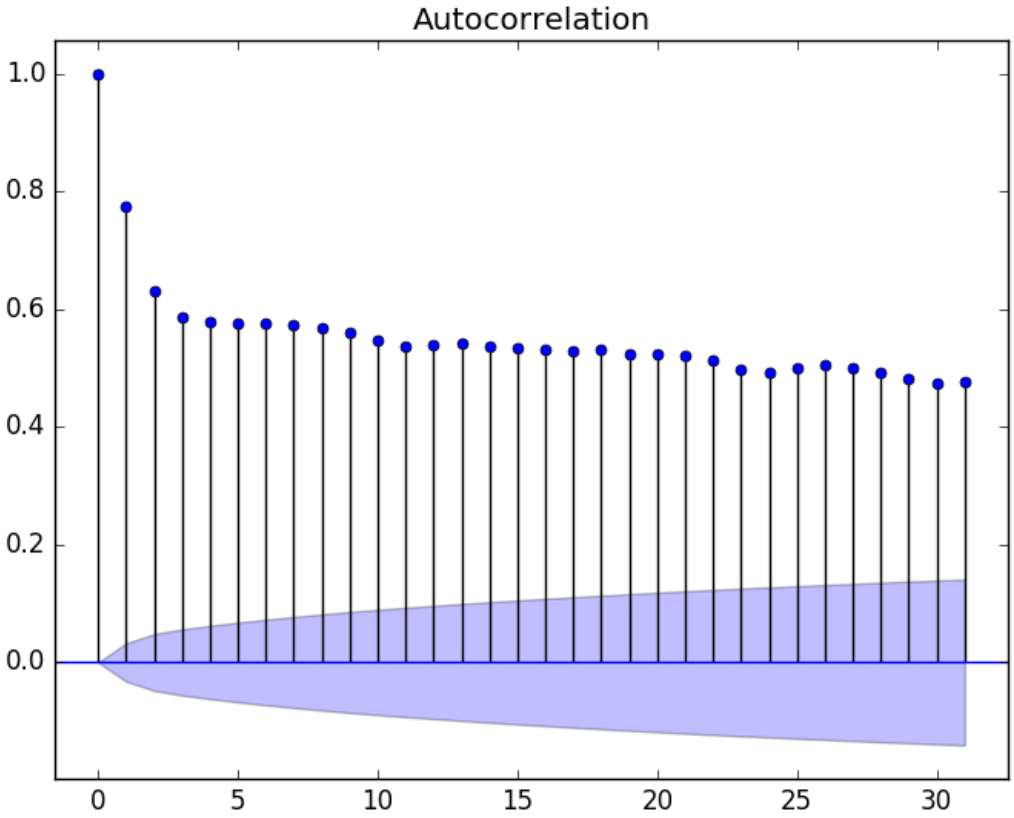

Figure 22.3: Autocorrelation plot of the Minimum Daily Temperatures dataset with Statsmodels.

Now that we know how to review the autocorrelation in our time series, let's look at modeling it with an autoregression. Before we do that, let's establish a baseline performance.

22.6 Persistence Model

Let's say that we want to develop a model to predict the last 7 days of minimum temperatures in the dataset given all prior observations. The simplest model that we could use to make predictions would be to persist the last observation. We can call this a persistence model and it provides a baseline of performance for the problem that we can use for comparison with an autoregression model.

We can develop a test harness for the problem by splitting the observations into training and test sets, with only the last 7 observations in the dataset assigned to the test set as *unseen* data that we wish to predict. The predictions are made using a walk-forward validation model so that we can persist the most recent observations for the next day. This means that we are not making a 7-day forecast, but 7 1-day forecasts.

```
# evaluate a persistence model
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
```

```
from matplotlib import pyplot
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train, test = X[1:len(X)-7], X[len(X)-7:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model
def model_persistence(x):
  return x
# walk-forward validation
predictions = list()
for x in test_X:
  yhat = model_persistence(x)
  predictions.append(yhat)
rmse = sqrt(mean_squared_error(test_y, predictions))
print('Test RMSE: %.3f' % rmse)
# plot predictions vs expected
pyplot.plot(test_y)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 22.6: Persistence model of the Minimum Daily Temperatures dataset.

Running the example prints the root mean squared error (RMSE). The value provides a baseline performance for the problem, in this case 1.850 degrees.

```
Test RMSE: 1.850
```

Listing 22.7: Example output of the persistence model on the Minimum Daily Temperatures dataset.

The expected values for the next 7 days are plotted compared to the predictions from the model.

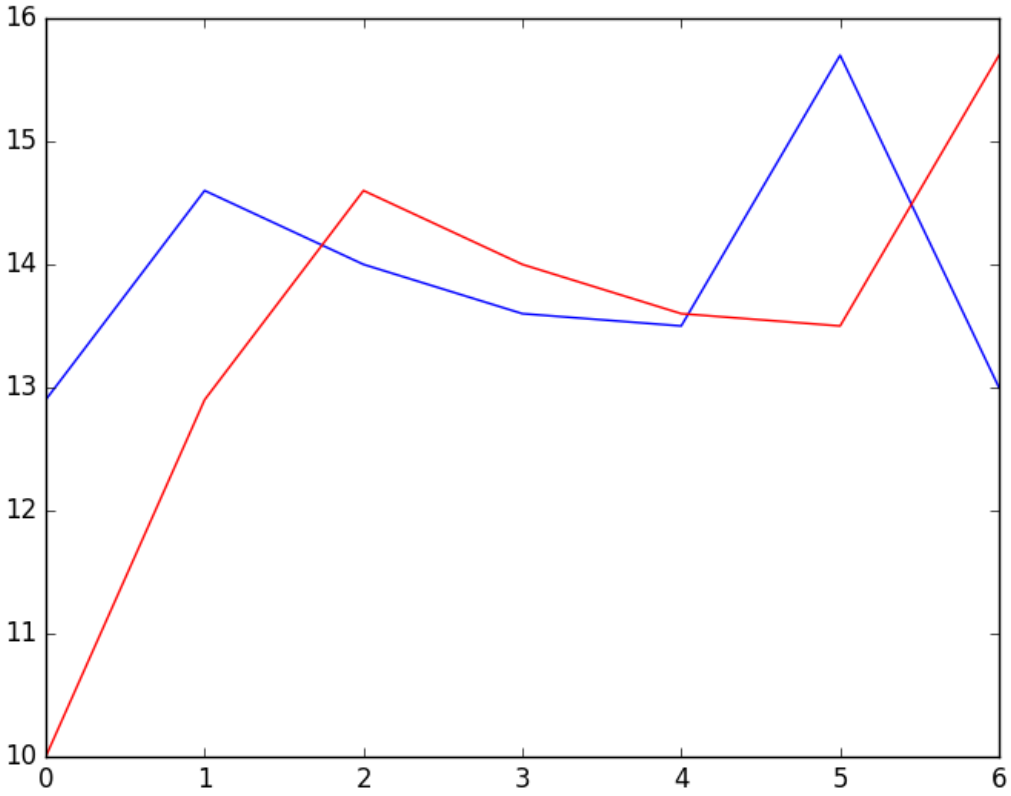

Figure 22.4: Line plot of the persistence forecast (red) on the Minimum Daily Temperatures dataset (blue).

22.7 Autoregression Model

An autoregression model is a linear regression model that uses lagged variables as input variables. We could calculate the linear regression model manually using the LinearRegession class in scikit-learn and manually specify the lag input variables to use. Alternately, the Statsmodels library provides an autoregression model that automatically selects an appropriate lag value using statistical tests and trains a linear regression model. It is provided in the `AR` class[5].

We can use this model by first creating the model `AR()` and then calling `fit()` to train it on our dataset. This returns an `ARResult` object[6]. Once fit, we can use the model to make a prediction by calling the `predict()` function for a number of observations in the future. This creates 1 7-day forecast, which is different from the persistence example above. The complete example is listed below.

```
# create and evaluate a static autoregressive model
from pandas import read_csv
from matplotlib import pyplot
```

[5] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.ar_model.AR.html

[6] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.ar_model.ARResults.html

```
from statsmodels.tsa.ar_model import AR
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
# split dataset
X = series.values
train, test = X[1:len(X)-7], X[len(X)-7:]
# train autoregression
model = AR(train)
model_fit = model.fit()
print('Lag: %s' % model_fit.k_ar)
print('Coefficients: %s' % model_fit.params)
# make predictions
predictions = model_fit.predict(start=len(train), end=len(train)+len(test)-1, dynamic=False)
for i in range(len(predictions)):
  print('predicted=%f, expected=%f' % (predictions[i], test[i]))
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
# plot results
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 22.8: Autoregression model of the Minimum Daily Temperatures dataset.

Running the example first prints the chosen optimal lag and the list of coefficients in the trained linear regression model. We can see that a 29-lag model was chosen and trained. This is interesting given how close this lag is to the average number of days in a month. The 7 day forecast is then printed and the root mean squared error of the forecast is summarized.

```
Lag: 29
Coefficients: [ 5.57543506e-01 5.88595221e-01 -9.08257090e-02 4.82615092e-02
   4.00650265e-02 3.93020055e-02 2.59463738e-02 4.46675960e-02
   1.27681498e-02 3.74362239e-02 -8.11700276e-04 4.79081949e-03
   1.84731397e-02 2.68908418e-02 5.75906178e-04 2.48096415e-02
   7.40316579e-03 9.91622149e-03 3.41599123e-02 -9.11961877e-03
   2.42127561e-02 1.87870751e-02 1.21841870e-02 -1.85534575e-02
  -1.77162867e-03 1.67319894e-02 1.97615668e-02 9.83245087e-03
   6.22710723e-03 -1.37732255e-03]
predicted=11.871275, expected=12.900000
predicted=13.053794, expected=14.600000
predicted=13.532591, expected=14.000000
predicted=13.243126, expected=13.600000
predicted=13.091438, expected=13.500000
predicted=13.146989, expected=15.700000
predicted=13.176153, expected=13.000000
Test RMSE: 1.225
```

Listing 22.9: Example output of the Autoregression model on the Minimum Daily Temperatures dataset.

A plot of the expected vs the predicted values is made. The forecast does look pretty good (about 1 degree Celsius out each day), with big deviation on day 5.

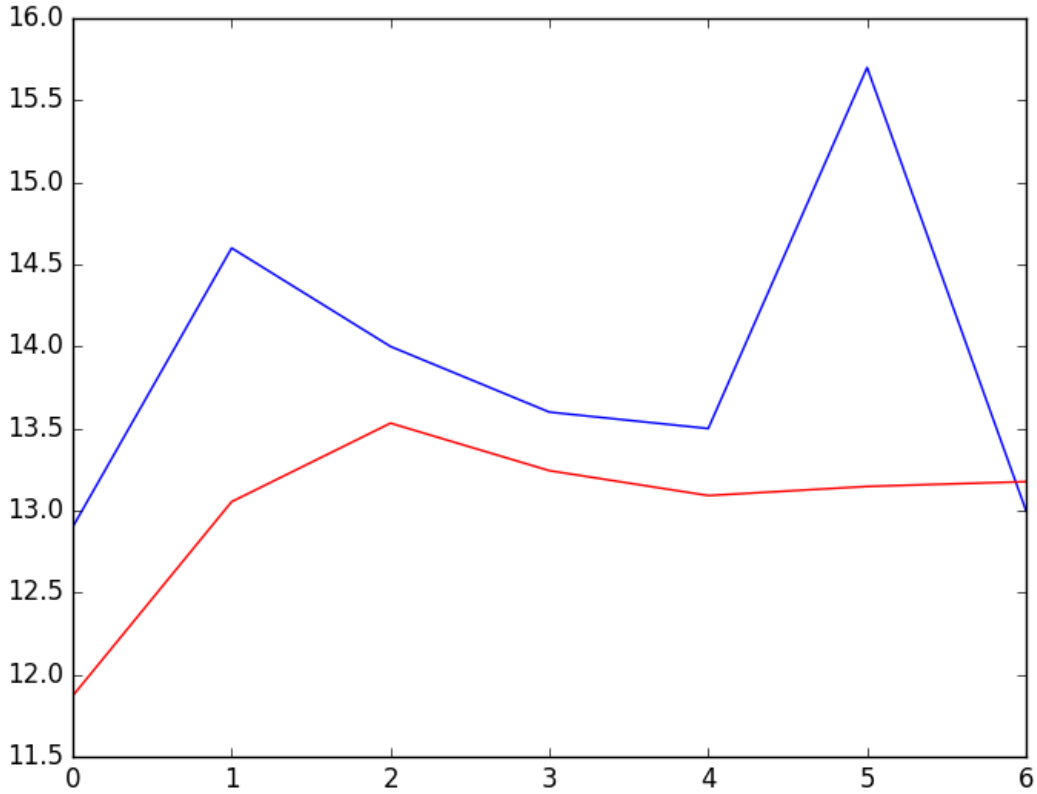

Figure 22.5: Line plot of the AR model forecast (red) on the Minimum Daily Temperatures dataset (blue).

The Statsmodels API does not make it easy to update the model as new observations become available. One way would be to re-train the AR model each day as new observations become available, and that may be a valid approach, if not computationally expensive. An alternative would be to use the learned coefficients and manually make predictions. This requires that the history of 29 prior observations be kept and that the coefficients be retrieved from the model and used in the regression equation to come up with new forecasts.

The coefficients are provided in an array with the intercept term followed by the coefficients for each lag variable starting at `t` to `t-n`. We simply need to use them in the right order on the history of observations, as follows:

$$yhat = b0 + (b1 \times X1) + (b2 \times X2) \ldots (bn \times Xn) \tag{22.3}$$

Below is the complete example.

```
# create and evaluate an updated autoregressive model
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.tsa.ar_model import AR
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
```

```
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
# split dataset
X = series.values
train, test = X[1:len(X)-7], X[len(X)-7:]
# train autoregression
model = AR(train)
model_fit = model.fit()
window = model_fit.k_ar
coef = model_fit.params
# walk forward over time steps in test
history = train[len(train)-window:]
history = [history[i] for i in range(len(history))]
predictions = list()
for t in range(len(test)):
  length = len(history)
  lag = [history[i] for i in range(length-window,length)]
  yhat = coef[0]
  for d in range(window):
    yhat += coef[d+1] * lag[window-d-1]
  obs = test[t]
  predictions.append(yhat)
  history.append(obs)
  print('predicted=%f, expected=%f' % (yhat, obs))
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
# plot
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 22.10: Manual predictions with AR model of the Minimum Daily Temperatures dataset.

Again, running the example prints the forecast and the mean squared error.

```
predicted=11.871275, expected=12.900000
predicted=13.659297, expected=14.600000
predicted=14.349246, expected=14.000000
predicted=13.427454, expected=13.600000
predicted=13.374877, expected=13.500000
predicted=13.479991, expected=15.700000
predicted=14.765146, expected=13.000000
Test RMSE: 1.204
```

Listing 22.11: Example output of the manual predictions with an AR model on the Minimum Daily Temperatures dataset.

We can see a small improvement in the forecast when comparing the RMSE scores from 1.225 to 1.204.

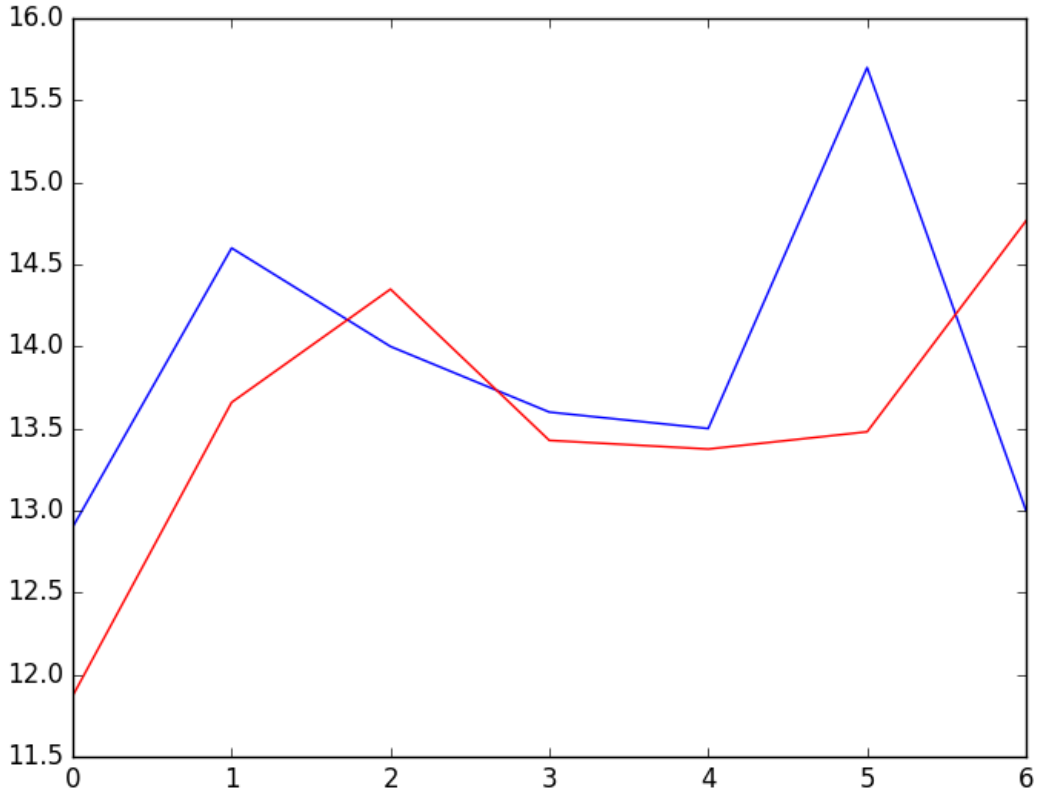

Figure 22.6: Line plot of the manual predictions with the AR model (red) on the Minimum Daily Temperatures dataset (blue).

22.8 Summary

In this tutorial, you discovered how to make autoregression forecasts for time series data using Python. Specifically, you learned:

- About autocorrelation and autoregression and how they can be used to better understand time series data.
- How to explore the autocorrelation in a time series using plots and statistical tests.
- How to train an autoregression model in Python and use it to make short-term and rolling forecasts.

22.8.1 Next

In the next lesson you will discover how to develop a moving average model.

Chapter 23

Moving Average Models for Forecasting

The residual errors from forecasts on a time series provide another source of information that we can model. Residual errors themselves form a time series that can have temporal structure. A simple autoregression model of this structure can be used to predict the forecast error, which in turn can be used to correct forecasts. This type of model is called a moving average model, the same name but very different from moving average smoothing. In this tutorial, you will discover how to model a residual error time series and use it to correct predictions with Python. After completing this tutorial, you will know:

- About how to model residual error time series using an autoregressive model.
- How to develop and evaluate a model of residual error time series.
- How to use a model of residual error to correct predictions and improve forecast skill.

Let's get started.

23.1 Model of Residual Errors

The difference between what was expected and what was predicted is called the residual error. It is calculated as:

$$residual_error = expected - predicted \tag{23.1}$$

Just like the input observations themselves, the residual errors from a time series can have temporal structure like trends, bias, and seasonality. Any temporal structure in the time series of residual forecast errors is useful as a diagnostic as it suggests information that could be incorporated into the predictive model. An ideal model would leave no structure in the residual error, just random fluctuations that cannot be modeled.

Structure in the residual error can also be modeled directly. There may be complex signals in the residual error that are difficult to directly incorporate into the model. Instead, you can create a model of the residual error time series and predict the expected error for your model. The predicted error can then be subtracted from the model prediction and in turn provide an additional lift in performance.

A simple and effective model of residual error is an autoregression. This is where some number of lagged error values are used to predict the error at the next time step. These lag errors are combined in a linear regression model, much like an autoregression model of the direct time series observations. An autoregression of the residual error time series is called a Moving Average (MA) model. This is confusing because it has nothing to do with the moving average smoothing process. Think of it as the sibling to the autoregressive (AR) process, except on lagged residual error rather than lagged raw observations.

In this tutorial, we will develop an autoregression model of the residual error time series. Before we dive in, let's look at a univariate dataset for which we will develop a model.

23.2 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`.

23.3 Persistence Forecast Model

The simplest forecast that we can make is to forecast that what happened in the previous time step will be the same as what will happen in the next time step. This is called the *naive forecast* or the persistence forecast model. This model will provide the predictions from which we can calculate the residual error time series. Alternately, we could develop an autoregression model of the time series and use that as our model. We will not develop an autoregression model in this case for brevity and to focus on the model of residual error.

We can implement the persistence model in Python. After the dataset is loaded, it is phrased as a supervised learning problem. A lagged version of the dataset is created where the current time step (`t`) is used as the input variable and the next time step (`t+1`) is taken as the output variable.

```
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
```

Listing 23.1: Create the lagged dataset.

Next, the dataset is split into training and test sets. A total of 66% of the data is kept for training and the remaining 34% is held for the test set. No training is required for the persistence model; this is just a standard test harness approach. Once split, the train and test sets are separated into their input and output components.

```
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
```

Listing 23.2: Split the lagged dataset.

```
# persistence model
predictions = [x for x in test_X]
```

Listing 23.3: Make persistence predictions.

The residual errors are then calculated as the difference between the expected outcome (`test_y`) and the prediction (`predictions`). The example puts this all together and gives us a set of residual forecast errors that we can explore this tutorial.

```
# calculate residual errors for a persistence forecast model
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from sklearn.metrics import mean_squared_error
from math import sqrt
# load data
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model
predictions = [x for x in test_X]
# skill of persistence model
rmse = sqrt(mean_squared_error(test_y, predictions))
print('Test RMSE: %.3f' % rmse)
# calculate residuals
residuals = [test_y[i]-predictions[i] for i in range(len(predictions))]
residuals = DataFrame(residuals)
print(residuals.head())
```

Listing 23.4: Persistence predictions and residual error for the Daily Female Births dataset.

The example then prints the RMSE of the persistence forecasts and the first 5 rows of the forecast residual errors.

```
Test RMSE: 9.151
      0
0   9.0
1 -10.0
2   3.0
3  -6.0
4  30.0
```

Listing 23.5: Example output of persistence predictions and residual error for the Daily Female Births dataset.

We now have a residual error time series that we can model.

23.4 Autoregression of Residual Error

We can model the residual error time series using an autoregression model. This is a linear regression model that creates a weighted linear sum of lagged residual error terms. For example:

$$error(t+1) = b0 + (b1 \times error(t)) + (b2 \times error(t-1)) \ldots + (bn \times error(t-n)) \quad (23.2)$$

We can use the autoregression model (AR) provided by the Statsmodels library. Building on the persistence model in the previous section, we can first train the model on the residual errors calculated on the training dataset. This requires that we make persistence predictions for each observation in the training dataset, then create the AR model, as follows.

```
# autoregressive model of residual errors
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from statsmodels.tsa.ar_model import AR
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model on training set
train_pred = [x for x in train_X]
# calculate residuals
train_resid = [train_y[i]-train_pred[i] for i in range(len(train_pred))]
# model the training set residuals
model = AR(train_resid)
model_fit = model.fit()
window = model_fit.k_ar
coef = model_fit.params
print('Lag=%d, Coef=%s' % (window, coef))
```

Listing 23.6: Autoregression model of persistence residual errors on the Daily Female Births dataset.

```
Lag=15, Coef=[ 0.10120699 -0.84940615 -0.77783609 -0.73345006 -0.68902061 -0.59270551
 -0.5376728 -0.42553356 -0.24861246 -0.19972102 -0.15954013 -0.11045476
 -0.14045572 -0.13299964 -0.12515801 -0.03615774]
```

Listing 23.7: Example output of autoregression model of persistence residual errors on the Daily Female Births dataset.

Next, we can step through the test dataset and for each time step we must:

1. Calculate the persistence prediction (`t+1 = t`).

2. Predict the residual error using the autoregression model.

The autoregression model requires the residual error of the 15 previous time steps. Therefore, we must keep these values handy. As we step through the test dataset timestep by timestep making predictions and estimating error, we can then calculate the actual residual error and update the residual error time series lag values (history) so that we can calculate the error at the next time step.

This is a walk forward forecast, or a rolling forecast, model. We end up with a time series of the residual forecast error from the train dataset and a predicted residual error on the test dataset. We can plot these and get a quick idea of how skillful the model is at predicting residual error. The complete example is listed below.

```
# forecast residual forecast error
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from statsmodels.tsa.ar_model import AR
from matplotlib import pyplot
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model on training set
train_pred = [x for x in train_X]
# calculate residuals
train_resid = [train_y[i]-train_pred[i] for i in range(len(train_pred))]
# model the training set residuals
model = AR(train_resid)
model_fit = model.fit()
window = model_fit.k_ar
coef = model_fit.params
# walk forward over time steps in test
history = train_resid[len(train_resid)-window:]
history = [history[i] for i in range(len(history))]
predictions = list()
expected_error = list()
for t in range(len(test_y)):
  # persistence
  yhat = test_X[t]
  error = test_y[t] - yhat
  expected_error.append(error)
  # predict error
  length = len(history)
  lag = [history[i] for i in range(length-window,length)]
  pred_error = coef[0]
  for d in range(window):
    pred_error += coef[d+1] * lag[window-d-1]
```

```
  predictions.append(pred_error)
  history.append(error)
  print('predicted error=%f, expected error=%f' % (pred_error, error))
# plot predicted error
pyplot.plot(expected_error)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 23.8: Forecast residual errors for the Daily Female Births dataset.

Running the example first prints the predicted and expected residual error for each time step in the test dataset.

```
...
predicted error=6.675538, expected error=3.000000
predicted error=3.419129, expected error=15.000000
predicted error=-7.160046, expected error=-4.000000
predicted error=-4.179003, expected error=7.000000
predicted error=-10.425124, expected error=-5.000000
```

Listing 23.9: Example output of forecasting residual errors on the Daily Female Births dataset.

Next, the actual residual error for the time series is plotted compared to the predicted residual error.

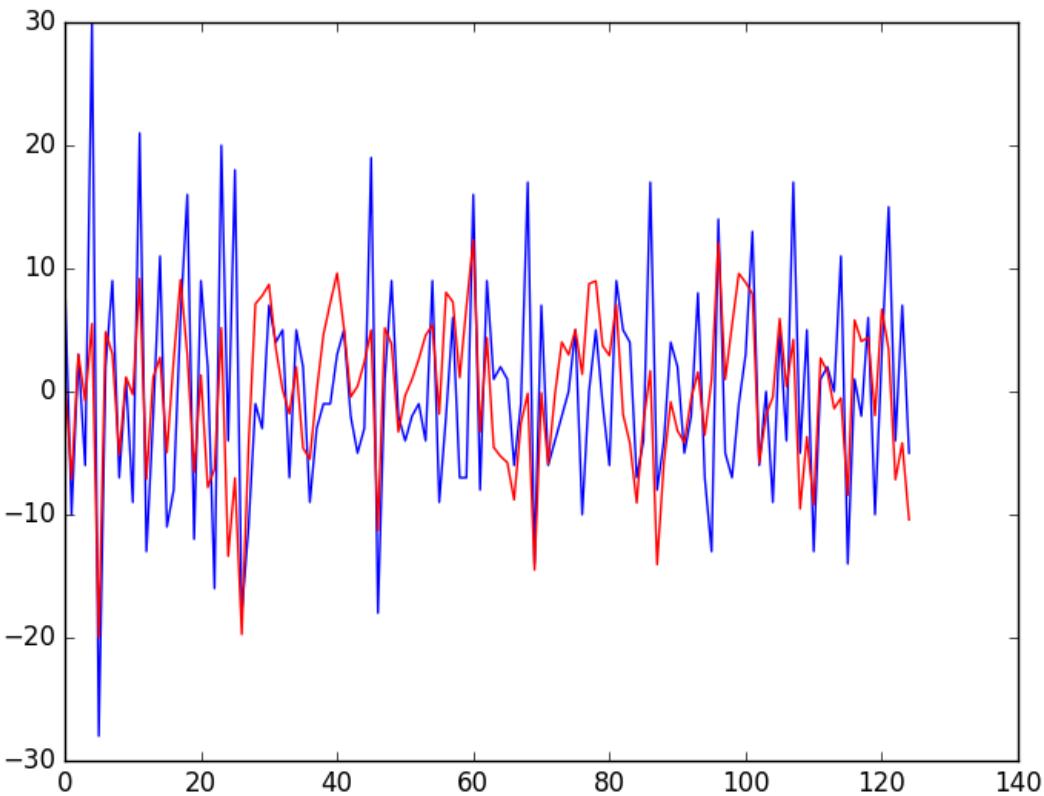

Figure 23.1: Line plot of expected residual error (blue) and forecast residual error (red) on the Daily Female Births dataset.

Now that we know how to model residual error, next we will look at how we can go about correcting forecasts and improving model skill.

23.5 Correct Predictions with a Model of Residuals

A model of forecast residual error is interesting, but it can also be useful to make better predictions. With a good estimate of forecast error at a time step, we can make better predictions. For example, we can add the expected forecast error to a prediction to correct it and in turn improve the skill of the model.

$$improved_forecast = forecast + estimated_error \tag{23.3}$$

Let's make this concrete with an example. Let's say that the expected value for a time step is 10. The model predicts 8 and estimates the error to be 3. The improved forecast would be:

$$\begin{aligned} improved_forecast &= forecast + estimated_error \\ &= 8 + 3 \\ &= 11 \end{aligned} \tag{23.4}$$

This takes the actual forecast error from 2 units to 1 unit. We can update the example from the previous section to add the estimated forecast error to the persistence forecast as follows:

```
# correct the prediction
yhat = yhat + pred_error
```

Listing 23.10: Correct predictions with forecast error.

The complete example is listed below.

```
# correct forecasts with a model of forecast residual errors
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from statsmodels.tsa.ar_model import AR
from matplotlib import pyplot
from sklearn.metrics import mean_squared_error
from math import sqrt
# load data
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# create lagged dataset
values = DataFrame(series.values)
dataframe = concat([values.shift(1), values], axis=1)
dataframe.columns = ['t', 't+1']
# split into train and test sets
X = dataframe.values
train_size = int(len(X) * 0.66)
train, test = X[1:train_size], X[train_size:]
train_X, train_y = train[:,0], train[:,1]
test_X, test_y = test[:,0], test[:,1]
# persistence model on training set
train_pred = [x for x in train_X]
# calculate residuals
```

```
train_resid = [train_y[i]-train_pred[i] for i in range(len(train_pred))]
# model the training set residuals
model = AR(train_resid)
model_fit = model.fit()
window = model_fit.k_ar
coef = model_fit.params
# walk forward over time steps in test
history = train_resid[len(train_resid)-window:]
history = [history[i] for i in range(len(history))]
predictions = list()
for t in range(len(test_y)):
  # persistence
  yhat = test_X[t]
  error = test_y[t] - yhat
  # predict error
  length = len(history)
  lag = [history[i] for i in range(length-window,length)]
  pred_error = coef[0]
  for d in range(window):
    pred_error += coef[d+1] * lag[window-d-1]
  # correct the prediction
  yhat = yhat + pred_error
  predictions.append(yhat)
  history.append(error)
  print('predicted=%f, expected=%f' % (yhat, test_y[t]))
# error
rmse = sqrt(mean_squared_error(test_y, predictions))
print('Test RMSE: %.3f' % rmse)
# plot predicted error
pyplot.plot(test_y)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 23.11: Correct predictions with forecast error on the Daily Female Births dataset.

Running the example prints the predictions and the expected outcome for each time step in the test dataset. The RMSE of the corrected forecasts is calculated to be 7.499 births per day, which is much better than the score of 9.151 for the persistence model alone.

```
...
predicted=40.675538, expected=37.000000
predicted=40.419129, expected=52.000000
predicted=44.839954, expected=48.000000
predicted=43.820997, expected=55.000000
predicted=44.574876, expected=50.000000
Test RMSE: 7.499
```

Listing 23.12: Example output of correcting predictions with forecast error on the Daily Female Births dataset.

Finally, the expected values for the test dataset are plotted compared to the corrected forecast. We can see that the persistence model has been aggressively corrected back to a time series that looks something like a moving average.

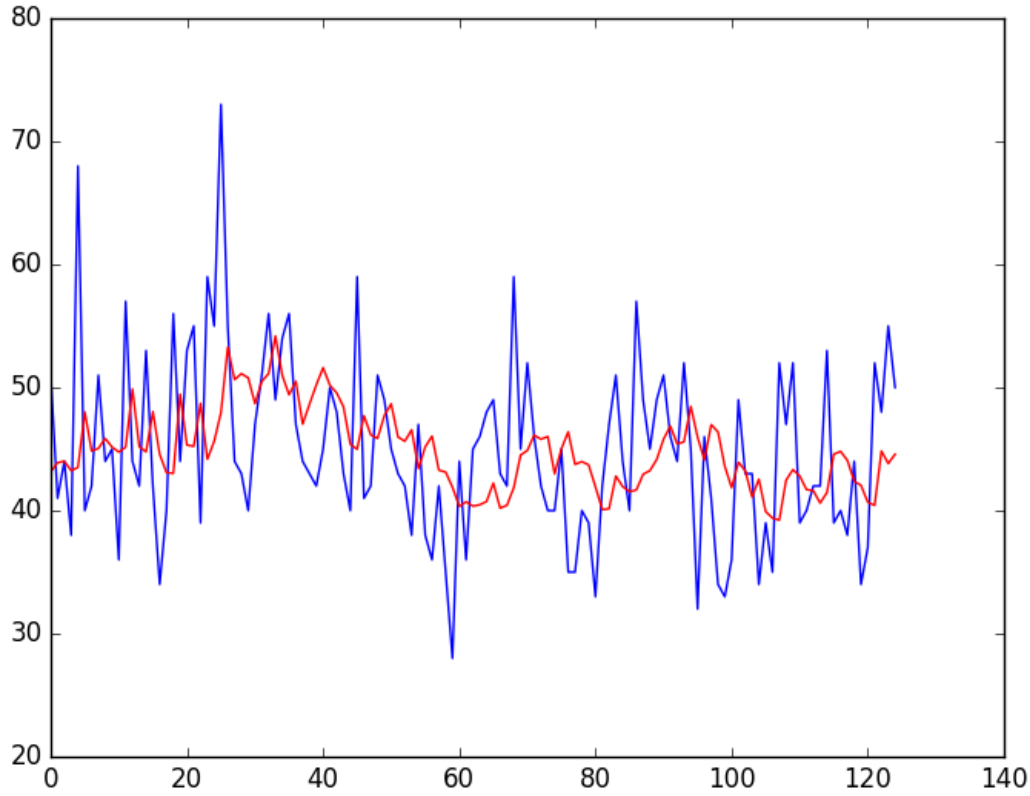

Figure 23.2: Line plot of expected values (blue) and corrected forecasts (red) on the Daily Female Births dataset.

23.6 Summary

In this tutorial, you discovered how to model residual error time series and use it to correct predictions with Python. Specifically, you learned:

- About the Moving Average (MA) approach to developing an autoregressive model to residual error.
- How to develop and evaluate a model of residual error to predict forecast error.
- How to use the predictions of forecast error to correct predictions and improve model skill.

23.6.1 Next

In the next lesson you will discover how to develop an ARIMA model.

Chapter 24

ARIMA Model for Forecasting

A popular and widely used statistical method for time series forecasting is the ARIMA model. ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a class of model that captures a suite of different standard temporal structures in time series data. In this tutorial, you will discover how to develop an ARIMA model for time series data with Python. After completing this tutorial, you will know:

- About the ARIMA model the parameters used and assumptions made by the model.
- How to fit an ARIMA model to data and use it to make forecasts.
- How to configure the ARIMA model on your time series problem.

Let's get started.

24.1 Autoregressive Integrated Moving Average Model

An ARIMA model is a class of statistical models for analyzing and forecasting time series data. It explicitly caters to a suite of standard structures in time series data, and as such provides a simple yet powerful method for making skillful time series forecasts. ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration. This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- **AR**: *Autoregression*. A model that uses the dependent relationship between an observation and some number of lagged observations.
- **I**: *Integrated*. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- **MA**: *Moving Average*. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Each of these components are explicitly specified in the model as a parameter. A standard notation is used of `ARIMA(p,d,q)` where the parameters are substituted with integer values to quickly indicate the specific ARIMA model being used.

The parameters of the ARIMA model are defined as follows:

- p: The number of lag observations included in the model, also called the lag order.
- d: The number of times that the raw observations are differenced, also called the degree of differencing.
- q: The size of the moving average window, also called the order of moving average.

A linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model. A value of 0 can be used for a parameter, which indicates to not use that element of the model. This way, the ARIMA model can be configured to perform the function of an ARMA model, and even a simple AR, I, or MA model.

Adopting an ARIMA model for a time series assumes that the underlying process that generated the observations is an ARIMA process. This may seem obvious, but helps to motivate the need to confirm the assumptions of the model in the raw observations and in the residual errors of forecasts from the model. Next, let's take a look at how we can use the ARIMA model in Python. We will start with loading a simple univariate time series.

24.2 Shampoo Sales Dataset

In this lesson, we will use the Shampoo Sales dataset as an example. This dataset describes the monthly number of sales of shampoo over a 3 year period. You can learn more about the dataset in Appendix A.1. Place the dataset in your current working directory with the filename `shampoo-sales.csv`. Below is an example of loading the Shampoo Sales dataset with Pandas with a custom function to parse the date-time field.

```
# load and plot dataset
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
# summarize first few rows
print(series.head())
# line plot
series.plot()
pyplot.show()
```

Listing 24.1: Load and plot the Shampoo Sales dataset.

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

```
Month
1901-01-01   266.0
1901-02-01   145.9
1901-03-01   183.1
1901-04-01   119.3
1901-05-01   180.3
```

Listing 24.2: Example output of the first 5 rows of the Shampoo Sales dataset.

The data is also plotted as a time series with the month along the x-axis and sales figures on the y-axis.

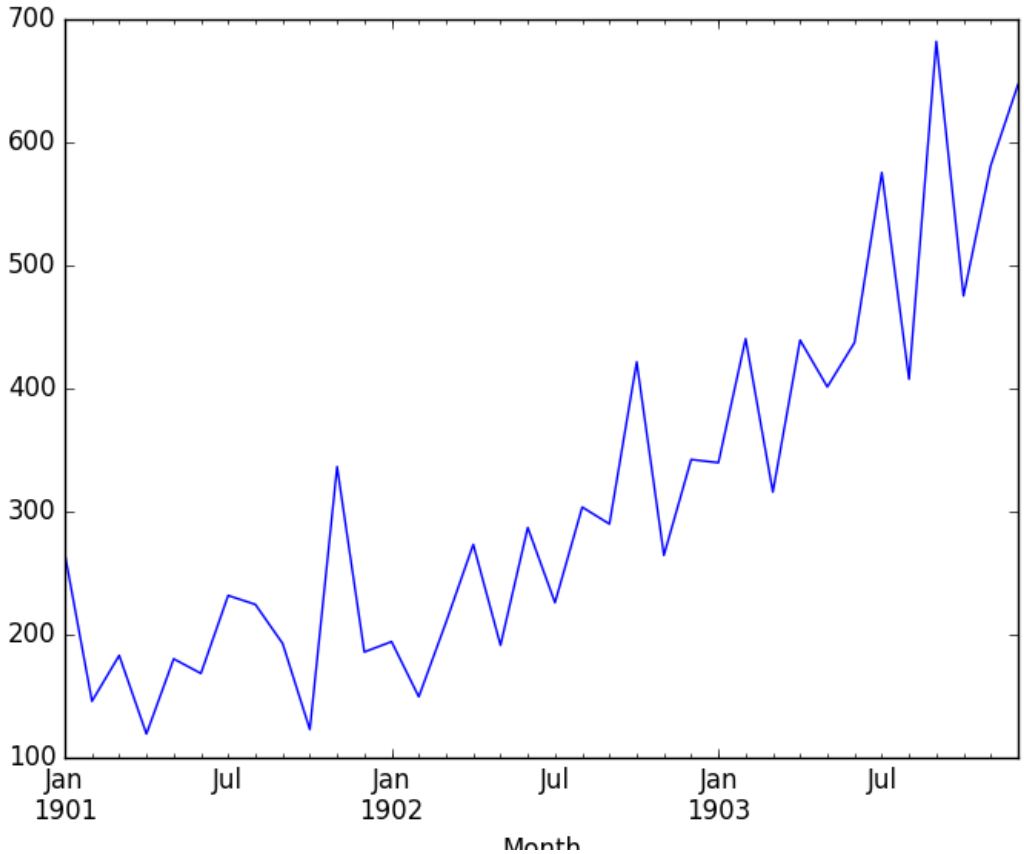

Figure 24.1: Line plot of the Shampoo Sales dataset.

We can see that the Shampoo Sales dataset has a clear trend. This suggests that the time series is not stationary and will require differencing to make it stationary, at least a difference order of 1. Let's also take a quick look at an autocorrelation plot of the time series. This is also built-in to Pandas. The example below plots the autocorrelation for a large number of lags in the time series.

```
# autocorrelation plot of time series
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from pandas.plotting import autocorrelation_plot
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
```

```
# autocorrelation plot
autocorrelation_plot(series)
pyplot.show()
```

Listing 24.3: Autocorrelation plot of the Shampoo Sales dataset.

Running the example, we can see that there is a positive correlation with the first 10-to-12 lags that is perhaps significant for the first 5 lags. A good starting point for the AR parameter of the model may be 5.

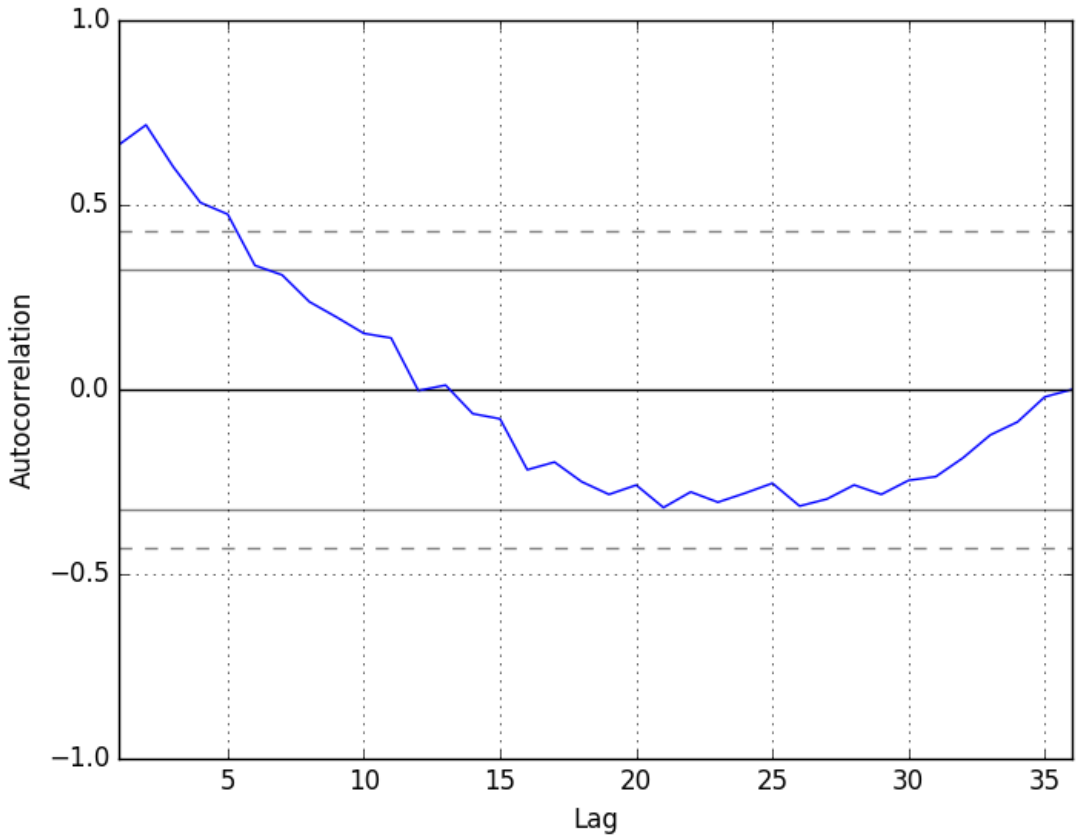

Figure 24.2: Autocorrelation plot of the Shampoo Sales dataset.

24.3 ARIMA with Python

The Statsmodels library provides the capability to fit an ARIMA model. An ARIMA model can be created using the Statsmodels library as follows:

1. Define the model by calling `ARIMA()`[1] and passing in the `p`, `d`, and `q` parameters.
2. The model is prepared on the training data by calling the `fit()` function[2].

[1] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMA.html

[2] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMA.fit.html

3. Predictions can be made by calling the `predict()` function[3] and specifying the index of the time or times to be predicted.

Let's start off with something simple. We will fit an ARIMA model to the entire Shampoo Sales dataset and review the residual errors. First, we fit an `ARIMA(5,1,0)` model. This sets the lag value to 5 for autoregression, uses a difference order of 1 to make the time series stationary, and uses a moving average model of 0. When fitting the model, a lot of debug information is provided about the fit of the linear regression model. We can turn this off by setting the `disp` argument to 0.

```
# fit an ARIMA model and plot residual errors
from pandas import read_csv
from pandas import datetime
from pandas import DataFrame
from statsmodels.tsa.arima_model import ARIMA
from matplotlib import pyplot
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
# fit model
model = ARIMA(series, order=(5,1,0))
model_fit = model.fit(disp=0)
# summary of fit model
print(model_fit.summary())
# line plot of residuals
residuals = DataFrame(model_fit.resid)
residuals.plot()
pyplot.show()
# density plot of residuals
residuals.plot(kind='kde')
pyplot.show()
# summary stats of residuals
print(residuals.describe())
```

Listing 24.4: ARIMA model of the Shampoo Sales dataset.

Running the example prints a summary of the fit model. This summarizes the coefficient values used as well as the skill of the fit on the on the in-sample observations.

```
                             ARIMA Model Results
==============================================================================
Dep. Variable:              D.Sales   No. Observations:                   35
Model:                 ARIMA(5, 1, 0)  Log Likelihood                -196.170
Method:                       css-mle  S.D. of innovations             64.241
Date:                Fri, 20 Jan 2017  AIC                            406.340
Time:                        15:01:24  BIC                            417.227
Sample:                    02-01-1901  HQIC                           410.098
                         - 12-01-1903
================================================================================
                   coef    std err          z      P>|z|      [95.0% Conf. Int.]
--------------------------------------------------------------------------------
```

[3] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMA.predict.html

```
const           12.0649     3.652     3.304     0.003        4.908   19.222
ar.L1.D.Sales   -1.1082     0.183    -6.063     0.000       -1.466   -0.750
ar.L2.D.Sales   -0.6203     0.282    -2.203     0.036       -1.172   -0.068
ar.L3.D.Sales   -0.3606     0.295    -1.222     0.231       -0.939    0.218
ar.L4.D.Sales   -0.1252     0.280    -0.447     0.658       -0.674    0.424
ar.L5.D.Sales    0.1289     0.191     0.673     0.506       -0.246    0.504
                                    Roots
=============================================================================
                Real          Imaginary         Modulus        Frequency
-----------------------------------------------------------------------------
AR.1          -1.0617         -0.5064j          1.1763         -0.4292
AR.2          -1.0617         +0.5064j          1.1763          0.4292
AR.3           0.0816         -1.3804j          1.3828         -0.2406
AR.4           0.0816         +1.3804j          1.3828          0.2406
AR.5           2.9315         -0.0000j          2.9315         -0.0000
-----------------------------------------------------------------------------
               0
count  35.000000
mean   -5.495213
std    68.132882
min   -133.296597
25%    -42.477935
50%     -7.186584
75%     24.748357
max    133.237980
```

Listing 24.5: Example output of the ARIMA model coefficients summary.

First, we get a line plot of the residual errors, suggesting that there may still be some trend information not captured by the model.

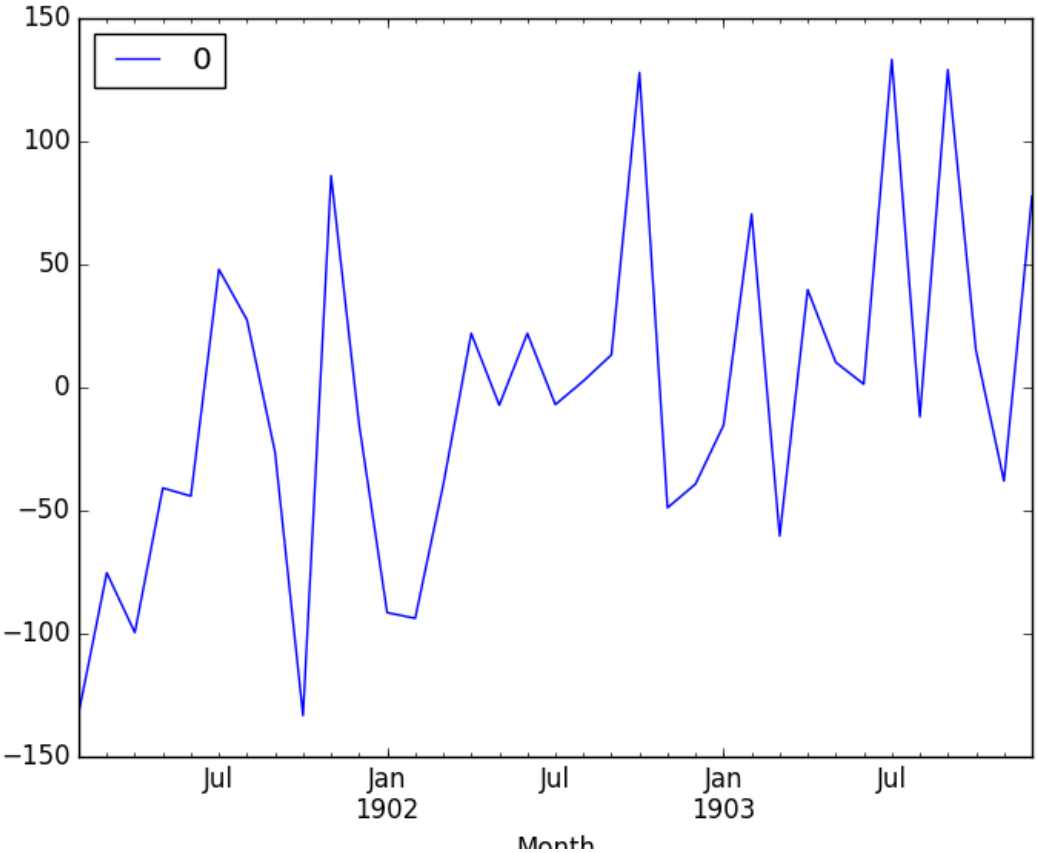

Figure 24.3: Line plot of ARIMA forecast residual errors on the Shampoo Sales dataset.

Next, we get a density plot of the residual error values, suggesting the errors are Gaussian, but may not be centered on zero.

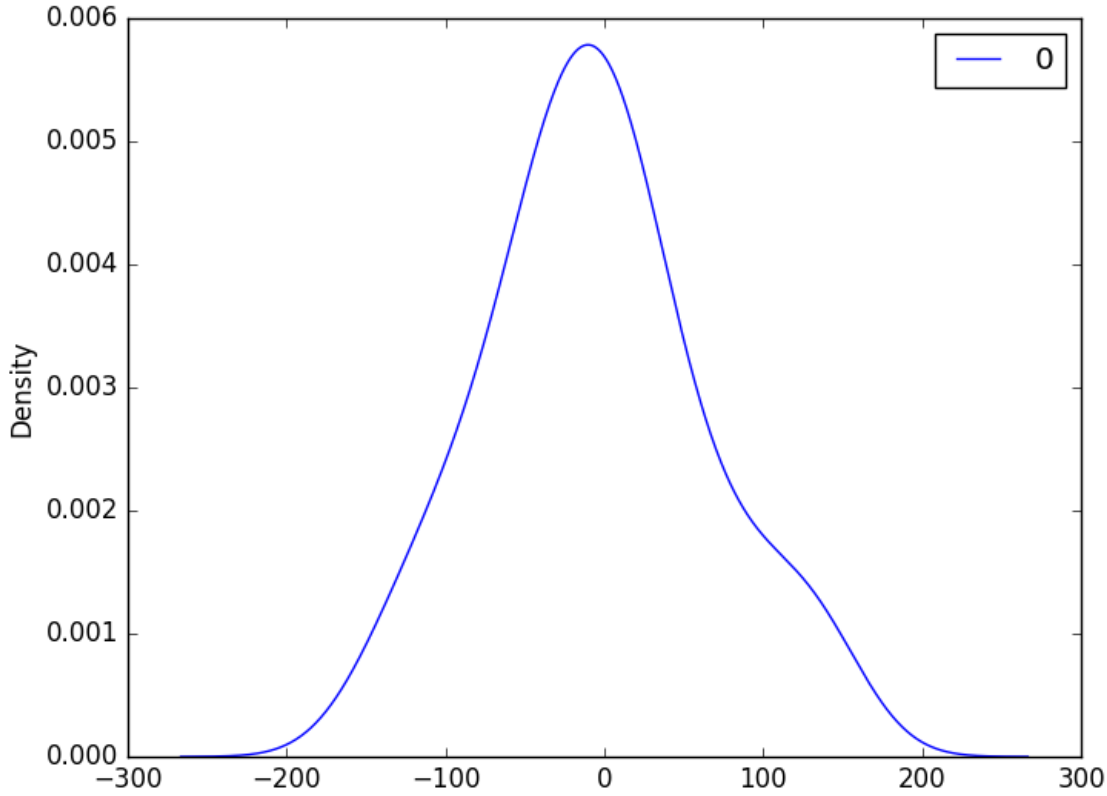

Figure 24.4: Density plot of ARIMA forecast residual errors on the Shampoo Sales dataset.

The distribution of the residual errors is displayed. The results show that indeed there is a bias in the prediction (a non-zero mean in the residuals).

```
count   35.000000
mean    -5.495213
std     68.132882
min   -133.296597
25%    -42.477935
50%     -7.186584
75%     24.748357
max    133.237980
```

Listing 24.6: Example output of the ARIMA model summary statistics of forecast error residuals.

Note, that although above we used the entire dataset for time series analysis, ideally we would perform this analysis on just the training dataset when developing a predictive model. Next, let's look at how we can use the ARIMA model to make forecasts.

24.4 Rolling Forecast ARIMA Model

The ARIMA model can be used to forecast future time steps. We can use the `predict()` function on the `ARIMAResults` object[4] to make predictions. It accepts the index of the time steps to make predictions as arguments. These indexes are relative to the start of the training dataset used to make predictions.

If we used 100 observations in the training dataset to fit the model, then the index of the next time step for making a prediction would be specified to the prediction function as `start=101, end=101`. This would return an array with one element containing the prediction. We also would prefer the forecasted values to be in the original scale, in case we performed any differencing ($d > 0$ when configuring the model). This can be specified by setting the `typ` argument to the value `'levels'`: `typ='levels'`.

Alternately, we can avoid all of these specifications by using the `forecast()` function[5] which performs a one-step forecast using the model. We can split the training dataset into train and test sets, use the train set to fit the model, and generate a prediction for each element on the test set.

A rolling forecast is required given the dependence on observations in prior time steps for differencing and the AR model. A crude way to perform this rolling forecast is to re-create the ARIMA model after each new observation is received. We manually keep track of all observations in a list called history that is seeded with the training data and to which new observations are appended each iteration. Putting this all together, below is an example of a rolling forecast with the ARIMA model in Python.

```
# evaluate an ARIMA model using a walk-forward validation
from pandas import read_csv
from pandas import datetime
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
# split into train and test sets
X = series.values
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:len(X)]
history = [x for x in train]
predictions = list()
# walk-forward validation
for t in range(len(test)):
  model = ARIMA(history, order=(5,1,0))
  model_fit = model.fit(disp=0)
  output = model_fit.forecast()
  yhat = output[0]
```

[4] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMAResults.html

[5] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMAResults.forecast.html

```
  predictions.append(yhat)
  obs = test[t]
  history.append(obs)
  print('predicted=%f, expected=%f' % (yhat, obs))
# evaluate forecasts
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
# plot forecasts against actual outcomes
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 24.7: ARIMA model with rolling forecasts on the Shampoo Sales dataset.

Running the example prints the prediction and expected value each iteration. We can also calculate a final root mean squared error score (RMSE) for the predictions, providing a point of comparison for other ARIMA configurations.

```
...
predicted=434.915566, expected=407.600000
predicted=507.923407, expected=682.000000
predicted=435.483082, expected=475.300000
predicted=652.743772, expected=581.300000
predicted=546.343485, expected=646.900000
Test RMSE: 83.417
```

Listing 24.8: Example output of the ARIMA model with rolling forecasts on the Shampoo Sales dataset.

A line plot is created showing the expected values compared to the rolling forecast predictions. We can see the values show some trend and are in the correct scale.

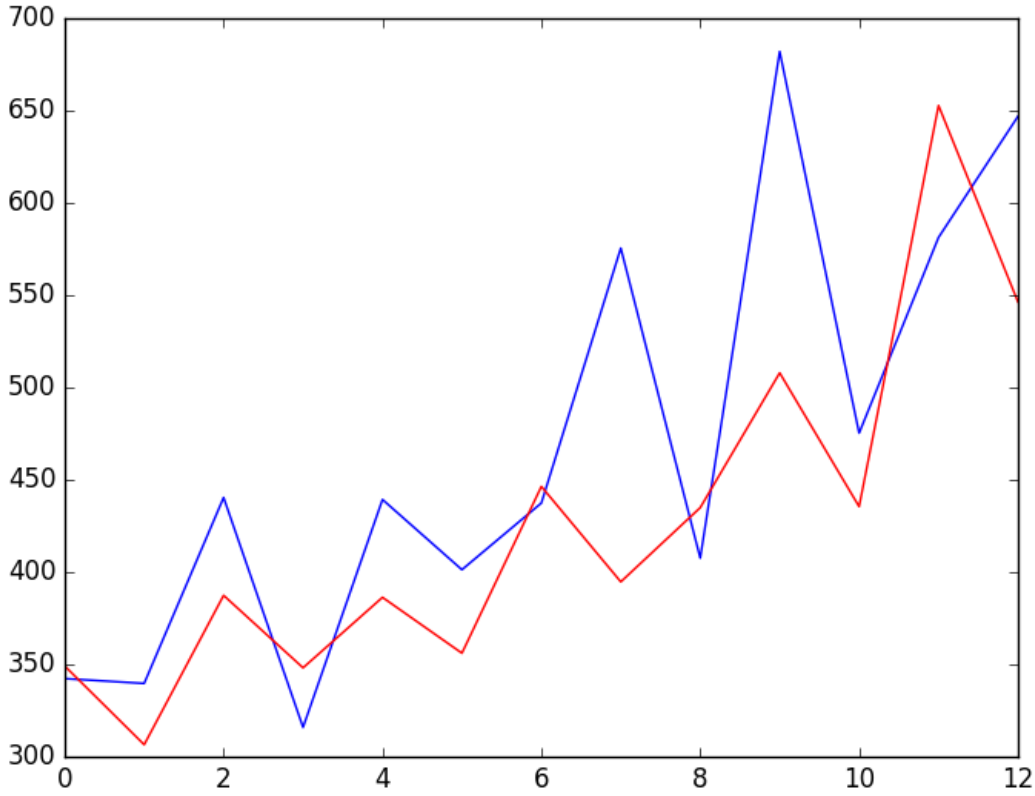

Figure 24.5: Line plot of expected values (blue) and rolling forecast (red) with an ARIMA model on the Shampoo Sales dataset.

The model could use further tuning of the `p`, `d`, and maybe even the `q` parameters.

24.5 Summary

In this tutorial, you discovered how to develop an ARIMA model for time series forecasting in Python. Specifically, you learned:

- About the ARIMA model, how it can be configured, and assumptions made by the model.
- How to perform a quick time series analysis using the ARIMA model.
- How to use an ARIMA model to forecast out of sample predictions.

24.5.1 Next

In the next lesson you will discover how to differentiate ACF and PACF plots and how to read them.

Chapter 25

Autocorrelation and Partial Autocorrelation

Autocorrelation and partial autocorrelation plots are heavily used in time series analysis and forecasting. These are plots that graphically summarize the strength of a relationship with an observation in a time series with observations at prior time steps. The difference between autocorrelation and partial autocorrelation can be difficult and confusing for beginners to time series forecasting. In this tutorial, you will discover how to calculate and plot autocorrelation and partial correlation plots with Python. After completing this tutorial, you will know:

- How to plot and review the autocorrelation function for a time series.
- How to plot and review the partial autocorrelation function for a time series.
- The difference between autocorrelation and partial autocorrelation functions for time series analysis.

Let's get started.

25.1 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`.

25.2 Correlation and Autocorrelation

Statistical correlation summarizes the strength of the relationship between two variables. We can assume the distribution of each variable fits a Gaussian (bell curve) distribution. If this is the case, we can use the Pearson's correlation coefficient to summarize the correlation between the variables. The Pearson's correlation coefficient is a number between -1 and 1 that describes a negative or positive correlation respectively. A value of zero indicates no correlation.

We can calculate the correlation for time series observations with observations with previous time steps, called lags. Because the correlation of the time series observations is calculated with

values of the same series at previous times, this is called a serial correlation, or an autocorrelation. A plot of the autocorrelation of a time series by lag is called the AutoCorrelation Function, or the acronym ACF. This plot is sometimes called a correlogram or an autocorrelation plot.

Below is an example of calculating and plotting the autocorrelation plot for the Minimum Daily Temperatures using the `plot_acf()` function[1] from the Statsmodels library.

```
# ACF plot of time series
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.graphics.tsaplots import plot_acf
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
plot_acf(series)
pyplot.show()
```

Listing 25.1: Create an ACF plot on the Minimum Daily Temperatures dataset.

Running the example creates a 2D plot showing the lag value along the x-axis and the correlation on the y-axis between -1 and 1. Confidence intervals are drawn as a cone. By default, this is set to a 95% confidence interval, suggesting that correlation values outside of this cone are very likely a correlation and not a statistical fluke.

[1] http://statsmodels.sourceforge.net/devel/generated/statsmodels.graphics.tsaplots.plot_acf.html

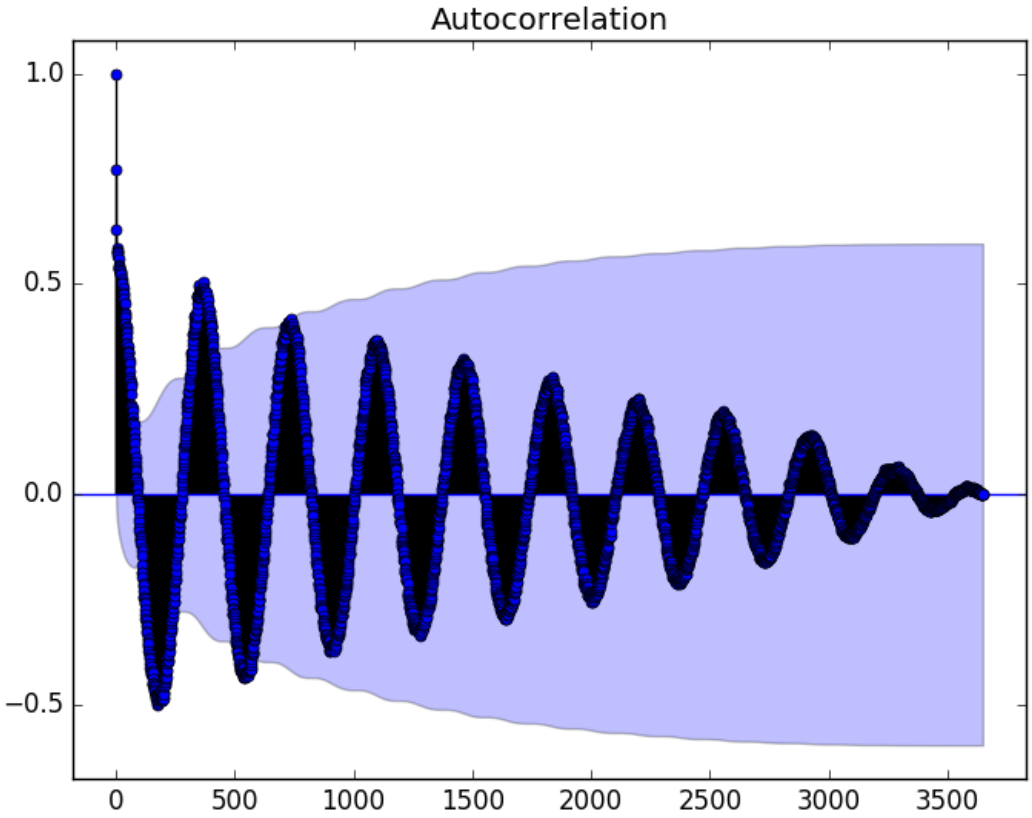

Figure 25.1: ACF plot on the Minimum Daily Temperatures dataset.

By default, all lag values are printed, which makes the plot noisy. We can limit the number of lags on the x-axis to 50 by setting the `lags` argument.

```
# zoomed-in ACF plot of time series
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.graphics.tsaplots import plot_acf
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
plot_acf(series, lags=50)
pyplot.show()
```

Listing 25.2: Create an ACF plot on the Minimum Daily Temperatures dataset.

The resulting plot is easier to read.

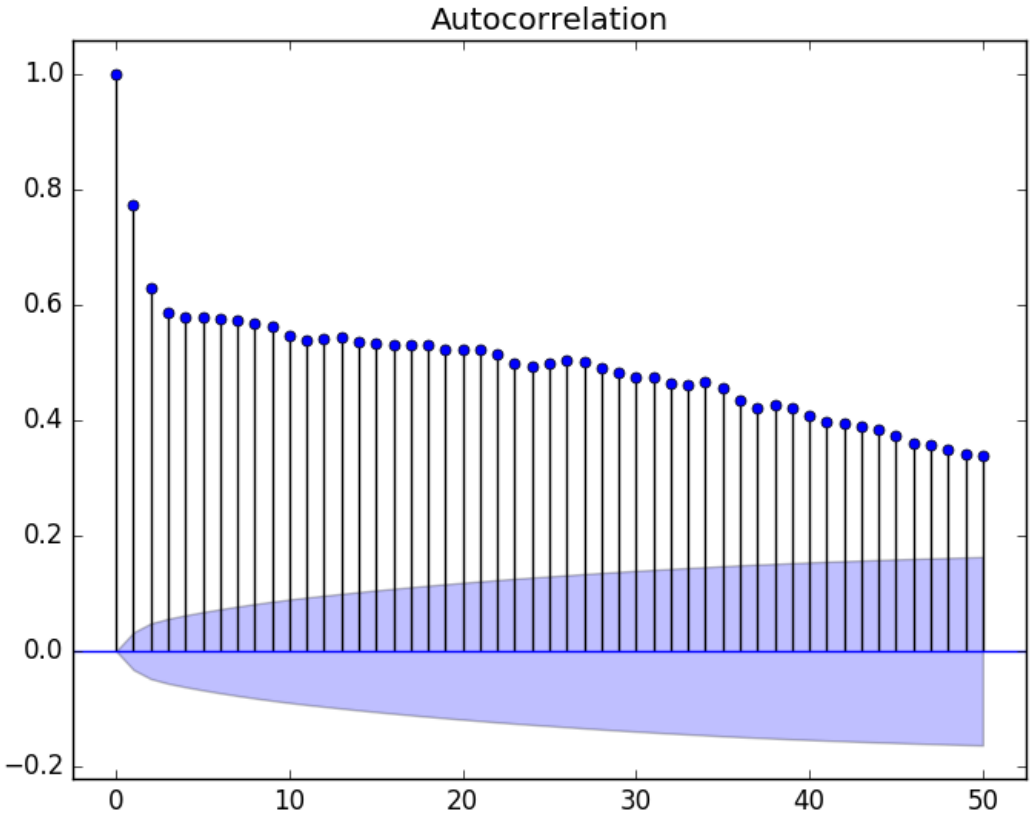

Figure 25.2: ACF plot with lags=50 on the Minimum Daily Temperatures dataset.

25.3 Partial Autocorrelation Function

A partial autocorrelation is a summary of the relationship between an observation in a time series with observations at prior time steps with the relationships of intervening observations removed.

> The partial autocorrelation at lag k is the correlation that results after removing the effect of any correlations due to the terms at shorter lags.
>
> — Page 81, Section 4.5.6 Partial Autocorrelations, *Introductory Time Series with R*.

The autocorrelation for an observation and an observation at a prior time step is comprised of both the direct correlation and indirect correlations. These indirect correlations are a linear function of the correlation of the observation, with observations at intervening time steps. It is these indirect correlations that the partial autocorrelation function seeks to remove. Without going into the math, this is the intuition for the partial autocorrelation.

The example below calculates and plots a partial autocorrelation function for the first 50 lags in the Minimum Daily Temperatures dataset using the `plot_pacf()` function[2] from the

[2] http://statsmodels.sourceforge.net/devel/generated/statsmodels.graphics.tsaplots.plot_pacf.html

Statsmodels library.

```
# PACF plot of time series
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.graphics.tsaplots import plot_pacf
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
plot_pacf(series, lags=50)
pyplot.show()
```

Listing 25.3: Create an PACF plot on the Minimum Daily Temperatures dataset.

Running the example creates a 2D plot of the partial autocorrelation for the first 50 lags.

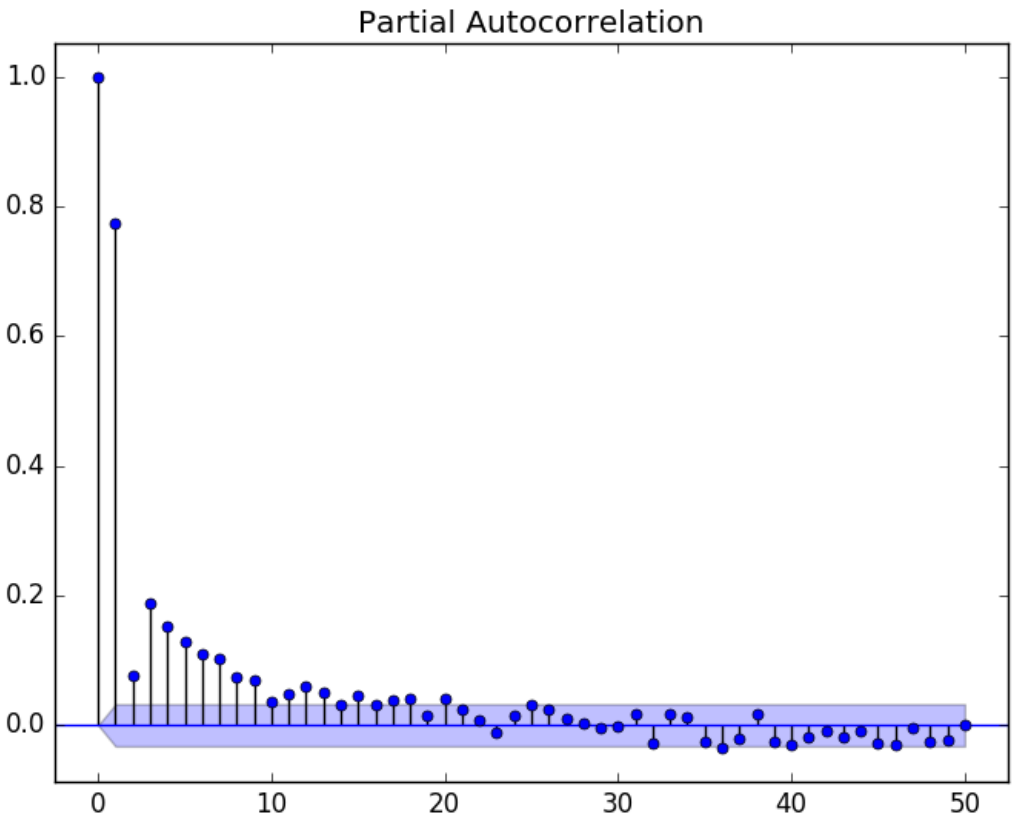

Figure 25.3: PACF plot with lags=50 on the Minimum Daily Temperatures dataset.

25.4 Intuition for ACF and PACF Plots

Plots of the autocorrelation function and the partial autocorrelation function for a time series tell a very different story. We can use the intuition for ACF and PACF above to explore some thought experiments.

25.4.1 Autoregression Intuition

Consider a time series that was generated by an autoregression (AR) process with a lag of k. We know that the ACF describes the autocorrelation between an observation and another observation at a prior time step that includes direct and indirect dependence information. This means we would expect the ACF for the `AR(k)` time series to be strong to a lag of k and the inertia of that relationship would carry on to subsequent lag values, trailing off at some point as the effect was weakened.

We know that the PACF only describes the direct relationship between an observation and its lag. This would suggest that there would be no correlation for lag values beyond k. This is exactly the expectation of the ACF and PACF plots for an `AR(k)` process.

25.4.2 Moving Average Intuition

Consider a time series that was generated by a moving average (MA) process with a lag of k. Remember that the moving average process is an autoregression model of the time series of residual errors from prior predictions. Another way to think about the moving average model is that it corrects future forecasts based on errors made on recent forecasts. We would expect the ACF for the `MA(k)` process to show a strong correlation with recent values up to the lag of k, then a sharp decline to low or no correlation. By definition, this is how the process was generated.

For the PACF, we would expect the plot to show a strong relationship to the lag and a trailing off of correlation from the lag onwards. Again, this is exactly the expectation of the ACF and PACF plots for an `MA(k)` process.

25.5 Summary

In this tutorial, you discovered how to calculate autocorrelation and partial autocorrelation plots for time series data with Python. Specifically, you learned:

- How to calculate and create an autocorrelation plot for time series data.
- How to calculate and create a partial autocorrelation plot for time series data.
- The difference and intuition for interpreting ACF and PACF plots.

25.5.1 Next

In the next lesson you will discover how you can use grid search to tune an ARIMA model.

Chapter 26

Grid Search ARIMA Model Hyperparameters

The ARIMA model for time series analysis and forecasting can be tricky to configure. There are 3 parameters that require estimation by iterative trial and error from reviewing diagnostic plots and using 40-year-old heuristic rules. We can automate the process of evaluating a large number of hyperparameters for the ARIMA model by using a grid search procedure. In this tutorial, you will discover how to tune the ARIMA model using a grid search of hyperparameters in Python. After completing this tutorial, you will know:

- A general procedure that you can use to tune the ARIMA hyperparameters for a rolling one-step forecast.
- How to apply ARIMA hyperparameter optimization on a standard univariate time series dataset.
- Ideas for extending the procedure for more elaborate and robust models.

Let's get started.

26.1 Grid Searching Method

Diagnostic plots of the time series can be used along with heuristic rules to determine the hyperparameters of the ARIMA model. These are good in most, but perhaps not all, situations. We can automate the process of training and evaluating ARIMA models on different combinations of model hyperparameters. In machine learning this is called a grid search or model tuning. In this tutorial, we will develop a method to grid search ARIMA hyperparameters for a one-step rolling forecast. The approach is broken down into two parts:

1. Evaluate an ARIMA model.
2. Evaluate sets of ARIMA parameters.

The code in this tutorial makes use of the scikit-learn, Pandas, and the Statsmodels Python libraries.

26.2 Evaluate ARIMA Model

We can evaluate an ARIMA model by preparing it on a training dataset and evaluating predictions on a test dataset. This approach involves the following steps:

1. Split the dataset into training and test sets.
2. Walk the time steps in the test dataset.
   (a) Train an ARIMA model.
   (b) Make a one-step prediction.
   (c) Store prediction; get and store actual observation.
3. Calculate error score for predictions compared to expected values.

We can implement this in Python as a new standalone function called `evaluate_arima_model()` that takes a time series dataset as input as well as a tuple with the `p`, `d`, and `q` parameters for the model to be evaluated. The dataset is split in two: 66% for the initial training dataset and the remaining 34% for the test dataset.

Each time step of the test set is iterated. Just one iteration provides a model that you could use to make predictions on new data. The iterative approach allows a new ARIMA model to be trained each time step. A prediction is made each iteration and stored in a list. This is so that at the end of the test set, all predictions can be compared to the list of expected values and an error score calculated. In this case, a root mean squared error score is calculated and returned. The complete function is listed below.

```
# evaluate an ARIMA model for a given order (p,d,q)
def evaluate_arima_model(X, arima_order):
  # prepare training dataset
  train_size = int(len(X) * 0.66)
  train, test = X[0:train_size], X[train_size:]
  history = [x for x in train]
  # make predictions
  predictions = list()
  for t in range(len(test)):
    model = ARIMA(history, order=arima_order)
    model_fit = model.fit(disp=0)
    yhat = model_fit.forecast()[0]
    predictions.append(yhat)
    history.append(test[t])
  # calculate out of sample error
  rmse = sqrt(mean_squared_error(test, predictions))
  return rmse
```

Listing 26.1: Function to evaluate an ARIMA model using out-of-sample error.

Now that we know how to evaluate one set of ARIMA hyperparameters, let's see how we can call this function repeatedly for a grid of parameters to evaluate.

26.3 Iterate ARIMA Parameters

Evaluating a suite of parameters is relatively straightforward. The user must specify a grid of `p`, `d`, and `q` ARIMA parameters to iterate. A model is created for each parameter and its performance evaluated by calling the `evaluate_arima_model()` function described in the previous section. The function must keep track of the lowest error score observed and the configuration that caused it. This can be summarized at the end of the function with a print to standard out.

We can implement this function called `evaluate_models()` as a series of four loops. There are two additional considerations. The first is to ensure the input data are floating point values (as opposed to integers or strings), as this can cause the ARIMA procedure to fail. Second, the Statsmodels ARIMA procedure internally uses numerical optimization procedures to find a set of coefficients for the model. These procedures can fail, which in turn can throw an exception. We must catch these exceptions and skip those configurations that cause a problem. This happens more often then you would think. Additionally, it is recommended that warnings be ignored for this code to avoid a lot of noise from running the procedure. This can be done as follows:

```
import warnings
warnings.filterwarnings("ignore")
```

Listing 26.2: Disable Python warnings.

Finally, even with all of these protections, the underlying code libraries may still report warnings to standard out, such as:

```
** On entry to DLASCL, parameter number 4 had an illegal value
```

Listing 26.3: Example error message from underlying libraries.

These have been removed from the results reported in this tutorial for brevity. The complete procedure for evaluating a grid of ARIMA hyperparameters is listed below.

```
# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
  dataset = dataset.astype('float32')
  best_score, best_cfg = float("inf"), None
  for p in p_values:
    for d in d_values:
      for q in q_values:
        order = (p,d,q)
        try:
          rmse = evaluate_arima_model(dataset, order)
          if rmse < best_score:
            best_score, best_cfg = rmse, order
          print('ARIMA%s RMSE=%.3f' % (order,rmse))
        except:
          continue
  print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))
```

Listing 26.4: Function to grid search configurations for an ARIMA model.

Now that we have a procedure to grid search ARIMA hyperparameters, let's test the procedure on two univariate time series problems. We will start with the Shampoo Sales dataset.

26.4 Shampoo Sales Case Study

In this section, we will use the Shampoo Sales dataset as an example. This dataset describes the monthly number of sales of shampoo over a 3 year period. You can learn more about the dataset in Appendix A.1. Place the dataset in your current working directory with the filename `shampoo-sales.csv`. The timestamps in the time series do not contain an absolute year component. We can use a custom date-parsing function when loading the data and baseline the year from 1900, as follows:

```
# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, parse_dates=[0], index_col=0,
    squeeze=True, date_parser=parser)
```

Listing 26.5: Load the Shampoo Sales dataset.

Once loaded, we can specify a site of `p`, `d`, and `q` values to search and pass them to the `evaluate_models()` function. We will try a suite of lag values (`p`) and just a few difference iterations (`d`) and residual error lag values (`q`).

```
# evaluate parameters
p_values = [0, 1, 2, 4, 6, 8, 10]
d_values = range(0, 3)
q_values = range(0, 3)
warnings.filterwarnings("ignore")
evaluate_models(series.values, p_values, d_values, q_values)
```

Listing 26.6: ARIMA configuration to grid search on the Shampoo Sales dataset.

Putting this all together with the generic procedures defined in the previous section, we can grid search ARIMA hyperparameters in the Shampoo Sales dataset. The complete code example is listed below.

```
# grid search ARIMA parameters for time series
import warnings
from math import sqrt
from pandas import read_csv
from pandas import datetime
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error

# evaluate an ARIMA model for a given order (p,d,q)
def evaluate_arima_model(X, arima_order):
  # prepare training dataset
  train_size = int(len(X) * 0.66)
  train, test = X[0:train_size], X[train_size:]
  history = [x for x in train]
  # make predictions
  predictions = list()
  for t in range(len(test)):
    model = ARIMA(history, order=arima_order)
    model_fit = model.fit(disp=0)
    yhat = model_fit.forecast()[0]
    predictions.append(yhat)
    history.append(test[t])
```

```
  # calculate out of sample error
  rmse = sqrt(mean_squared_error(test, predictions))
  return rmse

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
  dataset = dataset.astype('float32')
  best_score, best_cfg = float("inf"), None
  for p in p_values:
    for d in d_values:
      for q in q_values:
        order = (p,d,q)
        try:
          rmse = evaluate_arima_model(dataset, order)
          if rmse < best_score:
            best_score, best_cfg = rmse, order
          print('ARIMA%s RMSE=%.3f' % (order,rmse))
        except:
          continue
  print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))

# load dataset
def parser(x):
  return datetime.strptime('190'+x, '%Y-%m')
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True, date_parser=parser)
# evaluate parameters
p_values = [0, 1, 2, 4, 6, 8, 10]
d_values = range(0, 3)
q_values = range(0, 3)
warnings.filterwarnings("ignore")
evaluate_models(series.values, p_values, d_values, q_values)
```

Listing 26.7: Grid search ARIMA configurations on the Shampoo Sales dataset.

Running the example prints the ARIMA parameters and RMSE for each successful evaluation completed. The best parameters of `ARIMA(4,2,1)` are reported at the end of the run with a root mean squared error of 68.519 sales.

```
ARIMA(0, 0, 0) RMSE=228.966
ARIMA(0, 0, 1) RMSE=195.308
ARIMA(0, 0, 2) RMSE=154.886
ARIMA(0, 1, 0) RMSE=134.176
ARIMA(0, 1, 1) RMSE=97.767
ARIMA(0, 2, 0) RMSE=259.499
ARIMA(0, 2, 1) RMSE=135.363
ARIMA(1, 0, 0) RMSE=152.029
ARIMA(1, 1, 0) RMSE=84.388
ARIMA(1, 1, 1) RMSE=83.688
ARIMA(1, 2, 0) RMSE=136.411
ARIMA(2, 1, 0) RMSE=75.432
ARIMA(2, 1, 1) RMSE=88.089
ARIMA(2, 2, 0) RMSE=99.302
ARIMA(4, 1, 0) RMSE=81.545
ARIMA(4, 1, 1) RMSE=82.440
ARIMA(4, 2, 0) RMSE=87.157
```

```
ARIMA(4, 2, 1) RMSE=68.519
ARIMA(6, 1, 0) RMSE=82.523
ARIMA(6, 2, 0) RMSE=79.127
ARIMA(8, 0, 0) RMSE=85.182
ARIMA(8, 1, 0) RMSE=81.114
Best ARIMA(4, 2, 1) RMSE=68.519
```

Listing 26.8: Example output of grid searching ARIMA configuration on the Shampoo Sales dataset.

26.5 Daily Female Births Case Study

In this section, 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`. This dataset can be easily loaded directly as a Pandas `Series`.

```
# load dataset
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
```

Listing 26.9: Load the Daily Female Births dataset.

To keep things simple, we will explore the same grid of ARIMA hyperparameters as in the previous section.

```
# evaluate parameters
p_values = [0, 1, 2, 4, 6, 8, 10]
d_values = range(0, 3)
q_values = range(0, 3)
warnings.filterwarnings("ignore")
evaluate_models(series.values, p_values, d_values, q_values)
```

Listing 26.10: ARIMA configuration to grid search on the Daily Female Births dataset.

Putting this all together, we can grid search ARIMA parameters on the Daily Female Births dataset. The complete code listing is provided below.

```
# grid search ARIMA parameters for time series
import warnings
from math import sqrt
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error

# evaluate an ARIMA model for a given order (p,d,q)
def evaluate_arima_model(X, arima_order):
  # prepare training dataset
  train_size = int(len(X) * 0.66)
  train, test = X[0:train_size], X[train_size:]
  history = [x for x in train]
  # make predictions
  predictions = list()
```

```
  for t in range(len(test)):
    model = ARIMA(history, order=arima_order)
    model_fit = model.fit(disp=0)
    yhat = model_fit.forecast()[0]
    predictions.append(yhat)
    history.append(test[t])
  # calculate out of sample error
  rmse = sqrt(mean_squared_error(test, predictions))
  return rmse

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
  dataset = dataset.astype('float32')
  best_score, best_cfg = float("inf"), None
  for p in p_values:
    for d in d_values:
      for q in q_values:
        order = (p,d,q)
        try:
          rmse = evaluate_arima_model(dataset, order)
          if rmse < best_score:
            best_score, best_cfg = rmse, order
          print('ARIMA%s RMSE=%.3f' % (order,rmse))
        except:
          continue
  print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))

# load dataset
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# evaluate parameters
p_values = [0, 1, 2, 4, 6, 8, 10]
d_values = range(0, 3)
q_values = range(0, 3)
warnings.filterwarnings("ignore")
evaluate_models(series.values, p_values, d_values, q_values)
```

Listing 26.11: Grid search ARIMA configurations on the Daily Female Births dataset.

Running the example prints the ARIMA parameters and root mean squared error for each configuration successfully evaluated. The mean best parameters are reported as `ARIMA(6,1,0)` with a root mean squared error of 7.293 births.

```
ARIMA(0, 0, 0) RMSE=8.189
ARIMA(0, 0, 1) RMSE=7.884
ARIMA(0, 0, 2) RMSE=7.771
ARIMA(0, 1, 0) RMSE=9.167
ARIMA(0, 1, 1) RMSE=7.527
ARIMA(0, 1, 2) RMSE=7.434
ARIMA(0, 2, 0) RMSE=15.698
ARIMA(0, 2, 1) RMSE=9.201
ARIMA(1, 0, 0) RMSE=7.802
ARIMA(1, 1, 0) RMSE=8.120
ARIMA(1, 1, 1) RMSE=7.425
ARIMA(1, 1, 2) RMSE=7.429
ARIMA(1, 2, 0) RMSE=11.990
```

```
ARIMA(2, 0, 0) RMSE=7.697
ARIMA(2, 1, 0) RMSE=7.713
ARIMA(2, 1, 1) RMSE=7.417
ARIMA(2, 2, 0) RMSE=10.373
ARIMA(4, 0, 0) RMSE=7.693
ARIMA(4, 1, 0) RMSE=7.578
ARIMA(4, 1, 1) RMSE=7.474
ARIMA(4, 2, 0) RMSE=8.956
ARIMA(6, 0, 0) RMSE=7.666
ARIMA(6, 1, 0) RMSE=7.293
ARIMA(6, 1, 1) RMSE=7.553
ARIMA(6, 2, 0) RMSE=8.352
ARIMA(8, 0, 0) RMSE=7.549
ARIMA(8, 1, 0) RMSE=7.569
ARIMA(8, 2, 0) RMSE=8.126
ARIMA(8, 2, 1) RMSE=7.608
ARIMA(10, 0, 0) RMSE=7.581
ARIMA(10, 1, 0) RMSE=7.574
ARIMA(10, 2, 0) RMSE=8.093
ARIMA(10, 2, 1) RMSE=7.608
ARIMA(10, 2, 2) RMSE=7.636
Best ARIMA(6, 1, 0) RMSE=7.293
```

Listing 26.12: Example output of grid searching ARIMA configuration on the Daily Female Births dataset.

26.6 Extensions

The grid search method used in this tutorial is simple and can easily be extended. This section lists some ideas to extend the approach you may wish to explore.

- **Seed Grid**. The classical diagnostic tools of ACF and PACF plots can still be used with the results used to seed the grid of ARIMA parameters to search.
- **Alternate Measures**. The search seeks to optimize the out-of-sample root mean squared error. This could be changed to another out-of-sample statistic, an in-sample statistic, such as AIC or BIC, or some combination of the two. You can choose a metric that is most meaningful on your project.
- **Residual Diagnostics**. Statistics can automatically be calculated on the residual forecast errors to provide an additional indication of the quality of the fit. Examples include statistical tests for whether the distribution of residuals is Gaussian and whether there is an autocorrelation in the residuals.
- **Update Model**. The ARIMA model is created from scratch for each one-step forecast. With careful inspection of the API, it may be possible to update the internal data of the model with new observations rather than recreating it from scratch.
- **Preconditions**. The ARIMA model can make assumptions about the time series dataset, such as normality and stationarity. These could be checked and a warning raised for a given of a dataset prior to a given model being trained.

26.7 Summary

In this tutorial, you discovered how to grid search the hyperparameters for the ARIMA model in Python. Specifically, you learned:

- A procedure that you can use to grid search ARIMA hyperparameters for a one-step rolling forecast.
- How to apply ARIMA hyperparameters tuning on standard univariate time series datasets.
- Ideas on how to further improve grid searching of ARIMA hyperparameters.

26.7.1 Next

In the next lesson you will discover how to finalize a model for making predictions on new data.

Chapter 27

Save Models and Make Predictions

Selecting a time series forecasting model is just the beginning. Using the chosen model in practice can pose challenges, including data transformations and storing the model parameters on disk. In this tutorial, you will discover how to finalize a time series forecasting model and use it to make predictions in Python. After completing this tutorial, you will know:

- How to finalize a model and save it and required data to file.
- How to load a finalized model from file and use it to make a prediction.
- How to update data associated with a finalized model in order to make subsequent predictions.

Let's get started.

27.1 Process for Making a Prediction

A lot is written about how to tune specific time series forecasting models, but little help is given to how to use a model to make predictions. Once you can build and tune forecast models for your data, the process of making a prediction involves the following steps:

1. **Model Selection**. This is where you choose a model and gather evidence and support to defend the decision.
2. **Model Finalization**. The chosen model is trained on all available data and saved to file for later use.
3. **Forecasting**. The saved model is loaded and used to make a forecast.
4. **Model Update**. Elements of the model are updated in the presence of new observations.

We will take a look at each of these elements in this tutorial, with a focus on saving and loading the model to and from file and using a loaded model to make predictions. Before we dive in, let's first look at a standard univariate dataset that we can use as the context for this tutorial.

27.2 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`.

27.3 Select Time Series Forecast Model

You must select a model. This is where the bulk of the effort will be in preparing the data, performing analysis, and ultimately selecting a model and model hyperparameters that best capture the relationships in the data. In this case, we can arbitrarily select an autoregression model (AR) with a lag of 6 on the differenced dataset. We can demonstrate this model below. First, the data is transformed by differencing, with each observation transformed as:

$$value(t) = obs(t) - obs(t-1) \tag{27.1}$$

Next, the `AR(6)` model is trained on 66% of the historical data. The regression coefficients learned by the model are extracted and used to make predictions in a rolling manner across the test dataset. As each time step in the test dataset is executed, the prediction is made using the coefficients and stored. The actual observation for the time step is then made available and stored to be used as a lag variable for future predictions.

```
# fit and evaluate an AR model
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.tsa.ar_model import AR
from sklearn.metrics import mean_squared_error
import numpy
from math import sqrt

# create a difference transform of the dataset
def difference(dataset):
  diff = list()
  for i in range(1, len(dataset)):
    value = dataset[i] - dataset[i - 1]
    diff.append(value)
  return numpy.array(diff)

# Make a prediction give regression coefficients and lag obs
def predict(coef, history):
  yhat = coef[0]
  for i in range(1, len(coef)):
    yhat += coef[i] * history[-i]
  return yhat

series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# split dataset
X = difference(series.values)
size = int(len(X) * 0.66)
train, test = X[0:size], X[size:]
```

```
# train autoregression
model = AR(train)
model_fit = model.fit(maxlag=6, disp=False)
window = model_fit.k_ar
coef = model_fit.params
# walk forward over time steps in test
history = [train[i] for i in range(len(train))]
predictions = list()
for t in range(len(test)):
  yhat = predict(coef, history)
  obs = test[t]
  predictions.append(yhat)
  history.append(obs)
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
# plot
pyplot.plot(test)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 27.1: AR model on the Daily Female Births dataset.

Running the example first prints the Root Mean Squared Error (RMSE) of the predictions, which is about 7 births on average. This is how well we expect the model to perform on average when making forecasts on new data.

```
Test RMSE: 7.259
```

Listing 27.2: Example output of AR model on the Daily Female Births dataset.

Finally, a graph is created showing the actual observations in the test dataset compared to the predictions.

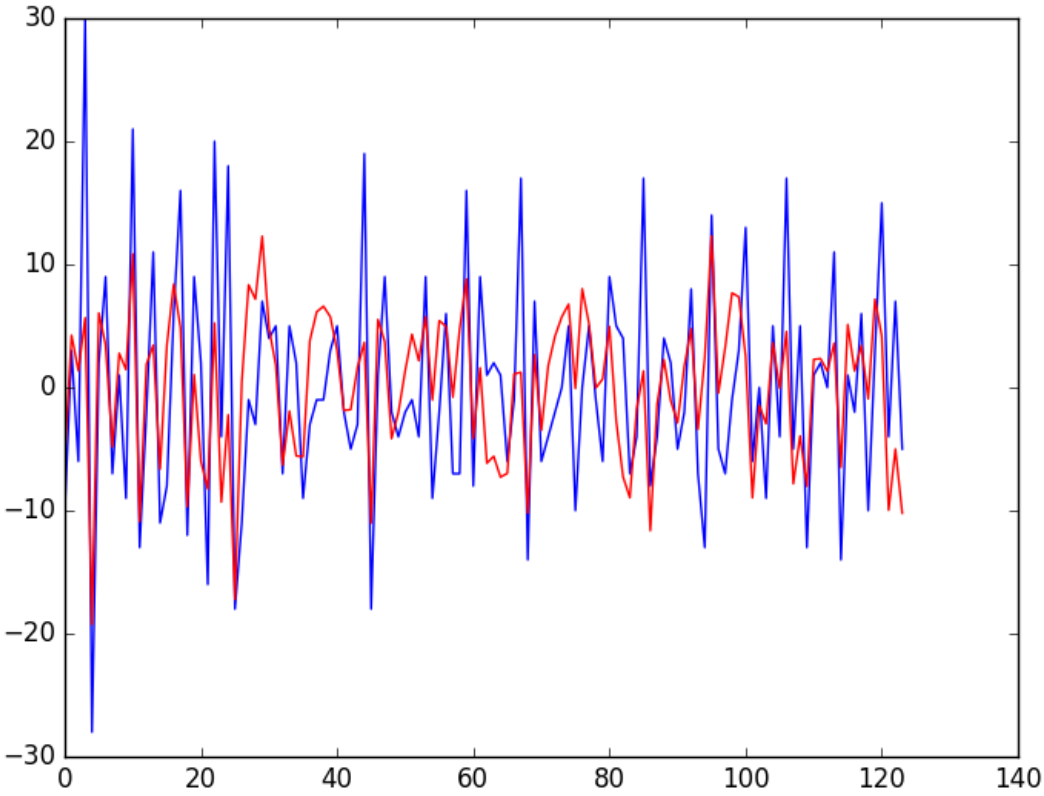

Figure 27.1: Line plot of expected values (blue) and AR model predictions (red) on the Daily Female Births dataset.

This may not be the very best possible model we could develop on this problem, but it is reasonable and skillful.

27.4 Finalize and Save Time Series Forecast Model

Once the model is selected, we must finalize it. This means save the salient information learned by the model so that we do not have to re-create it every time a prediction is needed. This involves first training the model on all available data and then saving the model to file.

The Statsmodels implementations of time series models do provide built-in capability to save and load models by calling `save()` and `load()` on the fit `ARResults` object[1]. For example, the code below will train an `AR(6)` model on the entire Female Births dataset and save it using the built-in `save()` function, which will essentially pickle the `ARResults` object.

The differenced training data must also be saved, both for the lag variables needed to make a prediction, and for knowledge of the number of observations seen, required by the `predict()` function of the `ARResults` object. Finally, we need to be able to transform the differenced

[1] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.ar_model.ARResults.html

dataset back into the original form. To do this, we must keep track of the last actual observation. This is so that the predicted differenced value can be added to it.

```
# fit an AR model and save the whole model to file
from pandas import read_csv
from statsmodels.tsa.ar_model import AR
import numpy

# create a difference transform of the dataset
def difference(dataset):
  diff = list()
  for i in range(1, len(dataset)):
    value = dataset[i] - dataset[i - 1]
    diff.append(value)
  return numpy.array(diff)

# load dataset
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
X = difference(series.values)
# fit model
model = AR(X)
model_fit = model.fit(maxlag=6, disp=False)
# save model to file
model_fit.save('ar_model.pkl')
# save the differenced dataset
numpy.save('ar_data.npy', X)
# save the last ob
numpy.save('ar_obs.npy', [series.values[-1]])
```

Listing 27.3: Save the AR model for the Daily Female Births dataset.

This code will create a file `ar_model.pkl` that you can load later and use to make predictions. The entire differenced training dataset is saved as `ar_data.npy` and the last observation is saved in the file `ar_obs.npy` as an array with one item.

The NumPy `save()` function[2] is used to save the differenced training data and the observation. The `load()` function[3] can then be used to load these arrays later. The snippet below will load the model, differenced data, and last observation.

```
# load the AR model from file
from statsmodels.tsa.ar_model import ARResults
import numpy
loaded = ARResults.load('ar_model.pkl')
print(loaded.params)
data = numpy.load('ar_data.npy')
last_ob = numpy.load('ar_obs.npy')
print(last_ob)
```

Listing 27.4: Load the AR model for the Daily Female Births dataset.

Running the example prints the coefficients and the last observation.

```
[ 0.12129822 -0.75275857 -0.612367 -0.51097172 -0.4176669 -0.32116469
 -0.23412997]
```

[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.save.html
[3] https://docs.scipy.org/doc/numpy/reference/generated/numpy.load.html

```
[50]
```

Listing 27.5: Example output of loading the AR model for the Daily Female Births dataset.

I think this is good for most cases, but is also pretty heavy. You are subject to changes to the Statsmodels API. My preference is to work with the coefficients of the model directly, as in the case above, evaluating the model using a rolling forecast. In this case, you could simply store the model coefficients and later load them and make predictions. The example below saves just the coefficients from the model, as well as the minimum differenced lag values required to make the next prediction and the last observation needed to transform the next prediction made.

```
# fit an AR model and manually save coefficients to file
from pandas import read_csv
from statsmodels.tsa.ar_model import AR
import numpy

# create a difference transform of the dataset
def difference(dataset):
  diff = list()
  for i in range(1, len(dataset)):
    value = dataset[i] - dataset[i - 1]
    diff.append(value)
  return numpy.array(diff)

# load dataset
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
X = difference(series.values)
# fit model
window_size = 6
model = AR(X)
model_fit = model.fit(maxlag=window_size, disp=False)
# save coefficients
coef = model_fit.params
numpy.save('man_model.npy', coef)
# save lag
lag = X[-window_size:]
numpy.save('man_data.npy', lag)
# save the last ob
numpy.save('man_obs.npy', [series.values[-1]])
```

Listing 27.6: Save the AR model coefficients for the Daily Female Births dataset.

The coefficients are saved in the local file `man_model.npy`, the lag history is saved in the file `man_data.npy`, and the last observation is saved in the file `man_obs.npy`. These values can then be loaded again as follows:

```
# load the manually saved model from file
import numpy
coef = numpy.load('man_model.npy')
print(coef)
lag = numpy.load('man_data.npy')
print(lag)
last_ob = numpy.load('man_obs.npy')
print(last_ob)
```

Listing 27.7: Load the AR model coefficients for the Daily Female Births dataset.

Running this example prints the loaded data for review. We can see the coefficients and last observation match the output from the previous example.

```
[ 0.12129822 -0.75275857 -0.612367 -0.51097172 -0.4176669 -0.32116469
 -0.23412997]
[-10  3  15  -4  7  -5]
[50]
```

Listing 27.8: Example output of loading the AR model coefficients for the Daily Female Births dataset.

Now that we know how to save a finalized model, we can use it to make forecasts.

27.5 Make a Time Series Forecast

Making a forecast involves loading the saved model and estimating the observation at the next time step. If the `ARResults` object was serialized, we can use the `predict()` function to predict the next time period. The example below shows how the next time period can be predicted. The model, training data, and last observation are loaded from file.

The period is specified to the `predict()` function as the next time index after the end of the training data set. This index may be stored directly in a file instead of storing the entire training data, which may be an efficiency. The prediction is made, which is in the context of the differenced dataset. To turn the prediction back into the original units, it must be added to the last known observation.

```
# load AR model from file and make a one-step prediction
from statsmodels.tsa.ar_model import ARResults
import numpy
# load model
model = ARResults.load('ar_model.pkl')
data = numpy.load('ar_data.npy')
last_ob = numpy.load('ar_obs.npy')
# make prediction
predictions = model.predict(start=len(data), end=len(data))
# transform prediction
yhat = predictions[0] + last_ob[0]
print('Prediction: %f' % yhat)
```

Listing 27.9: Load the AR model and make a prediction for the Daily Female Births dataset.

Running the example prints the prediction.

```
Prediction: 46.755211
```

Listing 27.10: Example output of loading the AR model and making a prediction for the Daily Female Births dataset.

We can also use a similar trick to load the raw coefficients and make a manual prediction. The complete example is listed below.

```
# load a coefficients and from file and make a manual prediction
import numpy

def predict(coef, history):
  yhat = coef[0]
  for i in range(1, len(coef)):
    yhat += coef[i] * history[-i]
  return yhat

# load model
coef = numpy.load('man_model.npy')
lag = numpy.load('man_data.npy')
last_ob = numpy.load('man_obs.npy')
# make prediction
prediction = predict(coef, lag)
# transform prediction
yhat = prediction + last_ob[0]
print('Prediction: %f' % yhat)
```

Listing 27.11: Complete example of making a prediction and updating the model for the Daily Female Births dataset.

Running the example, we achieve the same prediction as we would expect, given the underlying model and method for making the prediction are the same.

```
Prediction: 46.755211
```

Listing 27.12: Example output making a prediction and updating the model for the Daily Female Births dataset.

27.6 Update Forecast Model

Our work is not done. Once the next real observation is made available, we must update the data associated with the model. Specifically, we must update:

1. The differenced training dataset used as inputs to make the subsequent prediction.
2. The last observation, providing a context for the predicted differenced value.

Let's assume the next actual observation in the series was 48. The new observation must first be differenced with the last observation. It can then be stored in the list of differenced observations. Finally, the value can be stored as the last observation.

In the case of the stored AR model, we can update the `ar_data.npy` and `ar_obs.npy` files. The complete example is listed below:

```
# update the data for the AR model with a new obs
import numpy
# get real observation
observation = 48
# load the saved data
data = numpy.load('ar_data.npy')
last_ob = numpy.load('ar_obs.npy')
```

```
# update and save differenced observation
diffed = observation - last_ob[0]
data = numpy.append(data, [diffed], axis=0)
numpy.save('ar_data.npy', data)
# update and save real observation
last_ob[0] = observation
numpy.save('ar_obs.npy', last_ob)
```

Listing 27.13: Update the differenced data and last observation for the AR model.

We can make the same changes for the data files for the manual case. Specifically, we can update the `man_data.npy` and `man_obs.npy` respectively. The complete example is listed below.

```
# update the data for the manual model with a new obs
import numpy
# get real observation
observation = 48
# update and save differenced observation
lag = numpy.load('man_data.npy')
last_ob = numpy.load('man_obs.npy')
diffed = observation - last_ob[0]
lag = numpy.append(lag[1:], [diffed], axis=0)
numpy.save('man_data.npy', lag)
# update and save real observation
last_ob[0] = observation
numpy.save('man_obs.npy', last_ob)
```

Listing 27.14: Complete example of updating observations for a model of the Daily Female Births dataset.

We have focused on one-step forecasts. These methods would work just as easily for multi-step forecasts, by using the model repetitively and using forecasts of previous time steps as input lag values to predict observations for subsequent time steps.

27.7 Extensions

Generally, it is a good idea to keep track of all the observations. This will allow you to:

- Provide a context for further time series analysis to understand new changes in the data.
- Train a new model in the future on the most recent data.
- Back-test new and different models to see if performance can be improved.

For small applications, perhaps you could store the raw observations in a file alongside your model. It may also be desirable to store the model coefficients and required lag data and last observation in plain text for easy review. For larger applications, perhaps a database system could be used to store the observations.

27.8 Summary

In this tutorial, you discovered how to finalize a time series model and use it to make predictions with Python. Specifically, you learned:

- How to save a time series forecast model to file.
- How to load a saved time series forecast from file and make a prediction.
- How to update a time series forecast model with new observations.

27.8.1 Next

In the next lesson you will discover how to calculate and review confidence intervals on time series forecasts.

Chapter 28

Forecast Confidence Intervals

Time series forecast models can both make predictions and provide a confidence interval for those predictions. Confidence intervals provide an upper and lower expectation for the real observation. These can be useful for assessing the range of real possible outcomes for a prediction and for better understanding the skill of the model In this tutorial, you will discover how to calculate and interpret confidence intervals for time series forecasts with Python.

Specifically, you will learn:

- How to make a forecast with an ARIMA model and gather forecast diagnostic information.
- How to interpret a confidence interval for a forecast and configure different intervals.
- How to plot the confidence interval in the context of recent observations.

Let's dive in.

28.1 ARIMA Forecast

The ARIMA implementation in the Statsmodels Python library can be used to fit an ARIMA model. It returns an `ARIMAResults` object. This object provides the `forecast()` function that can be used to make predictions about future time steps and default to predicting the value at the next time step after the end of the training data. Assuming we are predicting just the next time step, the `forecast()` method returns three values:

- **Forecast**. The forecasted value in the units of the training time series.
- **Standard error**. The standard error for the model.
- **Confidence interval**. The 95% confidence interval for the forecast.

In this tutorial, we will better understand the confidence interval provided with an ARIMA forecast. Before we dive in, let's first look at the Daily Female Births dataset that we will use as the context for this tutorial.

28.2 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`.

28.3 Forecast Confidence Interval

In this section, we will train an ARIMA model, use it to make a prediction, and inspect the confidence interval. First, we will split the training dataset into a training and test dataset. Almost all observations will be used for training and we will hold back the last single observation as a test dataset for which we will make a prediction.

An `ARIMA(5,1,1)` model is trained. This is not the optimal model for this problem, just a good model for demonstration purposes. The trained model is then used to make a prediction by calling the `forecast()` function. The results of the forecast are then printed. The complete example is listed below.

```
# summarize the confidence interval on an ARIMA forecast
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
# load dataset
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# split into train and test sets
X = series.values
X = X.astype('float32')
size = len(X) - 1
train, test = X[0:size], X[size:]
# fit an ARIMA model
model = ARIMA(train, order=(5,1,1))
model_fit = model.fit(disp=False)
# forecast
forecast, stderr, conf = model_fit.forecast()
# summarize forecast and confidence intervals
print('Expected: %.3f' % test[0])
print('Forecast: %.3f' % forecast)
print('Standard Error: %.3f' % stderr)
print('95%% Confidence Interval: %.3f to %.3f' % (conf[0][0], conf[0][1]))
```

Listing 28.1: Calculate the confidence interval of a forecast on the Daily Female Births dataset.

Running the example prints the expected value from the test set followed by the predicted value, standard error, and confidence interval for the forecast.

```
Expected: 50.000
Forecast: 45.878
Standard Error: 6.996
95% Confidence Interval: 32.167 to 59.590
```

Listing 28.2: Example ARIMA forecast and confidence interval on the Daily Female Births dataset.

28.4 Interpreting the Confidence Interval

The `forecast()` function allows the confidence interval to be specified. The `alpha` argument on the `forecast()` function specifies the confidence level. It is set by default to `alpha=0.05`, which is a 95% confidence interval. This is a sensible and widely used confidence interval. An `alpha` of 0.05 means that the ARIMA model will estimate the upper and lower values around the forecast where there is a only a 5% chance that the real value will not be in that range.

Put another way, the 95% confidence interval suggests that there is a high likelihood that the real observation will be within the range. In the above example, the forecast was 45.878. The 95% confidence interval suggested that the real observation was highly likely to fall within the range of values between 32.167 and 59.590. The real observation was 50.0 and was well within this range. We can tighten the range of likely values a few ways:

- We can ask for a range that is narrower but increases the statistical likelihood of a real observation falling outside of the range.
- We can develop a model that has more predictive power and in turn makes more accurate predictions.

Further, the confidence interval is also limited by the assumptions made by the model, such as the distribution of errors made by the model fit a Gaussian distribution with a zero mean value (e.g. white noise). Extending the example above, we can report our forecast with a few different commonly used confidence intervals of 80%, 90%, 95% and 99%. The complete example is listed below.

```
# summarize multiple confidence intervals on an ARIMA forecast
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
# load data
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# split data into train and test setes
X = series.values
X = X.astype('float32')
size = len(X) - 1
train, test = X[0:size], X[size:]
# fit an ARIMA model
model = ARIMA(train, order=(5,1,1))
model_fit = model.fit(disp=False)
# summarize confidence intervals
intervals = [0.2, 0.1, 0.05, 0.01]
for a in intervals:
  forecast, stderr, conf = model_fit.forecast(alpha=a)
  print('%.1f%% Confidence Interval: %.3f between %.3f and %.3f' % ((1-a)*100, forecast,
      conf[0][0], conf[0][1]))
```

Listing 28.3: Calculate multiple different confidence intervals of a forecast on the Daily Female Births dataset.

Running the example prints the forecasts and confidence intervals for each alpha value. We can see that we get the same forecast value each time and an interval that expands as our desire for a *safer* interval increases. We can see that an 80% captures our actual value just fine in this specific case.

```
80.0% Confidence Interval: 45.878 between 36.913 and 54.844
90.0% Confidence Interval: 45.878 between 34.371 and 57.386
95.0% Confidence Interval: 45.878 between 32.167 and 59.590
99.0% Confidence Interval: 45.878 between 27.858 and 63.898
```

Listing 28.4: Example output for multiple difference confidence intervals for an ARIMA forecast on the Daily Female Births dataset.

28.5 Plotting the Confidence Interval

The confidence interval can be plotted directly. The `ARIMAResults` object provides the `plot_predict()` function[1] that can be used to make a forecast and plot the results showing recent observations, the forecast, and confidence interval. As with the `forecast()` function, the confidence interval can be configured by specifying the alpha argument. The default is 0.05 (95% confidence), which is a sensible default. The example below shows the same forecast from above plotted using this function.

```
# plot the confidence intervals for an ARIMA forecast
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
# load data
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# split into train and test sets
X = series.values
X = X.astype('float32')
size = len(X) - 1
train, test = X[0:size], X[size:]
# fit an ARIMA model
model = ARIMA(train, order=(5,1,1))
model_fit = model.fit(disp=False)
# plot some history and the forecast with confidence intervals
model_fit.plot_predict(len(train)-10, len(train)+1)
pyplot.legend(loc='upper left')
pyplot.show()
```

Listing 28.5: Plot the confidence interval for forecasts on the Daily Female Births dataset.

The `plot_predict()` function will plot the observed y values if the prediction interval covers the training data. In this case, we predict the previous 10 days and the next 1 day. This is useful to see the prediction carry on from in sample to out of sample time indexes. This is contracted with the actual observations from the last 10 days. Finally, we can see the confidence interval as a gray cone around the predicted value. This is useful to get a spatial feeling for the range of possible values that an observation in the next time step may take.

[1] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.arima_model.ARIMAResults.plot_predict.html

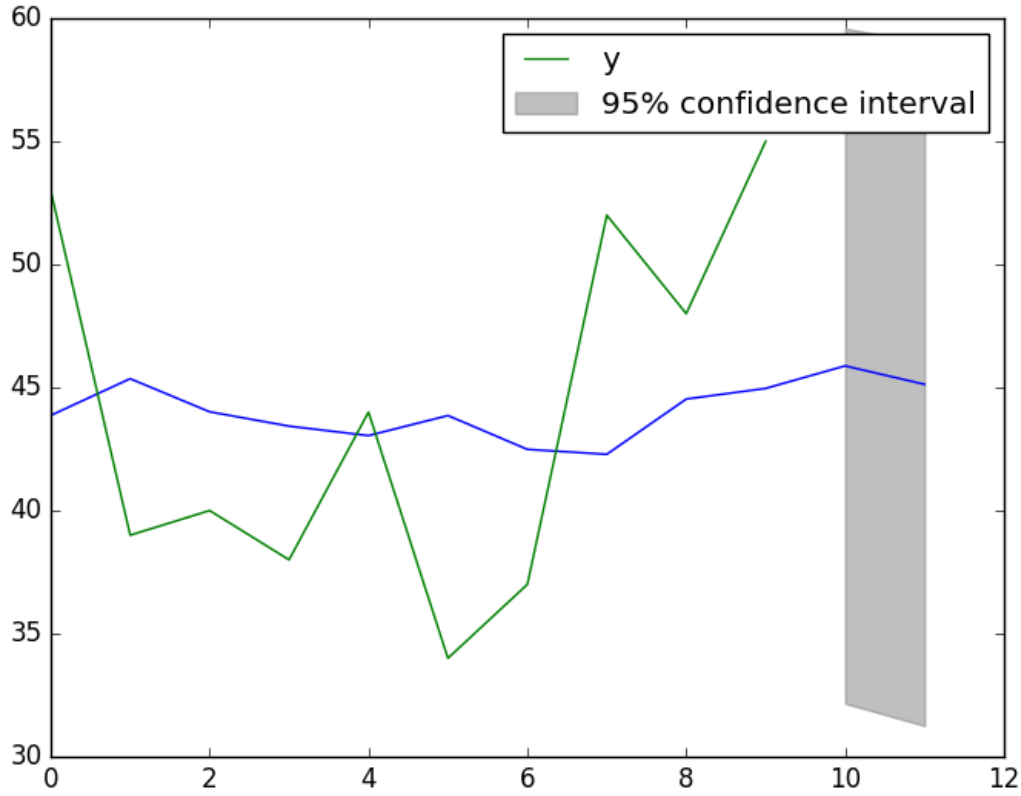

Figure 28.1: Line plot of expected (green) and forecast (blue) with a 95% confidence interval (gray) on the Daily Female Births dataset.

28.6 Summary

In this tutorial, you discovered how to calculate and interpret the confidence interval for a time series forecast with Python. Specifically, you learned:

- How to report forecast diagnostic statistics when making a point forecast.
- How to interpret and configure the confidence interval for a time series forecast.
- How to plot a forecast and confidence interval in the context of recent observations.

28.6.1 Next

This concludes Part V. Next, in Part VI you will discover how to tie together all of the lessons into projects, starting with a project template.

Part VI

Projects

Chapter 29

Time Series Forecast Projects

A time series forecast process is a set of steps or a recipe that leads you from defining your problem through to the outcome of having a time series forecast model or set of predictions. In this lesson, you will discover time series forecast processes that you can use to guide you through your forecast project. After reading this lesson, you will know:

- The 5-Step forecasting task by Hyndman and Athanasopoulos to guide you from problem definition to using and evaluating your forecast model.
- The iterative forecast development process by Shmueli and Lichtendahl to guide you from defining your goal to implementing forecasts.
- Suggestions and tips for working through your own time series forecasting project.

Let's get started.

29.1 5-Step Forecasting Task

The 5 basic steps in a forecasting task are summarized by Hyndman and Athanasopoulos in their book *Forecasting: principles and practice*[1]. These steps are:

1. **Problem Definition**. The careful consideration of who requires the forecast and how the forecast will be used. This is described as the most difficult part of the process, most likely because it is entirely problem specific and subjective.
2. **Gathering Information**. The collection of historical data to analyze and model. This also includes getting access to domain experts and gathering information that can help to best interpret the historical information, and ultimately the forecasts that will be made.
3. **Preliminary Exploratory Analysis**. The use of simple tools, like graphing and summary statistics, to better understand the data. Review plots and summarize and note obvious temporal structures, like trends seasonality, anomalies like missing data, corruption, and outliers, and any other structures that may impact forecasting.

[1] http://www.amazon.com/dp/0987507109?tag=inspiredalgor-20

4. **Choosing and Fitting Models**. Evaluate two, three, or a suite of models of varying types on the problem. Models may be chosen for evaluation based on the assumptions they make and whether the dataset conforms. Models are configured and fit to the historical data.

5. **Using and Evaluating a Forecasting Model**. The model is used to make forecasts and the performance of those forecasts is evaluated and skill of the models estimated. This may involve back-testing with historical data or waiting for new observations to become available for comparison.

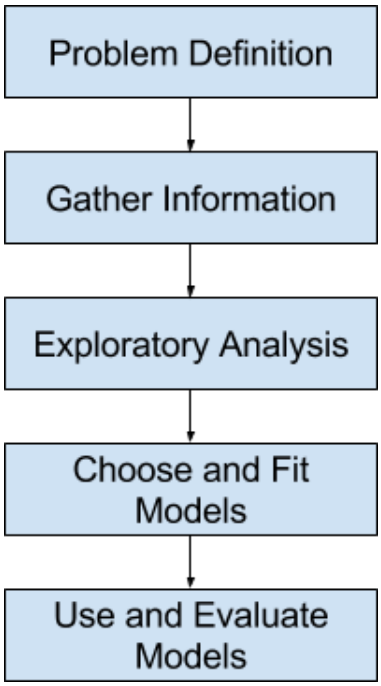

Figure 29.1: 5-Step Forecasting Task.

This 5-step process provides a strong overview from starting off with an idea or problem statement and leading through to a model that can be used to make predictions. The focus of the process is on understanding the problem and fitting a good model.

> Each model is itself an artificial construct that is based on a set of assumptions (explicit and implicit) and usually involves one or more parameters which must be "fitted" using the known historical data.
>
> — Page 22, *Forecasting: principles and practice.*

29.2 Iterative Forecast Development Process

The authors Shmueli and Lichtendahl in their book *Practical Time Series Forecasting with R: A Hands-On Guide*[2] suggest an 8-step process. This process extends beyond the development of a

[2] http://www.amazon.com/dp/0997847913?tag=inspiredalgor-20

model and making forecasts and involves iterative loops. Their process can be summarized as follows:

1. Define Goal.
2. Get Data.
3. Explore and Visualize Series.
4. Pre-Process Data.
5. Partition Series.
6. Apply Forecasting Method/s.
7. Evaluate and Compare Performance.
8. Implement Forecasts/Systems.

Below are the iterative loops within the process:

- *Explore and Visualize Series* ⇒ *Get Data*. Data exploration can lead to questions that require access to new data.
- *Evaluate and Compare Performance* ⇒ *Apply Forecasting Method/s*. The evaluation of models may raise questions or ideas for new methods or new method configurations to try.

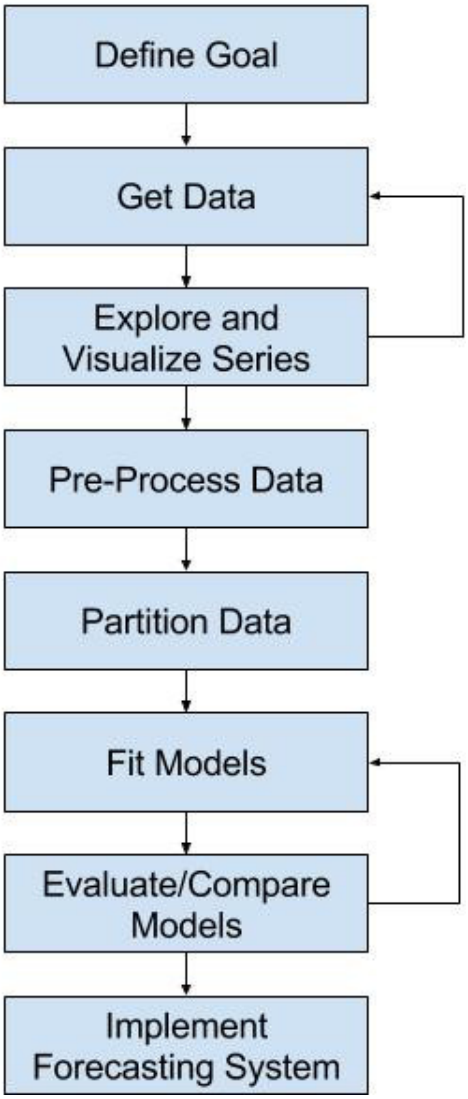

Figure 29.2: Iterative Forecast Development Process.

The process is more focused on the ongoing development and refinement of one or more models on the problem until an acceptable level of performance is achieved. This process can continue where models are revised and updated as new data and new insights are made available.

> Of course, the process does not end once forecasts are generated, because forecasting is typically an ongoing goal. Hence, forecast accuracy is monitored and sometimes forecasting method is adapted or changed to accommodate changes in the goal or the data over time
>
> — Page 16, *Practical Time Series Forecasting with R: A Hands-On Guide.*

29.3 Suggestions and Tips

This section lists 10 suggestions and tips to consider when working through your time series forecasting project. The thrust of these suggestions is centered on the premise that you cannot

know what will work, let alone which methods will work well on your problem beforehand. And that the best source of knowledge on a forecasting project comes from the results of trial and error with real historical data.

- Select or devise a time series forecast process that is tailored to your project, tools, team, and level of expertise.
- Write down all assumptions and questions you have during analysis and forecasting work, then revisit them later and seek to answer them with small experiments on historical data.
- Review a large number of plots of your data at different time scales, zooms, and transforms of observations in an effort to help make exploitable structures present in the data obvious to you.
- Develop a robust test harness for evaluating models using a meaningful performance measure and a reliable test strategy, such as walk-forward validation (rolling forecast).
- Start with simple naive forecast models to provide a baseline of performance for more sophisticated methods to improve upon.
- Create a large number of perspectives or views on your time series data, including a suite of automated transforms, and evaluate each with one or a suite of models in order to help automatically discover non-intuitive representations and model combinations that result in good predictions for your problem.
- Try a suite of models of differing types on your problem, from simple to more advanced approaches.
- Try a suite of configurations for a given problem, including configurations that have worked well on other problems.
- Try automated hyperparameter optimization methods for models to flush out a suite of well-performing models as well as non-intuitive model configurations that you would not have tried manually.
- Devise automated tests of performance and skill for ongoing predictions to help to automatically determine if and when a model has become stale and requires review or retraining.

29.4 Summary

In this lesson, you discovered processes that you can use to work through time series forecasting problems. Specifically, you learned:

- The 5 steps of working through a time series forecast task by Hyndman and Athanasopoulos.
- The 8 step iterative process of defining a goal and implementing a forecast system by Shmueli and Lichtendahl.
- The 10 suggestions and practical tips to consider when working through your time series forecasting project.

29.4.1 Next

In the next project you will develop models to forecast Monthly Armed Robberies in Boston.

Chapter 30

Project: Monthly Armed Robberies in Boston

Time series forecasting is a process, and the only way to get good forecasts is to practice this process. In this tutorial, you will discover how to forecast the number of monthly armed robberies in Boston with Python. Working through this tutorial will provide you with a framework for the steps and the tools for working through your own time series forecasting problems. After completing this tutorial, you will know:

- How to check your Python environment and carefully define a time series forecasting problem.
- How to create a test harness for evaluating models, develop a baseline forecast, and better understand your problem with the tools of time series analysis.
- How to develop an autoregressive integrated moving average model, save it to file, and later load it to make predictions for new time steps.

Let's get started.

30.1 Overview

In this tutorial, we will work through a time series forecasting project from end-to-end, from downloading the dataset and defining the problem to training a final model and making predictions. This project is not exhaustive, but shows how you can get good results quickly by working through a time series forecasting problem systematically. The steps of this project that we will work through are as follows:

1. Problem Description.
2. Test Harness.
3. Persistence.
4. Data Analysis.
5. ARIMA Models.

6. Model Validation.

This will provide a template for working through a time series prediction problem that you can use on your own dataset.

30.2 Problem Description

The problem is to predict the number of monthly armed robberies in Boston, USA. The dataset provides the number of monthly armed robberies in Boston from January 1966 to October 1975, or just under 10 years of data. The values are a count and there are 118 observations. The dataset is credited to McCleary and Hay (1980). Below is a sample of the first few rows of the dataset.

```
"Month","Robberies"
"1966-01",41
"1966-02",39
"1966-03",50
"1966-04",40
"1966-05",43
```

Listing 30.1: Sample of the first few rows of the dataset.

You can learn more about this dataset and download it directly from DataMarket[1]. Download the dataset as a CSV file and place it in your current working directory with the filename `robberies.csv`.

30.3 Test Harness

We must develop a test harness to investigate the data and evaluate candidate models. This involves two steps:

1. Defining a Validation Dataset.
2. Developing a Method for Model Evaluation.

30.3.1 Validation Dataset

The dataset is not current. This means that we cannot easily collect updated data to validate the model. Therefore we will pretend that it is October 1974 and withhold the last one year of data from analysis and model selection. This final year of data will be used to validate the final model. The code below will load the dataset as a Pandas `Series` and split into two, one for model development (`dataset.csv`) and the other for validation (`validation.csv`).

```
# split into a training and validation dataset
from pandas import read_csv
series = read_csv('robberies.csv', header=0, index_col=0, parse_dates=True, squeeze=True)
split_point = len(series) - 12
dataset, validation = series[0:split_point], series[split_point:]
```

[1] https://goo.gl/pKxME8

```
print('Dataset %d, Validation %d' % (len(dataset), len(validation)))
dataset.to_csv('dataset.csv')
validation.to_csv('validation.csv')
```

Listing 30.2: Split the raw data into training and validation datasets.

Running the example creates two files and prints the number of observations in each.

```
Dataset 106, Validation 12
```

Listing 30.3: Example output splitting the raw data into train and validation datasets.

The specific contents of these files are:

- `dataset.csv`: Observations from January 1966 to October 1974 (106 observations)
- `validation.csv`: Observations from November 1974 to October 1975 (12 observations)

The validation dataset is 10% of the original dataset. Note that the saved datasets do not have a header line, therefore we do not need to cater to this when working with these files later.

30.3.2 Model Evaluation

Model evaluation will only be performed on the data in `dataset.csv` prepared in the previous section. Model evaluation involves two elements:

1. Performance Measure.
2. Test Strategy.

Performance Measure

The observations are a count of robberies. We will evaluate the performance of predictions using the root mean squared error (RMSE). This will give more weight to predictions that are grossly wrong and will have the same units as the original data. Any transforms to the data must be reversed before the RMSE is calculated and reported to make the performance between different methods directly comparable.

We can calculate the RMSE using the helper function from the scikit-learn library `mean_squared_error()` that calculates the mean squared error between a list of expected values (the test set) and the list of predictions. We can then take the square root of this value to give us an RMSE score. For example:

```
from sklearn.metrics import mean_squared_error
from math import sqrt
...
test = ...
predictions = ...
mse = mean_squared_error(test, predictions)
rmse = sqrt(mse)
print('RMSE: %.3f' % rmse)
```

Listing 30.4: Example of evaluating predictions using RMSE.

Test Strategy

Candidate models will be evaluated using walk-forward validation. This is because a rolling-forecast type model is required from the problem definition. This is where one-step forecasts are needed given all available data. The walk-forward validation will work as follows:

1. The first 50% of the dataset will be held back to train the model.
2. The remaining 50% of the dataset will be iterated and test the model.
3. For each step in the test dataset:
   (a) A model will be trained.
   (b) A one-step prediction made and the prediction stored for later evaluation.
   (c) The actual observation from the test dataset will be added to the training dataset for the next iteration.
4. The predictions made during the iteration of the test dataset will be evaluated and an RMSE score reported.

Given the small size of the data, we will allow a model to be re-trained given all available data prior to each prediction. We can write the code for the test harness using simple NumPy and Python code. Firstly, we can split the dataset into train and test sets directly. We're careful to always convert a loaded dataset to `float32` in case the loaded data still has some String or Integer data types.

```
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
```

Listing 30.5: Example of splitting data into train and test sets.

Next, we can iterate over the time steps in the test dataset. The train dataset is stored in a Python list as we need to easily append a new observation each iteration and NumPy array concatenation feels like overkill. The prediction made by the model is called `yhat` for convention, as the outcome or observation is referred to as `y` and `yhat` (a `y` with a mark above) is the mathematical notation for the prediction of the `y` variable. The prediction and observation are printed each observation for a sanity check prediction in case there are issues with the model.

```
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # predict
  yhat = ...
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
```

Listing 30.6: Example of walk-forward validation.

30.4 Persistence

The first step before getting bogged down in data analysis and modeling is to establish a baseline of performance. This will provide both a template for evaluating models using the proposed test harness and a performance measure by which all more elaborate predictive models can be compared. The baseline prediction for time series forecasting is called the naive forecast, or persistence. This is where the observation from the previous time step is used as the prediction for the observation at the next time step. We can plug this directly into the test harness defined in the previous section. The complete code listing is provided below.

```
# evaluate a persistence model
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from math import sqrt
# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # predict
  yhat = history[-1]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
```

Listing 30.7: Persistence forecast model on the Boston Robberies dataset.

Running the test harness prints the prediction and observation for each iteration of the test dataset. The example ends by printing the RMSE for the model. In this case, we can see that the persistence model achieved an RMSE of 51.844. This means that on average, the model was wrong by about 51 robberies for each prediction made.

```
...
>Predicted=241.000, Expected=287
>Predicted=287.000, Expected=355
>Predicted=355.000, Expected=460
>Predicted=460.000, Expected=364
>Predicted=364.000, Expected=487
RMSE: 51.844
```

Listing 30.8: Example output of the persistence forecast model on the Boston Robberies dataset.

We now have a baseline prediction method and performance; now we can start digging into our data.

30.5 Data Analysis

We can use summary statistics and plots of the data to quickly learn more about the structure of the prediction problem. In this section, we will look at the data from four perspectives:

1. Summary Statistics.
2. Line Plot.
3. Density Plots.
4. Box and Whisker Plot.

30.5.1 Summary Statistics

Open the data `dataset.csv` file and/or the original `robberies.csv` file in a text editor and look at the data. A quick check suggests that there are no obviously missing observations. We may have noticed this earlier if we tried to force the series to floating point values and values like `NaN` or `?` were in the data. Summary statistics provide a quick look at the limits of observed values. It can help to get a quick idea of what we are working with. The example below calculates and prints summary statistics for the time series.

```
# summary statistics of time series
from pandas import read_csv
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
print(series.describe())
```

Listing 30.9: Summary statistics of the Boston Robberies dataset.

Running the example provides a number of summary statistics to review. Some observations from these statistics include:

- The number of observations (count) matches our expectation, meaning we are handling the data correctly.
- The mean is about 173, which we might consider our level in this series.
- The standard deviation (average spread from the mean) is relatively large at 112 robberies.
- The percentiles along with the standard deviation do suggest a large spread to the data.

```
count   106.000000
mean    173.103774
std     112.231133
min      29.000000
25%      74.750000
50%     144.500000
75%     271.750000
max     487.000000
```

Listing 30.10: Example output of summary statistics on the Boston Robberies dataset.

The large spread in this series will likely make highly accurate predictions difficult if it is caused by random fluctuation (e.g. not systematic).

30.5.2 Line Plot

A line plot of a time series can provide a lot of insight into the problem. The example below creates and shows a line plot of the dataset.

```
# line plots of time series
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
series.plot()
pyplot.show()
```

Listing 30.11: Line plot of the Boston Robberies dataset.

Run the example and review the plot. Note any obvious temporal structures in the series. Some observations from the plot include:

- There is an increasing trend of robberies over time.
- There do not appear to be any obvious outliers.
- There are relatively large fluctuations from year to year, up and down.
- The fluctuations at later years appear larger than fluctuations at earlier years.
- The trend means the dataset is almost certainly non-stationary and the apparent change in fluctuation may also contribute.

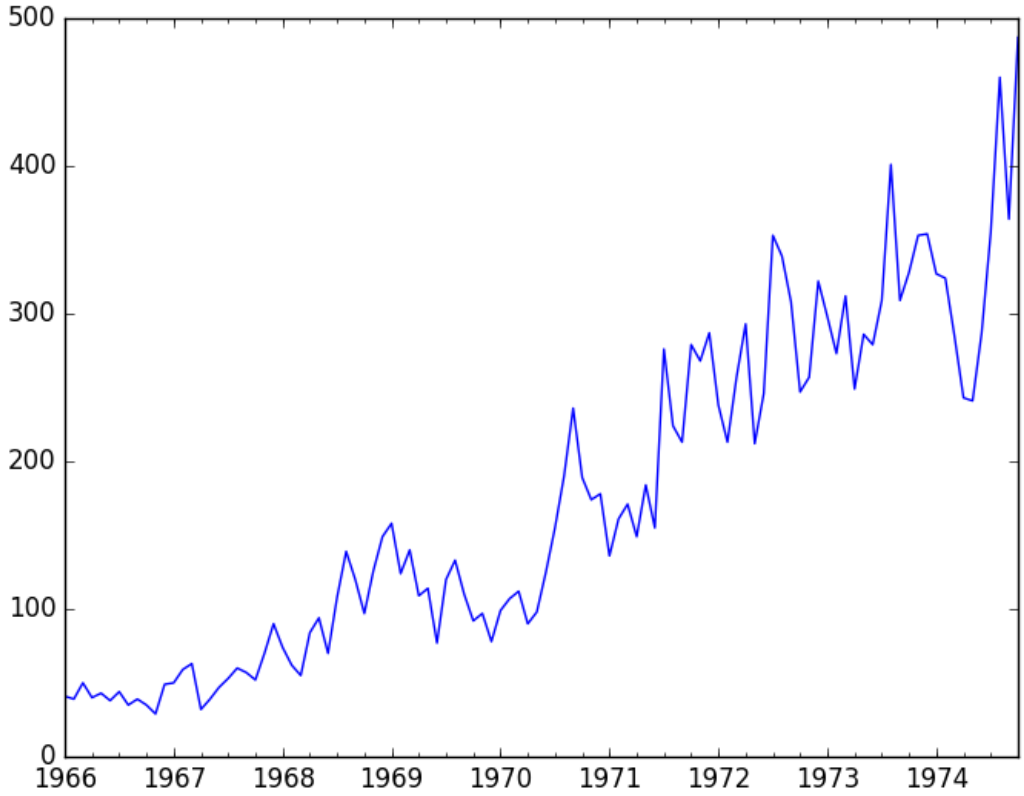

Figure 30.1: Line plot of the training set for the Boston Robberies dataset.

These simple observations suggest we may see benefit in modeling the trend and removing it from the time series. Alternately, we could use differencing to make the series stationary for modeling. We may even need two levels of differencing if there is a growth trend in the fluctuations in later years.

30.5.3 Density Plot

Reviewing plots of the density of observations can provide further insight into the structure of the data. The example below creates a histogram and density plot of the observations without any temporal structure.

```
# density plots of time series
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
pyplot.figure(1)
pyplot.subplot(211)
series.hist()
pyplot.subplot(212)
series.plot(kind='kde')
pyplot.show()
```

Listing 30.12: Density plots of the Boston Robberies dataset.

Run the example and review the plots. Some observations from the plots include:

- The distribution is not Gaussian.
- The distribution is left shifted and may be exponential or a double Gaussian.

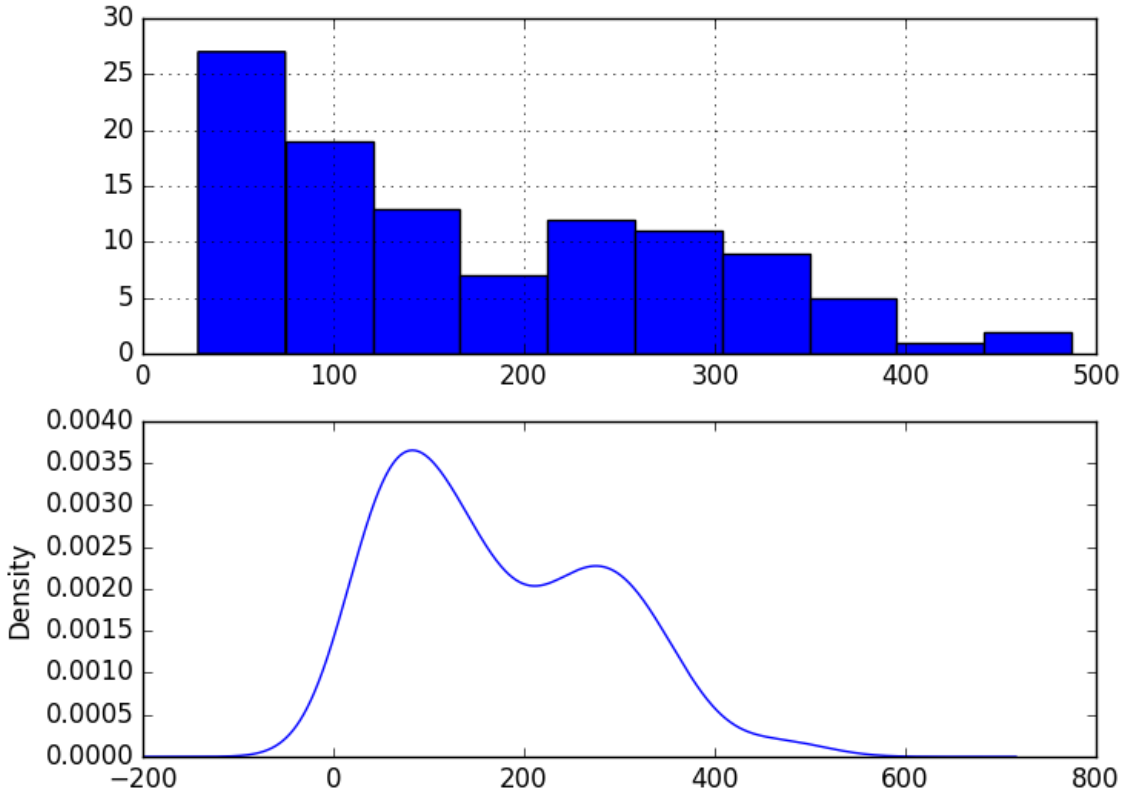

Figure 30.2: Density plots of the training set for the Boston Robberies dataset.

We may see some benefit in exploring some power transforms of the data prior to modeling.

30.5.4 Box and Whisker Plots

We can group the monthly data by year and get an idea of the spread of observations for each year and how this may be changing. We do expect to see some trend (increasing mean or median), but it may be interesting to see how the rest of the distribution may be changing. The example below groups the observations by year and creates one box and whisker plot for each year of observations. The last year (1974) only contains 10 months and may not be a useful comparison with the other 12 months of observations in the other years. Therefore only data between 1966 and 1973 was plotted.

```
# boxplots of time series
from pandas import read_csv
from pandas import DataFrame
from pandas import Grouper
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
print(series)
groups = series['1966':'1973'].groupby(Grouper(freq='A'))
years = DataFrame()
for name, group in groups:
  years[name.year] = group.values
years.boxplot()
pyplot.show()
```

Listing 30.13: Box and whisker plots of the Boston Robberies dataset.

Running the example creates 8 box and whisker plots side-by-side, one for each of the 8 years of selected data. Some observations from reviewing the plot include:

- The median values for each year (red line) show a trend that may not be linear.
- The spread, or middle 50% of the data (blue boxes), differ, but perhaps not consistently over time.
- The earlier years, perhaps first 2, are quite different from the rest of the dataset.

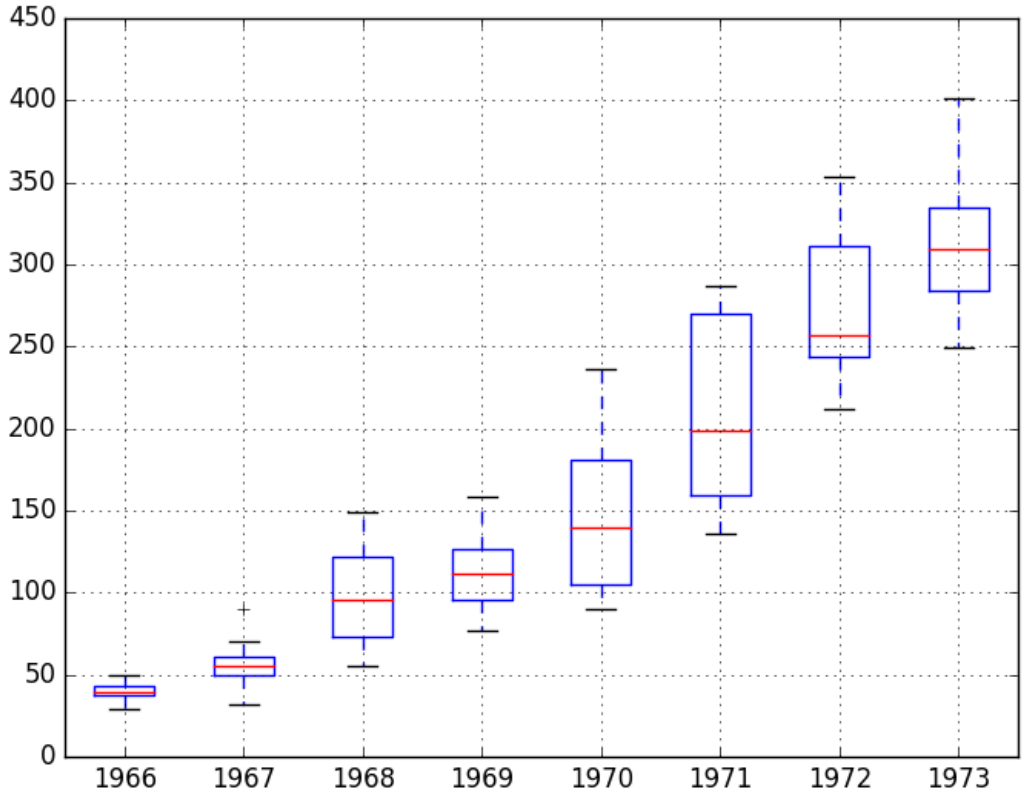

Figure 30.3: Monthly box and whisker plots of the training set for the Boston Robberies dataset.

The observations suggest that the year-to-year fluctuations may not be systematic and hard to model. They also suggest that there may be some benefit in clipping the first two years of data from modeling if it is indeed quite different. This yearly view of the data is an interesting avenue and could be pursued further by looking at summary statistics from year-to-year and changes in summary stats from year-to-year. Next, we can start looking at predictive models of the series.

30.6 ARIMA Models

In this section, we will develop Autoregressive Integrated Moving Average, or ARIMA, models for the problem. We will approach this in four steps:

1. Developing a manually configured ARIMA model.
2. Using a grid search of ARIMA to find an optimized model.
3. Analysis of forecast residual errors to evaluate any bias in the model.
4. Explore improvements to the model using power transforms.

30.6.1 Manually Configured ARIMA

Nonseasonal `ARIMA(p,d,q)` requires three parameters and is traditionally configured manually. Analysis of the time series data assumes that we are working with a stationary time series. The time series is almost certainly non-stationary. We can make it stationary by first differencing the series and using a statistical test to confirm that the result is stationary. The example below creates a stationary version of the series and saves it to file `stationary.csv`.

```
# statistical test for the stationarity of the time series
from pandas import read_csv
from pandas import Series
from statsmodels.tsa.stattools import adfuller

# create a differenced time series
def difference(dataset):
  diff = list()
  for i in range(1, len(dataset)):
    value = dataset[i] - dataset[i - 1]
    diff.append(value)
  return Series(diff)

series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
X = series.values
# difference data
stationary = difference(X)
stationary.index = series.index[1:]
# check if stationary
result = adfuller(stationary)
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))
# save
stationary.to_csv('stationary.csv')
```

Listing 30.14: Create a stationary version of the Boston Robberies dataset.

Running the example outputs the result of a statistical significance test of whether the 1-lag differenced series is stationary. Specifically, the augmented Dickey-Fuller test. The results show that the test statistic value -3.980946 is smaller than the critical value at 5% of -2.893. This suggests that we can reject the null hypothesis with a significance level of less than 5% (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 1-lag differenced time series is stationary or does not have time-dependent structure.

```
ADF Statistic: -3.980946
p-value: 0.001514
Critical Values:
  1%: -3.503
  5%: -2.893
  10%: -2.584
```

Listing 30.15: Example output of stationarity test on the differenced Boston Robberies dataset.

This suggests that at least one level of differencing is required. The `d` parameter in our ARIMA model should at least be a value of 1. The next step is to select the lag values for the Autoregression (AR) and Moving Average (MA) parameters, `p` and `q` respectively. We can do this by reviewing Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots. The example below creates ACF and PACF plots for the series.

```
# ACF and PACF plots of time series
from pandas import read_csv
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
pyplot.figure()
pyplot.subplot(211)
plot_acf(series, ax=pyplot.gca())
pyplot.subplot(212)
plot_pacf(series, ax=pyplot.gca())
pyplot.show()
```

Listing 30.16: Create ACF and PACF plots of the Boston Robberies dataset.

Run the example and review the plots for insights into how to set the `p` and `q` variables for the ARIMA model. Below are some observations from the plots.

- The ACF shows a significant lag for 10-11 months.
- The PACF shows a significant lag for perhaps 2 months.
- Both the ACF and PACF show a drop-off at the same point, perhaps suggesting a mix of AR and MA.

A good starting point for the `p` and `q` values is 1 or 2.

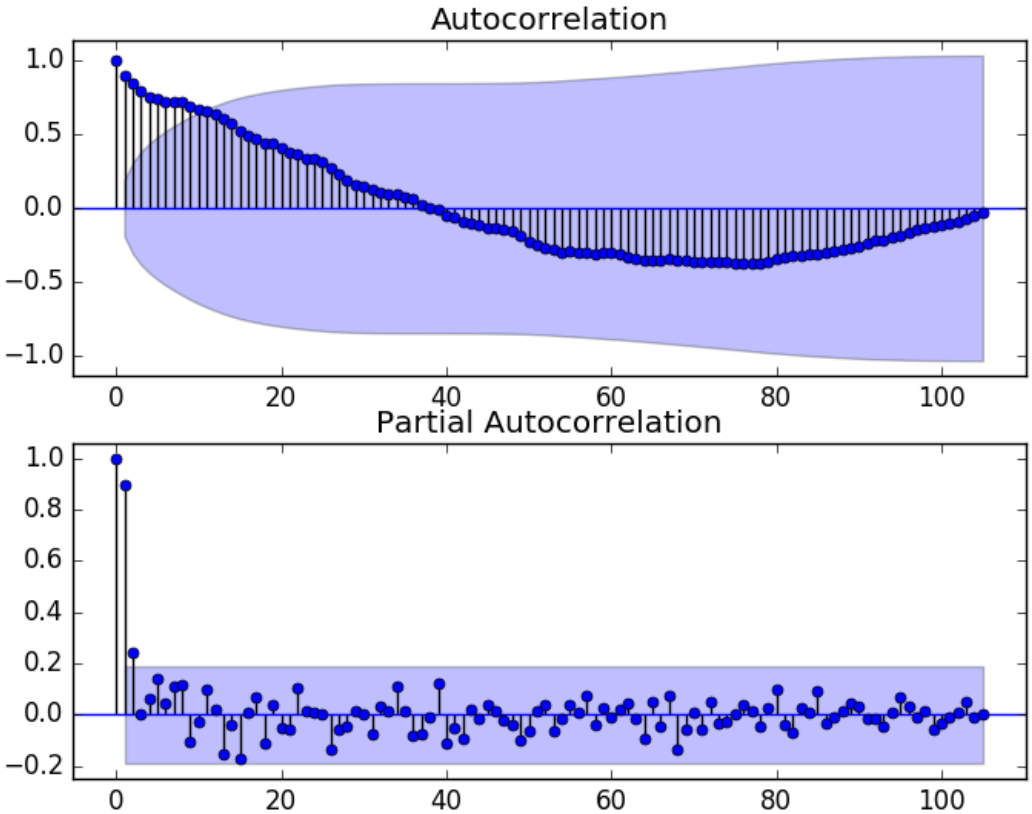

Figure 30.4: ACF and PACF plots of the training set for the Boston Robberies dataset.

This quick analysis suggests an `ARIMA(11,1,2)` on the raw data may be a good starting point. Experimentation shows that this configuration of ARIMA does not converge and results in errors by the underlying library, as do similarly large AR values. Some experimentation shows that the model does not appear to be stable, with non-zero AR and MA orders defined at the same time. The model can be simplified to `ARIMA(0,1,2)`. The example below demonstrates the performance of this ARIMA model on the test harness.

```
# evaluate manually configured ARIMA model
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA
from math import sqrt
# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
```

```
  # predict
  model = ARIMA(history, order=(0,1,2))
  model_fit = model.fit(disp=0)
  yhat = model_fit.forecast()[0]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
```

Listing 30.17: Manual ARIMA model for the Boston Robberies dataset.

Running this example results in an RMSE of 49.821, which is lower than the persistence model.

```
...
>Predicted=280.614, Expected=287
>Predicted=302.079, Expected=355
>Predicted=340.210, Expected=460
>Predicted=405.172, Expected=364
>Predicted=333.755, Expected=487
RMSE: 49.821
```

Listing 30.18: Example output of a manual ARIMA model for the Boston Robberies dataset.

This is a good start, but we may be able to get improved results with a better configured ARIMA model.

30.6.2 Grid Search ARIMA Hyperparameters

Many ARIMA configurations are unstable on this dataset, but there may be other hyperparameters that result in a well-performing model. In this section, we will search values of `p`, `d`, and `q` for combinations that do not result in error, and find the combination that results in the best performance. We will use a grid search to explore all combinations in a subset of integer values. Specifically, we will search all combinations of the following parameters:

- `p`: 0 to 12.
- `d`: 0 to 3.
- `q`: 0 to 12.

This is $(13 \times 4 \times 13)$, or 676, runs of the test harness and will take some time to execute. The complete worked example with the grid search version of the test harness is listed below.

```
# grid search ARIMA parameters for time series
import warnings
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
```

```
# evaluate an ARIMA model for a given order (p,d,q) and return RMSE
def evaluate_arima_model(X, arima_order):
  # prepare training dataset
  X = X.astype('float32')
  train_size = int(len(X) * 0.50)
  train, test = X[0:train_size], X[train_size:]
  history = [x for x in train]
  # make predictions
  predictions = list()
  for t in range(len(test)):
    model = ARIMA(history, order=arima_order)
    model_fit = model.fit(disp=0)
    yhat = model_fit.forecast()[0]
    predictions.append(yhat)
    history.append(test[t])
  # calculate out of sample error
  rmse = sqrt(mean_squared_error(test, predictions))
  return rmse

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
  dataset = dataset.astype('float32')
  best_score, best_cfg = float("inf"), None
  for p in p_values:
    for d in d_values:
      for q in q_values:
        order = (p,d,q)
        try:
          rmse = evaluate_arima_model(dataset, order)
          if rmse < best_score:
            best_score, best_cfg = rmse, order
          print('ARIMA%s RMSE=%.3f' % (order,rmse))
        except:
          continue
  print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))

# load dataset
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# evaluate parameters
p_values = range(0,13)
d_values = range(0, 4)
q_values = range(0, 13)
warnings.filterwarnings("ignore")
evaluate_models(series.values, p_values, d_values, q_values)
```

Listing 30.19: Grid Search ARIMA models for the Boston Robberies dataset.

Running the example runs through all combinations and reports the results on those that converge without error. The example takes a little less than 2 hours to run on modern hardware. The results show that the best configuration discovered was `ARIMA(0,1,2)`; coincidently, that was demonstrated in the previous section.

```
...
ARIMA(6, 1, 0) RMSE=52.437
ARIMA(6, 2, 0) RMSE=58.307
ARIMA(7, 1, 0) RMSE=51.104
```

```
ARIMA(7, 1, 1) RMSE=52.340
ARIMA(8, 1, 0) RMSE=51.759
Best ARIMA(0, 1, 2) RMSE=49.821
```

Listing 30.20: Example output of grid searching ARIMA models for the Boston Robberies dataset.

30.6.3 Review Residual Errors

A good final check of a model is to review residual forecast errors. Ideally, the distribution of residual errors should be a Gaussian with a zero mean. We can check this by plotting the residuals with a histogram and density plots. The example below calculates the residual errors for predictions on the test set and creates these density plots.

```
# plot residual errors for ARIMA model
from pandas import read_csv
from pandas import DataFrame
from statsmodels.tsa.arima_model import ARIMA
from matplotlib import pyplot
# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # predict
  model = ARIMA(history, order=(0,1,2))
  model_fit = model.fit(disp=0)
  yhat = model_fit.forecast()[0]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
# errors
residuals = [test[i]-predictions[i] for i in range(len(test))]
residuals = DataFrame(residuals)
pyplot.figure()
pyplot.subplot(211)
residuals.hist(ax=pyplot.gca())
pyplot.subplot(212)
residuals.plot(kind='kde', ax=pyplot.gca())
pyplot.show()
```

Listing 30.21: Plot density of residual errors from ARIMA model on the Boston Robberies dataset.

Running the example creates the two plots. The graphs suggest a Gaussian-like distribution with a longer right tail. This is perhaps a sign that the predictions are biased, and in this case that perhaps a power-based transform of the raw data before modeling might be useful.

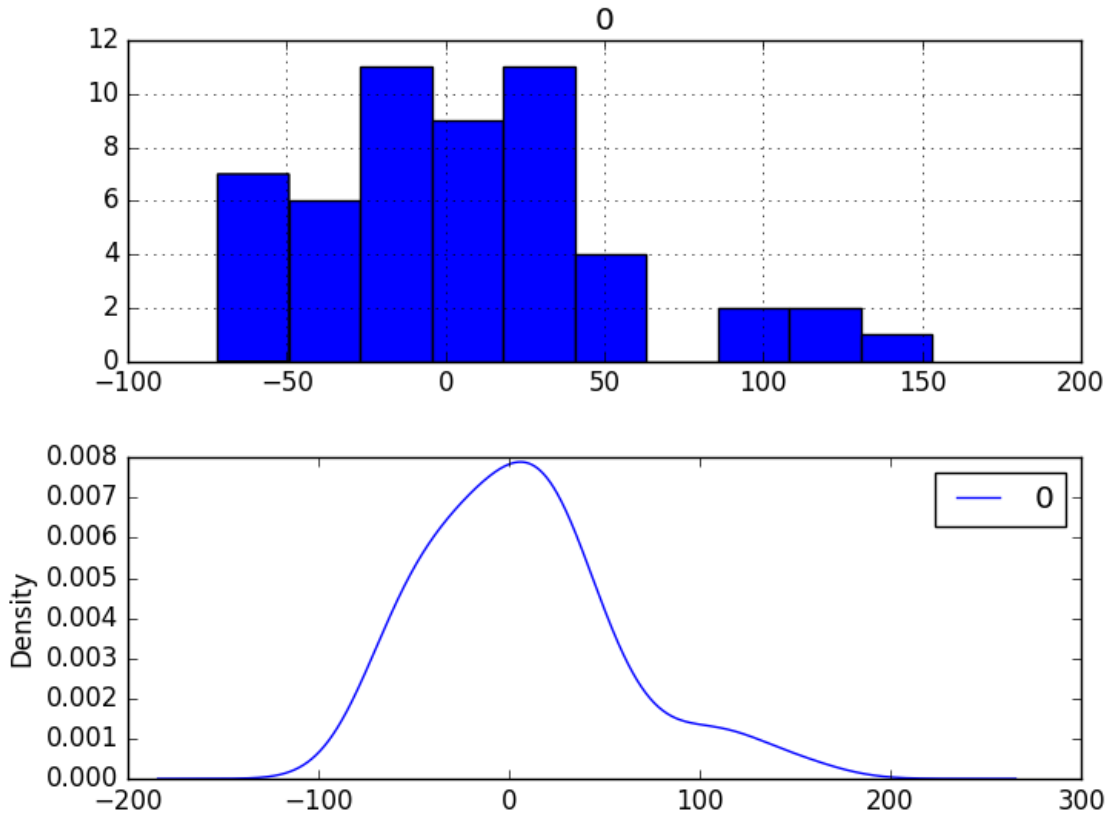

Figure 30.5: Density plots of the residual errors from ARIMA models on the Boston Robberies dataset.

It is also a good idea to check the time series of the residual errors for any type of autocorrelation. If present, it would suggest that the model has more opportunity to model the temporal structure in the data. The example below re-calculates the residual errors and creates ACF and PACF plots to check for any significant autocorrelation.

```
# ACF and PACF plots of forecast residual errors
from pandas import read_csv
from pandas import DataFrame
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from matplotlib import pyplot
# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
```

```
for i in range(len(test)):
  # predict
  model = ARIMA(history, order=(0,1,2))
  model_fit = model.fit(disp=0)
  yhat = model_fit.forecast()[0]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
# errors
residuals = [test[i]-predictions[i] for i in range(len(test))]
residuals = DataFrame(residuals)
pyplot.figure()
pyplot.subplot(211)
plot_acf(residuals, ax=pyplot.gca())
pyplot.subplot(212)
plot_pacf(residuals, ax=pyplot.gca())
pyplot.show()
```

Listing 30.22: ACF and PACF plots of residual errors from ARIMA model on the Boston Robberies dataset.

The results suggest that what little autocorrelation is present in the time series has been captured by the model.

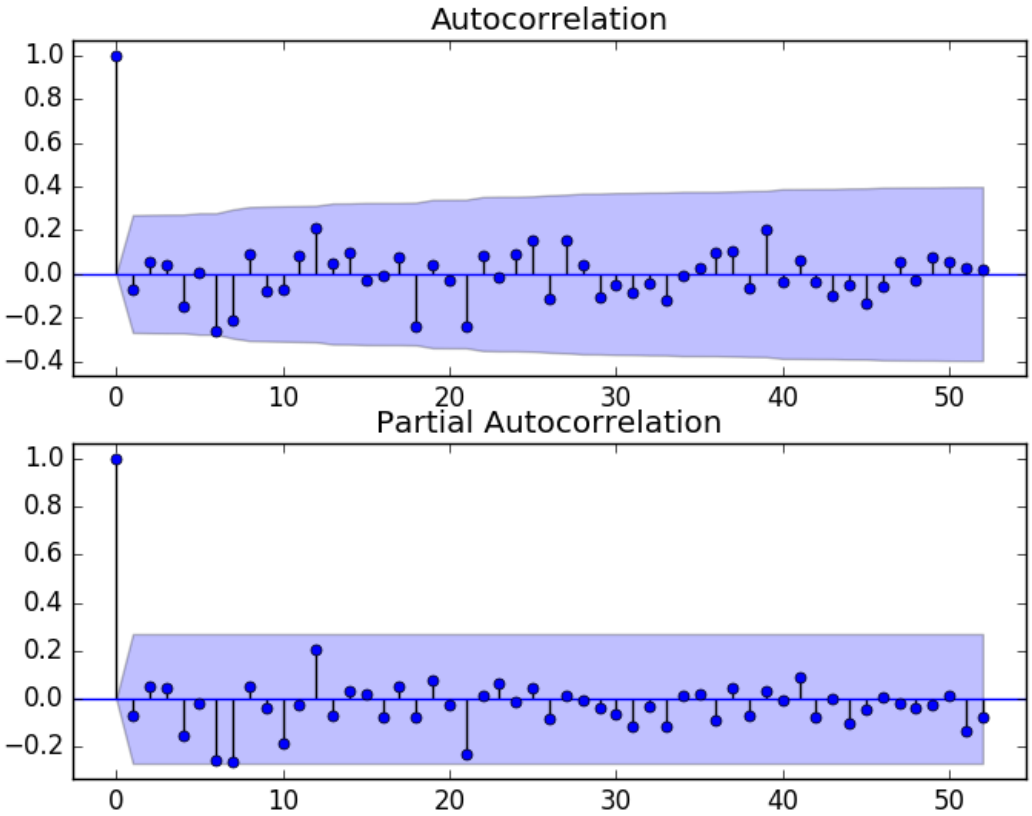

Figure 30.6: ACF and PACF plots of the residual errors from ARIMA models on the Boston Robberies dataset.

30.6.4 Box-Cox Transformed Dataset

The Box-Cox transform is a method that is able to evaluate a suite of power transforms, including, but not limited to, log, square root, and reciprocal transforms of the data. The example below performs a log transform of the data and generates some plots to review the effect on the time series.

```
# plots of box-cox transformed dataset
from pandas import read_csv
from scipy.stats import boxcox
from matplotlib import pyplot
from statsmodels.graphics.gofplots import qqplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
X = series.values
transformed, lam = boxcox(X)
print('Lambda: %f' % lam)
pyplot.figure(1)
# line plot
pyplot.subplot(311)
pyplot.plot(transformed)
# histogram
pyplot.subplot(312)
```

```
pyplot.hist(transformed)
# q-q plot
pyplot.subplot(313)
qqplot(transformed, line='r', ax=pyplot.gca())
pyplot.show()
```

Listing 30.23: Box-Cox transforms and plots on the Boston Robberies dataset.

Running the example creates three graphs: a line chart of the transformed time series, a histogram showing the distribution of transformed values, and a Q-Q plot showing how the distribution of values compared to an idealized Gaussian distribution. Some observations from these plots are follows:

- The large fluctuations have been removed from the line plot of the time series.
- The histogram shows a flatter or more uniform (well behaved) distribution of values.
- The Q-Q plot is reasonable, but still not a perfect fit for a Gaussian distribution.

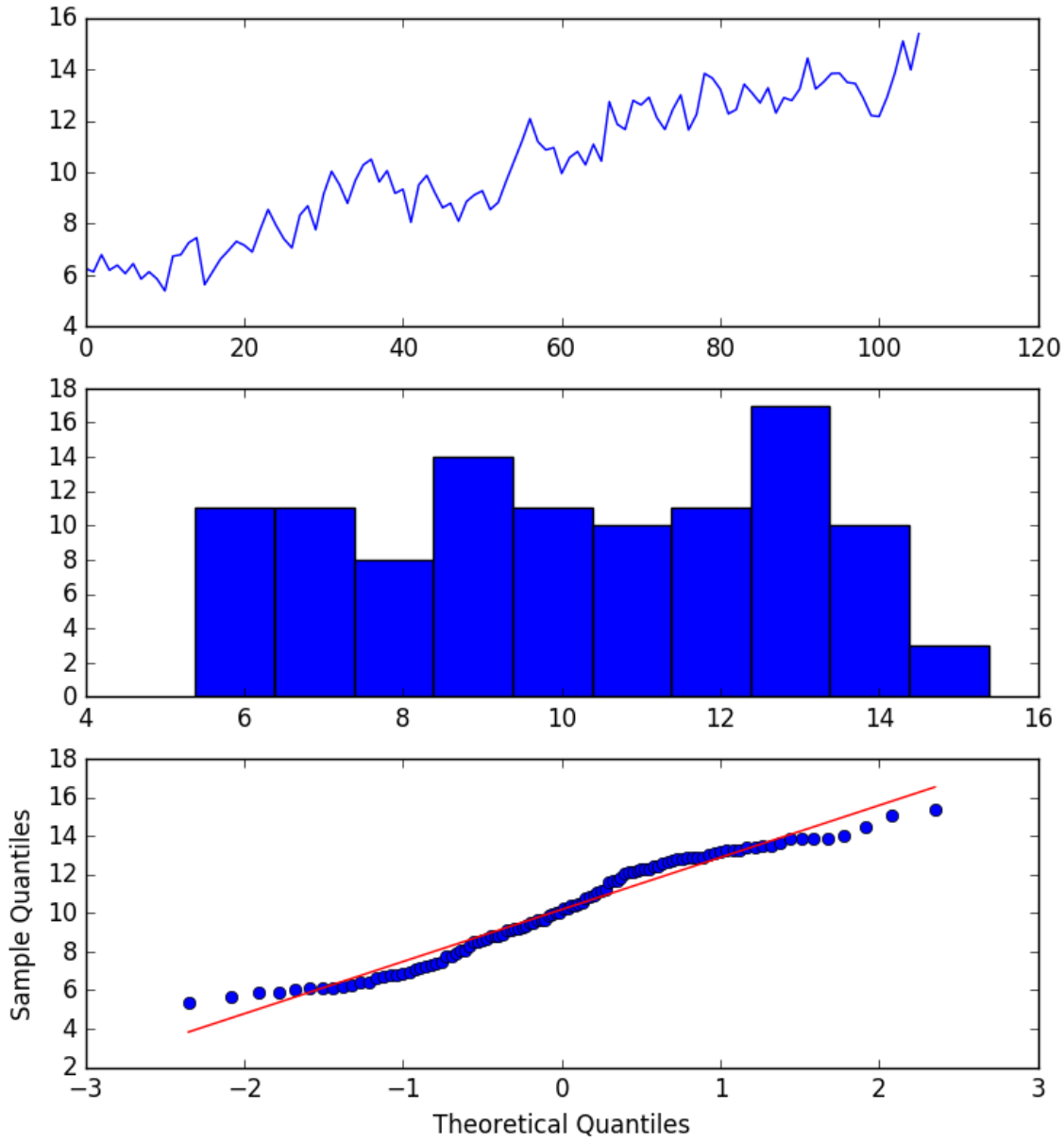

Figure 30.7: Line, Density and Q-Q Plots of the Box-Cox transformed version of the Boston Robberies dataset.

Undoubtedly, the Box-Cox transform has done something to the time series and may be useful. Before proceeding to test the ARIMA model with the transformed data, we must have a way to reverse the transform in order to convert predictions made with a model trained on the transformed data back into the original scale. The `boxcox()` function used in the example finds an ideal `lambda` value by optimizing a cost function. The `lambda` is used in the following

function to transform the data:

$$\begin{aligned} transform &= log(x),\ \text{IF } lambda = 0 \\ transform &= \frac{x^{lambda} - 1}{lambda},\ \text{IF } lambda \neq 0 \end{aligned} \tag{30.1}$$

This transform function can be reversed directly, as follows:

$$\begin{aligned} x &= exp(transform),\ \text{IF } lambda = 0 \\ x &= exp(\frac{log(lambda \times transform + 1)}{lambda}),\ \text{IF } lambda \neq 0 \end{aligned} \tag{30.2}$$

This inverse Box-Cox transform function can be implemented in Python as follows:

```
# invert Box-Cox transform
from math import log
from math import exp
def boxcox_inverse(value, lam):
  if lam == 0:
    return exp(value)
  return exp(log(lam * value + 1) / lam)
```

Listing 30.24: Function to invert the Box-Cox transform.

We can re-evaluate the `ARIMA(0,1,2)` model with the Box-Cox transform. This involves first transforming the history prior to fitting the ARIMA model, then inverting the transform on the prediction before storing it for later comparison with the expected values. The `boxcox()` function can fail. In practice, I have seen this and it appears to be signaled by a returned `lambda` value of less than -5. By convention, `lambda` values are evaluated between -5 and 5.

A check is added for a `lambda` value less than -5, and if this the case, a `lambda` value of 1 is assumed and the raw history is used to fit the model. A `lambda` value of 1 is the same as `no-transform` and therefore the inverse transform has no effect. The complete example is listed below.

```
# evaluate ARIMA models with box-cox transformed time series
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA
from math import sqrt
from math import log
from math import exp
from scipy.stats import boxcox

# invert box-cox transform
def boxcox_inverse(value, lam):
  if lam == 0:
    return exp(value)
  return exp(log(lam * value + 1) / lam)

# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
```

```
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # transform
  transformed, lam = boxcox(history)
  if lam < -5:
    transformed, lam = history, 1
  # predict
  model = ARIMA(transformed, order=(0,1,2))
  model_fit = model.fit(disp=0)
  yhat = model_fit.forecast()[0]
  # invert transformed prediction
  yhat = boxcox_inverse(yhat, lam)
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
```

Listing 30.25: ARIMA model on Box-Cox transformed version of the Boston Robberies dataset.

Running the example prints the predicted and expected value each iteration. Note, you may see warnings when using the `boxcox()` transform function; for example:

```
RuntimeWarning: overflow encountered in square
llf -= N / 2.0 * np.log(np.sum((y - y_mean)**2. / N, axis=0))
```

Listing 30.26: Example errors printed while modeling the Box-Cox transformed version of the Boston Robberies dataset.

These can be ignored for now. The final RMSE of the model on the transformed data was 49.443. This is a smaller error than the ARIMA model on untransformed data, but only slightly, and it may or may not be statistically different.

```
...
>Predicted=276.253, Expected=287
>Predicted=299.811, Expected=355
>Predicted=338.997, Expected=460
>Predicted=404.509, Expected=364
>Predicted=349.336, Expected=487
RMSE: 49.443
```

Listing 30.27: Example output of the ARIMA model on Box-Cox transformed version of the Boston Robberies dataset.

We will use this model with the Box-Cox transform as the final model.

30.7 Model Validation

After models have been developed and a final model selected, it must be validated and finalized. Validation is an optional part of the process, but one that provides a last check to ensure we

have not fooled or lied to ourselves. This section includes the following steps:

- **Finalize Model**: Train and save the final model.
- **Make Prediction**: Load the finalized model and make a prediction.
- **Validate Model**: Load and validate the final model.

30.7.1 Finalize Model

Finalizing the model involves fitting an ARIMA model on the entire dataset, in this case, on a transformed version of the entire dataset. Once fit, the model can be saved to file for later use. Because a Box-Cox transform is also performed on the data, we need to know the chosen `lambda` so that any predictions from the model can be converted back to the original, untransformed scale. The example below fits an `ARIMA(0,1,2)` model on the Box-Cox transform dataset and saves the whole fit object and the `lambda` value to file.

There is a bug in the current version of the Statsmodels library that results in an error when you try to load a saved ARIMA model from file. Learn more about this bug and the workaround in Appendix B. The example below saves the fit model to file in the correct state so that it can be loaded successfully later.

```
# finalize model and save to file with workaround
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
from scipy.stats import boxcox
import numpy

# monkey patch around bug in ARIMA class
def __getnewargs__(self):
  return ((self.endog),(self.k_lags, self.k_diff, self.k_ma))

ARIMA.__getnewargs__ = __getnewargs__

# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
# transform data
transformed, lam = boxcox(X)
# fit model
model = ARIMA(transformed, order=(0,1,2))
model_fit = model.fit(disp=0)
# save model
model_fit.save('model.pkl')
numpy.save('model_lambda.npy', [lam])
```

Listing 30.28: Save the finalized ARIMA model with workaround.

Running the example creates two local files:

- `model.pkl` This is the ARIMAResult object from the call to `ARIMA.fit()`. This includes the coefficients and all other internal data returned when fitting the model.

- `model_lambda.npy` This is the `lambda` value stored as a one-row, one-column NumPy array.

This is probably overkill and all that is really needed for operational use are the AR and MA coefficients from the model, the `d` parameter for the number of differences, perhaps the lag observations and model residuals, and the `lambda` value for the transform.

30.7.2 Make Prediction

A natural case may be to load the model and make a single forecast. This is relatively straightforward and involves restoring the saved model and the `lambda` and calling the `forecast()` function. The example below loads the model, makes a prediction for the next time step, inverses the Box-Cox transform, and prints the prediction.

```
# load the finalized model and make a prediction
from statsmodels.tsa.arima_model import ARIMAResults
from math import exp
from math import log
import numpy

# invert box-cox transform
def boxcox_inverse(value, lam):
  if lam == 0:
    return exp(value)
  return exp(log(lam * value + 1) / lam)

model_fit = ARIMAResults.load('model.pkl')
lam = numpy.load('model_lambda.npy')
yhat = model_fit.forecast()[0]
yhat = boxcox_inverse(yhat, lam)
print('Predicted: %.3f' % yhat)
```

Listing 30.29: Load finalized ARIMA model and make a prediction.

Running the example prints the prediction of about 452. If we peek inside `validation.csv`, we can see that the value on the first row for the next time period is 452. The model got it 100% correct, which is very impressive (or lucky).

```
Predicted: 452.043
```

Listing 30.30: Example output when loading the finalized ARIMA and making one prediction.

30.7.3 Validate Model

We can load the model and use it in a pretend operational manner. In the test harness section, we saved the final 12 months of the original dataset in a separate file to validate the final model. We can load this `validation.csv` file now and use it see how well our model really is on unseen data. There are two ways we might proceed:

- Load the model and use it to forecast the next 12 months. The forecast beyond the first one or two months will quickly start to degrade in skill.

- Load the model and use it in a rolling-forecast manner, updating the transform and model for each time step. This is the preferred method as it is how one would use this model in practice, as it would achieve the best performance.

As with model evaluation in previous sections, we will make predictions in a rolling-forecast manner. This means that we will step over lead times in the validation dataset and take the observations as an update to the history.

```
# evaluate the finalized model on the validation dataset
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.arima_model import ARIMAResults
from scipy.stats import boxcox
from sklearn.metrics import mean_squared_error
from math import sqrt
from math import exp
from math import log
import numpy

# invert box-cox transform
def boxcox_inverse(value, lam):
  if lam == 0:
    return exp(value)
  return exp(log(lam * value + 1) / lam)

# load and prepare datasets
dataset = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
X = dataset.values.astype('float32')
history = [x for x in X]
validation = read_csv('validation.csv', header=None, index_col=0, parse_dates=True,
    squeeze=True)
y = validation.values.astype('float32')
# load model
model_fit = ARIMAResults.load('model.pkl')
lam = numpy.load('model_lambda.npy')
# make first prediction
predictions = list()
yhat = model_fit.forecast()[0]
yhat = boxcox_inverse(yhat, lam)
predictions.append(yhat)
history.append(y[0])
print('>Predicted=%.3f, Expected=%3.f' % (yhat, y[0]))
# rolling forecasts
for i in range(1, len(y)):
  # transform
  transformed, lam = boxcox(history)
  if lam < -5:
    transformed, lam = history, 1
  # predict
  model = ARIMA(transformed, order=(0,1,2))
  model_fit = model.fit(disp=0)
  yhat = model_fit.forecast()[0]
  # invert transformed prediction
  yhat = boxcox_inverse(yhat, lam)
  predictions.append(yhat)
```

```
  # observation
  obs = y[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(y, predictions))
print('RMSE: %.3f' % rmse)
pyplot.plot(y)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 30.31: Load and validate the finalized model.

Running the example prints each prediction and expected value for the time steps in the validation dataset. The final RMSE for the validation period is predicted at 53 robberies. This is not too different to the expected error of 49, but I would expect that it is also not too different from a simple persistence model.

```
>Predicted=452.043, Expected=452
>Predicted=423.088, Expected=391
>Predicted=408.378, Expected=500
>Predicted=482.454, Expected=451
>Predicted=445.944, Expected=375
>Predicted=413.881, Expected=372
>Predicted=413.209, Expected=302
>Predicted=355.159, Expected=316
>Predicted=363.515, Expected=398
>Predicted=406.365, Expected=394
>Predicted=394.186, Expected=431
>Predicted=428.174, Expected=431
RMSE: 53.078
```

Listing 30.32: Example output from validating the finalized model.

A plot of the predictions compared to the validation dataset is also provided. The forecast does have the characteristic of a persistence forecast. This does suggest that although this time series does have an obvious trend, it is still a reasonably difficult problem.

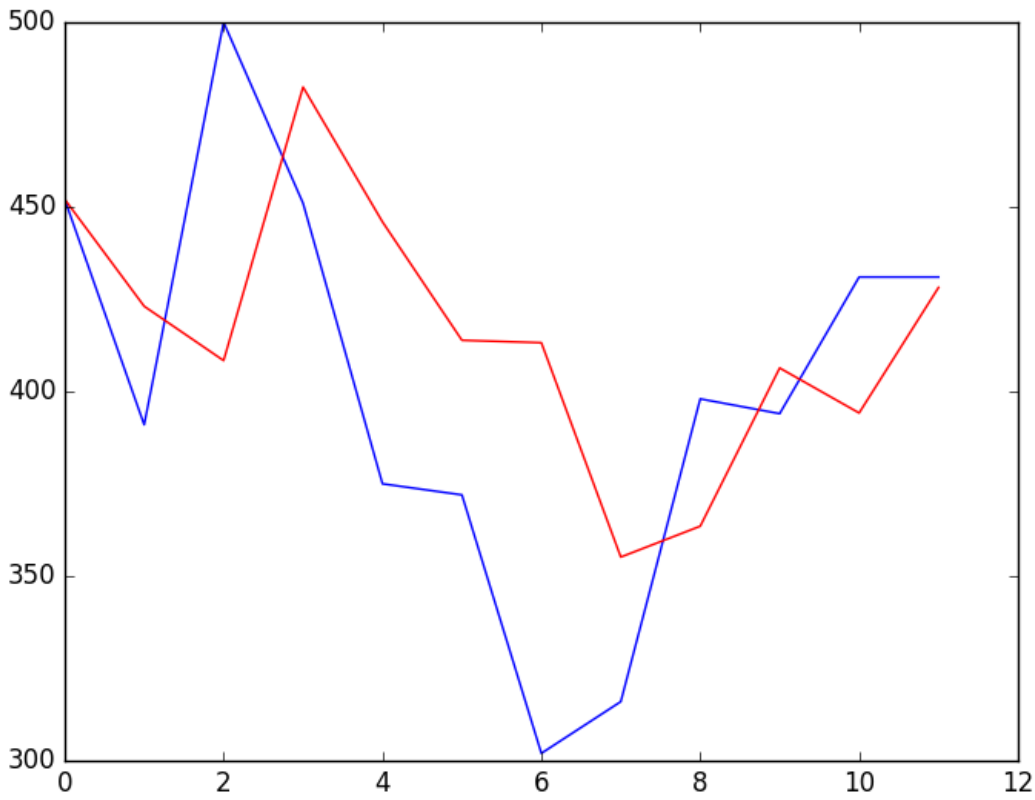

Figure 30.8: Line plot of expected validation values (blue) compared to predicted validation values (red).

30.8 Extensions

This tutorial was not exhaustive; there may be more that you can do to improve the result. This section lists some ideas.

- **Statistical Significance Tests**. Use a statistical test to check if the difference in results between different models is statistically significant. The Student t-test would be a good place to start.
- **Grid Search with Data Transforms**. Repeat the grid search in the ARIMA hyperparameters with the Box-Cox transform and see if a different and better set of parameters can be achieved.
- **Inspect Residuals**. Investigate the residual forecast errors on the final model with Box-Cox transforms to see if there is a further bias and/or autocorrelation that can be addressed.
- **Lean Model Saving**. Simplify model saving to only store the required coefficients rather than the entire `ARIMAResults` object.

- **Manually Handle Trend**. Model the trend directly with a linear or nonlinear model and explicitly remove it from the series. This may result in better performance if the trend is nonlinear and can be modeled better than the linear case.
- **Confidence Interval**. Display the confidence intervals for the predictions on the validation dataset.
- **Data Selection**. Consider modeling the problem without the first two years of data and see if this has an impact on forecast skill.

30.9 Summary

In this tutorial, you discovered the steps and the tools for a time series forecasting project with Python. We covered a lot of ground in this tutorial, specifically:

- How to develop a test harness with a performance measure and evaluation method and how to quickly develop a baseline forecast and skill.
- How to use time series analysis to raise ideas for how to best model the forecast problem.
- How to develop an ARIMA model, save it, and later load it to make predictions on new data.

30.9.1 Next

In the next project you will develop models to forecast Annual Water Usage in Baltimore.

Chapter 31

Project: Annual Water Usage in Baltimore

Time series forecasting is a process, and the only way to get good forecasts is to practice this process. In this tutorial, you will discover how to forecast the annual water usage in Baltimore with Python. Working through this tutorial will provide you with a framework for the steps and the tools for working through your own time series forecasting problems. After completing this tutorial, you will know:

- How to confirm your Python environment and carefully define a time series forecasting problem.
- How to create a test harness for evaluating models, develop a baseline forecast, and better understand your problem with the tools of time series analysis.
- How to develop an autoregressive integrated moving average model, save it to file, and later load it to make predictions for new time steps.

Let's get started.

31.1 Overview

In this tutorial, we will work through a time series forecasting project from end-to-end, from downloading the dataset and defining the problem to training a final model and making predictions. This project is not exhaustive, but shows how you can get good results quickly by working through a time series forecasting problem systematically. The steps of this project that we will work through are as follows.

1. Problem Description.
2. Test Harness.
3. Persistence.
4. Data Analysis.
5. ARIMA Models.

6. Model Validation.

This will provide a template for working through a time series prediction problem that you can use on your own dataset.

31.2 Problem Description

The problem is to predict annual water usage. The dataset provides the annual water usage in Baltimore from 1885 to 1963, or 79 years of data. The values are in the units of liters per capita per day, and there are 79 observations. The dataset is credited to Hipel and McLeod, 1994. Below is a sample of the first few rows of the dataset.

```
"Year","Water"
"1885",356
"1886",386
"1887",397
"1888",397
"1889",413
```

Listing 31.1: Sample of the first few rows of the dataset.

You can learn more about this dataset and download it directly from DataMarket[1]. Download the dataset as a CSV file and place it in your current working directory with the filename `water.csv`.

31.3 Test Harness

We must develop a test harness to investigate the data and evaluate candidate models. This involves two steps:

1. Defining a Validation Dataset.
2. Developing a Method for Model Evaluation.

31.3.1 Validation Dataset

The dataset is not current. This means that we cannot easily collect updated data to validate the model. Therefore, we will pretend that it is 1953 and withhold the last 10 years of data from analysis and model selection. This final decade of data will be used to validate the final model. The code below will load the dataset as a Pandas `Series` and split into two, one for model development (`dataset.csv`) and the other for validation (`validation.csv`).

```
# separate out a validation dataset
from pandas import read_csv
series = read_csv('water.csv', header=0, index_col=0, parse_dates=True, squeeze=True)
split_point = len(series) - 10
dataset, validation = series[0:split_point], series[split_point:]
print('Dataset %d, Validation %d' % (len(dataset), len(validation)))
dataset.to_csv('dataset.csv')
```

[1] https://goo.gl/BPB0ra

```
validation.to_csv('validation.csv')
```

Listing 31.2: Split the raw data into training and validation datasets.

Running the example creates two files and prints the number of observations in each.

```
Dataset 69, Validation 10
```

Listing 31.3: Example output splitting the raw data into train and validation datasets.

The specific contents of these files are:

- `dataset.csv`: Observations from 1885 to 1953 (69 observations).
- `validation.csv`: Observations from 1954 to 1963 (10 observations).

The validation dataset is about 12% of the original dataset. Note that the saved datasets do not have a header line, therefore we do not need to cater to this when working with these files later.

31.3.2 Model Evaluation

The RMSE performance measure and walk-forward validation will be used for model evaluation as was described for Project 1 in Section 30.3.2.

31.4 Persistence

The first step before getting bogged down in data analysis and modeling is to establish a baseline of performance. This will provide both a template for evaluating models using the proposed test harness and a performance measure by which all more elaborate predictive models can be compared. The baseline prediction for time series forecasting is called the naive forecast, or persistence.

This is where the observation from the previous time step is used as the prediction for the observation at the next time step. We can plug this directly into the test harness defined in the previous section. The complete code listing is provided below.

```
# evaluate a persistence model
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from math import sqrt
# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # predict
  yhat = history[-1]
```

```
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
```

Listing 31.4: Persistence model on the Baltimore Water Usage dataset.

Running the test harness prints the prediction and observation for each iteration of the test dataset. The example ends by printing the RMSE for the model. In this case, we can see that the persistence model achieved an RMSE of 21.975. This means that on average, the model was wrong by about 22 liters per capita per day for each prediction made.

```
...
>Predicted=613.000, Expected=598
>Predicted=598.000, Expected=575
>Predicted=575.000, Expected=564
>Predicted=564.000, Expected=549
>Predicted=549.000, Expected=538
RMSE: 21.975
```

Listing 31.5: Example output of the persistence model on the Baltimore Water Usage dataset.

We now have a baseline prediction method and performance; now we can start digging into our data.

31.5 Data Analysis

We can use summary statistics and plots of the data to quickly learn more about the structure of the prediction problem. In this section, we will look at the data from four perspectives:

1. Summary Statistics.
2. Line Plot.
3. Density Plots.
4. Box and Whisker Plot.

31.5.1 Summary Statistics

Summary statistics provide a quick look at the limits of observed values. It can help to get a quick idea of what we are working with. The example below calculates and prints summary statistics for the time series.

```
# summary statistics of time series
from pandas import read_csv
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
print(series.describe())
```

Listing 31.6: Summary statistics on the Baltimore Water Usage dataset.

Running the example provides a number of summary statistics to review. Some observations from these statistics include:

- The number of observations (count) matches our expectation, meaning we are handling the data correctly.
- The mean is about 500, which we might consider our level in this series.
- The standard deviation and percentiles suggest a reasonably tight spread around the mean.

```
count    69.000000
mean    500.478261
std      73.901685
min     344.000000
25%     458.000000
50%     492.000000
75%     538.000000
max     662.000000
```

Listing 31.7: Example output of summary statistics on the Baltimore Water Usage dataset.

31.5.2 Line Plot

A line plot of a time series dataset can provide a lot of insight into the problem. The example below creates and shows a line plot of the dataset.

```
# line plot of time series
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
series.plot()
pyplot.show()
```

Listing 31.8: Create a line plot of the Baltimore Water Usage dataset.

Run the example and review the plot. Note any obvious temporal structures in the series. Some observations from the plot include:

- There looks to be an increasing trend in water usage over time.
- There do not appear to be any obvious outliers, although there are some large fluctuations.
- There is a downward trend for the last few years of the series.

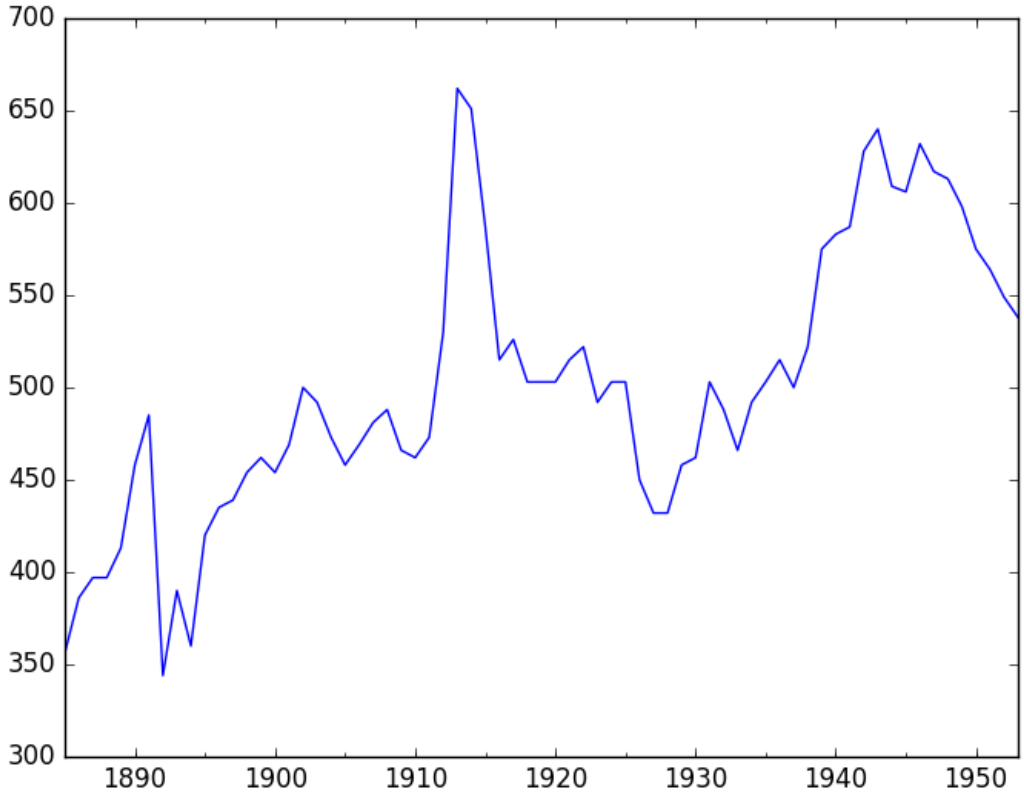

Figure 31.1: Line plot of the training set for the Baltimore Water Usage dataset.

There may be some benefit in explicitly modeling the trend component and removing it. You may also explore using differencing with one or two levels in order to make the series stationary.

31.5.3 Density Plot

Reviewing plots of the density of observations can provide further insight into the structure of the data. The example below creates a histogram and density plot of the observations without any temporal structure.

```
# density plots of time series
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
pyplot.figure(1)
pyplot.subplot(211)
series.hist()
pyplot.subplot(212)
series.plot(kind='kde')
pyplot.show()
```

Listing 31.9: Create a density plots of the Baltimore Water Usage dataset.

Run the example and review the plots. Some observations from the plots include:

- The distribution is not Gaussian, but is pretty close.
- The distribution has a long right tail and may suggest an exponential distribution or a double Gaussian.

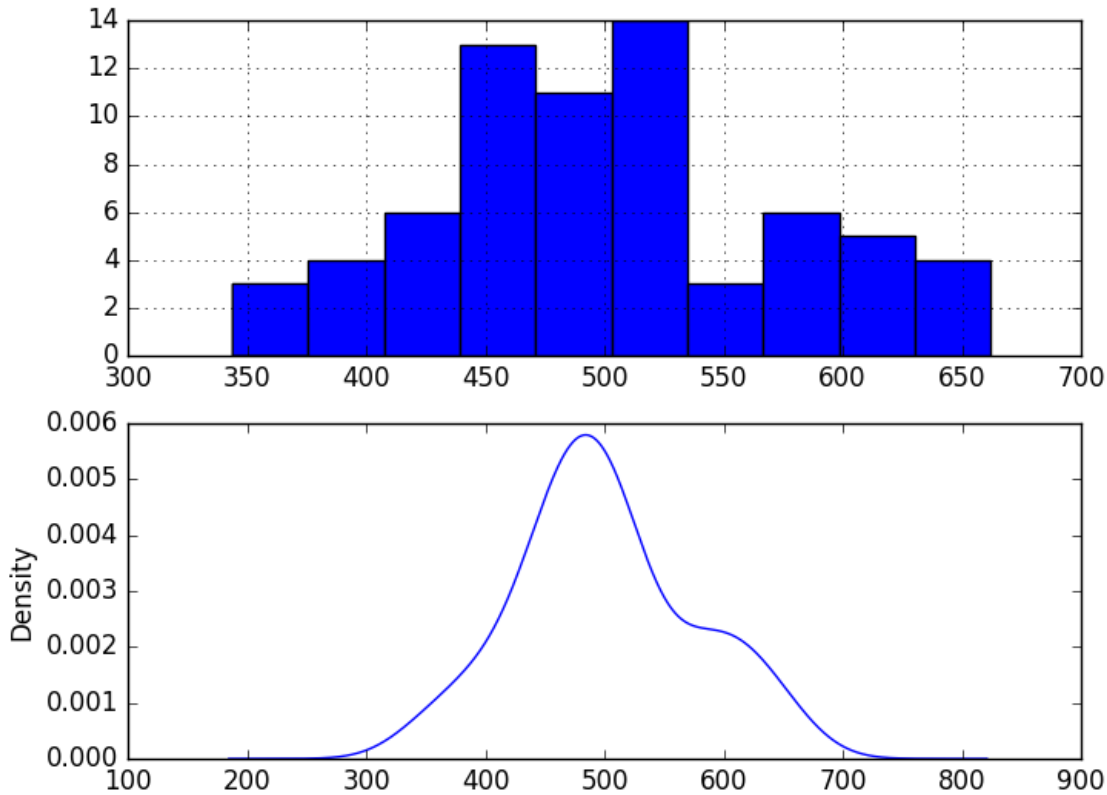

Figure 31.2: Density plots of the training set for the Baltimore Water Usage dataset.

This suggests it may be worth exploring some power transforms of the data prior to modeling.

31.5.4 Box and Whisker Plots

We can group the annual data by decade and get an idea of the spread of observations for each decade and how this may be changing. We do expect to see some trend (increasing mean or median), but it may be interesting to see how the rest of the distribution may be changing. The example below groups the observations by decade and creates one box and whisker plot for each decade of observations. The last decade only contains 9 years and may not be a useful comparison with the other decades. Therefore only data between 1885 and 1944 was plotted.

```
# boxplots of time series
from pandas import read_csv
from pandas import DataFrame
from pandas import Grouper
from matplotlib import pyplot
```

```
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
groups = series.groupby(Grouper(freq='10YS'))
decades = DataFrame()
for name, group in groups:
  if len(group.values) is 10:
    decades[name.year] = group.values
decades.boxplot()
pyplot.show()
```

Listing 31.10: Create a box and whisker plots of the Baltimore Water Usage dataset.

Running the example creates 6 box and whisker plots side-by-side, one for the 6 decades of selected data. Some observations from reviewing the plot include:

- The median values for each year (red line) may show an increasing trend that may not be linear.
- The spread, or middle 50% of the data (blue boxes), does show some variability.
- There maybe outliers in some decades (crosses outside of the box and whiskers).
- The second to last decade seems to have a lower average consumption, perhaps related to the first world war.

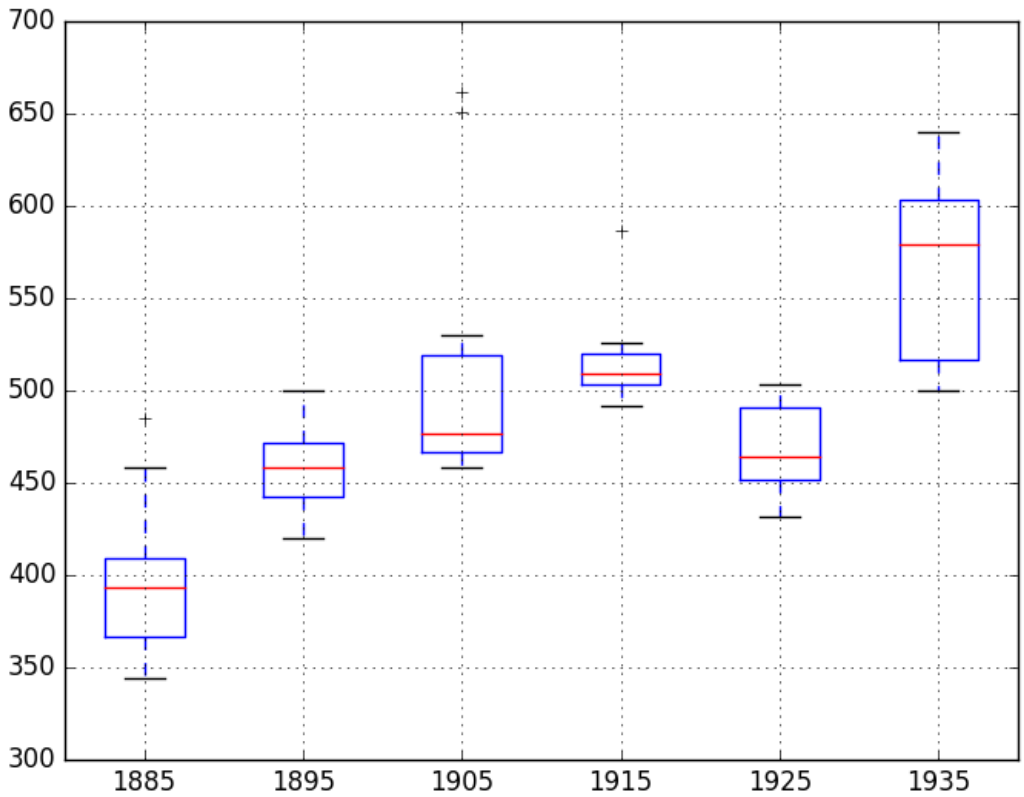

Figure 31.3: Box and whisker plots of the training set for the Baltimore Water Usage dataset.

This yearly view of the data is an interesting avenue and could be pursued further by looking at summary statistics from decade-to-decade and changes in summary statistics.

31.6 ARIMA Models

In this section, we will develop Autoregressive Integrated Moving Average or ARIMA models for the problem. We will approach modeling by both manual and automatic configuration of the ARIMA model. This will be followed by a third step of investigating the residual errors of the chosen model. As such, this section is broken down into 3 steps:

1. Manually Configure the ARIMA.
2. Automatically Configure the ARIMA.
3. Review Residual Errors.

31.6.1 Manually Configured ARIMA

The `ARIMA(p,d,q)` model requires three parameters and is traditionally configured manually. Analysis of the time series data assumes that we are working with a stationary time series. The time series is likely non-stationary. We can make it stationary by first differencing the series and using a statistical test to confirm that the result is stationary. The example below creates a stationary version of the series and saves it to file `stationary.csv`.

```
# create and summarize a stationary version of the time series
from pandas import read_csv
from pandas import Series
from statsmodels.tsa.stattools import adfuller
from matplotlib import pyplot

# create a differenced series
def difference(dataset):
  diff = list()
  for i in range(1, len(dataset)):
    value = dataset[i] - dataset[i - 1]
    diff.append(value)
  return Series(diff)

series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
X = series.values
X = X.astype('float32')
# difference data
stationary = difference(X)
stationary.index = series.index[1:]
# check if stationary
result = adfuller(stationary)
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))
# plot differenced data
```

```
stationary.plot()
pyplot.show()
# save
stationary.to_csv('stationary.csv')
```

Listing 31.11: Create a stationary version of the dataset and plot the differenced result.

Running the example outputs the result of a statistical significance test of whether the differenced series is stationary. Specifically, the augmented Dickey-Fuller test. The results show that the test statistic value -6.126719 is smaller than the critical value at 1% of -3.534. 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: -6.126719
p-value: 0.000000
Critical Values:
  5%: -2.906
  10%: -2.591
  1%: -3.534
```

Listing 31.12: Example output of a statistical test on the differenced Baltimore Water Usage dataset.

This suggests that at least one level of differencing is required. The `d` parameter in our ARIMA model should at least be a value of 1. A plot of the differenced data is also created. It suggests that this has indeed removed the increasing trend.

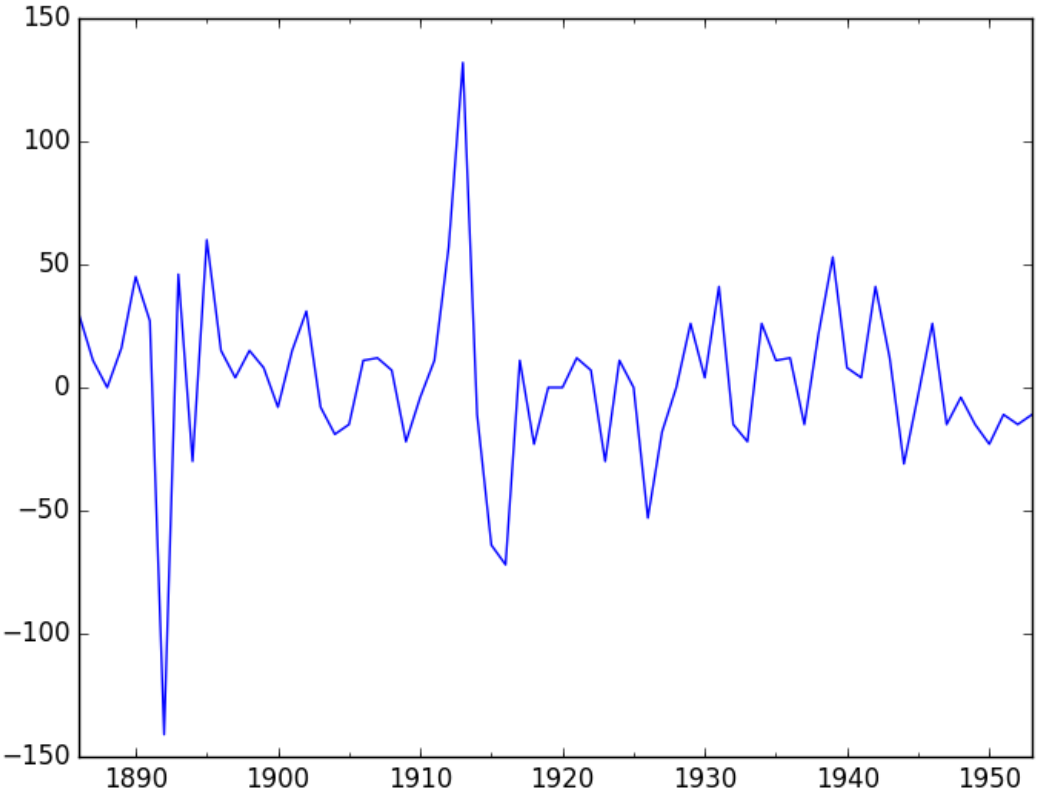

Figure 31.4: Line plot of the differenced Baltimore Water Usage dataset.

The next first step is to select the lag values for the Autoregression (AR) and Moving Average (MA) parameters, p and q respectively. We can do this by reviewing Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots. The example below creates ACF and PACF plots for the series.

```
# ACF and PACF plots of the time series
from pandas import read_csv
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
pyplot.figure()
pyplot.subplot(211)
plot_acf(series, ax=pyplot.gca())
pyplot.subplot(212)
plot_pacf(series, ax=pyplot.gca())
pyplot.show()
```

Listing 31.13: ACF and PACF plots of the Baltimore Water Usage dataset.

Run the example and review the plots for insights into how to set the p and q variables for the ARIMA model. Below are some observations from the plots.

- The ACF shows significant lags to 4 time steps.

- The PACF shows significant lags to 1 time step.

A good starting point for the `p` is 4 and `q` as 1.

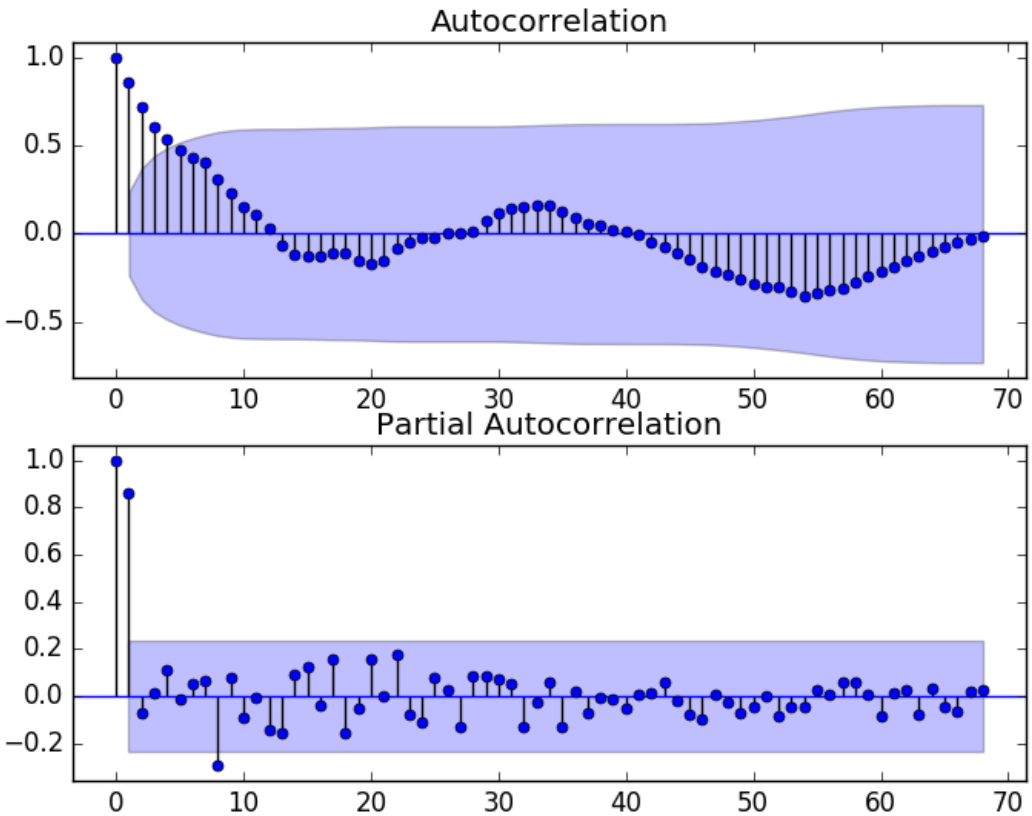

Figure 31.5: ACF and PACF plots of the Baltimore Water Usage dataset.

This quick analysis suggests an `ARIMA(4,1,1)` on the raw data may be a good starting point. The complete manual ARIMA example is listed below.

```
# evaluate a manually configured ARIMA model
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA
from math import sqrt
# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
```

```
  # predict
  model = ARIMA(history, order=(4,1,1))
  model_fit = model.fit(disp=0)
  yhat = model_fit.forecast()[0]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
```

Listing 31.14: Manual ARIMA model on the Baltimore Water Usage dataset.

Running this example results in an RMSE of 31.097, which is higher than the persistence model above.

```
...
>Predicted=609.030, Expected=598
>Predicted=601.832, Expected=575
>Predicted=580.316, Expected=564
>Predicted=574.084, Expected=549
>Predicted=562.009, Expected=538
RMSE: 31.097
```

Listing 31.15: Example output of the manual ARIMA model on the Baltimore Water Usage dataset.

31.6.2 Grid Search ARIMA Hyperparameters

The ACF and PACF plots suggest that we cannot do better than a persistence model on this dataset. To confirm this analysis, we can grid search a suite of ARIMA hyperparameters and check that no models result in better out of sample RMSE performance. In this section, we will search values of `p`, `d`, and `q` for combinations (skipping those that fail to converge), and find the combination that results in the best performance. We will use a grid search to explore all combinations in a subset of integer values. Specifically, we will search all combinations of the following parameters:

- `p`: 0 to 4.
- `d`: 0 to 2.
- `q`: 0 to 4.

This is $(5 \times 3 \times 5)$, or 300 potential runs of the test harness, and will take some time to execute. We will also disable the automatic addition of a trend constant from the model by setting the `trend` argument to `nc` for no constant when calling `fit()`. The complete worked example with the grid search version of the test harness is listed below.

```
# grid search ARIMA parameters for a time series
import warnings
from pandas import read_csv
```

```
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt

# evaluate an ARIMA model for a given order (p,d,q) and return RMSE
def evaluate_arima_model(X, arima_order):
  # prepare training dataset
  X = X.astype('float32')
  train_size = int(len(X) * 0.50)
  train, test = X[0:train_size], X[train_size:]
  history = [x for x in train]
  # make predictions
  predictions = list()
  for t in range(len(test)):
    model = ARIMA(history, order=arima_order)
    # model_fit = model.fit(disp=0)
    model_fit = model.fit(trend='nc', disp=0)
    yhat = model_fit.forecast()[0]
    predictions.append(yhat)
    history.append(test[t])
  # calculate out of sample error
  rmse = sqrt(mean_squared_error(test, predictions))
  return rmse

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
  dataset = dataset.astype('float32')
  best_score, best_cfg = float("inf"), None
  for p in p_values:
    for d in d_values:
      for q in q_values:
        order = (p,d,q)
        try:
          rmse = evaluate_arima_model(dataset, order)
          if rmse < best_score:
            best_score, best_cfg = rmse, order
          print('ARIMA%s RMSE=%.3f' % (order,rmse))
        except:
          continue
  print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))

# load dataset
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# evaluate parameters
p_values = range(0, 5)
d_values = range(0, 3)
q_values = range(0, 5)
warnings.filterwarnings("ignore")
evaluate_models(series.values, p_values, d_values, q_values)
```

Listing 31.16: Grid Search ARIMA models on the Baltimore Water Usage dataset.

Running the example runs through all combinations and reports the results on those that converge without error. The example takes a little over 2 minutes to run on modern hardware. The results show that the best configuration discovered was `ARIMA(2,1,0)` with an RMSE of 21.733, slightly lower than the persistence model tested earlier, but may or may not be

significantly different.

```
...
ARIMA(4, 1, 0) RMSE=24.802
ARIMA(4, 1, 1) RMSE=25.103
ARIMA(4, 2, 0) RMSE=27.089
ARIMA(4, 2, 1) RMSE=25.932
ARIMA(4, 2, 2) RMSE=25.418
Best ARIMA(2, 1, 0) RMSE=21.733
```

Listing 31.17: Example output of grid searching ARIMA models on the Baltimore Water Usage dataset.

We will select this `ARIMA(2,1,0)` model going forward.

31.6.3 Review Residual Errors

A good final check of a model is to review residual forecast errors. Ideally, the distribution of residual errors should be a Gaussian with a zero mean. We can check this by using summary statistics and plots to investigate the residual errors from the `ARIMA(2,1,0)` model. The example below calculates and summarizes the residual forecast errors.

```
# summarize residual errors for an ARIMA model
from pandas import read_csv
from pandas import DataFrame
from statsmodels.tsa.arima_model import ARIMA
from matplotlib import pyplot
# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # predict
  model = ARIMA(history, order=(2,1,0))
  model_fit = model.fit(trend='nc', disp=0)
  yhat = model_fit.forecast()[0]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
# errors
residuals = [test[i]-predictions[i] for i in range(len(test))]
residuals = DataFrame(residuals)
print(residuals.describe())
pyplot.figure()
pyplot.subplot(211)
residuals.hist(ax=pyplot.gca())
pyplot.subplot(212)
residuals.plot(kind='kde', ax=pyplot.gca())
pyplot.show()
```

Listing 31.18: Plot and summarize errors of an ARIMA model on the Baltimore Water Usage dataset.

Running the example first describes the distribution of the residuals. We can see that the distribution has a right shift and that the mean is non-zero at 1.081624. This is perhaps a sign that the predictions are biased.

```
count 35.000000
mean   1.081624
std   22.022566
min  -52.103811
25%  -16.202283
50%   -0.459801
75%   12.085091
max   51.284336
```

Listing 31.19: Example output of summary statistics on the residual errors of an ARIMA model on the Baltimore Water Usage dataset.

The distribution of residual errors is also plotted. The graphs suggest a Gaussian-like distribution with a longer right tail, providing further evidence that perhaps a power transform might be worth exploring.

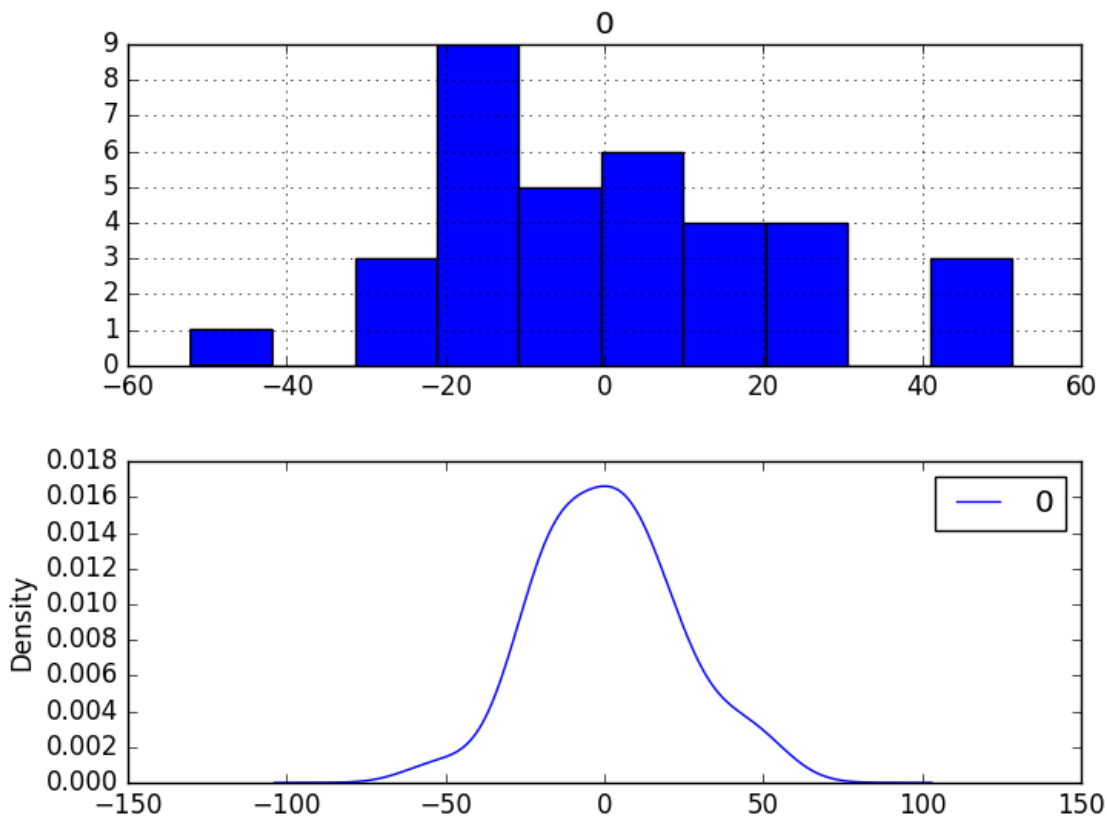

Figure 31.6: Density plots of residual errors on the Baltimore Water Usage dataset.

We could use this information to bias-correct predictions by adding the mean residual error of 1.081624 to each forecast made. The example below performs this bias-correction.

```
# summarize residual errors from bias corrected forecasts
from pandas import read_csv
from pandas import DataFrame
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA
from math import sqrt
from matplotlib import pyplot
# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
bias = 1.081624
for i in range(len(test)):
  # predict
  model = ARIMA(history, order=(2,1,0))
  model_fit = model.fit(trend='nc', disp=0)
  yhat = bias + float(model_fit.forecast()[0])
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
# summarize residual errors
residuals = [test[i]-predictions[i] for i in range(len(test))]
residuals = DataFrame(residuals)
print(residuals.describe())
# plot residual errors
pyplot.figure()
pyplot.subplot(211)
residuals.hist(ax=pyplot.gca())
pyplot.subplot(212)
residuals.plot(kind='kde', ax=pyplot.gca())
pyplot.show()
```

Listing 31.20: Plot and summarize bias corrected errors of an ARIMA model on the Baltimore Water Usage dataset.

The performance of the predictions is improved very slightly from 21.733 to 21.706, which may or may not be significant. The summary of the forecast residual errors shows that the mean was indeed moved to a value very close to zero.

```
RMSE: 21.706
                0
count 3.500000e+01
mean -3.537544e-07
```

```
std    2.202257e+01
min   -5.318543e+01
25%   -1.728391e+01
50%   -1.541425e+00
75%    1.100347e+01
max    5.020271e+01
```

Listing 31.21: Example output of summary statistics of residual errors of a bias corrected ARIMA model on the Baltimore Water Usage dataset.

Finally, density plots of the residual error do show a small shift towards zero.

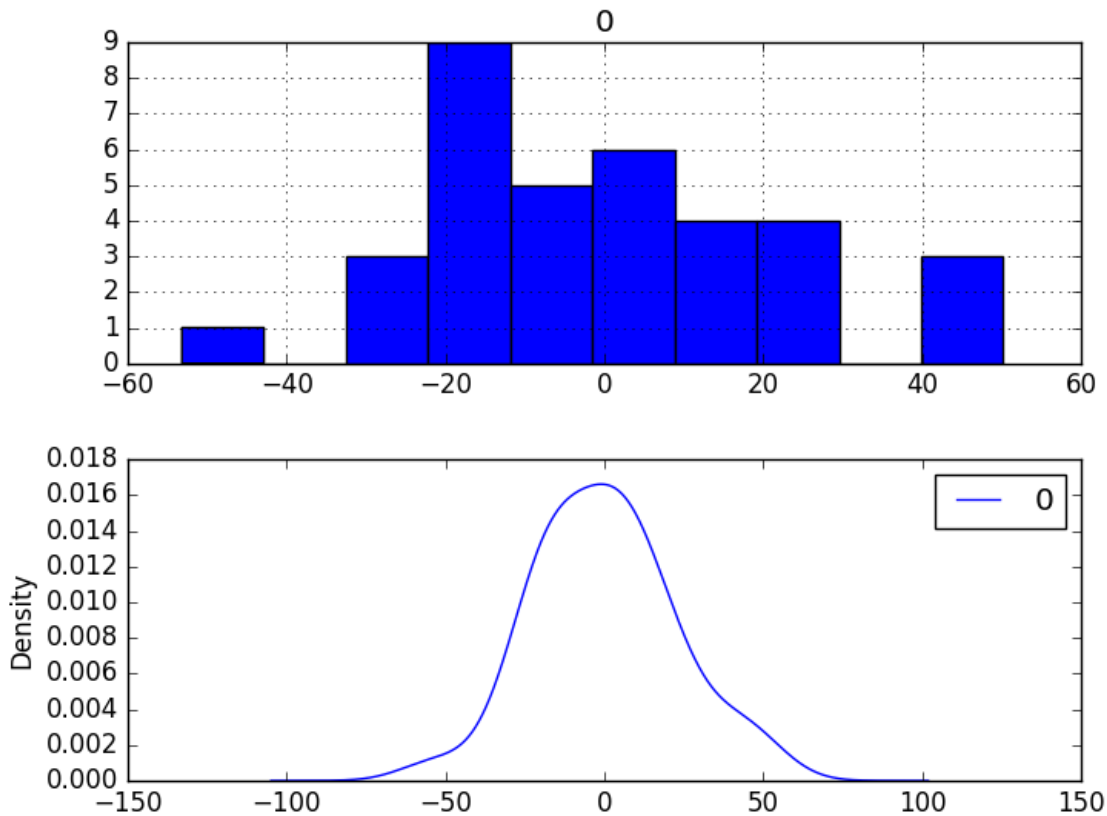

Figure 31.7: Density plots of residual errors of a bias corrected model on the Baltimore Water Usage dataset.

It is debatable whether this bias correction is worth it, but we will use it for now.

31.7 Model Validation

After models have been developed and a final model selected, it must be validated and finalized. Validation is an optional part of the process, but one that provides a last check to ensure we have not fooled or misled ourselves. This section includes the following steps:

- **Finalize Model**: Train and save the final model.
- **Make Prediction**: Load the finalized model and make a prediction.
- **Validate Model**: Load and validate the final model.

31.7.1 Finalize Model

Finalizing the model involves fitting an ARIMA model on the entire dataset, in this case, on a transformed version of the entire dataset. Once fit, the model can be saved to file for later use. The example below trains an `ARIMA(2,1,0)` model on the dataset and saves the whole fit object and the bias to file. There is a bug in the current version of the Statsmodels library that results in an error when you try to load a saved ARIMA model from file. Learn more about this bug and the workaround in Appendix B. The example below saves the fit model to file in the correct state so that it can be loaded successfully later.

```
# save finalized model to file
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
import numpy

# monkey patch around bug in ARIMA class
def __getnewargs__(self):
  return ((self.endog),(self.k_lags, self.k_diff, self.k_ma))

ARIMA.__getnewargs__ = __getnewargs__

# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
# fit model
model = ARIMA(X, order=(2,1,0))
model_fit = model.fit(trend='nc', disp=0)
# bias constant, could be calculated from in-sample mean residual
bias = 1.081624
# save model
model_fit.save('model.pkl')
numpy.save('model_bias.npy', [bias])
```

Listing 31.22: Save a finalized ARIMA model to file for the Baltimore Water Usage dataset.

Running the example creates two local files:

- `model.pkl` This is the `ARIMAResult` object from the call to `ARIMA.fit()`. This includes the coefficients and all other internal data returned when fitting the model.
- `model_bias.npy` This is the bias value stored as a one-row, one-column NumPy array.

31.7.2 Make Prediction

A natural case may be to load the model and make a single forecast. This is relatively straightforward and involves restoring the saved model and the bias and calling the `forecast()`

function. The example below loads the model, makes a prediction for the next time step, and prints the prediction.

```
# load finalized model and make a prediction
from statsmodels.tsa.arima_model import ARIMAResults
import numpy
model_fit = ARIMAResults.load('model.pkl')
bias = numpy.load('model_bias.npy')
yhat = bias + float(model_fit.forecast()[0])
print('Predicted: %.3f' % yhat)
```

Listing 31.23: Load the finalized ARIMA model and make a prediction on the Baltimore Water Usage dataset.

Running the example prints the prediction of about 540.

```
Predicted: 540.013
```

Listing 31.24: Example output loading the ARIMA model and making a single prediction on the Baltimore Water Usage dataset.

If we peek inside `validation.csv`, we can see that the value on the first row for the next time period is 568. The prediction is in the right ballpark.

31.7.3 Validate Model

We can load the model and use it in a pretend operational manner. In the test harness section, we saved the final 10 years of the original dataset in a separate file to validate the final model. We can load this `validation.csv` file now and use it to see how well our model really is on unseen data. There are two ways we might proceed:

- Load the model and use it to forecast the next 10 years. The forecast beyond the first one or two years will quickly start to degrade in skill.
- Load the model and use it in a rolling-forecast manner, updating the transform and model for each time step. This is the preferred method as it is how one would use this model in practice as it would achieve the best performance.

As with model evaluation in the previous sections, we will make predictions in a rolling-forecast manner. This means that we will step over lead times in the validation dataset and take the observations as an update to the history.

```
# load and evaluate the finalized model on the validation dataset
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.arima_model import ARIMAResults
from sklearn.metrics import mean_squared_error
from math import sqrt
import numpy
# load and prepare datasets
dataset = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
X = dataset.values.astype('float32')
history = [x for x in X]
```

```
validation = read_csv('validation.csv', header=None, index_col=0, parse_dates=True,
    squeeze=True)
y = validation.values.astype('float32')
# load model
model_fit = ARIMAResults.load('model.pkl')
bias = numpy.load('model_bias.npy')
# make first prediction
predictions = list()
yhat = bias + float(model_fit.forecast()[0])
predictions.append(yhat)
history.append(y[0])
print('>Predicted=%.3f, Expected=%3.f' % (yhat, y[0]))
# rolling forecasts
for i in range(1, len(y)):
  # predict
  model = ARIMA(history, order=(2,1,0))
  model_fit = model.fit(trend='nc', disp=0)
  yhat = bias + float(model_fit.forecast()[0])
  predictions.append(yhat)
  # observation
  obs = y[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(y, predictions))
print('RMSE: %.3f' % rmse)
pyplot.plot(y)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 31.25: Load and validate the finalized model on the Baltimore Water Usage dataset.

Running the example prints each prediction and expected value for the time steps in the validation dataset. The final RMSE for the validation period is predicted at 16 liters per capita per day. This is not too different from the expected error of 21, but I would expect that it is also not too different from a simple persistence model.

```
>Predicted=540.013, Expected=568
>Predicted=571.589, Expected=575
>Predicted=573.289, Expected=579
>Predicted=579.561, Expected=587
>Predicted=588.063, Expected=602
>Predicted=603.022, Expected=594
>Predicted=593.178, Expected=587
>Predicted=588.558, Expected=587
>Predicted=588.797, Expected=625
>Predicted=627.941, Expected=613
RMSE: 16.532
```

Listing 31.26: Example output loading and validating the finalized model on the Baltimore Water Usage dataset.

A plot of the predictions compared to the validation dataset is also provided. The forecast does have the characteristics of a persistence forecast. This suggests that although this time series does have an obvious trend, it is still a reasonably difficult problem.

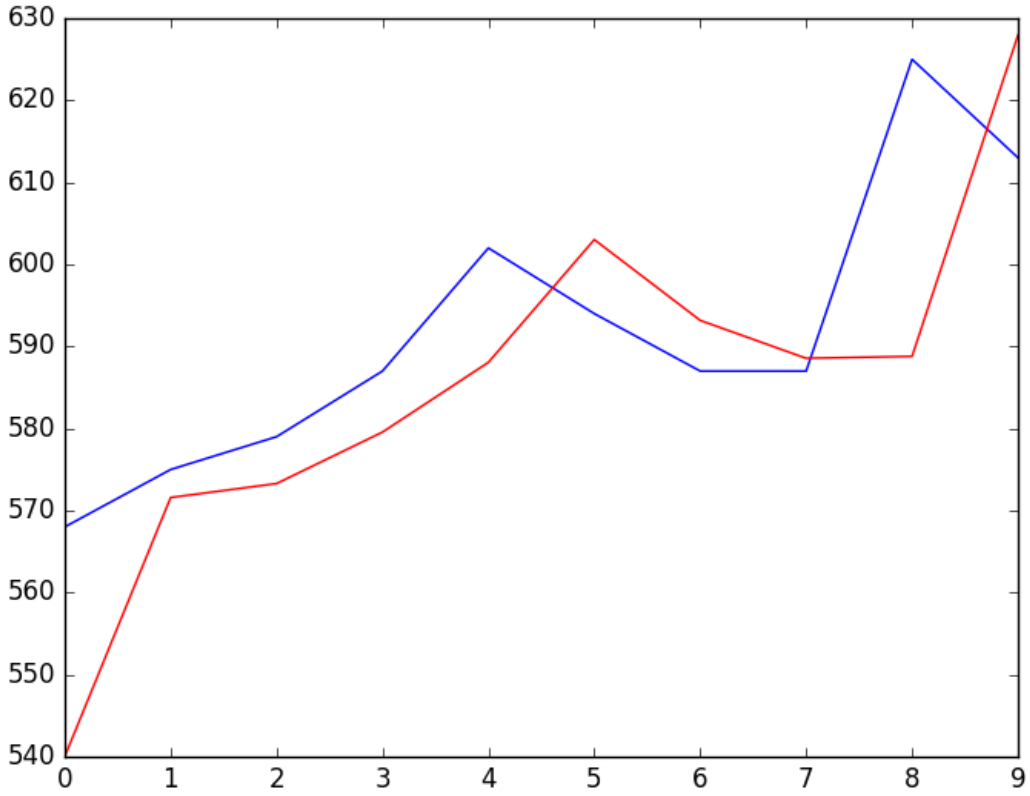

Figure 31.8: Line plot of the expected values (blue) and predictions (red) for the validation dataset.

31.8 Summary

In this tutorial, you discovered the steps and the tools for a time series forecasting project with Python. We covered a lot of ground in this tutorial; specifically:

- How to develop a test harness with a performance measure and evaluation method and how to quickly develop a baseline forecast and skill.
- How to use time series analysis to raise ideas for how to best model the forecast problem.
- How to develop an ARIMA model, save it, and later load it to make predictions on new data.

31.8.1 Next

In the next project you will develop models to forecast Monthly Sales of French Champagne.

Chapter 32

Project: Monthly Sales of French Champagne

Time series forecasting is a process, and the only way to get good forecasts is to practice this process. In this tutorial, you will discover how to forecast the monthly sales of French champagne with Python. Working through this tutorial will provide you with a framework for the steps and the tools for working through your own time series forecasting problems. After completing this tutorial, you will know:

- How to confirm your Python environment and carefully define a time series forecasting problem.
- How to create a test harness for evaluating models, develop a baseline forecast, and better understand your problem with the tools of time series analysis.
- How to develop an autoregressive integrated moving average model, save it to file, and later load it to make predictions for new time steps.

Let's get started.

32.1 Overview

In this tutorial, we will work through a time series forecasting project from end-to-end, from downloading the dataset and defining the problem to training a final model and making predictions. This project is not exhaustive, but shows how you can get good results quickly by working through a time series forecasting problem systematically.

The steps of this project that we will through are as follows.

1. Problem Description.
2. Test Harness.
3. Persistence.
4. Data Analysis.
5. ARIMA Models.

6. Model Validation.

This will provide a template for working through a time series prediction problem that you can use on your own dataset.

32.2 Problem Description

The problem is to predict the number of monthly sales of champagne for the Perrin Freres label (named for a region in France). The dataset provides the number of monthly sales of champagne from January 1964 to September 1972, or just under 10 years of data. The values are a count of millions of sales and there are 105 observations. The dataset is credited to Makridakis and Wheelwright, 1989. Below is a sample of the first few rows of the dataset.

```
"Month","Sales"
"1964-01",2815
"1964-02",2672
"1964-03",2755
"1964-04",2721
"1964-05",2946
```

Listing 32.1: Sample of the first few rows of the dataset.

You can learn more about this dataset and download it directly from DataMarket[1]. Download the dataset as a CSV file and place it in your current working directory with the filename `champagne.csv`.

32.3 Test Harness

We must develop a test harness to investigate the data and evaluate candidate models. This involves two steps:

1. Defining a Validation Dataset.
2. Developing a Method for Model Evaluation.

32.3.1 Validation Dataset

The dataset is not current. This means that we cannot easily collect updated data to validate the model. Therefore we will pretend that it is September 1971 and withhold the last one year of data from analysis and model selection. This final year of data will be used to validate the final model. The code below will load the dataset as a Pandas `Series` and split into two, one for model development (`dataset.csv`) and the other for validation (`validation.csv`).

```
# separate out a validation dataset
from pandas import read_csv
series = read_csv('champagne.csv', header=0, index_col=0, parse_dates=True, squeeze=True)
split_point = len(series) - 12
dataset, validation = series[0:split_point], series[split_point:]
```

[1] https://goo.gl/OwxEqK

```
print('Dataset %d, Validation %d' % (len(dataset), len(validation)))
dataset.to_csv('dataset.csv')
validation.to_csv('validation.csv')
```

Listing 32.2: Split the raw data into training and validation datasets.

Running the example creates two files and prints the number of observations in each.

```
Dataset 93, Validation 12
```

Listing 32.3: Example output splitting the raw data into train and validation datasets.

The specific contents of these files are:

- `dataset.csv`: Observations from January 1964 to September 1971 (93 observations).
- `validation.csv`: Observations from October 1971 to September 1972 (12 observations).

The validation dataset is about 11% of the original dataset. Note that the saved datasets do not have a header line, therefore we do not need to cater for this when working with these files later.

32.3.2 Model Evaluation

The RMSE performance measure and walk-forward validation will be used for model evaluation as was described for Project 1 in Section 30.3.2.

32.4 Persistence

The first step before getting bogged down in data analysis and modeling is to establish a baseline of performance. This will provide both a template for evaluating models using the proposed test harness and a performance measure by which all more elaborate predictive models can be compared. The baseline prediction for time series forecasting is called the naive forecast, or persistence.

This is where the observation from the previous time step is used as the prediction for the observation at the next time step. We can plug this directly into the test harness defined in the previous section. The complete code listing is provided below.

```
# evaluate persistence model on time series
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from math import sqrt
# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
```

```
  # predict
  yhat = history[-1]
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
```

Listing 32.4: Persistence model on the Champagne Sales dataset.

Running the test harness prints the prediction and observation for each iteration of the test dataset. The example ends by printing the RMSE for the model. In this case, we can see that the persistence model achieved an RMSE of 3186.501. This means that on average, the model was wrong by about 3,186 million sales for each prediction made.

```
...
>Predicted=4676.000, Expected=5010
>Predicted=5010.000, Expected=4874
>Predicted=4874.000, Expected=4633
>Predicted=4633.000, Expected=1659
>Predicted=1659.000, Expected=5951
RMSE: 3186.501
```

Listing 32.5: Example output of the persistence model on the Champagne Sales dataset.

We now have a baseline prediction method and performance; now we can start digging into our data.

32.5 Data Analysis

We can use summary statistics and plots of the data to quickly learn more about the structure of the prediction problem. In this section, we will look at the data from five perspectives:

1. Summary Statistics.
2. Line Plot.
3. Seasonal Line Plots
4. Density Plots.
5. Box and Whisker Plot.

32.5.1 Summary Statistics

Summary statistics provide a quick look at the limits of observed values. It can help to get a quick idea of what we are working with. The example below calculates and prints summary statistics for the time series.

```
# summary statistics of time series
from pandas import read_csv
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
print(series.describe())
```

Listing 32.6: Summary statistics on the Champagne Sales dataset.

Running the example provides a number of summary statistics to review. Some observations from these statistics include:

- The number of observations (count) matches our expectation, meaning we are handling the data correctly.
- The mean is about 4,641, which we might consider our level in this series.
- The standard deviation (average spread from the mean) is relatively large at 2,486 sales.
- The percentiles along with the standard deviation do suggest a large spread to the data.

```
count      93.000000
mean     4641.118280
std      2486.403841
min      1573.000000
25%      3036.000000
50%      4016.000000
75%      5048.000000
max     13916.000000
```

Listing 32.7: Example output of summary statistics on the Champagne Sales dataset.

32.5.2 Line Plot

A line plot of a time series can provide a lot of insight into the problem. The example below creates and shows a line plot of the dataset.

```
# line plot of time series
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
series.plot()
pyplot.show()
```

Listing 32.8: Create a line plot of the Champagne Sales dataset.

Run the example and review the plot. Note any obvious temporal structures in the series. Some observations from the plot include:

- There may be an increasing trend of sales over time.
- There appears to be systematic seasonality to the sales for each year.
- The seasonal signal appears to be growing over time, suggesting a multiplicative relationship (increasing change).

- There do not appear to be any obvious outliers.
- The seasonality suggests that the series is almost certainly non-stationary.

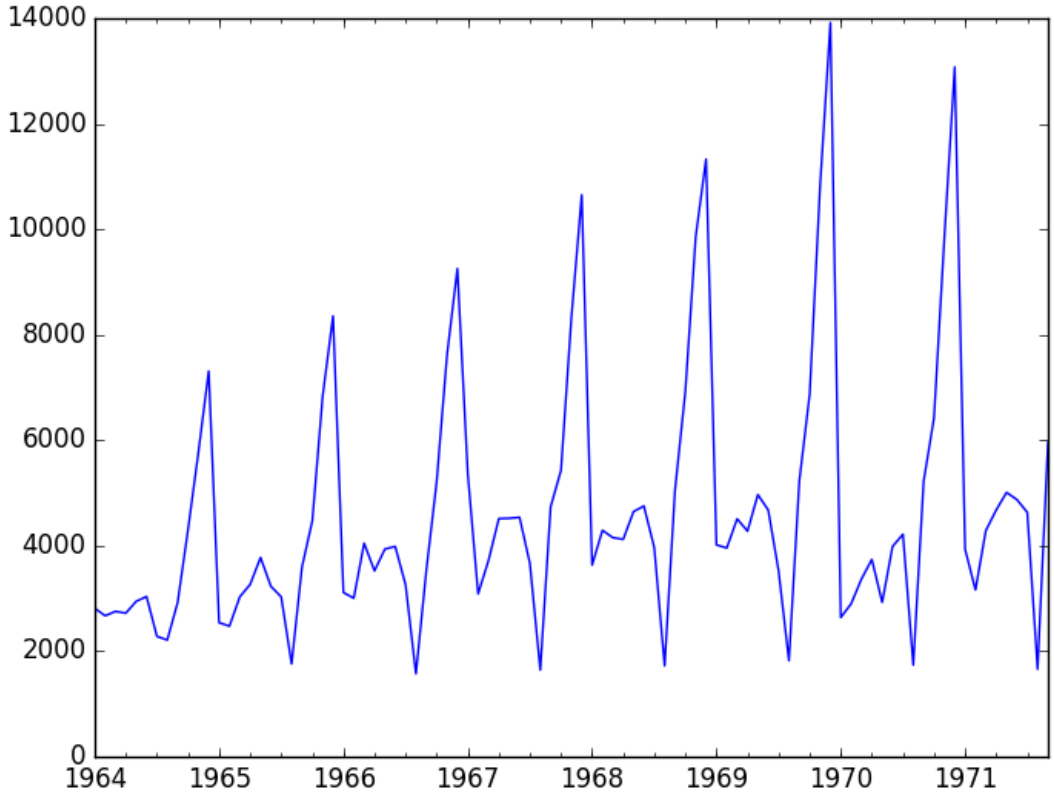

Figure 32.1: Line plot of the training set for the Champagne Sales dataset.

There may be benefit in explicitly modeling the seasonal component and removing it. You may also explore using differencing with one or two levels in order to make the series stationary. The increasing trend or growth in the seasonal component may suggest the use of a log or other power transform.

32.5.3 Seasonal Line Plots

We can confirm the assumption that the seasonality is a yearly cycle by eyeballing line plots of the dataset by year. The example below takes the 7 full years of data as separate groups and creates one line plot for each. The line plots are aligned vertically to help spot any year-to-year pattern.

```
# multiple line plots of time series
from pandas import read_csv
from pandas import DataFrame
from pandas import Grouper
```

```
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
groups = series['1964':'1970'].groupby(Grouper(freq='A'))
years = DataFrame()
pyplot.figure()
i = 1
n_groups = len(groups)
for name, group in groups:
  pyplot.subplot((n_groups*100) + 10 + i)
  i += 1
  pyplot.plot(group)
pyplot.show()
```

Listing 32.9: Create multiple yearly line plots of the Champagne Sales dataset.

Running the example creates the stack of 7 line plots. We can clearly see a dip each August and a rise from each August to December. This pattern appears the same each year, although at different levels. This will help with any explicitly season-based modeling later.

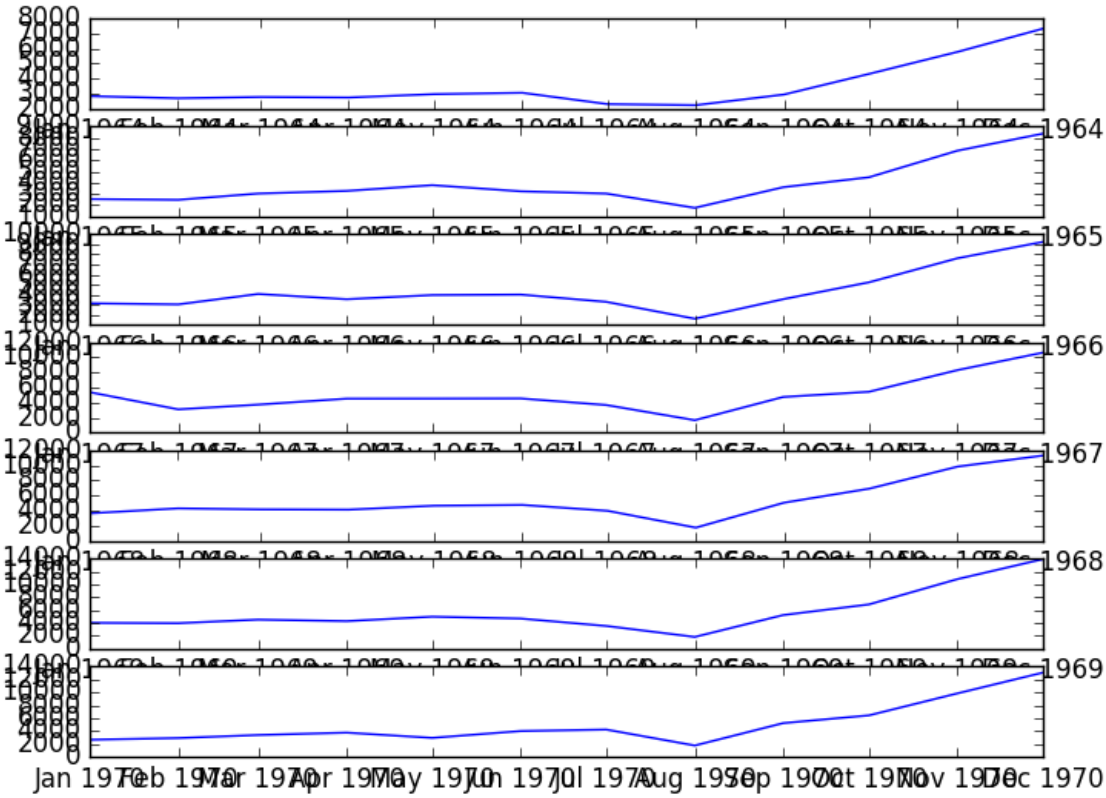

Figure 32.2: Multiple yearly line plots of the training set for the Champagne Sales dataset.

It might have been easier if all season line plots were added to the one graph to help contrast the data for each year.

32.5.4 Density Plot

Reviewing plots of the density of observations can provide further insight into the structure of the data. The example below creates a histogram and density plot of the observations without any temporal structure.

```
# density plots of time series
from pandas import read_csv
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
pyplot.figure(1)
pyplot.subplot(211)
series.hist()
pyplot.subplot(212)
series.plot(kind='kde')
pyplot.show()
```

Listing 32.10: Create a density plots of the Champagne Sales dataset.

Run the example and review the plots. Some observations from the plots include:

- The distribution is not Gaussian.
- The shape has a long right tail and may suggest an exponential distribution

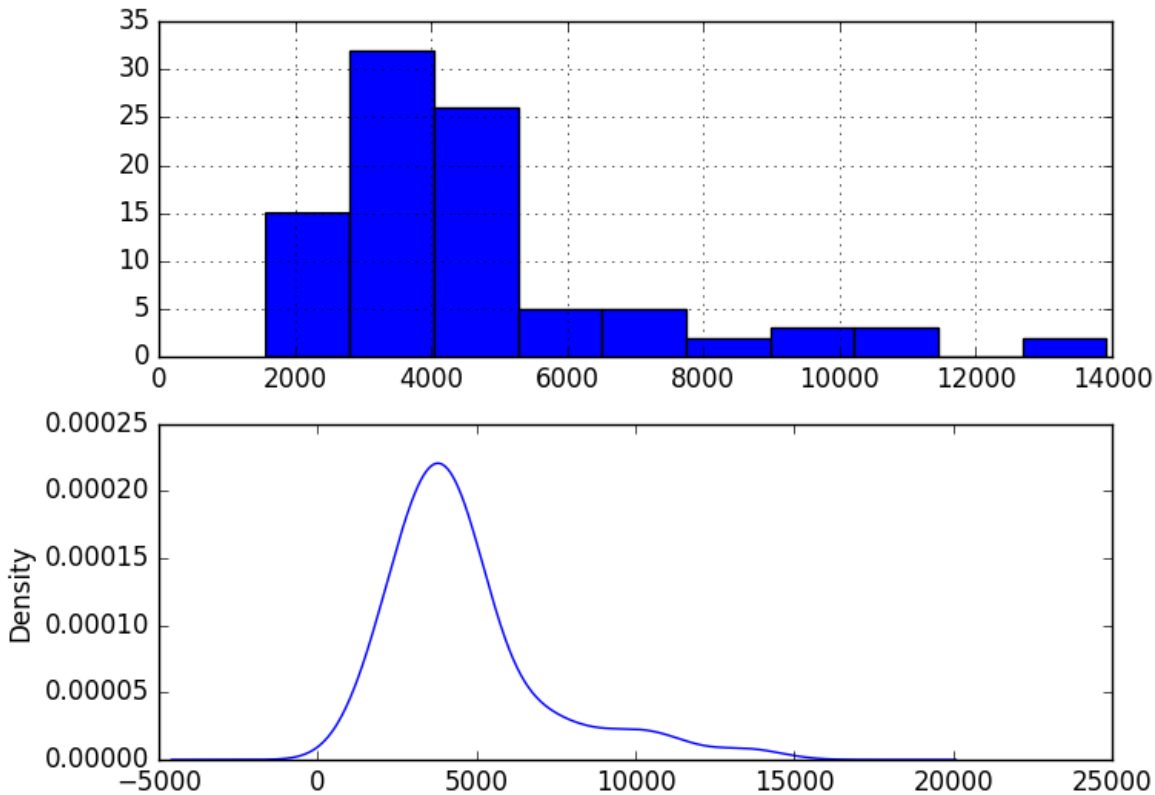

Figure 32.3: Density plots of the training set for the Champagne Sales dataset.

This lends more support to exploring some power transforms of the data prior to modeling.

32.5.5 Box and Whisker Plots

We can group the monthly data by year and get an idea of the spread of observations for each year and how this may be changing. We do expect to see some trend (increasing mean or median), but it may be interesting to see how the rest of the distribution may be changing. The example below groups the observations by year and creates one box and whisker plot for each year of observations. The last year (1971) only contains 9 months and may not be a useful comparison with the 12 months of observations for other years. Therefore, only data between 1964 and 1970 was plotted.

```
# boxplots of time series
from pandas import read_csv
from pandas import DataFrame
from pandas import Grouper
from matplotlib import pyplot
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
groups = series['1964':'1970'].groupby(Grouper(freq='A'))
years = DataFrame()
for name, group in groups:
  years[name.year] = group.values
years.boxplot()
pyplot.show()
```

Listing 32.11: Create a box and whisker plots of the Champagne Sales dataset.

Running the example creates 7 box and whisker plots side-by-side, one for each of the 7 years of selected data. Some observations from reviewing the plots include:

- The median values for each year (red line) may show an increasing trend.
- The spread or middle 50% of the data (blue boxes) does appear reasonably stable.
- There are outliers each year (black crosses); these may be the tops or bottoms of the seasonal cycle.
- The last year, 1970, does look different from the trend in prior years

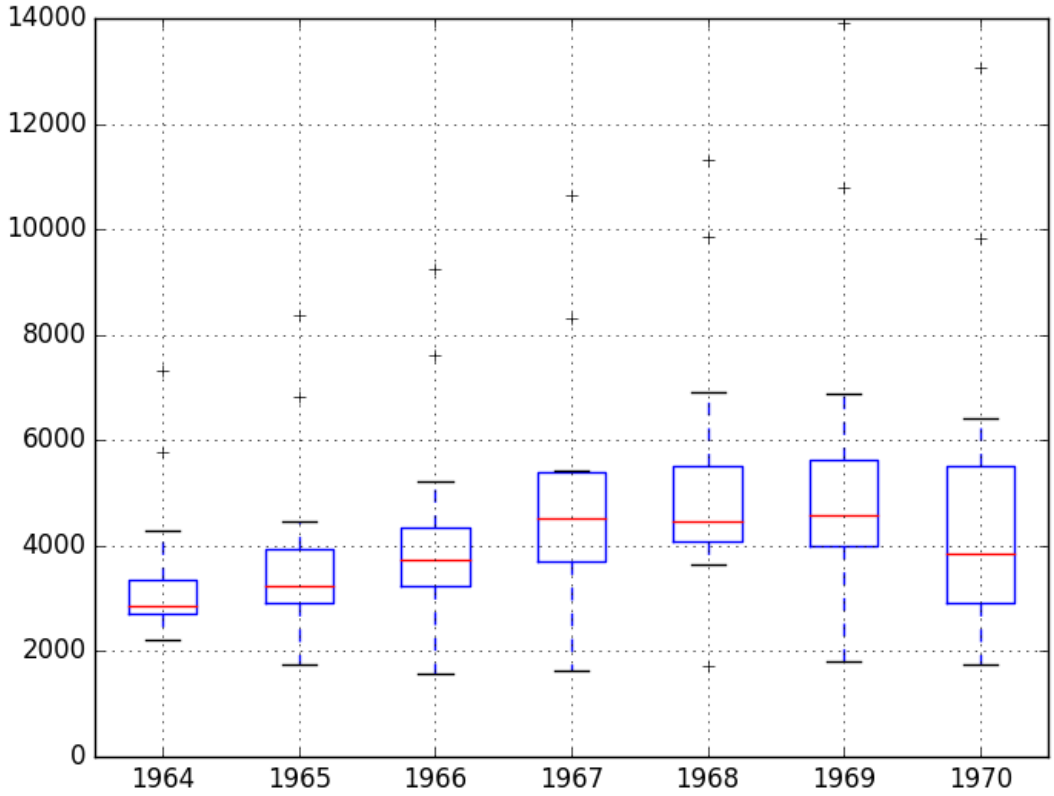

Figure 32.4: Box and whisker plots of the training set for the Champagne Sales dataset.

The observations suggest perhaps some growth trend over the years and outliers that may be a part of the seasonal cycle. This yearly view of the data is an interesting avenue and could be pursued further by looking at summary statistics from year-to-year and changes in summary stats from year-to-year.

32.6 ARIMA Models

In this section, we will develop Autoregressive Integrated Moving Average, or ARIMA, models for the problem. We will approach modeling by both manual and automatic configuration of the ARIMA model. This will be followed by a third step of investigating the residual errors of the chosen model. As such, this section is broken down into 3 steps:

1. Manually Configure the ARIMA.
2. Automatically Configure the ARIMA.
3. Review Residual Errors.

32.6.1 Manually Configured ARIMA

The `ARIMA(p,d,q)` model requires three parameters and is traditionally configured manually. Analysis of the time series data assumes that we are working with a stationary time series. The time series is almost certainly non-stationary. We can make it stationary this by first differencing the series and using a statistical test to confirm that the result is stationary.

The seasonality in the series is seemingly year-to-year. Seasonal data can be differenced by subtracting the observation from the same time in the previous cycle, in this case the same month in the previous year. This does mean that we will lose the first year of observations as there is no prior year to difference with. The example below creates a deseasonalized version of the series and saves it to file `stationary.csv`.

```
# create and summarize stationary version of time series
from pandas import read_csv
from pandas import Series
from statsmodels.tsa.stattools import adfuller
from matplotlib import pyplot

# create a differenced series
def difference(dataset, interval=1):
  diff = list()
  for i in range(interval, len(dataset)):
    value = dataset[i] - dataset[i - interval]
    diff.append(value)
  return Series(diff)

series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
X = series.values
X = X.astype('float32')
# difference data
months_in_year = 12
stationary = difference(X, months_in_year)
stationary.index = series.index[months_in_year:]
# check if stationary
result = adfuller(stationary)
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))
# save
stationary.to_csv('stationary.csv')
# plot
stationary.plot()
pyplot.show()
```

Listing 32.12: Create a stationary version of the dataset and plot the differenced result.

Running the example outputs the result of a statistical significance test of whether the differenced series is stationary. Specifically, the augmented Dickey-Fuller test. The results show that the test statistic value -7.134898 is smaller than the critical value at 1% of -3.515. 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: -7.134898
p-value: 0.000000
Critical Values:
  5%: -2.898
  1%: -3.515
  10%: -2.586
```

Listing 32.13: Example output of a statistical test on the differenced Champagne Sales dataset.

For reference, the seasonal difference operation can be inverted by adding the observation for the same month the year before. This is needed in the case that predictions are made by a model fit on seasonally differenced data. The function to invert the seasonal difference operation is listed below for completeness.

```
# invert differenced value
def inverse_difference(history, yhat, interval=1):
  return yhat + history[-interval]
```

Listing 32.14: Function to invert differencing.

A plot of the differenced dataset is also created. The plot does not show any obvious seasonality or trend, suggesting the seasonally differenced dataset is a good starting point for modeling. We will use this dataset as an input to the ARIMA model. It also suggests that no further differencing may be required, and that the `d` parameter may be set to 0.

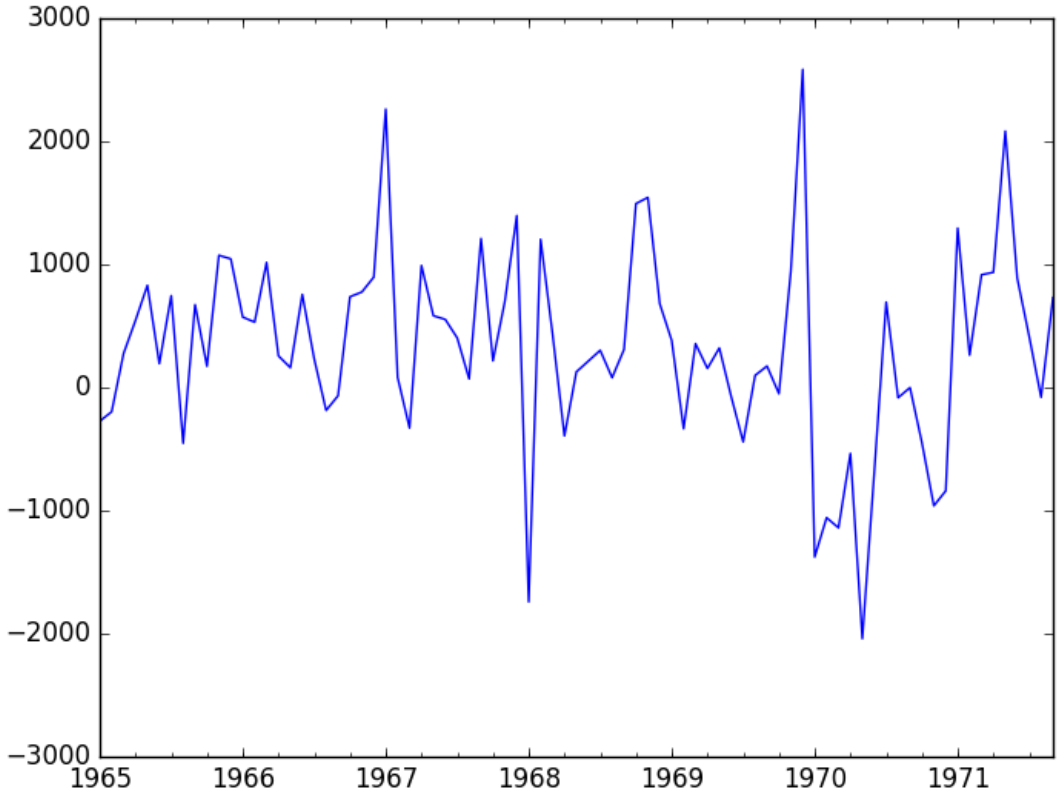

Figure 32.5: Line plot of the differenced Champagne Sales dataset.

The next first step is to select the lag values for the Autoregression (AR) and Moving Average (MA) parameters, `p` and `q` respectively. We can do this by reviewing Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots. Note, we are now using the seasonally differenced `stationary.csv` as our dataset. This is because the manual seasonal differencing performed is different from the `lag=1` differencing performed by the ARIMA model with the `d` parameter. The example below creates ACF and PACF plots for the series.

```
# ACF and PACF plots of time series
from pandas import read_csv
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from matplotlib import pyplot
series = read_csv('stationary.csv', header=None, index_col=0, parse_dates=True,
    squeeze=True)
pyplot.figure()
pyplot.subplot(211)
plot_acf(series, ax=pyplot.gca())
pyplot.subplot(212)
plot_pacf(series, ax=pyplot.gca())
pyplot.show()
```

Listing 32.15: ACF and PACF plots of the Champagne Sales dataset.

Run the example and review the plots for insights into how to set the `p` and `q` variables for the ARIMA model. Below are some observations from the plots.

- The ACF shows a significant lag for 1 month.
- The PACF shows a significant lag for 1 month, with perhaps some significant lag at 12 and 13 months.
- Both the ACF and PACF show a drop-off at the same point, perhaps suggesting a mix of AR and MA.

A good starting point for the `p` and `q` values is also 1. The PACF plot also suggests that there is still some seasonality present in the differenced data. We may consider a better model of seasonality, such as modeling it directly and explicitly removing it from the model rather than seasonal differencing.

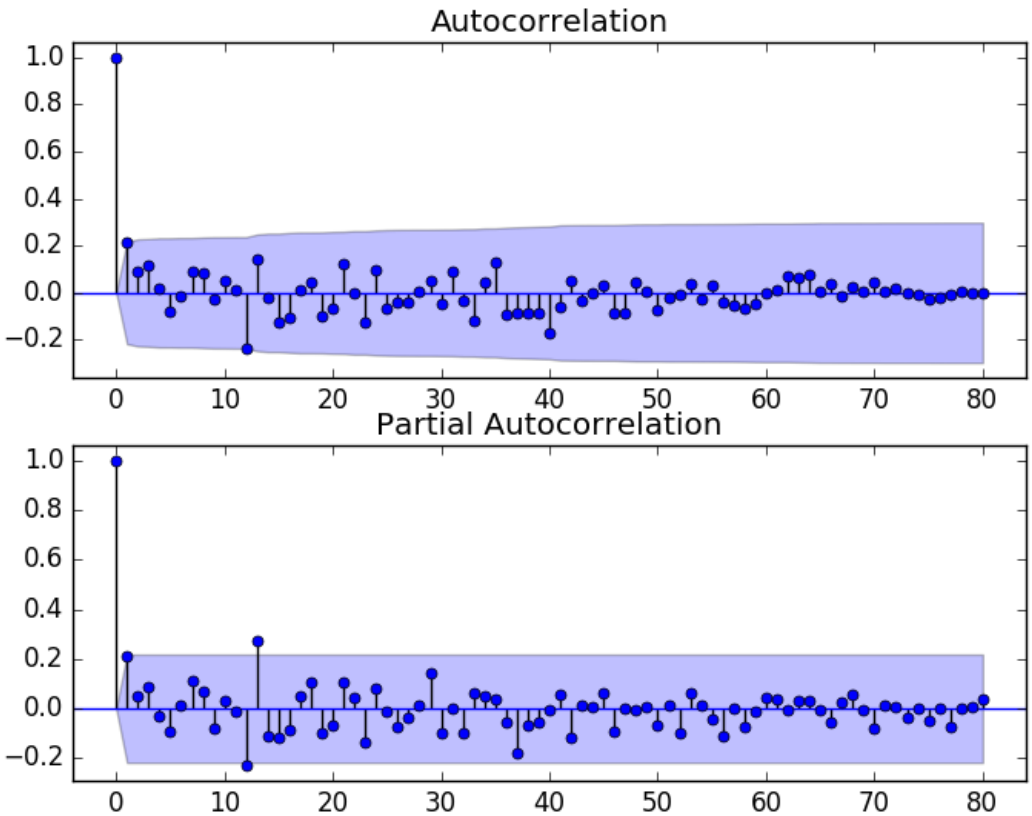

Figure 32.6: ACF and PACF plots of the differenced Champagne Sales dataset.

This quick analysis suggests an `ARIMA(1,0,1)` on the stationary data may be a good starting point. The historic observations will be seasonally differenced prior to the fitting of each ARIMA model. The differencing will be inverted for all predictions made to make them directly comparable to the expected observation in the original sale count units. Experimentation shows that this configuration of ARIMA does not converge and results in errors by the underlying

library. Further experimentation showed that adding one level of differencing to the stationary data made the model more stable. The model can be extended to `ARIMA(1,1,1)`.

We will also disable the automatic addition of a trend constant from the model by setting the `trend` argument to `nc` for no constant in the call to `fit()`. From experimentation, I find that this can result in better forecast performance on some problems. The example below demonstrates the performance of this ARIMA model on the test harness.

```
# evaluate manually configured ARIMA model
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA
from math import sqrt

# create a differenced series
def difference(dataset, interval=1):
  diff = list()
  for i in range(interval, len(dataset)):
    value = dataset[i] - dataset[i - interval]
    diff.append(value)
  return diff

# invert differenced value
def inverse_difference(history, yhat, interval=1):
  return yhat + history[-interval]

# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # difference data
  months_in_year = 12
  diff = difference(history, months_in_year)
  # predict
  model = ARIMA(diff, order=(1,1,1))
  model_fit = model.fit(trend='nc', disp=0)
  yhat = model_fit.forecast()[0]
  yhat = inverse_difference(history, yhat, months_in_year)
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
```

Listing 32.16: Manual ARIMA model on the Champagne Sales dataset.

Note, you may see a warning message from the underlying linear algebra library; this can be

ignored for now. Running this example results in an RMSE of 956.942, which is dramatically better than the persistence RMSE of 3186.501.

```
...
>Predicted=3157.018, Expected=5010
>Predicted=4615.082, Expected=4874
>Predicted=4624.998, Expected=4633
>Predicted=2044.097, Expected=1659
>Predicted=5404.428, Expected=5951
RMSE: 956.942
```

Listing 32.17: Example output of the manual ARIMA model on the Champagne Sales dataset.

This is a great start, but we may be able to get improved results with a better configured ARIMA model.

32.6.2 Grid Search ARIMA Hyperparameters

The ACF and PACF plots suggest that an `ARIMA(1,0,1)` or similar may be the best that we can do. To confirm this analysis, we can grid search a suite of ARIMA hyperparameters and check that no models result in better out of sample RMSE performance. In this section, we will search values of `p`, `d`, and `q` for combinations (skipping those that fail to converge), and find the combination that results in the best performance on the test set. We will use a grid search to explore all combinations in a subset of integer values. Specifically, we will search all combinations of the following parameters:

- `p`: 0 to 6.
- `d`: 0 to 2.
- `q`: 0 to 6.

This is $(7 \times 3 \times 7)$, or 147, potential runs of the test harness and will take some time to execute. It may be interesting to evaluate MA models with a lag of 12 or 13 as were noticed as potentially interesting from reviewing the ACF and PACF plots. Experimentation suggested that these models may not be stable, resulting in errors in the underlying mathematical libraries. The complete worked example with the grid search version of the test harness is listed below.

```
# grid search ARIMA parameters for time series
import warnings
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt
import numpy

# create a differenced series
def difference(dataset, interval=1):
  diff = list()
  for i in range(interval, len(dataset)):
    value = dataset[i] - dataset[i - interval]
    diff.append(value)
  return numpy.array(diff)
```

```
# invert differenced value
def inverse_difference(history, yhat, interval=1):
  return yhat + history[-interval]

# evaluate an ARIMA model for a given order (p,d,q) and return RMSE
def evaluate_arima_model(X, arima_order):
  # prepare training dataset
  X = X.astype('float32')
  train_size = int(len(X) * 0.50)
  train, test = X[0:train_size], X[train_size:]
  history = [x for x in train]
  # make predictions
  predictions = list()
  for t in range(len(test)):
    # difference data
    months_in_year = 12
    diff = difference(history, months_in_year)
    model = ARIMA(diff, order=arima_order)
    model_fit = model.fit(trend='nc', disp=0)
    yhat = model_fit.forecast()[0]
    yhat = inverse_difference(history, yhat, months_in_year)
    predictions.append(yhat)
    history.append(test[t])
  # calculate out of sample error
  rmse = sqrt(mean_squared_error(test, predictions))
  return rmse

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
  dataset = dataset.astype('float32')
  best_score, best_cfg = float("inf"), None
  for p in p_values:
    for d in d_values:
      for q in q_values:
        order = (p,d,q)
        try:
          rmse = evaluate_arima_model(dataset, order)
          if rmse < best_score:
            best_score, best_cfg = rmse, order
          print('ARIMA%s RMSE=%.3f' % (order,rmse))
        except:
          continue
  print('Best ARIMA%s RMSE=%.3f' % (best_cfg, best_score))

# load dataset
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# evaluate parameters
p_values = range(0, 7)
d_values = range(0, 3)
q_values = range(0, 7)
warnings.filterwarnings("ignore")
evaluate_models(series.values, p_values, d_values, q_values)
```

Listing 32.18: Grid Search ARIMA models on the Champagne Sales dataset.

Running the example runs through all combinations and reports the results on those that converge without error. The example takes a little over 2 minutes to run on modern hardware. The results show that the best configuration discovered was `ARIMA(0,0,1)` with an RMSE of 939.464, slightly lower than the manually configured ARIMA from the previous section. This difference may or may not be statistically significant.

```
...
ARIMA(5, 1, 2) RMSE=1003.200
ARIMA(5, 2, 1) RMSE=1053.728
ARIMA(6, 0, 0) RMSE=996.466
ARIMA(6, 1, 0) RMSE=1018.211
ARIMA(6, 1, 1) RMSE=1023.762
Best ARIMA(0, 0, 1) RMSE=939.464
```

Listing 32.19: Example output of grid searching ARIMA models on the Champagne Sales dataset.

We will select this `ARIMA(0,0,1)` model going forward.

32.6.3 Review Residual Errors

A good final check of a model is to review residual forecast errors. Ideally, the distribution of residual errors should be a Gaussian with a zero mean. We can check this by using summary statistics and plots to investigate the residual errors from the `ARIMA(0,0,1)` model. The example below calculates and summarizes the residual forecast errors.

```
# summarize ARIMA forecast residuals
from pandas import read_csv
from pandas import DataFrame
from statsmodels.tsa.arima_model import ARIMA
from matplotlib import pyplot

# create a differenced series
def difference(dataset, interval=1):
  diff = list()
  for i in range(interval, len(dataset)):
    value = dataset[i] - dataset[i - interval]
    diff.append(value)
  return diff

# invert differenced value
def inverse_difference(history, yhat, interval=1):
  return yhat + history[-interval]

# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
```

```
  # difference data
  months_in_year = 12
  diff = difference(history, months_in_year)
  # predict
  model = ARIMA(diff, order=(0,0,1))
  model_fit = model.fit(trend='nc', disp=0)
  yhat = model_fit.forecast()[0]
  yhat = inverse_difference(history, yhat, months_in_year)
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
# errors
residuals = [test[i]-predictions[i] for i in range(len(test))]
residuals = DataFrame(residuals)
print(residuals.describe())
# plot
pyplot.figure()
pyplot.subplot(211)
residuals.hist(ax=pyplot.gca())
pyplot.subplot(212)
residuals.plot(kind='kde', ax=pyplot.gca())
pyplot.show()
```

Listing 32.20: Plot and summarize errors of an ARIMA model on the Champagne Sales dataset.

Running the example first describes the distribution of the residuals. We can see that the distribution has a right shift and that the mean is non-zero at 165.904728. This is perhaps a sign that the predictions are biased.

```
count   47.000000
mean   165.904728
std    934.696199
min   -2164.247449
25%    -289.651596
50%     191.759548
75%     732.992187
max    2367.304748
```

Listing 32.21: Example output of summary statistics on the residual errors of an ARIMA model on the Champagne Sales dataset.

The distribution of residual errors is also plotted. The graphs suggest a Gaussian-like distribution with a bumpy left tail, providing further evidence that perhaps a power transform might be worth exploring.

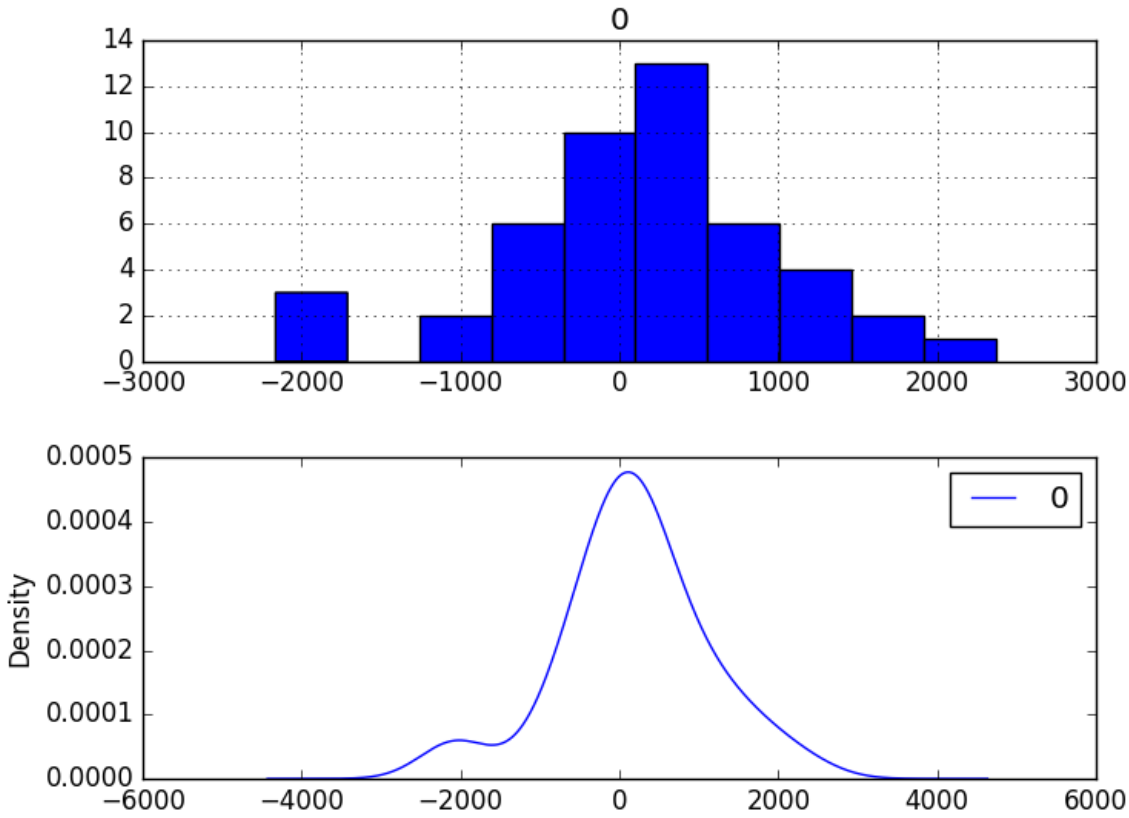

Figure 32.7: Density plots of residual errors on the Champagne Sales dataset.

We could use this information to bias-correct predictions by adding the mean residual error of 165.904728 to each forecast made. The example below performs this bias correlation.

```
# plots of residual errors of bias corrected forecasts
from pandas import read_csv
from pandas import DataFrame
from statsmodels.tsa.arima_model import ARIMA
from matplotlib import pyplot
from sklearn.metrics import mean_squared_error
from math import sqrt

# create a differenced series
def difference(dataset, interval=1):
  diff = list()
  for i in range(interval, len(dataset)):
    value = dataset[i] - dataset[i - interval]
    diff.append(value)
  return diff

# invert differenced value
def inverse_difference(history, yhat, interval=1):
  return yhat + history[-interval]

# load data
```

```
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
bias = 165.904728
for i in range(len(test)):
  # difference data
  months_in_year = 12
  diff = difference(history, months_in_year)
  # predict
  model = ARIMA(diff, order=(0,0,1))
  model_fit = model.fit(trend='nc', disp=0)
  yhat = model_fit.forecast()[0]
  yhat = bias + inverse_difference(history, yhat, months_in_year)
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
# report performance
rmse = sqrt(mean_squared_error(test, predictions))
print('RMSE: %.3f' % rmse)
# errors
residuals = [test[i]-predictions[i] for i in range(len(test))]
residuals = DataFrame(residuals)
print(residuals.describe())
# plot
pyplot.figure()
pyplot.subplot(211)
residuals.hist(ax=pyplot.gca())
pyplot.subplot(212)
residuals.plot(kind='kde', ax=pyplot.gca())
pyplot.show()
```

Listing 32.22: Plot and summarize bias corrected errors of an ARIMA model on the Champagne Sales dataset.

The performance of the predictions is improved very slightly from 939.464 to 924.699, which may or may not be significant. The summary of the forecast residual errors shows that the mean was indeed moved to a value very close to zero.

```
RMSE: 924.699
                 0
count  4.700000e+01
mean   4.965016e-07
std    9.346962e+02
min   -2.330152e+03
25%   -4.555563e+02
50%    2.585482e+01
75%    5.670875e+02
max    2.201400e+03
```

Listing 32.23: Example output of summary statistics of residual errors of a bias corrected

ARIMA model on the Champagne Sales dataset.

Finally, density plots of the residual error do show a small shift towards zero. It is debatable whether this bias correction is worth it, but we will use it for now.

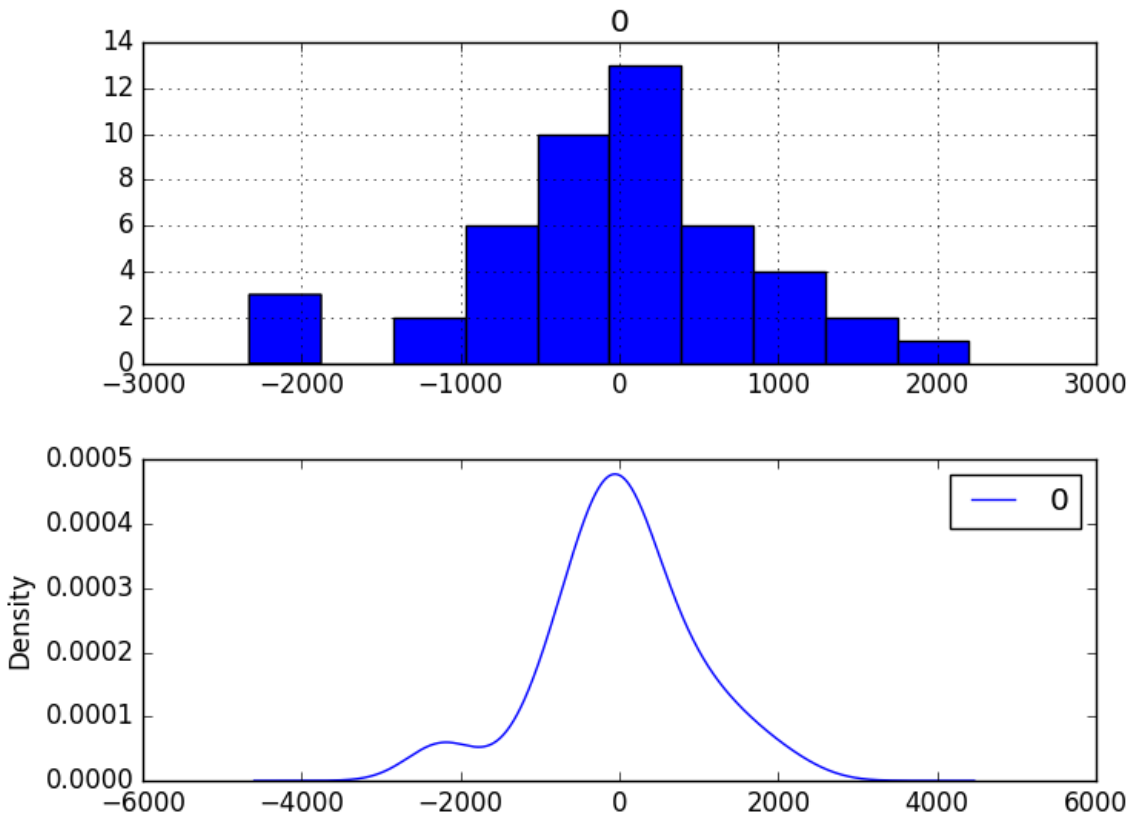

Figure 32.8: Density plots of residual errors of a bias corrected model on the Champagne Sales dataset.

It is also a good idea to check the time series of the residual errors for any type of autocorrelation. If present, it would suggest that the model has more opportunity to model the temporal structure in the data. The example below re-calculates the residual errors and creates ACF and PACF plots to check for any significant autocorrelation.

```
# ACF and PACF plots of residual errors of bias corrected forecasts
from pandas import read_csv
from pandas import DataFrame
from statsmodels.tsa.arima_model import ARIMA
from matplotlib import pyplot
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf

# create a differenced series
def difference(dataset, interval=1):
  diff = list()
  for i in range(interval, len(dataset)):
```

```
    value = dataset[i] - dataset[i - interval]
    diff.append(value)
  return diff

# invert differenced value
def inverse_difference(history, yhat, interval=1):
  return yhat + history[-interval]

# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
train_size = int(len(X) * 0.50)
train, test = X[0:train_size], X[train_size:]
# walk-forward validation
history = [x for x in train]
predictions = list()
for i in range(len(test)):
  # difference data
  months_in_year = 12
  diff = difference(history, months_in_year)
  # predict
  model = ARIMA(diff, order=(0,0,1))
  model_fit = model.fit(trend='nc', disp=0)
  yhat = model_fit.forecast()[0]
  yhat = inverse_difference(history, yhat, months_in_year)
  predictions.append(yhat)
  # observation
  obs = test[i]
  history.append(obs)
# errors
residuals = [test[i]-predictions[i] for i in range(len(test))]
residuals = DataFrame(residuals)
print(residuals.describe())
# plot
pyplot.figure()
pyplot.subplot(211)
plot_acf(residuals, ax=pyplot.gca())
pyplot.subplot(212)
plot_pacf(residuals, ax=pyplot.gca())
pyplot.show()
```

Listing 32.24: ACF and PACF plots of forecast residuals for the bias-corrected ARIMA model on the Champagne Sales dataset.

The results suggest that what little autocorrelation is present in the time series has been captured by the model.

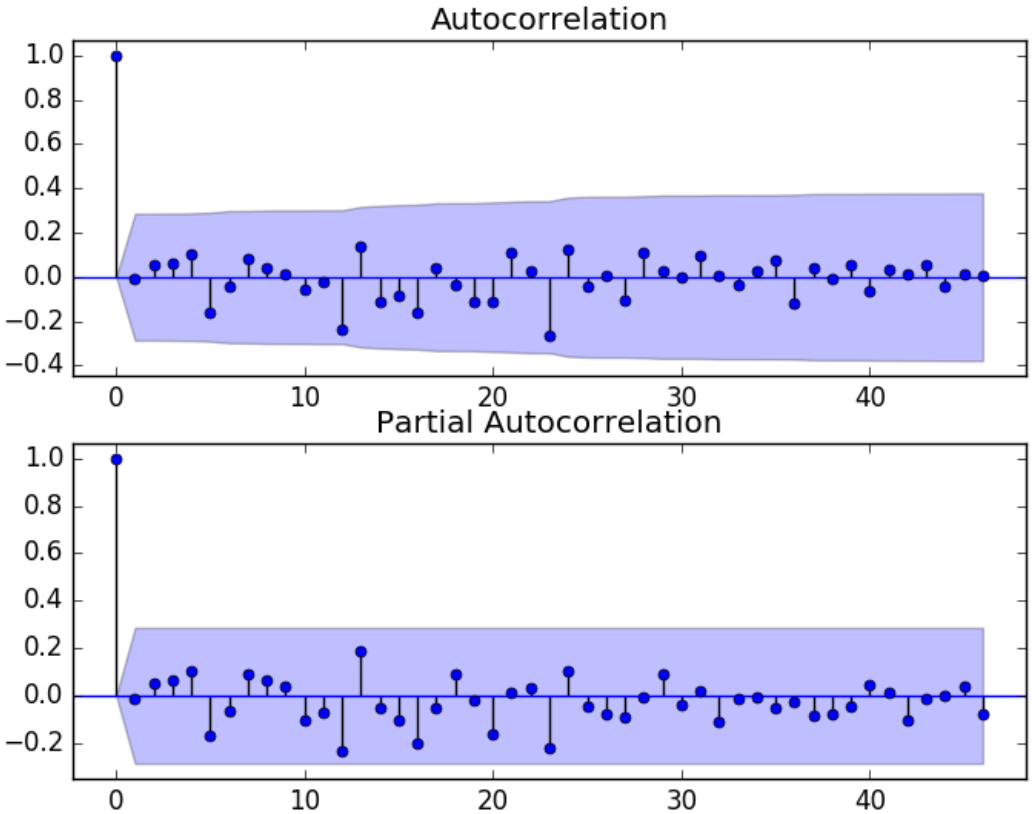

Figure 32.9: ACF and PACF plots of residual errors on the bias corrected ARIMA model of the Champagne Sales dataset.

32.7 Model Validation

After models have been developed and a final model selected, it must be validated and finalized. Validation is an optional part of the process, but one that provides a last check to ensure we have not fooled or misled ourselves. This section includes the following steps:

- **Finalize Model**: Train and save the final model.
- **Make Prediction**: Load the finalized model and make a prediction.
- **Validate Model**: Load and validate the final model.

32.7.1 Finalize Model

Finalizing the model involves fitting an ARIMA model on the entire dataset, in this case on a transformed version of the entire dataset. Once fit, the model can be saved to file for later use. The example below trains an `ARIMA(0,0,1)` model on the dataset and saves the whole fit object and the bias to file. There is a bug in the current version of the Statsmodels library that results in an error when you try to load a saved ARIMA model from file. Learn more about

this bug and the workaround in Appendix B. The example below saves the fit model to file in the correct state so that it can be loaded successfully later.

```
# save finalized model
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
import numpy

# monkey patch around bug in ARIMA class
def __getnewargs__(self):
  return ((self.endog),(self.k_lags, self.k_diff, self.k_ma))

ARIMA.__getnewargs__ = __getnewargs__

# create a differenced series
def difference(dataset, interval=1):
  diff = list()
  for i in range(interval, len(dataset)):
    value = dataset[i] - dataset[i - interval]
    diff.append(value)
  return diff

# load data
series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
# difference data
months_in_year = 12
diff = difference(X, months_in_year)
# fit model
model = ARIMA(diff, order=(0,0,1))
model_fit = model.fit(trend='nc', disp=0)
# bias constant, could be calculated from in-sample mean residual
bias = 165.904728
# save model
model_fit.save('model.pkl')
numpy.save('model_bias.npy', [bias])
```

Listing 32.25: Save a finalized ARIMA model to file for the Champagne Sales dataset.

Running the example creates two local files:

- `model.pkl` This is the `ARIMAResult` object from the call to `ARIMA.fit()`. This includes the coefficients and all other internal data returned when fitting the model.
- `model_bias.npy` This is the bias value stored as a one-row, one-column NumPy array.

32.7.2 Make Prediction

A natural case may be to load the model and make a single forecast. This is relatively straightforward and involves restoring the saved model and the bias and calling the `forecast()` function. To invert the seasonal differencing, the historical data must also be loaded. The example below loads the model, makes a prediction for the next time step, and prints the prediction.

```
# load finalized model and make a prediction
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMAResults
import numpy

# invert differenced value
def inverse_difference(history, yhat, interval=1):
  return yhat + history[-interval]

series = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
months_in_year = 12
model_fit = ARIMAResults.load('model.pkl')
bias = numpy.load('model_bias.npy')
yhat = float(model_fit.forecast()[0])
yhat = bias + inverse_difference(series.values, yhat, months_in_year)
print('Predicted: %.3f' % yhat)
```

Listing 32.26: Load the finalized ARIMA model and make a prediction on the Champagne Sales dataset.

Running the example prints the prediction of about 6794.

```
Predicted: 6794.773
```

Listing 32.27: Example output loading the ARIMA model and making a single prediction on the Champagne Sales dataset.

If we peek inside `validation.csv`, we can see that the value on the first row for the next time period is 6981. The prediction is in the right ballpark.

32.7.3 Validate Model

We can load the model and use it in a pretend operational manner. In the test harness section, we saved the final 12 months of the original dataset in a separate file to validate the final model. We can load this `validation.csv` file now and use it see how well our model really is on unseen data. There are two ways we might proceed:

- Load the model and use it to forecast the next 12 months. The forecast beyond the first one or two months will quickly start to degrade in skill.
- Load the model and use it in a rolling-forecast manner, updating the transform and model for each time step. This is the preferred method as it is how one would use this model in practice as it would achieve the best performance.

As with model evaluation in previous sections, we will make predictions in a rolling-forecast manner. This means that we will step over lead times in the validation dataset and take the observations as an update to the history.

```
# load and evaluate the finalized model on the validation dataset
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.arima_model import ARIMAResults
```

```
from sklearn.metrics import mean_squared_error
from math import sqrt
import numpy

# create a differenced series
def difference(dataset, interval=1):
  diff = list()
  for i in range(interval, len(dataset)):
    value = dataset[i] - dataset[i - interval]
    diff.append(value)
  return diff

# invert differenced value
def inverse_difference(history, yhat, interval=1):
  return yhat + history[-interval]

# load and prepare datasets
dataset = read_csv('dataset.csv', header=None, index_col=0, parse_dates=True, squeeze=True)
X = dataset.values.astype('float32')
history = [x for x in X]
months_in_year = 12
validation = read_csv('validation.csv', header=None, index_col=0, parse_dates=True,
    squeeze=True)
y = validation.values.astype('float32')
# load model
model_fit = ARIMAResults.load('model.pkl')
bias = numpy.load('model_bias.npy')
# make first prediction
predictions = list()
yhat = float(model_fit.forecast()[0])
yhat = bias + inverse_difference(history, yhat, months_in_year)
predictions.append(yhat)
history.append(y[0])
print('>Predicted=%.3f, Expected=%3.f' % (yhat, y[0]))
# rolling forecasts
for i in range(1, len(y)):
  # difference data
  months_in_year = 12
  diff = difference(history, months_in_year)
  # predict
  model = ARIMA(diff, order=(0,0,1))
  model_fit = model.fit(trend='nc', disp=0)
  yhat = model_fit.forecast()[0]
  yhat = bias + inverse_difference(history, yhat, months_in_year)
  predictions.append(yhat)
  # observation
  obs = y[i]
  history.append(obs)
  print('>Predicted=%.3f, Expected=%3.f' % (yhat, obs))
# report performance
rmse = sqrt(mean_squared_error(y, predictions))
print('RMSE: %.3f' % rmse)
pyplot.plot(y)
pyplot.plot(predictions, color='red')
pyplot.show()
```

Listing 32.28: Load and validate the finalized model on the Champagne Sales dataset.

Running the example prints each prediction and expected value for the time steps in the validation dataset. The final RMSE for the validation period is predicted at 361.110 million sales. This is much better than the expectation of an error of a little more than 924 million sales per month.

```
>Predicted=6794.773, Expected=6981
>Predicted=10101.763, Expected=9851
>Predicted=13219.067, Expected=12670
>Predicted=3996.535, Expected=4348
>Predicted=3465.934, Expected=3564
>Predicted=4522.683, Expected=4577
>Predicted=4901.336, Expected=4788
>Predicted=5190.094, Expected=4618
>Predicted=4930.190, Expected=5312
>Predicted=4944.785, Expected=4298
>Predicted=1699.409, Expected=1413
>Predicted=6085.324, Expected=5877
RMSE: 361.110
```

Listing 32.29: Example output loading and validating the finalized model on the Champagne Sales dataset.

A plot of the predictions compared to the validation dataset is also provided. At this scale on the plot, the 12 months of forecast sales figures look fantastic.

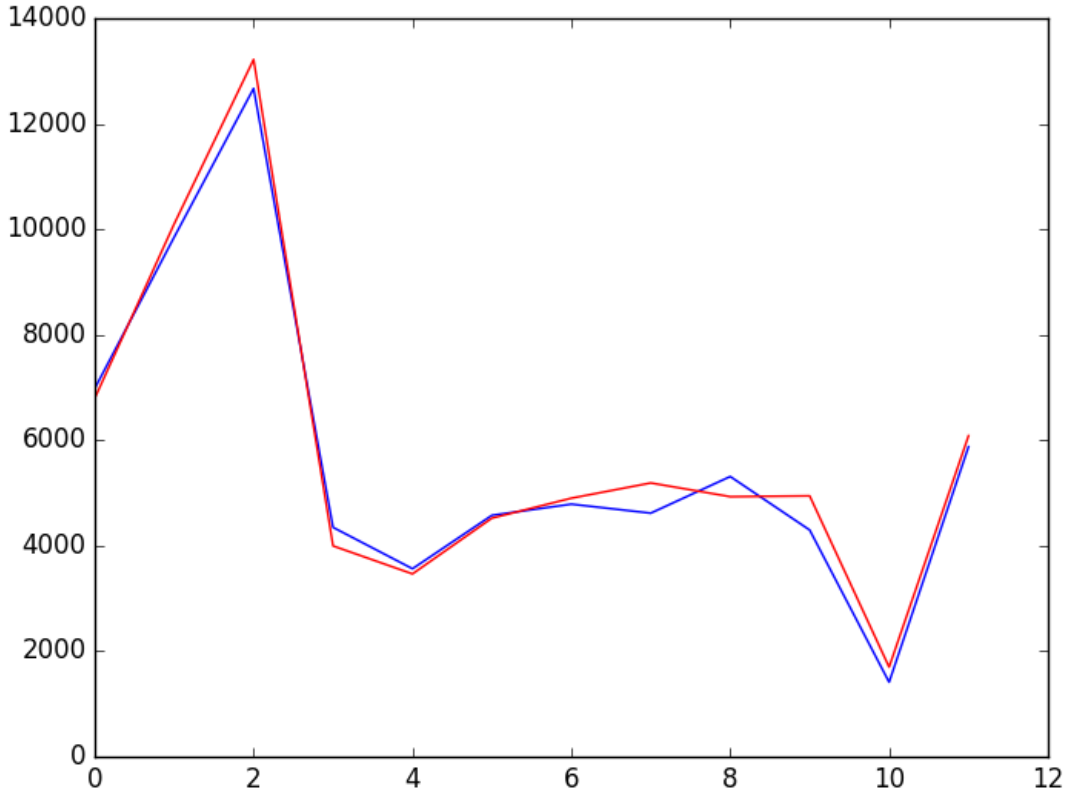

Figure 32.10: Line plot of the expected values (blue) and predictions (red) for the validation dataset.

32.8 Summary

In this tutorial, you discovered the steps and the tools for a time series forecasting project with Python. We have covered a lot of ground in this tutorial; specifically:

- How to develop a test harness with a performance measure and evaluation method and how to quickly develop a baseline forecast and skill.
- How to use time series analysis to raise ideas for how to best model the forecast problem.
- How to develop an ARIMA model, save it, and later load it to make predictions on new data.

32.8.1 Next

This concludes Part VI. Next in Part VII we will conclude the book and look at resources that you can use to dive deeper into the subject.

Part VII

Conclusions

Chapter 33

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come.

- **Time series fundamentals**: You know about time series problems, how they differ from other types of machine learning problems and how to frame time series as supervised learning.
- **Time series data preparation**: You know how to load time series data, construct new features, resample the frequency of the data, visualize, transform and smooth time series data.
- **Time series structures and dynamics**: You know about the standard decompositional approach to time series, how to identify and neutralize trends, seasonality and test for stationarity.
- **Time series model evaluation**: You know how to evaluate forecast models for time series, differentiate the performance measures, develop persistence models, visualize residual errors and reframe time series problems.
- **Time series forecasting**: You know how to develop linear forecast models including autoregression, moving average and ARIMA models using the Box-Jenkins method and how to tune these models, finalize the models and calculate confidence intervals on forecasts.
- **Tie lessons together**: You know how to tie together all of these skills into time series forecasting projects and deliver a finalized model that can make a prediction about the future.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable skill of being able to implement and work through time series forecasting problems using Python. This is a platform that is used by a majority of working data scientist professionals. The sky is the limit.

I want to take a moment and sincerely thank you for letting me help you start your time series journey with Python. I hope you keep learning and have fun as you continue to master machine learning.

Chapter 34

Further Reading

This is just the beginning of your journey with time series forecasting with Python. As you start to work on methods or expand your existing knowledge of algorithms you may need help. This chapter points out some of the best sources of help.

34.1 Applied Time Series

There are a wealth of good books on applied time series suitable for the machine learning perspective. Most are focused on time series using the R platform. Nevertheless, even though the code is not directly usable, the concepts, methods and descriptions are still valuable. This section lists some useful books on applied time series that you may consider.

- Rob J. Hyndman and George Athanasopoulos, **Forecasting: principles and practice**, 2013.
  http://amzn.to/2lln93c
- Galit Shmueli and Kenneth Lichtendahl, **Practical Time Series Forecasting with R: A Hands-On Guide**, 2016.
  http://amzn.to/2k3QpuV
- Paul S. P. Cowpertwait and Andrew V. Metcalfem **Introductory Time Series with R**, 2009.
  http://amzn.to/2kIQscS

34.2 Time Series

There are a ton of textbooks on time series analysis and forecasting. Most are intended for undergraduate and postgraduate courses, requiring knowledge of statistics, calculus and probability. If I were to choose one, it would be the seminal book on the topic by the developers of the Box-Jenkins method:

- George E. P. Box, Gwilym M. Jenkins, et al., **Time Series Analysis: Forecasting and Control**, 2015.
  http://amzn.to/2kIPc9j

34.3 Machine Learning

Although this is an introductory time series book, it paves the way for using more advanced machine learning methods on time series data. This section lists some books on applied machine learning with Python that you may consider to help take the next step.

- Jason Brownlee, **Machine Learning Mastery With Python**, 2017. (my book)
  https://machinelearningmastery.com/machine-learning-with-python/
- Sebastian Raschka, **Python Machine Learning**, 2015.
  http://amzn.to/2lpiv01

34.4 Getting Help

What if you need even more help? Below are some resources where you can ask more detailed technical questions and get helpful technical answers:

- **Cross Validated**: Ask questions related to time series forecasting or machine learning.
  http://stats.stackexchange.com
- **Stack Overflow**: Ask questions related to Python programming or library usage.
  http://stackoverflow.com

These are excellent resources both for posting unique questions, but also for searching through the answers to questions on related topics.

34.5 Contact the Author

You are not alone. If you ever have any questions about time series forecasting or this book, please contact me directly. I will do my best to help.

Jason Brownlee
Jason@MachineLearningMastery.com

Part VIII

Appendix

Appendix A

Standard Time Series Datasets

This appendix describes standard time series datasets used in lessons throughput this book. The website Data Market provides access to a large number of time series datasets. Specifically, the Time Series Data Library[1] created by Rob Hyndman, Professor of Statistics at Monash University, Australia. Five different datasets taken from Data Market are described in this Appendix, they are:

1. Shampoo Sales Dataset.
2. Minimum Daily Temperatures Dataset.
3. Monthly Sunspots Dataset.
4. Daily Female Births Dataset.
5. Airline Passengers Dataset.

Note: When downloading datasets from Data Market, you will need to manually clean up the file before use. Specifically, after downloading a dataset, open it in your favorite text editor and make the following changes:

- Header column names are way too long and detailed. Change them to simple one-word column names like *sales* or *temperature*.
- The files contain footer information which breaks file loading routines in Python. Scroll to the end of the file and delete all footer information and empty lines.
- Some datasets (like the Minimum Daily Temperatures dataset) contain a question mark character (`?`) that prevents the file from loading correctly in Python. Find and delete question mark characters from downloaded files.

[1] https://datamarket.com/data/list/?q=provider:tsdl

A.1 Shampoo Sales Dataset

This dataset describes the monthly number of sales of shampoo over a 3 year period. The units are a sales count and there are 36 observations. The original dataset is credited to Makridakis, Wheelwright and Hyndman (1998). Download the dataset and place it in your current working directory with the filename `shampoo-sales.csv`. The code below loads the Shampoo Sales dataset, prints the first 5 rows of data and graphs the data as a line plot.

```
# load the shampoo sales dataset
from matplotlib import pyplot
from pandas import read_csv
series = read_csv('shampoo-sales.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
print(series.head())
series.plot()
pyplot.show()
```

Listing A.1: Create a line plot of the Shampoo Sales dataset.

Below is a sample of the first 5 rows of the loaded dataset.

```
Month
1-01   266.0
1-02   145.9
1-03   183.1
1-04   119.3
1-05   180.3
```

Listing A.2: Example output of printing the first 5 rows of the Shampoo Sales dataset.

Below is a line plot of the entire dataset.

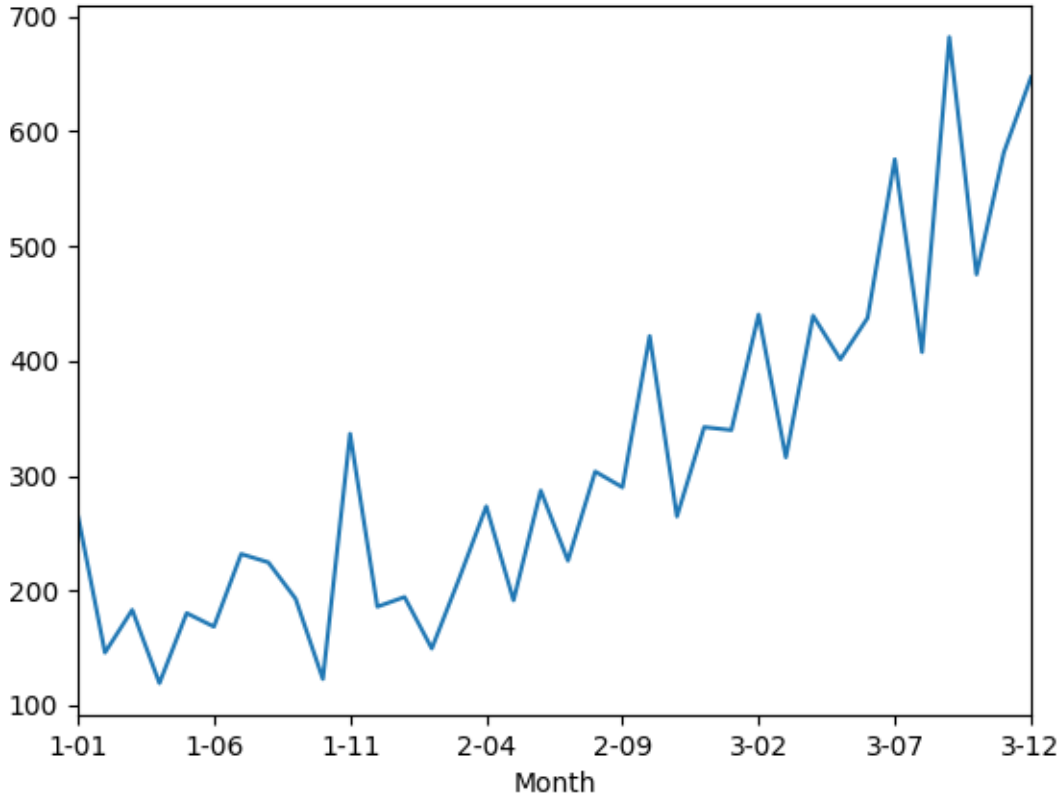

Figure A.1: Line plot of the Shampoo Sales dataset.

Learn more about the dataset on DataMarket: https://goo.gl/mZM1gk

A.2 Minimum Daily Temperatures Dataset

This dataset describes the minimum daily temperatures over 10 years (1981-1990) in the city Melbourne, Australia. The units are in degrees Celsius and there are 3650 observations. The source of the data is credited as the Australian Bureau of Meteorology. Download the dataset and place it in your current working directory with the filename `daily-minimum-temperatures.csv`. The code below loads the Minimum Daily Temperatures dataset, prints the first 5 rows of data and graphs the data as a line plot.

```
# load the minimum temperatures dataset
from matplotlib import pyplot
from pandas import read_csv
series = read_csv('daily-minimum-temperatures.csv', header=0, index_col=0,
    parse_dates=True, squeeze=True)
print(series.head())
series.plot()
pyplot.show()
```

Listing A.3: Create a line plot of the Minimum Daily Temperatures dataset.

Below is a sample of the first 5 rows of the loaded dataset.

```
Date
1981-01-01   20.7
1981-01-02   17.9
1981-01-03   18.8
1981-01-04   14.6
1981-01-05   15.8
```

Listing A.4: Example output of printing the first 5 rows of the Minimum Daily Temperatures dataset.

Below is a line plot of the entire dataset.

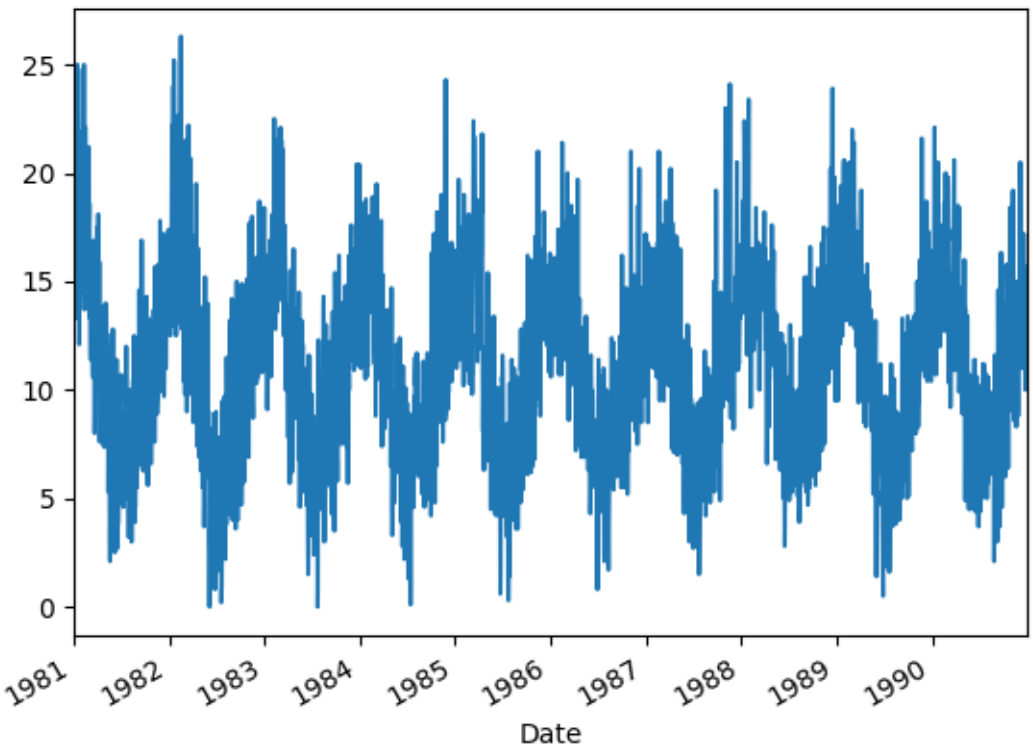

Figure A.2: Line plot of the Minimum Daily Temperatures dataset.

Learn more about the dataset on DataMarket: https://goo.gl/aGWZU2

A.3 Monthly Sunspots Dataset

This dataset describes a monthly count of the number of observed sunspots for just over 230 years (1749-1983). The units are a count and there are 2,820 observations. The source of the dataset is credited to Andrews & Herzberg (1985). Download the dataset and place it in your

current working directory with the filename `sunspots.csv`. The code below loads the Monthly Sunspot dataset, prints the first 5 rows of data and graphs the data as a line plot.

```
# load the sunspots dataset
from matplotlib import pyplot
from pandas import read_csv
series = read_csv('sunspots.csv', header=0, index_col=0, parse_dates=True, squeeze=True)
print(series.head())
series.plot()
pyplot.show()
```

Listing A.5: Create a line plot of the Monthly Sunspot dataset.

Below is a sample of the first 5 rows of the loaded dataset.

```
Month
1749-01-01   58.0
1749-02-01   62.6
1749-03-01   70.0
1749-04-01   55.7
1749-05-01   85.0
```

Listing A.6: Example output of printing the first 5 rows of the Monthly Sunspot dataset.

Below is a line plot of the entire dataset.

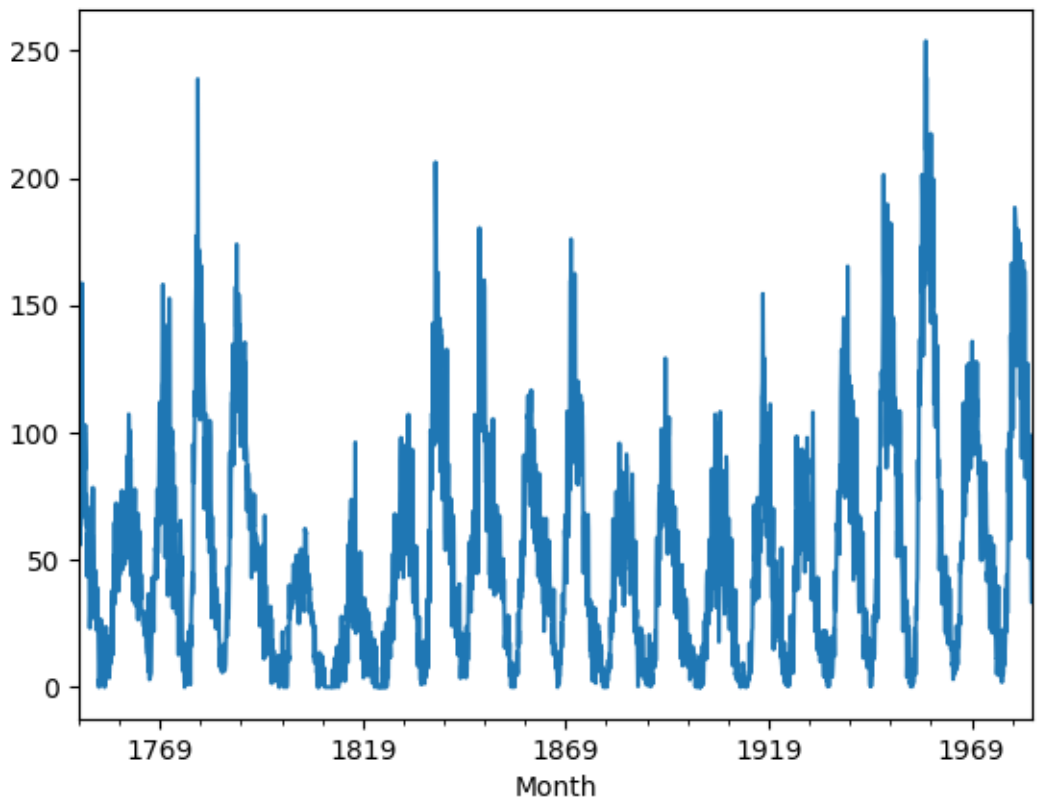

Figure A.3: Line plot of the Monthly Sunspot dataset.

Learn more about the dataset on DataMarket: https://goo.gl/NsK9x6

A.4 Daily Female Births Dataset

This dataset describes the number of daily female births in California in 1959. The units are a count and there are 365 observations. The source of the dataset is credited to Newton (1988). Download the dataset and place it in your current working directory with the filename `daily-total-female-births.csv`. The code below loads the Daily Female Births dataset, prints the first 5 rows of data and graphs the data as a line plot.

```
# load the female births dataset
from matplotlib import pyplot
from pandas import read_csv
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
print(series.head())
series.plot()
pyplot.show()
```

Listing A.7: Create a line plot of the Daily Female Births dataset.

Below is a sample of the first 5 rows of the loaded dataset.

```
Date
1959-01-01   35
1959-01-02   32
1959-01-03   30
1959-01-04   31
1959-01-05   44
```

Listing A.8: Example output of printing the first 5 rows of the Daily Female Births dataset.

Below is a line plot of the entire dataset.

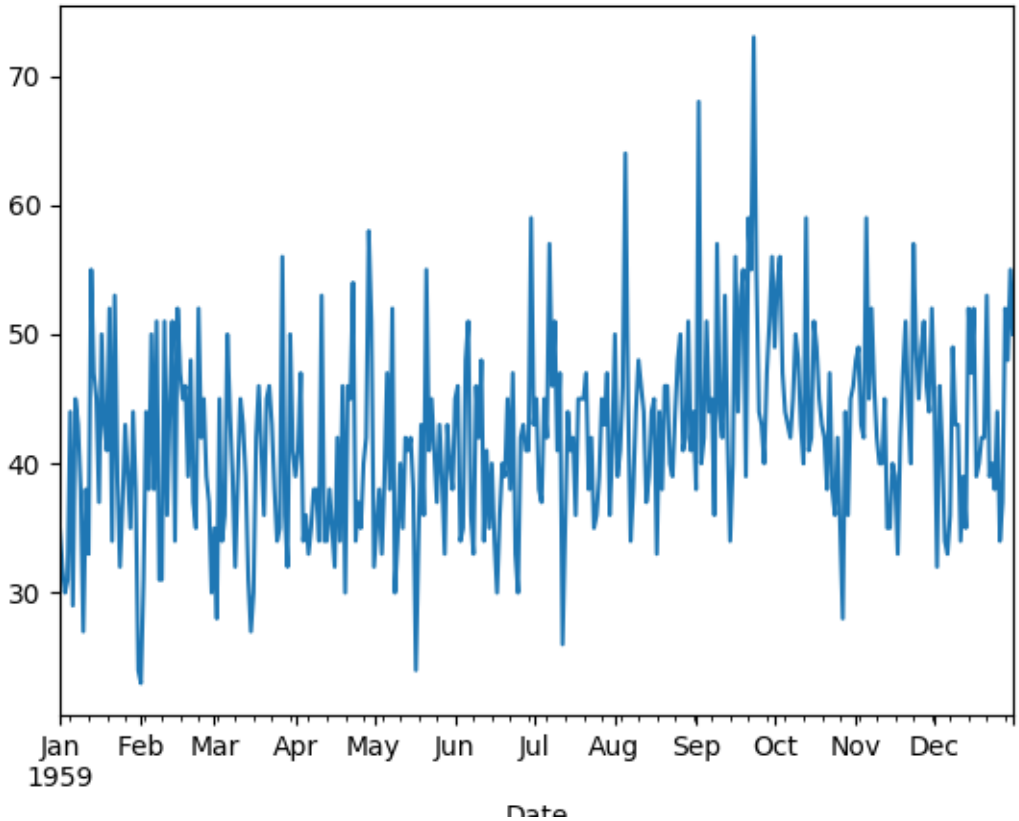

Figure A.4: Line plot of the Daily Female Births dataset.

Learn more about the dataset on DataMarket: https://goo.gl/6RBLka

A.5 Airline Passengers Dataset

This dataset describes the total number of airline passengers over time. The units are a count of the number of airline passengers in thousands. There are 144 monthly observations from 1949 to 1960. Download the dataset and place it in your current working directory with the filename `airline-passengers.csv`. The code below loads the Airline Passengers dataset, prints the first 5 rows of data and graphs the data as a line plot.

```
# load the airline passengers dataset
from matplotlib import pyplot
from pandas import read_csv
series = read_csv('airline-passengers.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
print(type(series))
print(series.head())
series.plot()
pyplot.show()
```

Listing A.9: Create a line plot of the Airline Passengers dataset.

Below is a sample of the first 5 rows of the loaded dataset.

```
Month
1949-01-01   112
1949-02-01   118
1949-03-01   132
1949-04-01   129
1949-05-01   121
```

Listing A.10: Example output of printing the first 5 rows of the Airline Passengers dataset.

Below is a line plot of the entire dataset.

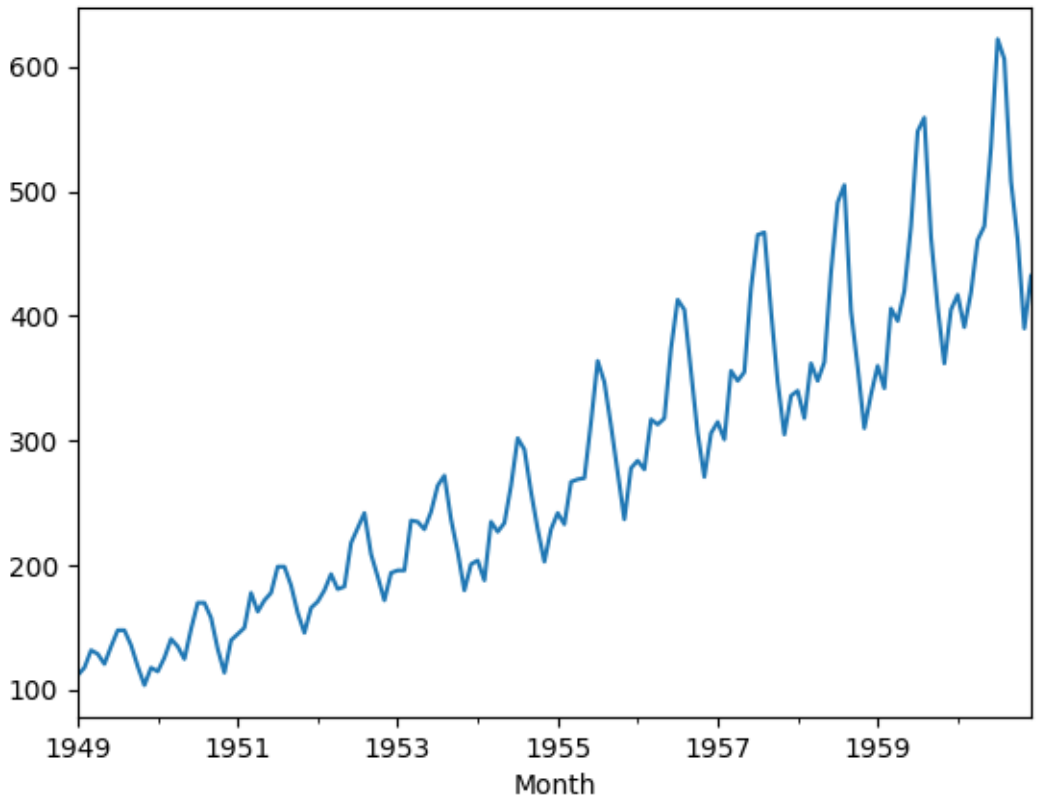

Figure A.5: Line plot of the Airline Passengers dataset.

Learn more about the dataset on DataMarket: https://goo.gl/znGq4F

Appendix B

Workaround for Saving ARIMA Models

The Autoregressive Integrated Moving Average Model, or ARIMA, is a popular linear model for time series analysis and forecasting. The Statsmodels library provides an implementation of ARIMA for use in Python. ARIMA models can be saved to file for later use in making predictions on new data. There is a bug in the current version of the Statsmodels library that prevents saved models from being loaded. In this tutorial, you will discover how to diagnose and work around this issue. Let's get started.

B.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`.

B.2 Python Environment

Confirm you are using the latest version of the Statsmodels library. You can do that by running the script below:

```
# check the version of statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
```

Listing B.1: Check Statsmodels version.

Running the script should produce a result showing Statsmodels 0.6.1 or 0.8.0.

```
statsmodels: 0.8.0
```

Listing B.2: Example output of checking the Statsmodels version.

You can use either Python 2 or 3.

B.3 ARIMA Model Save Bug

We can easily train an ARIMA model on the Daily Female Births dataset. The code snippet below trains an `ARIMA(1,1,1)` on the dataset. The `model.fit()` function returns an `ARIMAResults` object on which we can call `save()` to save the model to file and `load()` to later load it.

```
# save and load an ARIMA model that causes an error
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.arima_model import ARIMAResults
# load data
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
# fit model
model = ARIMA(X, order=(1,1,1))
model_fit = model.fit()
# save model
model_fit.save('model.pkl')
# load model
loaded = ARIMAResults.load('model.pkl')
```

Listing B.3: Save and load an ARIMA model that causes an error.

Running this example will train the model and save it to file without problem. An error will be reported when you try to load the model from file.

```
Traceback (most recent call last):
  File "...", line 16, in <module>
    loaded = ARIMAResults.load('model.pkl')
  File ".../site-packages/statsmodels/base/model.py", line 1529, in load
    return load_pickle(fname)
  File ".../site-packages/statsmodels/iolib/smpickle.py", line 41, in load_pickle
    return cPickle.load(fin)
TypeError: __new__() takes at least 3 arguments (1 given)
```

Listing B.4: Example error message from loading the ARIMA model.

Specifically, note the line:

```
TypeError: __new__() takes at least 3 arguments (1 given)
```

Listing B.5: Specific error from loading the ARIMA model.

Alternately, if you are running a more recent version of Statsmodels, such as 0.8.0 you may get an error message such as:

```
AttributeError: 'ARIMA' object has no attribute 'dates'
```

Listing B.6: Alternate error from loading the ARIMA model.

So far so good, so how do we fix it?

B.4 ARIMA Model Save Bug Workaround

Zae Myung Kim discovered this bug in September 2016 and reported the fault. You can read all about it here:

- BUG: Implemented `__getnewargs__()` method for unpickling.
  https://github.com/statsmodels/statsmodels/pull/3217

The bug occurs because a function required by pickle (the library used to serialize Python objects) has not been defined in Statsmodels. A function `__getnewargs__`[1] must be defined in the ARIMA model prior to saving that defines the arguments needed to construct the object. We can work around this issue. The fix involves two things:

1. Defining an implementation of the `__getnewargs__()` function suitable for the ARIMA object.
2. Adding the new function to ARIMA.

Thankfully, Zae Myung Kim provided an example of the function in his bug report so we can just use that directly:

```
def __getnewargs__(self):
  return ((self.endog),(self.k_lags, self.k_diff, self.k_ma))
```

Listing B.7: Example of defining the missing pickle function.

Python allows us to monkey patch an object, even one from a library like Statsmodels. We can define a new function on an existing object using assignment. We can do this for the `__getnewargs__()` function on the ARIMA object as follows:

```
ARIMA.__getnewargs__ = __getnewargs__
```

Listing B.8: Example of adding the missing function to the ARIMA class.

The complete example of training, saving, and loading an ARIMA model in Python with the monkey patch is listed below:

```
# save and load an ARIMA model with a workaround
from pandas import read_csv
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.arima_model import ARIMAResults

# monkey patch around bug in ARIMA class
def __getnewargs__(self):
  return ((self.endog),(self.k_lags, self.k_diff, self.k_ma))
ARIMA.__getnewargs__ = __getnewargs__

# load data
series = read_csv('daily-total-female-births.csv', header=0, index_col=0, parse_dates=True,
    squeeze=True)
# prepare data
X = series.values
X = X.astype('float32')
```

[1] https://docs.python.org/2/library/pickle.html#object.__getnewargs__

```
# fit model
model = ARIMA(X, order=(1,1,1))
model_fit = model.fit()
# save model
model_fit.save('model.pkl')
# load model
loaded = ARIMAResults.load('model.pkl')
```

Listing B.9: Save and load an ARIMA model with workaround.

Running the example now successfully loads the model without error.

B.5 Summary

In this lesson, you discovered how to work around a bug in the Statsmodels ARIMA implementation that prevented you from saving and loading an ARIMA model to and from file. You discovered how to write a monkey patch to work around the bug and how to demonstrate that it has indeed been fixed.