Machine Learning Mastery with Python

Understand Your Data, Create Accurate Models and Work Projects End-To-End

Jason Brownlee

Jason Brownlee

Machine Learning Mastery With Python

Understand Your Data, Create Accurate Models and Work Projects End-To-End

Machine Learning Mastery With Python

Edition: v1.3

Contents

Preface

I think Python is an amazing platform for machine learning. There are so many algorithms and so much power ready to use. I am often asked the question: *How do you use Python for machine learning?* This book is my definitive answer to that question. It contains my very best knowledge and ideas on how to work through predictive modeling machine learning projects using the Python ecosystem. It is the book that I am also going to use as a refresher at the start of a new project. I'm really proud of this book and I hope that you find it a useful companion on your machine learning journey with Python.

Jason Brownlee
Melbourne, Australia
2016

Part I

Introduction

Chapter 1

Welcome

Welcome to *Machine Learning Mastery With Python*. This book is your guide to applied machine learning with Python. You will discover the step-by-step process that you can use to get started and become good at machine learning for predictive modeling with the Python ecosystem.

1.1 Learn Python Machine Learning The Wrong Way

Here is what you should NOT do when you start studying machine learning in Python.

1. Get really good at Python programming and Python syntax.
2. Deeply study the underlying theory and parameters for machine learning algorithms in scikit-learn.
3. Avoid or lightly touch on all of the other tasks needed to complete a real project.

I think that this approach can work for some people, but it is a really slow and a roundabout way of getting to your goal. It teaches you that you need to spend all your time learning how to use individual machine learning algorithms. It also does not teach you the process of building predictive machine learning models in Python that you can actually use to make predictions. Sadly, this is the approach used to teach machine learning that I see in almost all books and online courses on the topic.

1.2 Machine Learning in Python

This book focuses on a specific sub-field of machine learning called predictive modeling. This is the field of machine learning that is the most useful in industry and the type of machine learning that the scikit-learn library in Python excels at facilitating. Unlike statistics, where models are used to *understand* data, predictive modeling is laser focused on developing models that make the *most accurate predictions* at the expense of explaining why predictions are made. Unlike the broader field of machine learning that could feasibly be used with data in any format, predictive modeling is primarily focused on tabular data (e.g. tables of numbers like in a spreadsheet).

This book was written around three themes designed to get you started and using Python for applied machine learning effectively and quickly. These three parts are as follows:

Lessons : Learn how the sub-tasks of a machine learning project map onto Python and the best practice way of working through each task.

Projects : Tie together all of the knowledge from the lessons by working through case study predictive modeling problems.

Recipes : Apply machine learning with a catalog of standalone recipes in Python that you can copy-and-paste as a starting point for new projects.

1.2.1 Lessons

You need to know how to complete the specific subtasks of a machine learning project using the Python ecosystem. Once you know how to complete a discrete task using the platform and get a result reliably, you can do it again and again on project after project. Let's start with an overview of the common tasks in a machine learning project. A predictive modeling machine learning project can be broken down into 6 top-level tasks:

1. **Define Problem**: Investigate and characterize the problem in order to better understand the goals of the project.
2. **Analyze Data**: Use descriptive statistics and visualization to better understand the data you have available.
3. **Prepare Data**: Use data transforms in order to better expose the structure of the prediction problem to modeling algorithms.
4. **Evaluate Algorithms**: Design a test harness to evaluate a number of standard algorithms on the data and select the top few to investigate further.
5. **Improve Results**: Use algorithm tuning and ensemble methods to get the most out of well-performing algorithms on your data.
6. **Present Results**: Finalize the model, make predictions and present results.

A blessing and a curse with Python is that there are so many techniques and so many ways to do the same thing with the platform. In part II of this book you will discover one easy or best practice way to complete each subtask of a general machine learning project. Below is a summary of the Lessons from Part II and the sub-tasks that you will learn about.

- Lesson 1: Python Ecosystem for Machine Learning.
- Lesson 2: Python and SciPy Crash Course.
- Lesson 3: Load Datasets from CSV.
- Lesson 4: Understand Data With Descriptive Statistics. (**Analyze Data**)
- Lesson 5: Understand Data With Visualization. (**Analyze Data**)
- Lesson 6: Pre-Process Data. (**Prepare Data**)

- Lesson 7: Feature Selection. (**Prepare Data**)
- Lesson 8: Resampling Methods. (**Evaluate Algorithms**)
- Lesson 9: Algorithm Evaluation Metrics. (**Evaluate Algorithms**)
- Lesson 10: Spot-Check Classification Algorithms. (**Evaluate Algorithms**)
- Lesson 11: Spot-Check Regression Algorithms. (**Evaluate Algorithms**)
- Lesson 12: Model Selection. (**Evaluate Algorithms**)
- Lesson 13: Pipelines. (**Evaluate Algorithms**)
- Lesson 14: Ensemble Methods. (**Improve Results**)
- Lesson 15: Algorithm Parameter Tuning. (**Improve Results**)
- Lesson 16: Model Finalization. (**Present Results**)

These lessons are intended to be read from beginning to end in order, showing you exactly how to complete each task in a predictive modeling machine learning project. Of course, you can dip into specific lessons again later to refresh yourself. Lessons are structured to demonstrate key API classes and functions, showing you how to use specific techniques for a common machine learning task. Each lesson was designed to be completed in under 30 minutes (depending on your level of skill and enthusiasm). It is possible to work through the entire book in one weekend. It also works if you want to dip into specific sections and use the book as a reference.

1.2.2 Projects

Recipes for common predictive modeling tasks are critically important, but they are also just the starting point. This is where most books and courses stop.

You need to piece the recipes together into end-to-end projects. This will show you how to actually deliver a model or make predictions on new data using Python. This book uses small well-understood machine learning datasets from the UCI Machine learning repository[1] in both the lessons and in the example projects. These datasets are available for free as CSV downloads. These datasets are excellent for practicing applied machine learning because:

- **They are small**, meaning they fit into memory and algorithms can model them in reasonable time.
- **They are well behaved**, meaning you often don't need to do a lot of feature engineering to get a good result.
- **They are benchmarks**, meaning that many people have used them before and you can get ideas of good algorithms to try and accuracy levels you should expect.

In Part III you will work through three projects:

[1] http://archive.ics.uci.edu/ml

Hello World Project (Iris flowers dataset) : This is a quick pass through the project steps without much tuning or optimizing on a dataset that is widely used as the *hello world* of machine learning.

Regression (Boston House Price dataset) : Work through each step of the project process with a regression problem.

Binary Classification (Sonar dataset) : Work through each step of the project process using all of the methods on a binary classification problem.

These projects unify all of the lessons from Part II. They also give you insight into the process for working through predictive modeling machine learning problems which is invaluable when you are trying to get a feeling for how to do this in practice. Also included in this section is a template for working through predictive modeling machine learning problems which you can use as a starting point for current and future projects. I find this useful myself to set the direction and setup important tasks (which are easy to forget) on new projects.

1.2.3 Recipes

Recipes are small standalone examples in Python that show you how to do one specific thing and get a result. For example, you could have a recipe that demonstrates how to use the Random Forest algorithm for classification. You could have another for normalizing the attributes of a dataset.

Recipes make the difference between a beginner who is having trouble and a fast learner capable of making accurate predictions quickly on any new project. A catalog of recipes provides a repertoire of skills that you can draw from when starting a new project. More formally, recipes are defined as follows:

- Recipes are code snippets not tutorials.
- Recipes provide just enough code to work.
- Recipes are demonstrative not exhaustive.
- Recipes run as-is and produce a result.
- Recipes assume that required libraries are installed.
- Recipes use built-in datasets or datasets provided in specific libraries.

You are starting your journey into machine learning with Python with my personal catalog of machine learning recipes provided with this book. All of the code from the lessons in Part II are available in your Python recipe catalog. There are also recipes for techniques not covered in this book, including usage of a large number of algorithms and many additional case studies. Recipes are divided into directories according to the common tasks of a machine learning project as listed above. The list below provides a summary of the recipes available.

- **Analyze Data**: Recipes to load, summarize and visualize data, including visualizations using univariate plots and multivariate plots.

- **Prepare Data**: Recipes for data preparation including data cleaning, feature selection and data transforms.
- **Algorithms**: Recipes for using a large number of machine learning algorithms, including linear, nonlinear, and decision trees for classification and regression.
- **Evaluate Algorithms**: Recipes for re-sampling methods, algorithm evaluation metrics and model selection.
- **Improve Results**: Recipes for algorithm tuning and ensemble methods.
- **Finalize Model**: Recipes to save and load models to disk.
- **Other**: Recipes for getting started with Python syntax and data structures used in SciPy.
- **Case Studies**: Case studies for binary classification, multiclass classification and regression problems.

This is an valuable resource that you can use to jump-start your current and future machine learning projects. You can also build upon this recipe catalog as you discover new techniques.

1.2.4 Your Outcomes From Reading This Book

This book will lead you from being a developer who is interested in machine learning with Python to a developer who has the resources and capability to work through a new dataset end-to-end using Python and develop accurate predictive models. Specifically, you will know:

- How to work through a small to medium sized dataset end-to-end.
- How to deliver a model that can make accurate predictions on new unseen data.
- How to complete all subtasks of a predictive modeling problem with Python.
- How to learn new and different techniques in Python and SciPy.
- How to get help with Python machine learning.

From here you can start to dive into the specifics of the functions, techniques and algorithms used with the goal of learning how to use them better in order to deliver more accurate predictive models, more reliably in less time.

1.3 What This Book is Not

This book was written for professional developers who want to know how to build reliable and accurate machine learning models in Python.

- **This is not a machine learning textbook**. We will not be getting into the basic theory of machine learning (e.g. induction, bias-variance trade-off, etc.). You are expected to have some familiarity with machine learning basics, or be able to pick them up yourself.

- **This is not an algorithm book**. We will not be working through the details of how specific machine learning algorithms work (e.g. Random Forests). You are expected to have some basic knowledge of machine learning algorithms or how to pick up this knowledge yourself.
- **This is not a Python programming book**. We will not be spending a lot of time on Python syntax and programming (e.g. basic programming tasks in Python). You are expected to be a developer who can pick up a new C-like language relatively quickly.

You can still get a lot out of this book if you are weak in one or two of these areas, but you may struggle picking up the language or require some more explanation of the techniques. If this is the case, see the *Getting More Help* chapter at the end of the book and seek out a good companion reference text.

1.4 Summary

I hope you are as excited as me to get started. In this introduction chapter you learned that this book is unconventional. Unlike other books and courses that focus heavily on machine learning algorithms in Python and focus on little else, this book will walk you through each step of a predictive modeling machine learning project.

- Part II of this book provides standalone lessons including a mixture of recipes and tutorials to build up your basic working skills and confidence in Python.
- Part III of this book will introduce a machine learning project template that you can use as a starting point on your own projects and walks you through three end-to-end projects.
- The recipes companion to this book provides a catalog of more than 90 machine learning recipes in Python. You can browse this invaluable resource, find useful recipes and copy-and-paste them into your current and future machine learning projects.
- Part IV will finish out the book. It will look back at how far you have come in developing your new found skills in applied machine learning with Python. You will also discover resources that you can use to get help if and when you have any questions about Python or the ecosystem.

1.4.1 Next Step

Next you will start Part II and your first lesson. You will take a closer look at the Python ecosystem for machine learning. You will discover what Python and SciPy are, why it is so powerful as a platform for machine learning and the different ways you should and should not use the platform.

Part II

Lessons

Chapter 2

Python Ecosystem for Machine Learning

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

1. Python and it's rising use for machine learning.
2. SciPy and the functionality it provides with NumPy, Matplotlib and Pandas.
3. scikit-learn that provides all of the machine learning algorithms.
4. How to setup your Python ecosystem for machine learning and what versions to use

Let's get started.

2.1 Python

Python is a general purpose interpreted programming language. It is easy to learn and use primarily because the language focuses on readability. The philosophy of Python is captured in the Zen of Python which includes phrases like:

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
```

Listing 2.1: Sample of the Zen of Python.

It is a popular language in general, consistently appearing in the top 10 programming languages in surveys on StackOverflow[1]. It's a dynamic language and very suited to interactive

[1] http://stackoverflow.com/research/developer-survey-2015

development and quick prototyping with the power to support the development of large applications. It is also widely used for machine learning and data science because of the excellent library support and because it is a general purpose programming language (unlike R or Matlab). For example, see the results of the Kaggle platform survey results in 2011[2] and the KDD Nuggets 2015 tool survey results[3].

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 for your production systems. Greatly simplifying the transition from development to production.

2.2 SciPy

SciPy is an ecosystem of Python libraries for mathematics, science and engineering. It is an add-on to Python that you will need for machine learning. The SciPy ecosystem is comprised of the following core modules relevant to machine learning:

- **NumPy**: A foundation for SciPy that allows you to efficiently work with data in arrays.
- **Matplotlib**: Allows you to create 2D charts and plots from data.
- **Pandas**: Tools and data structures to organize and analyze your data.

To be effective at machine learning in Python you must install and become familiar with SciPy. Specifically:

- You will prepare your data as NumPy arrays for modeling in machine learning algorithms.
- You will use Matplotlib (and wrappers of Matplotlib in other frameworks) to create plots and charts of your data.
- You will use Pandas to load explore and better understand your data.

2.3 scikit-learn

The scikit-learn library is how you can develop and practice machine learning in Python. It is built upon and requires the SciPy ecosystem. The name *scikit* suggests that it is a SciPy plug-in or toolkit. 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.

Like Python and SciPy, scikit-learn is open source and is usable commercially under the BSD license. This means that you can learn about machine learning, develop models and put them into operations all with the same ecosystem and code. A powerful reason to use scikit-learn.

[2] http://blog.kaggle.com/2011/11/27/kagglers-favorite-tools/

[3] http://www.kdnuggets.com/polls/2015/analytics-data-mining-data-science-software-used.html

2.4 Python Ecosystem Installation

There are multiple ways to install the Python ecosystem for machine learning. In this section we cover how to install the Python ecosystem for machine learning.

2.4.1 How To Install Python

The first step is to install Python. I prefer to use and recommend Python 2.7. The instructions for installing Python will be specific to your platform. For instructions see *Downloading Python*[4] in the *Python Beginners Guide*. Once installed you can confirm the installation was successful. Open a command line and type:

```
python --version
```

Listing 2.2: Print the version of Python installed.

You should see a response like the following:

```
Python 2.7.11
```

Listing 2.3: Example Python version.

The examples in this book assume that you are using this version of Python 2 or newer. The examples in this book have not been tested with Python 3.

2.4.2 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. yum 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 page *Installing the SciPy Stack*[5]. When installing SciPy, ensure that you install the following packages as a minimum:

- scipy
- numpy
- matplotlib
- pandas

Once installed, you can confirm that the installation was successful. Open the Python interactive environment by typing `python` at the command line, then type in and run the following Python code to print the versions of the installed libraries.

```
# scipy
import scipy
print('scipy: {}'.format(scipy.__version__))
# numpy
import numpy
print('numpy: {}'.format(numpy.__version__))
```

[4] https://wiki.python.org/moin/BeginnersGuide/Download
[5] http://scipy.org/install.html

```
# matplotlib
import matplotlib
print('matplotlib: {}'.format(matplotlib.__version__))
# pandas
import pandas
print('pandas: {}'.format(pandas.__version__))
```

Listing 2.4: Print the versions of the SciPy stack.

On my workstation at the time of writing I see the following output.

```
scipy: 0.17.0
numpy: 1.10.4
matplotlib: 1.5.1
pandas: 0.17.1
```

Listing 2.5: Example versions of the SciPy stack.

The examples in this book assume you have these version of the SciPy libraries or newer. If you have an error, you may need to consult the documentation for your platform.

2.4.3 How To Install scikit-learn

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[6], but they are limited to using the Python `pip` and `conda` package managers. Like SciPy, you can confirm that scikit-learn was installed successfully. Start your Python interactive environment and type and run the following code.

```
# scikit-learn
import sklearn
print('sklearn: {}'.format(sklearn.__version__))
```

Listing 2.6: Print the versions of scikit-learn.

It will print the version of the scikit-learn library installed. On my workstation at the time of writing I see the following output:

```
sklearn: 0.17.1
```

Listing 2.7: Example versions of scikit-learn.

The examples in this book assume you have this version of scikit-learn or newer.

2.4.4 How To Install The Ecosystem: An Easier Way

If you are not confident at installing software on your machine, there is an easier option for you. There is a distribution called Anaconda that you can download and install for free[7]. 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 machine learning with the Python Environment.

[6] http://scikit-learn.org/stable/install.html
[7] https://www.continuum.io/downloads

2.5 Summary

In this chapter you discovered the Python ecosystem for machine learning. You learned about:

- Python and it's rising use for machine learning.
- SciPy and the functionality it provides with NumPy, Matplotlib and Pandas.
- scikit-learn that provides all of the machine learning algorithms.

You also learned how to install the Python ecosystem for machine learning on your workstation.

2.5.1 Next

In the next lesson you will get a crash course in the Python and SciPy ecosystem, designed specifically to get a developer like you up to speed with ecosystem very fast.

Chapter 3

Crash Course in Python and SciPy

You do not need to be a Python developer to get started using the Python ecosystem for machine learning. As a developer who already knows how to program in one or more programming languages, you are able to pick up a new language like Python very quickly. You just need to know a few properties of the language to transfer what you already know to the new language. After completing this lesson you will know:

1. How to navigate Python language syntax.
2. Enough NumPy, Matplotlib and Pandas to read and write machine learning Python scripts.
3. A foundation from which to build a deeper understanding of machine learning tasks in Python.

If you already know a little Python, this chapter will be a friendly reminder for you. Let's get started.

3.1 Python Crash Course

When getting started in Python you need to know a few key details about the language syntax to be able to read and understand Python code. This includes:

- Assignment.
- Flow Control.
- Data Structures.
- Functions.

We will cover each of these topics in turn with small standalone examples that you can type and run. Remember, whitespace has meaning in Python.

3.1.1 Assignment

As a programmer, assignment and types should not be surprising to you.

Strings

```
# Strings
data = 'hello world'
print(data[0])
print(len(data))
print(data)
```

Listing 3.1: Example of working with strings.

Notice how you can access characters in the string using array syntax. Running the example prints:

```
h
11
hello world
```

Listing 3.2: Output of example working with strings.

Numbers

```
# Numbers
value = 123.1
print(value)
value = 10
print(value)
```

Listing 3.3: Example of working with numbers.

Running the example prints:

```
123.1
10
```

Listing 3.4: Output of example working with numbers.

Boolean

```
# Boolean
a = True
b = False
print(a, b)
```

Listing 3.5: Example of working with booleans.

Running the example prints:

```
(True, False)
```

Listing 3.6: Output of example working with booleans.

Multiple Assignment

```
# Multiple Assignment
a, b, c = 1, 2, 3
print(a, b, c)
```

Listing 3.7: Example of working with multiple assignment.

This can also be very handy for unpacking data in simple data structures. Running the example prints:

```
(1, 2, 3)
```

Listing 3.8: Output of example working with multiple assignment.

No Value

```
# No value
a = None
print(a)
```

Listing 3.9: Example of working with no value.

Running the example prints:

```
None
```

Listing 3.10: Output of example working with no value.

3.1.2 Flow Control

There are three main types of flow control that you need to learn: If-Then-Else conditions, For-Loops and While-Loops.

If-Then-Else Conditional

```
value = 99
if value == 99:
  print 'That is fast'
elif value > 200:
  print 'That is too fast'
else:
  print 'That is safe'
```

Listing 3.11: Example of working with an If-Then-Else conditional.

Notice the colon (:) at the end of the condition and the meaningful tab intend for the code block under the condition. Running the example prints:

```
If-Then-Else conditional
```

Listing 3.12: Output of example working with an If-Then-Else conditional.

For-Loop

```
# For-Loop
for i in range(10):
  print i
```

Listing 3.13: Example of working with a For-Loop.

Running the example prints:

```
0
1
2
3
4
5
6
7
8
9
```

Listing 3.14: Output of example working with a For-Loop.

While-Loop

```
# While-Loop
i = 0
while i < 10:
  print i
  i += 1
```

Listing 3.15: Example of working with a While-Loop.

Running the example prints:

```
0
1
2
3
4
5
6
7
8
9
```

Listing 3.16: Output of example working with a While-Loop.

3.1.3 Data Structures

There are three data structures in Python that you will find the most used and useful. They are tuples, lists and dictionaries.

Tuple

Tuples are read-only collections of items.

```
a = (1, 2, 3)
print a
```

Listing 3.17: Example of working with a Tuple.

Running the example prints:

```
(1, 2, 3)
```

Listing 3.18: Output of example working with a Tuple.

List

Lists use the square bracket notation and can be index using array notation.

```
mylist = [1, 2, 3]
print("Zeroth Value: %d") % mylist[0]
mylist.append(4)
print("List Length: %d") % len(mylist)
for value in mylist:
  print value
```

Listing 3.19: Example of working with a List.

Notice that we are using some simple `printf`-like functionality to combine strings and variables when printing. Running the example prints:

```
Zeroth Value: 1
List Length: 4
1
2
3
4
```

Listing 3.20: Output of example working with a List.

Dictionary

Dictionaries are mappings of names to values, like key-value pairs. Note the use of the curly bracket and colon notations when defining the dictionary.

```
mydict = {'a': 1, 'b': 2, 'c': 3}
print("A value: %d") % mydict['a']
mydict['a'] = 11
print("A value: %d") % mydict['a']
print("Keys: %s") % mydict.keys()
print("Values: %s") % mydict.values()
for key in mydict.keys():
  print mydict[key]
```

Listing 3.21: Example of working with a Dictionary.

Running the example prints:

```
A value: 1
A value: 11
Keys: ['a', 'c', 'b']
Values: [11, 3, 2]
11
3
2
```

Listing 3.22: Output of example working with a Dictionary.

Functions

The biggest gotcha with Python is the whitespace. Ensure that you have an empty new line after indented code. The example below defines a new function to calculate the sum of two values and calls the function with two arguments.

```
# Sum function
def mysum(x, y):
  return x + y

# Test sum function
result = mysum(1, 3)
print(result)
```

Listing 3.23: Example of working with a custom function.

Running the example prints:

```
4
```

Listing 3.24: Output of example working with a custom function.

3.2 NumPy Crash Course

NumPy provides the foundation data structures and operations for SciPy. These are arrays (`ndarrays`) that are efficient to define and manipulate.

3.2.1 Create Array

```
# define an array
import numpy
mylist = [1, 2, 3]
myarray = numpy.array(mylist)
print(myarray)
print(myarray.shape)
```

Listing 3.25: Example of creating a NumPy array.

Notice how we easily converted a Python list to a NumPy array. Running the example prints:

```
[1 2 3]
(3,)
```

Listing 3.26: Output of example creating a NumPy array.

3.2.2 Access Data

Array notation and ranges can be used to efficiently access data in a NumPy array.

```
# access values
import numpy
mylist = [[1, 2, 3], [3, 4, 5]]
myarray = numpy.array(mylist)
print(myarray)
print(myarray.shape)
print("First row: %s") % myarray[0]
print("Last row: %s") % myarray[-1]
print("Specific row and col: %s") % myarray[0, 2]
print("Whole col: %s") % myarray[:, 2]
```

Listing 3.27: Example of working with a NumPy array.

Running the example prints:

```
[[1 2 3]
 [3 4 5]]
(2, 3)
First row: [1 2 3]
Last row: [3 4 5]
Specific row and col: 3
Whole col: [3 5]
```

Listing 3.28: Output of example working with a NumPy array.

3.2.3 Arithmetic

NumPy arrays can be used directly in arithmetic.

```
# arithmetic
import numpy
myarray1 = numpy.array([2, 2, 2])
myarray2 = numpy.array([3, 3, 3])
print("Addition: %s") % (myarray1 + myarray2)
print("Multiplication: %s") % (myarray1 * myarray2)
```

Listing 3.29: Example of doing arithmetic with NumPy arrays.

Running the example prints:

```
Addition: [5 5 5]
Multiplication: [6 6 6]
```

Listing 3.30: Output of example of doing arithmetic with NumPy arrays.

There is a lot more to NumPy arrays but these examples give you a flavor of the efficiencies they provide when working with lots of numerical data. See Chapter 24 for resources to learn more about the NumPy API.

3.3 Matplotlib Crash Course

Matplotlib can be used for creating plots and charts. The library is generally used as follows:

- Call a plotting function with some data (e.g. `.plot()`).
- Call many functions to setup the properties of the plot (e.g. labels and colors).
- Make the plot visible (e.g. `.show()`).

3.3.1 Line Plot

The example below creates a simple line plot from one dimensional data.

```
# basic line plot
import matplotlib.pyplot as plt
import numpy
myarray = numpy.array([1, 2, 3])
plt.plot(myarray)
plt.xlabel('some x axis')
plt.ylabel('some y axis')
plt.show()
```

Listing 3.31: Example of creating a line plot with Matplotlib.

Running the example produces:

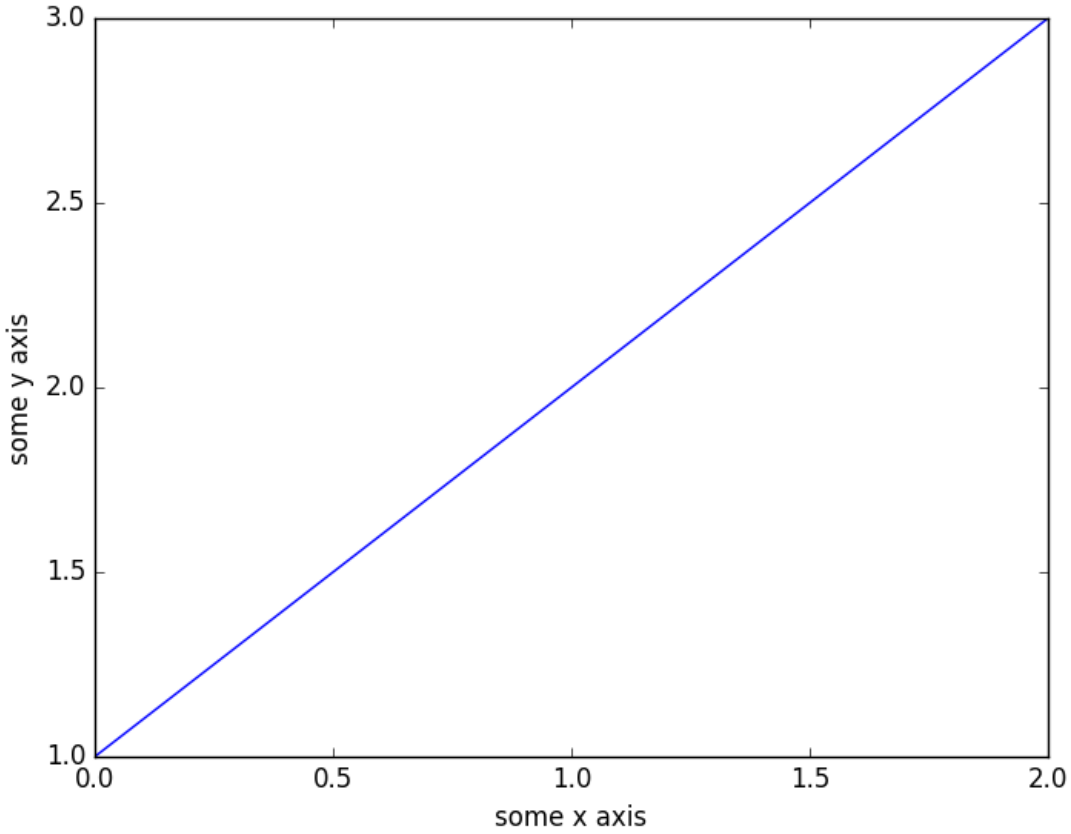

Figure 3.1: Line Plot with Matplotlib

3.3.2 Scatter Plot

Below is a simple example of creating a scatter plot from two dimensional data.

```
# basic scatter plot
import matplotlib.pyplot as plt
import numpy
x = numpy.array([1, 2, 3])
y = numpy.array([2, 4, 6])
plt.scatter(x,y)
plt.xlabel('some x axis')
plt.ylabel('some y axis')
plt.show()
```

Listing 3.32: Example of creating a line plot with Matplotlib.

Running the example produces:

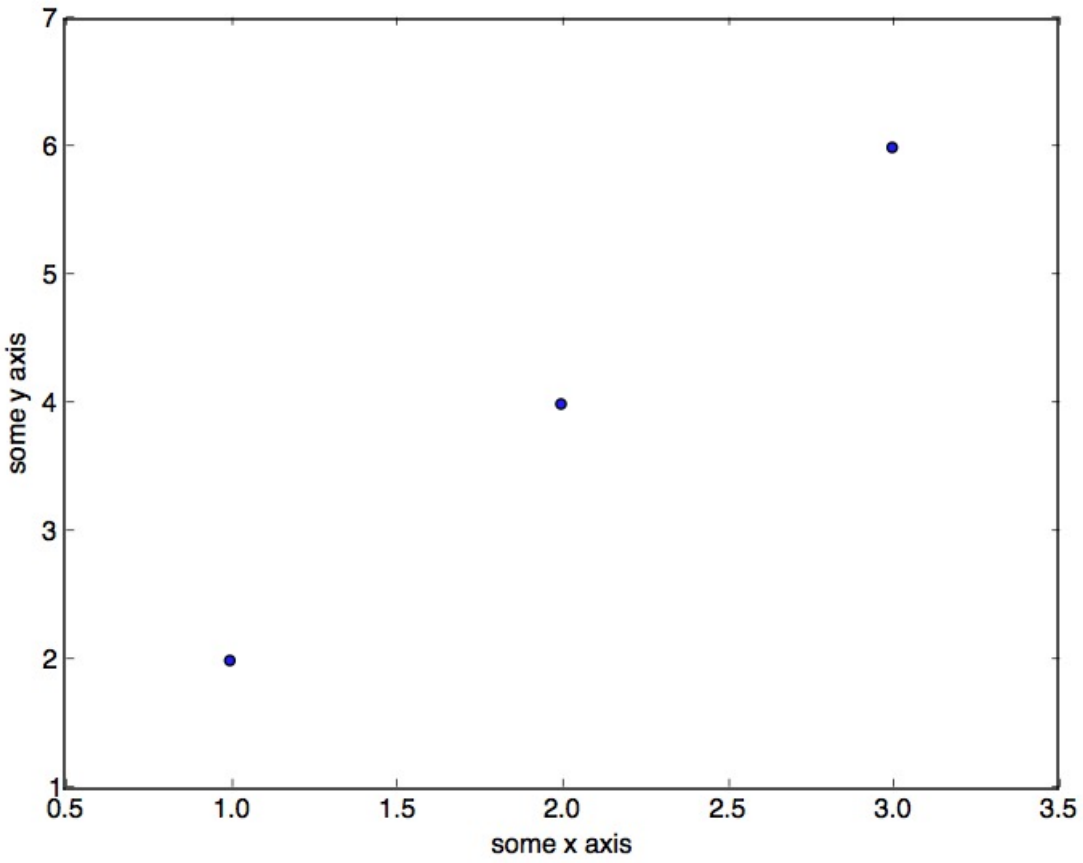

Figure 3.2: Scatter Plot with Matplotlib

There are many more plot types and many more properties that can be set on a plot to configure it. See Chapter 24 for resources to learn more about the Matplotlib API.

3.4 Pandas Crash Course

Pandas provides data structures and functionality to quickly manipulate and analyze data. The key to understanding Pandas for machine learning is understanding the Series and DataFrame data structures.

3.4.1 Series

A series is a one dimensional array where the rows and columns can be labeled.

```
# series
import numpy
import pandas
myarray = numpy.array([1, 2, 3])
rownames = ['a', 'b', 'c']
myseries = pandas.Series(myarray, index=rownames)
```

```
print(myseries)
```

Listing 3.33: Example of creating a Pandas Series.

Running the example prints:

```
a    1
b    2
c    3
```

Listing 3.34: Output of example of creating a Pandas Series.

You can access the data in a series like a NumPy array and like a dictionary, for example:

```
print(myseries[0])
print(myseries['a'])
```

Listing 3.35: Example of accessing data in a Pandas Series.

Running the example prints:

```
1
1
```

Listing 3.36: Output of example of accessing data in a Pandas Series.

3.4.2 DataFrame

A data frame is a multi-dimensional array where the rows and the columns can be labeled.

```
# dataframe
import numpy
import pandas
myarray = numpy.array([[1, 2, 3], [4, 5, 6]])
rownames = ['a', 'b']
colnames = ['one', 'two', 'three']
mydataframe = pandas.DataFrame(myarray, index=rownames, columns=colnames)
print(mydataframe)
```

Listing 3.37: Example of creating a Pandas DataFrame.

Running the example prints:

```
   one  two  three
a    1    2      3
b    4    5      6
```

Listing 3.38: Output of example of creating a Pandas DataFrame.

Data can be index using column names.

```
print("method 1:")
print("one column: %s") % mydataframe['one']
print("method 2:")
print("one column: %s") % mydataframe.one
```

Listing 3.39: Example of accessing data in a Pandas DataFrame.

Running the example prints:

```
method 1:
a    1
b    4
method 2:
a    1
b    4
```

Listing 3.40: Output of example of accessing data in a Pandas DataFrame.

Pandas is a very powerful tool for slicing and dicing you data. See Chapter 24 for resources to learn more about the Pandas API.

3.5 Summary

You have covered a lot of ground in this lesson. You discovered basic syntax and usage of Python and three key Python libraries used for machine learning:

- NumPy.
- Matplotlib.
- Pandas.

3.5.1 Next

You now know enough syntax and usage information to read and understand Python code for machine learning and to start creating your own scripts. In the next lesson you will discover how you can very quickly and easily load standard machine learning datasets in Python.

Chapter 4

How To Load Machine Learning Data

You must be able to load your data before you can start your machine learning project. The most common format for machine learning data is CSV files. There are a number of ways to load a CSV file in Python. In this lesson you will learn three ways that you can use to load your CSV data in Python:

1. Load CSV Files with the Python Standard Library.
2. Load CSV Files with NumPy.
3. Load CSV Files with Pandas.

Let's get started.

4.1 Considerations When Loading CSV Data

There are a number of considerations when loading your machine learning data from CSV files. For reference, you can learn a lot about the expectations for CSV files by reviewing the CSV request for comment titled *Common Format and MIME Type for Comma-Separated Values (CSV) Files*[1].

4.1.1 File Header

Does your data have a file header? If so this can help in automatically assigning names to each column of data. If not, you may need to name your attributes manually. Either way, you should explicitly specify whether or not your CSV file had a file header when loading your data.

4.1.2 Comments

Does your data have comments? Comments in a CSV file are indicated by a hash (#) at the start of a line. If you have comments in your file, depending on the method used to load your data, you may need to indicate whether or not to expect comments and the character to expect to signify a comment line.

[1] https://tools.ietf.org/html/rfc4180

4.1.3 Delimiter

The standard delimiter that separates values in fields is the comma (,) character. Your file could use a different delimiter like tab or white space in which case you must specify it explicitly.

4.1.4 Quotes

Sometimes field values can have spaces. In these CSV files the values are often quoted. The default quote character is the double quotation marks character. Other characters can be used, and you must specify the quote character used in your file.

4.2 Pima Indians Dataset

The Pima Indians dataset is used to demonstrate data loading in this lesson. It will also be used in many of the lessons to come. This dataset describes the medical records for Pima Indians and whether or not each patient will have an onset of diabetes within five years. As such it is a classification problem. It is a good dataset for demonstration because all of the input attributes are numeric and the output variable to be predicted is binary (0 or 1). The data is freely available from the UCI Machine Learning Repository[2].

4.3 Load CSV Files with the Python Standard Library

The Python API provides the module `CSV` and the function `reader()` that can be used to load CSV files. Once loaded, you can convert the CSV data to a NumPy array and use it for machine learning. For example, you can download[3] the Pima Indians dataset into your local directory with the filename `pima-indians-diabetes.data.csv`. All fields in this dataset are numeric and there is no header line.

```
# Load CSV (using python)
import csv
import numpy
filename = 'pima-indians-diabetes.data.csv'
raw_data = open(filename, 'rb')
reader = csv.reader(raw_data, delimiter=',', quoting=csv.QUOTE_NONE)
x = list(reader)
data = numpy.array(x).astype('float')
print(data.shape)
```

Listing 4.1: Example of loading a CSV file using the Python standard library.

The example loads an object that can iterate over each row of the data and can easily be converted into a NumPy array. Running the example prints the shape of the array.

```
(768, 9)
```

Listing 4.2: Output of example loading a CSV file using the Python standard library.

[2] https://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes
[3] https://goo.gl/vhm1eU

For more information on the `csv.reader()` function, see *CSV File Reading and Writing in the Python* API documentation[4].

4.4 Load CSV Files with NumPy

You can load your CSV data using NumPy and the `numpy.loadtxt()` function. This function assumes no header row and all data has the same format. The example below assumes that the file `pima-indians-diabetes.data.csv` is in your current working directory.

```
# Load CSV
import numpy
filename = 'pima-indians-diabetes.data.csv'
raw_data = open(filename, 'rb')
data = numpy.loadtxt(raw_data, delimiter=",")
print(data.shape)
```

Listing 4.3: Example of loading a CSV file using NumPy.

Running the example will load the file as a `numpy.ndarray`[5] and print the shape of the data:

```
(768, 9)
```

Listing 4.4: Output of example loading a CSV file using NumPy.

This example can be modified to load the same dataset directly from a URL as follows:

```
# Load CSV from URL using NumPy
import numpy
import urllib
url = "https://goo.gl/vhm1eU"
raw_data = urllib.urlopen(url)
dataset = numpy.loadtxt(raw_data, delimiter=",")
print(dataset.shape)
```

Listing 4.5: Example of loading a CSV URL using NumPy.

Again, running the example produces the same resulting shape of the data.

```
(768, 9)
```

Listing 4.6: Output of example loading a CSV URL using NumPy.

For more information on the `numpy.loadtxt()`[6] function see the API documentation.

4.5 Load CSV Files with Pandas

You can load your CSV data using Pandas and the `pandas.read_csv()` function. This function is very flexible and is perhaps my recommended approach for loading your machine learning data. The function returns a pandas.DataFrame[7] that you can immediately start summarizing and plotting. The example below assumes that the `pima-indians-diabetes.data.csv` file is in the current working directory.

[4] https://docs.python.org/2/library/csv.html
[5] http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.ndarray.html
[6] http://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.loadtxt.html
[7] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html

```
# Load CSV using Pandas
import pandas
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(filename, names=names)
print(data.shape)
```

Listing 4.7: Example of loading a CSV file using Pandas.

Note that in this example we explicitly specify the names of each attribute to the DataFrame. Running the example displays the shape of the data:

```
(768, 9)
```

Listing 4.8: Output of example loading a CSV file using Pandas.

We can also modify this example to load CSV data directly from a URL.

```
# Load CSV using Pandas from URL
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
print(data.shape)
```

Listing 4.9: Example of loading a CSV URL using Pandas.

Again, running the example downloads the CSV file, parses it and displays the shape of the loaded DataFrame.

```
(768, 9)
```

Listing 4.10: Output of example loading a CSV URL using Pandas.

To learn more about the `pandas.read_csv()`[8] function you can refer to the API documentation.

4.6 Summary

In this chapter you discovered how to load your machine learning data in Python. You learned three specific techniques that you can use:

- Load CSV Files with the Python Standard Library.
- Load CSV Files with NumPy.
- Load CSV Files with Pandas.

Generally I recommend that you load your data with Pandas in practice and all subsequent examples in this book will use this method.

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

4.6.1 Next

Now that you know how to load your CSV data using Python it is time to start looking at it. In the next lesson you will discover how to use simple descriptive statistics to better understand your data.

Chapter 5

Understand Your Data With Descriptive Statistics

You must understand your data in order to get the best results. In this chapter you will discover 7 recipes that you can use in Python to better understand your machine learning data. After reading this lesson you will know how to:

1. Take a peek at your raw data.
2. Review the dimensions of your dataset.
3. Review the data types of attributes in your data.
4. Summarize the distribution of instances across classes in your dataset.
5. Summarize your data using descriptive statistics.
6. Understand the relationships in your data using correlations.
7. Review the skew of the distributions of each attribute.

Each recipe is demonstrated by loading the Pima Indians Diabetes classification dataset from the UCI Machine Learning repository. Open your Python interactive environment and try each recipe out in turn. Let's get started.

5.1 Peek at Your Data

There is no substitute for looking at the raw data. Looking at the raw data can reveal insights that you cannot get any other way. It can also plant seeds that may later grow into ideas on how to better pre-process and handle the data for machine learning tasks. You can review the first 20 rows of your data using the `head()` function on the Pandas DataFrame.

```
# View first 20 rows
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
peek = data.head(20)
```

```
print(peek)
```

Listing 5.1: Example of reviewing the first few rows of data.

You can see that the first column lists the row number, which is handy for referencing a specific observation.

```
    preg  plas  pres  skin  test  mass   pedi  age  class
0      6   148    72    35     0  33.6  0.627   50      1
1      1    85    66    29     0  26.6  0.351   31      0
2      8   183    64     0     0  23.3  0.672   32      1
3      1    89    66    23    94  28.1  0.167   21      0
4      0   137    40    35   168  43.1  2.288   33      1
5      5   116    74     0     0  25.6  0.201   30      0
6      3    78    50    32    88  31.0  0.248   26      1
7     10   115     0     0     0  35.3  0.134   29      0
8      2   197    70    45   543  30.5  0.158   53      1
9      8   125    96     0     0   0.0  0.232   54      1
10     4   110    92     0     0  37.6  0.191   30      0
11    10   168    74     0     0  38.0  0.537   34      1
12    10   139    80     0     0  27.1  1.441   57      0
13     1   189    60    23   846  30.1  0.398   59      1
14     5   166    72    19   175  25.8  0.587   51      1
15     7   100     0     0     0  30.0  0.484   32      1
16     0   118    84    47   230  45.8  0.551   31      1
17     7   107    74     0     0  29.6  0.254   31      1
18     1   103    30    38    83  43.3  0.183   33      0
19     1   115    70    30    96  34.6  0.529   32      1
```

Listing 5.2: Output of reviewing the first few rows of data.

5.2 Dimensions of Your Data

You must have a very good handle on how much data you have, both in terms of rows and columns.

- Too many rows and algorithms may take too long to train. Too few and perhaps you do not have enough data to train the algorithms.
- Too many features and some algorithms can be distracted or suffer poor performance due to the curse of dimensionality.

You can review the shape and size of your dataset by printing the `shape` property on the Pandas DataFrame.

```
# Dimensions of your data
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
shape = data.shape
print(shape)
```

Listing 5.3: Example of reviewing the shape of the data.

The results are listed in rows then columns. You can see that the dataset has 768 rows and 9 columns.

```
(768, 9)
```

Listing 5.4: Output of reviewing the shape of the data.

5.3 Data Type For Each Attribute

The type of each attribute is important. Strings may need to be converted to floating point values or integers to represent categorical or ordinal values. You can get an idea of the types of attributes by peeking at the raw data, as above. You can also list the data types used by the DataFrame to characterize each attribute using the `dtypes` property.

```
# Data Types for Each Attribute
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
types = data.dtypes
print(types)
```

Listing 5.5: Example of reviewing the data types of the data.

You can see that most of the attributes are integers and that `mass` and `pedi` are floating point types.

```
preg      int64
plas      int64
pres      int64
skin      int64
test      int64
mass    float64
pedi    float64
age       int64
class     int64
dtype: object
```

Listing 5.6: Output of reviewing the data types of the data.

5.4 Descriptive Statistics

Descriptive statistics can give you great insight into the shape of each attribute. Often you can create more summaries than you have time to review. The `describe()` function on the Pandas DataFrame lists 8 statistical properties of each attribute. They are:

- Count.
- Mean.
- Standard Deviation.

- Minimum Value.
- 25th Percentile.
- 50th Percentile (Median).
- 75th Percentile.
- Maximum Value.

```
# Statistical Summary
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
pandas.set_option('display.width', 100)
pandas.set_option('precision', 3)
description = data.describe()
print(description)
```

Listing 5.7: Example of reviewing a statistical summary of the data.

You can see that you do get a lot of data. You will note some calls to `pandas.set_option()` in the recipe to change the precision of the numbers and the preferred width of the output. This is to make it more readable for this example. When describing your data this way, it is worth taking some time and reviewing observations from the results. This might include the presence of `NA` values for missing data or surprising distributions for attributes.

```
         preg    plas    pres    skin    test    mass    pedi     age   class
count 768.000 768.000 768.000 768.000 768.000 768.000 768.000 768.000 768.000
mean    3.845 120.895  69.105  20.536  79.799  31.993   0.472  33.241   0.349
std     3.370  31.973  19.356  15.952 115.244   7.884   0.331  11.760   0.477
min     0.000   0.000   0.000   0.000   0.000   0.000   0.078  21.000   0.000
25%     1.000  99.000  62.000   0.000   0.000  27.300   0.244  24.000   0.000
50%     3.000 117.000  72.000  23.000  30.500  32.000   0.372  29.000   0.000
75%     6.000 140.250  80.000  32.000 127.250  36.600   0.626  41.000   1.000
max    17.000 199.000 122.000  99.000 846.000  67.100   2.420  81.000   1.000
```

Listing 5.8: Output of reviewing a statistical summary of the data.

5.5 Class Distribution (Classification Only)

On classification problems you need to know how balanced the class values are. Highly imbalanced problems (a lot more observations for one class than another) are common and may need special handling in the data preparation stage of your project. You can quickly get an idea of the distribution of the class attribute in Pandas.

```
# Class Distribution
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
class_counts = data.groupby('class').size()
```

```
print(class_counts)
```

Listing 5.9: Example of reviewing a class breakdown of the data.

You can see that there are nearly double the number of observations with class 0 (no onset of diabetes) than there are with class 1 (onset of diabetes).

```
class
0    500
1    268
```

Listing 5.10: Output of reviewing a class breakdown of the data.

5.6 Correlations Between Attributes

Correlation refers to the relationship between two variables and how they may or may not change together. The most common method for calculating correlation is Pearson's Correlation Coefficient, that assumes a normal distribution of the attributes involved. A correlation of -1 or 1 shows a full negative or positive correlation respectively. Whereas a value of 0 shows no correlation at all. Some machine learning algorithms like linear and logistic regression can suffer poor performance if there are highly correlated attributes in your dataset. As such, it is a good idea to review all of the pairwise correlations of the attributes in your dataset. You can use the `corr()` function on the Pandas DataFrame to calculate a correlation matrix.

```
# Pairwise Pearson correlations
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
pandas.set_option('display.width', 100)
pandas.set_option('precision', 3)
correlations = data.corr(method='pearson')
print(correlations)
```

Listing 5.11: Example of reviewing correlations of attributes in the data.

The matrix lists all attributes across the top and down the side, to give correlation between all pairs of attributes (twice, because the matrix is symmetrical). You can see the diagonal line through the matrix from the top left to bottom right corners of the matrix shows perfect correlation of each attribute with itself.

```
        preg   plas   pres   skin   test   mass   pedi    age  class
preg   1.000  0.129  0.141 -0.082 -0.074  0.018 -0.034  0.544  0.222
plas   0.129  1.000  0.153  0.057  0.331  0.221  0.137  0.264  0.467
pres   0.141  0.153  1.000  0.207  0.089  0.282  0.041  0.240  0.065
skin  -0.082  0.057  0.207  1.000  0.437  0.393  0.184 -0.114  0.075
test  -0.074  0.331  0.089  0.437  1.000  0.198  0.185 -0.042  0.131
mass   0.018  0.221  0.282  0.393  0.198  1.000  0.141  0.036  0.293
pedi  -0.034  0.137  0.041  0.184  0.185  0.141  1.000  0.034  0.174
age    0.544  0.264  0.240 -0.114 -0.042  0.036  0.034  1.000  0.238
class  0.222  0.467  0.065  0.075  0.131  0.293  0.174  0.238  1.000
```

Listing 5.12: Output of reviewing correlations of attributes in the data.

5.7 Skew of Univariate Distributions

Skew refers to a distribution that is assumed Gaussian (normal or bell curve) that is shifted or squashed in one direction or another. Many machine learning algorithms assume a Gaussian distribution. Knowing that an attribute has a skew may allow you to perform data preparation to correct the skew and later improve the accuracy of your models. You can calculate the skew of each attribute using the `skew()` function on the Pandas DataFrame.

```
# Skew for each attribute
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
skew = data.skew()
print(skew)
```

Listing 5.13: Example of reviewing skew of attribute distributions in the data.

The skew result show a positive (right) or negative (left) skew. Values closer to zero show less skew.

```
preg     0.901674
plas     0.173754
pres    -1.843608
skin     0.109372
test     2.272251
mass    -0.428982
pedi     1.919911
age      1.129597
class    0.635017
```

Listing 5.14: Output of reviewing skew of attribute distributions in the data.

5.8 Tips To Remember

This section gives you some tips to remember when reviewing your data using summary statistics.

- **Review the numbers**. Generating the summary statistics is not enough. Take a moment to pause, read and really think about the numbers you are seeing.
- **Ask why**. Review your numbers and ask a lot of questions. How and why are you seeing specific numbers. Think about how the numbers relate to the problem domain in general and specific entities that observations relate to.
- **Write down ideas**. Write down your observations and ideas. Keep a small text file or note pad and jot down all of the ideas for how variables may relate, for what numbers mean, and ideas for techniques to try later. The things you write down now while the data is fresh will be very valuable later when you are trying to think up new things to try.

5.9 Summary

In this chapter you discovered the importance of describing your dataset before you start work on your machine learning project. You discovered 7 different ways to summarize your dataset using Python and Pandas:

- Peek At Your Data.
- Dimensions of Your Data.
- Data Types.
- Class Distribution.
- Data Summary.
- Correlations.
- Skewness.

5.9.1 Next

Another excellent way that you can use to better understand your data is by generating plots and charts. In the next lesson you will discover how you can visualize your data for machine learning in Python.

Chapter 6

Understand Your Data With Visualization

You must understand your data in order to get the best results from machine learning algorithms. The fastest way to learn more about your data is to use data visualization. In this chapter you will discover exactly how you can visualize your machine learning data in Python using Pandas. Recipes in this chapter use the Pima Indians onset of diabetes dataset introduced in Chapter 4. Let's get started.

6.1 Univariate Plots

In this section we will look at three techniques that you can use to understand each attribute of your dataset independently.

- Histograms.
- Density Plots.
- Box and Whisker Plots.

6.1.1 Histograms

A fast way to get an idea of the distribution of each attribute is to look at histograms. Histograms group data into bins and provide you a count of the number of observations in each bin. From the shape of the bins you can quickly get a feeling for whether an attribute is Gaussian, skewed or even has an exponential distribution. It can also help you see possible outliers.

```
# Univariate Histograms
import matplotlib.pyplot as plt
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
data.hist()
plt.show()
```

Listing 6.1: Example of creating histogram plots.

We can see that perhaps the attributes `age`, `pedi` and `test` may have an exponential distribution. We can also see that perhaps the `mass` and `pres` and `plas` attributes may have a Gaussian or nearly Gaussian distribution. This is interesting because many machine learning techniques assume a Gaussian univariate distribution on the input variables.

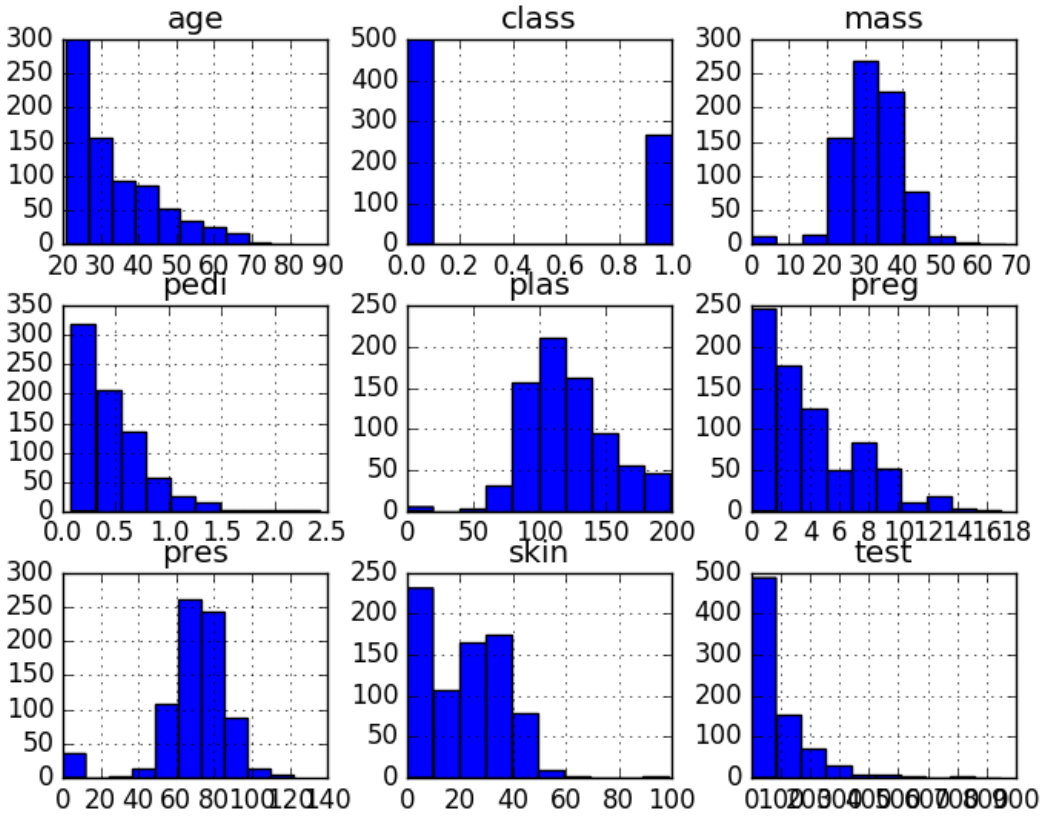

Figure 6.1: Histograms of each attribute

6.1.2 Density Plots

Density plots are another way of getting a quick idea of the distribution of each attribute. The plots look like an abstracted histogram with a smooth curve drawn through the top of each bin, much like your eye tried to do with the histograms.

```
# Univariate Density Plots
import matplotlib.pyplot as plt
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
data.plot(kind='density', subplots=True, layout=(3,3), sharex=False)
plt.show()
```

Listing 6.2: Example of creating density plots.

We can see the distribution for each attribute is clearer than the histograms.

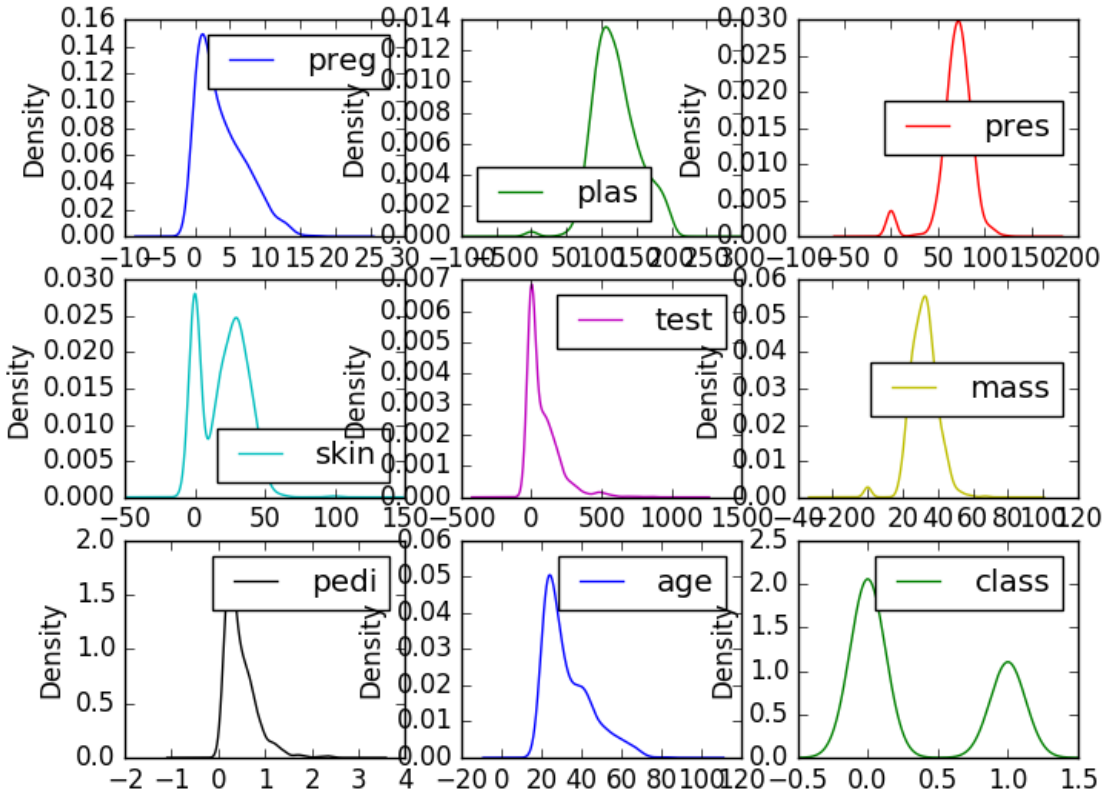

Figure 6.2: Density plots of each attribute

6.1.3 Box and Whisker Plots

Another useful way to review the distribution of each attribute is to use Box and Whisker Plots or boxplots for short. Boxplots summarize the distribution of each attribute, drawing a line for the median (middle value) and a box around the 25th and 75th percentiles (the middle 50% of the data). The whiskers give an idea of the spread of the data and dots outside of the whiskers show candidate outlier values (values that are 1.5 times greater than the size of spread of the middle 50% of the data).

```
# Box and Whisker Plots
import matplotlib.pyplot as plt
import pandas
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
data.plot(kind='box', subplots=True, layout=(3,3), sharex=False, sharey=False)
plt.show()
```

Listing 6.3: Example of creating box and whisker plots.

We can see that the spread of attributes is quite different. Some like `age`, `test` and `skin` appear quite skewed towards smaller values.

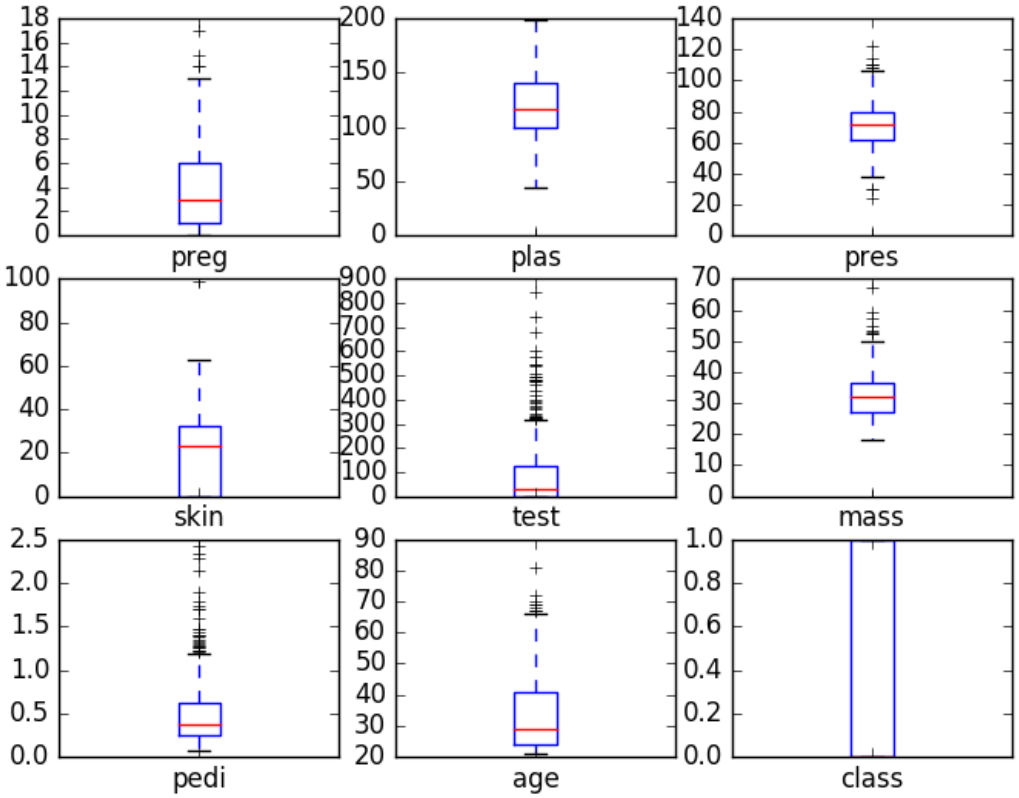

Figure 6.3: Box and whisker plots of each attribute

6.2 Multivariate Plots

This section provides examples of two plots that show the interactions between multiple variables in your dataset.

- Correlation Matrix Plot.
- Scatter Plot Matrix.

6.2.1 Correlation Matrix Plot

Correlation gives an indication of how related the changes are between two variables. If two variables change in the same direction they are positively correlated. If they change in opposite directions together (one goes up, one goes down), then they are negatively correlated. You can calculate the correlation between each pair of attributes. This is called a correlation matrix. You can then plot the correlation matrix and get an idea of which variables have a high correlation

with each other. This is useful to know, because some machine learning algorithms like linear and logistic regression can have poor performance if there are highly correlated input variables in your data.

```
# Correction Matrix Plot
import matplotlib.pyplot as plt
import pandas
import numpy
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
correlations = data.corr()
# plot correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = numpy.arange(0,9,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names)
ax.set_yticklabels(names)
plt.show()
```

Listing 6.4: Example of creating a correlation matrix plot.

We can see that the matrix is symmetrical, i.e. the bottom left of the matrix is the same as the top right. This is useful as we can see two different views on the same data in one plot. We can also see that each variable is perfectly positively correlated with each other (as you would have expected) in the diagonal line from top left to bottom right.

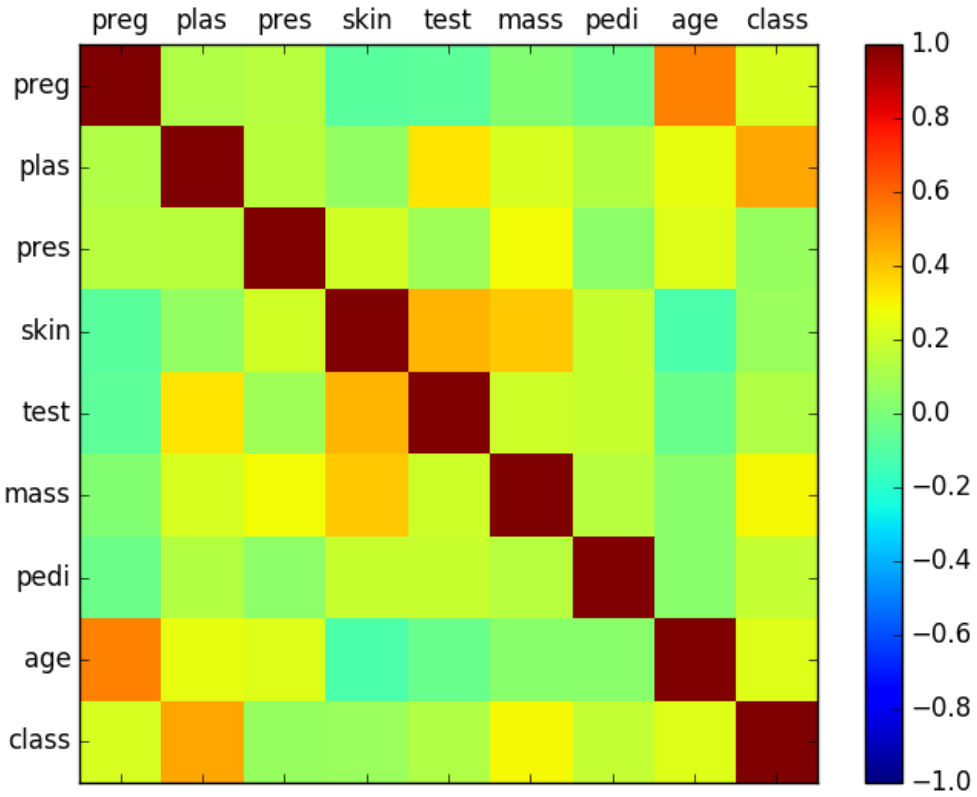

Figure 6.4: Correlation matrix plot.

The example is not generic in that it specifies the names for the attributes along the axes as well as the number of ticks. This recipe cam be made more generic by removing these aspects as follows:

```
# Correction Matrix Plot (generic)
import matplotlib.pyplot as plt
import pandas
import numpy
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
correlations = data.corr()
# plot correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
plt.show()
```

Listing 6.5: Example of creating a generic correlation matrix plot.

Generating the plot, you can see that it gives the same information although making it a little harder to see what attributes are correlated by name. Use this generic plot as a first cut

to understand the correlations in your dataset and customize it like the first example in order to read off more specific data if needed.

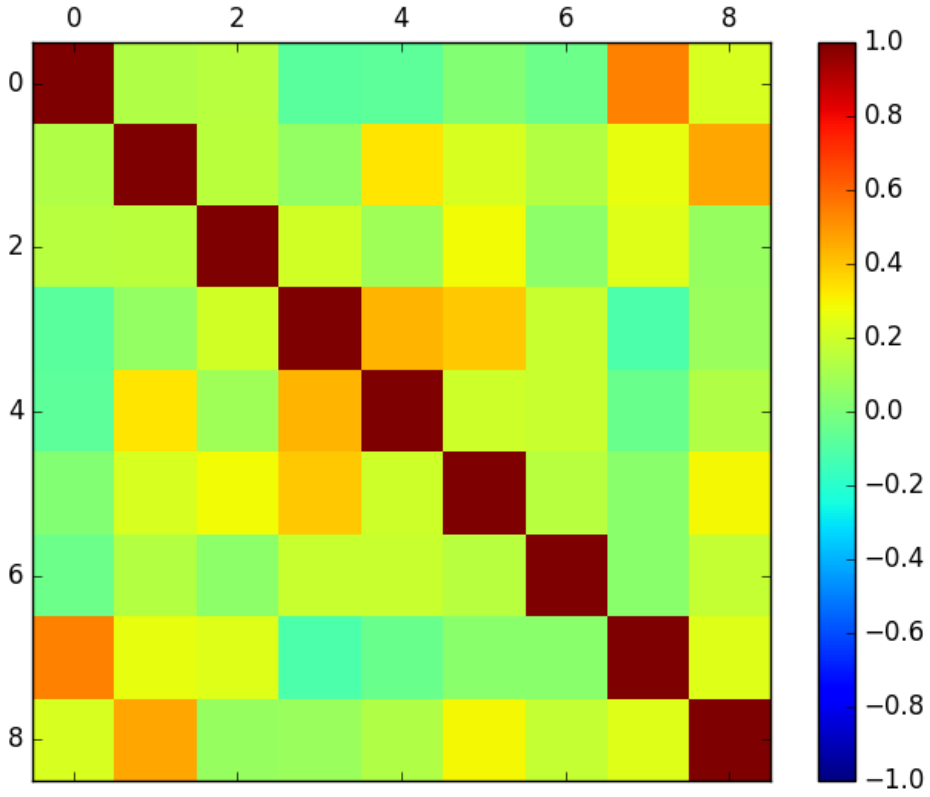

Figure 6.5: Generic Correlation matrix plot.

6.2.2 Scatter Plot Matrix

A scatter plot shows the relationship between two variables as dots in two dimensions, one axis for each attribute. You can create a scatter plot for each pair of attributes in your data. Drawing all these scatter plots together is called a scatter plot matrix. Scatter plots are useful for spotting structured relationships between variables, like whether you could summarize the relationship between two variables with a line. Attributes with structured relationships may also be correlated and good candidates for removal from your dataset.

```
# Scatter Plot Matrix
import matplotlib.pyplot as plt
import pandas
from pandas.tools.plotting import scatter_matrix
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pandas.read_csv(url, names=names)
scatter_matrix(data)
plt.show()
```

Listing 6.6: Example of creating a scatter plot matrix.

Like the Correlation Matrix Plot above, the scatter plot matrix is symmetrical. This is useful to look at the pairwise relationships from different perspectives. Because there is little point of drawing a scatter plot of each variable with itself, the diagonal shows histograms of each attribute.

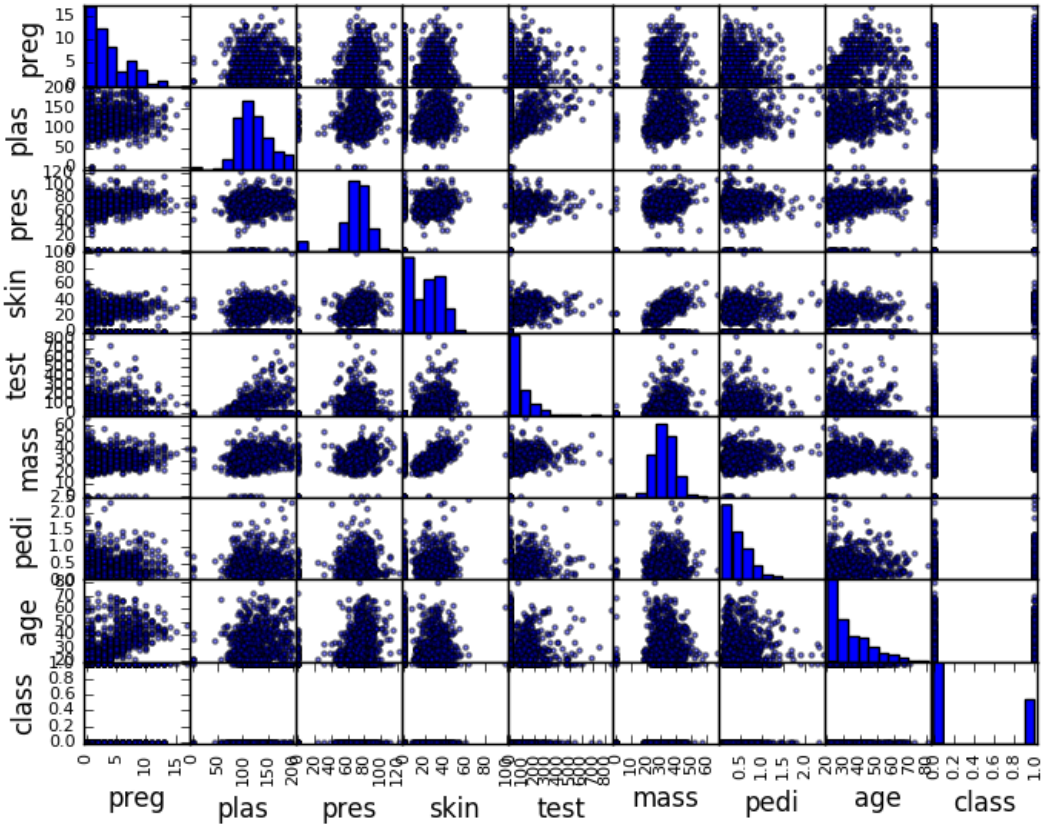

Figure 6.6: Scatter plot matrix of the data.

6.3 Summary

In this chapter you discovered a number of ways that you can better understand your machine learning data in Python using Pandas. Specifically, you learned how to plot your data using:

- Histograms.
- Density Plots.
- Box and Whisker Plots.
- Correlation Matrix Plot.
- Scatter Plot Matrix.

6.3.1 Next

Now that you know two ways to learn more about your data, you are ready to start manipulating it. In the next lesson you will discover how you can prepare your data to best expose the structure of your problem to modeling algorithms.

Chapter 7

Prepare Your Data For Machine Learning

Many machine learning algorithms make assumptions about your data. It is often a very good idea to prepare your data in such way to best expose the structure of the problem to the machine learning algorithms that you intend to use. In this chapter you will discover how to prepare your data for machine learning in Python using scikit-learn. After completing this lesson you will know how to:

1. Rescale data.
2. Standardize data.
3. Normalize data.
4. Binarize data.

Let's get started.

7.1 Need For Data Pre-processing

You almost always need to pre-process your data. It is a required step. A difficulty is that different algorithms make different assumptions about your data and may require different transforms. Further, when you follow all of the rules and prepare your data, sometimes algorithms can deliver better results without pre-processing.

Generally, I would recommend creating many different views and transforms of your data, then exercise a handful of algorithms on each view of your dataset. This will help you to flush out which data transforms might be better at exposing the structure of your problem in general.

7.2 Data Transforms

In this lesson you will work through 4 different data pre-processing recipes for machine learning. The Pima Indian diabetes dataset is used in each recipe. Each recipe follows the same structure:

- Load the dataset from a URL.

- Split the dataset into the input and output variables for machine learning.
- Apply a pre-processing transform to the input variables.
- Summarize the data to show the change.

The scikit-learn library provides two standard idioms for transforming data. Each are useful in different circumstances. The transforms are calculated in such a way that they can be applied to your training data and any samples of data you may have in the future. The scikit-learn documentation has some information on how to use various different pre-processing methods:

- Fit and Multiple Transform.
- Combined Fit-And-Transform.

The Fit and Multiple Transform method is the preferred approach. You call the `fit()` function to prepare the parameters of the transform once on your data. Then later you can use the `transform()` function on the same data to prepare it for modeling and again on the test or validation dataset or new data that you may see in the future. The Combined Fit-And-Transform is a convenience that you can use for one off tasks. This might be useful if you are interested in plotting or summarizing the transformed data. You can review the `preprocess` API in scikit-learn here[1].

7.3 Rescale Data

When your data is comprised of attributes with varying scales, many machine learning algorithms can benefit from rescaling the attributes to all have the same scale. Often this is referred to as normalization and attributes are often rescaled into the range between 0 and 1. This is useful for optimization algorithms in used in the core of machine learning algorithms like gradient descent. It is also useful for algorithms that weight inputs like regression and neural networks and algorithms that use distance measures like k-Nearest Neighbors. You can rescale your data using scikit-learn using the `MinMaxScaler` class[2].

```
# Rescale data (between 0 and 1)
import pandas
import scipy
import numpy
from sklearn.preprocessing import MinMaxScaler
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
# separate array into input and output components
X = array[:,0:8]
Y = array[:,8]
scaler = MinMaxScaler(feature_range=(0, 1))
rescaledX = scaler.fit_transform(X)
# summarize transformed data
```

[1] http://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing
[2] http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

```
numpy.set_printoptions(precision=3)
print(rescaledX[0:5,:])
```

Listing 7.1: Example of rescaling data.

After rescaling you can see that all of the values are in the range between 0 and 1.

```
[[ 0.353 0.744  0.59   0.354  0.     0.501  0.234  0.483]
 [ 0.059 0.427  0.541  0.293  0.     0.396  0.117  0.167]
 [ 0.471 0.92   0.525  0.     0.     0.347  0.254  0.183]
 [ 0.059 0.447  0.541  0.232  0.111  0.419  0.038  0.   ]
 [ 0.    0.688  0.328  0.354  0.199  0.642  0.944  0.2  ]]
```

Listing 7.2: Output of rescaling data.

7.4 Standardize Data

Standardization is a useful technique to transform attributes with a Gaussian distribution and differing means and standard deviations to a standard Gaussian distribution with a mean of 0 and a standard deviation of 1. It is most suitable for techniques that assume a Gaussian distribution in the input variables and work better with rescaled data, such as linear regression, logistic regression and linear discriminate analysis. You can standardize data using scikit-learn with the `StandardScaler` class[3].

```
# Standardize data (0 mean, 1 stdev)
from sklearn.preprocessing import StandardScaler
import pandas
import numpy
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
# separate array into input and output components
X = array[:,0:8]
Y = array[:,8]
scaler = StandardScaler().fit(X)
rescaledX = scaler.transform(X)
# summarize transformed data
numpy.set_printoptions(precision=3)
print(rescaledX[0:5,:])
```

Listing 7.3: Example of standardizing data.

The values for each attribute now have a mean value of 0 and a standard deviation of 1.

```
[[ 0.64   0.848  0.15   0.907 -0.693 0.204 0.468  1.426]
 [-0.845 -1.123 -0.161 0.531 -0.693 -0.684 -0.365 -0.191]
 [ 1.234  1.944 -0.264 -1.288 -0.693 -1.103 0.604 -0.106]
 [-0.845 -0.998 -0.161 0.155  0.123 -0.494 -0.921 -1.042]
 [-1.142  0.504 -1.505 0.907  0.766  1.41   5.485 -0.02 ]]
```

Listing 7.4: Output of rescaling data.

[3] http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

7.5 Normalize Data

Normalizing in scikit-learn refers to rescaling each observation (row) to have a length of 1 (called a unit norm or a vector with the length of 1 in linear algebra). This pre-processing method can be useful for sparse datasets (lots of zeros) with attributes of varying scales when using algorithms that weight input values such as neural networks and algorithms that use distance measures such as k-Nearest Neighbors. You can normalize data in Python with scikit-learn using the `Normalizer` class[4].

```
# Normalize data (length of 1)
from sklearn.preprocessing import Normalizer
import pandas
import numpy
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
# separate array into input and output components
X = array[:,0:8]
Y = array[:,8]
scaler = Normalizer().fit(X)
normalizedX = scaler.transform(X)
# summarize transformed data
numpy.set_printoptions(precision=3)
print(normalizedX[0:5,:])
```

Listing 7.5: Example of normalizing data.

The rows are normalized to length 1.

```
[[ 0.034 0.828 0.403 0.196 0.    0.188 0.004 0.28 ]
 [ 0.008 0.716 0.556 0.244 0.    0.224 0.003 0.261]
 [ 0.04  0.924 0.323 0.    0.    0.118 0.003 0.162]
 [ 0.007 0.588 0.436 0.152 0.622 0.186 0.001 0.139]
 [ 0.    0.596 0.174 0.152 0.731 0.188 0.01  0.144]]
```

Listing 7.6: Output of normalizing data.

7.6 Binarize Data (Make Binary)

You can transform your data using a binary threshold. All values above the threshold are marked 1 and all equal to or below are marked as 0. This is called *binarizing* your data or *thresholding* your data. It can be useful when you have probabilities that you want to make crisp values. It is also useful when feature engineering and you want to add new features that indicate something meaningful. You can create new binary attributes in Python using scikit-learn with the `Binarizer` class[5].

```
# binarization
from sklearn.preprocessing import Binarizer
import pandas
```

[4] http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Normalizer.html
[5] http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.Binarizer.html

```
import numpy
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
# separate array into input and output components
X = array[:,0:8]
Y = array[:,8]
binarizer = Binarizer(threshold=0.0).fit(X)
binaryX = binarizer.transform(X)
# summarize transformed data
numpy.set_printoptions(precision=3)
print(binaryX[0:5,:])
```

Listing 7.7: Example of binarizing data.

You can see that all values equal or less than 0 are marked 0 and all of those above 0 are marked 1.

```
[[ 1.  1.  1.  1.  0.  1.  1.  1.]
 [ 1.  1.  1.  1.  0.  1.  1.  1.]
 [ 1.  1.  1.  0.  0.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.]
 [ 0.  1.  1.  1.  1.  1.  1.  1.]]
```

Listing 7.8: Output of normalizing data.

7.7 Summary

In this chapter you discovered how you can prepare your data for machine learning in Python using scikit-learn. You now have recipes to:

- Rescale data.
- Standardize data.
- Normalize data.
- Binarize data.

7.7.1 Next

You now know how to transform your data to best expose the structure of your problem to the modeling algorithms. In the next lesson you will discover how to select the features of your data that are most relevant to making predictions.

Chapter 8

Feature Selection For Machine Learning

The data features that you use to train your machine learning models have a huge influence on the performance you can achieve. Irrelevant or partially relevant features can negatively impact model performance. In this chapter you will discover automatic feature selection techniques that you can use to prepare your machine learning data in Python with scikit-learn. After completing this lesson you will know how to use:

1. Univariate Selection.
2. Recursive Feature Elimination.
3. Principle Component Analysis.
4. Feature Importance.

Let's get started.

8.1 Feature Selection

Feature selection is a process where you automatically select those features in your data that contribute most to the prediction variable or output in which you are interested. Having irrelevant features in your data can decrease the accuracy of many models, especially linear algorithms like linear and logistic regression. Three benefits of performing feature selection before modeling your data are:

- **Reduces Overfitting**: Less redundant data means less opportunity to make decisions based on noise.
- **Improves Accuracy**: Less misleading data means modeling accuracy improves.
- **Reduces Training Time**: Less data means that algorithms train faster.

You can learn more about feature selection with scikit-learn in the article Feature selection[1]. Each feature selection recipes will use the Pima Indians onset of diabetes dataset.

[1] http://scikit-learn.org/stable/modules/feature_selection.html

8.2 Univariate Selection

Statistical tests can be used to select those features that have the strongest relationship with the output variable. The scikit-learn library provides the `SelectKBest` class[2] that can be used with a suite of different statistical tests to select a specific number of features. The example below uses the chi-squared (chi^2) statistical test for non-negative features to select 4 of the best features from the Pima Indians onset of diabetes dataset.

```
# Feature Extraction with Univariate Statistical Tests (Chi-squared for classification)
import pandas
import numpy
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
# load data
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
test = SelectKBest(score_func=chi2, k=4)
fit = test.fit(X, Y)
# summarize scores
numpy.set_printoptions(precision=3)
print(fit.scores_)
features = fit.transform(X)
# summarize selected features
print(features[0:5,:])
```

Listing 8.1: Example of univariate feature selection.

You can see the scores for each attribute and the 4 attributes chosen (those with the highest scores): `plas`, `test`, `mass` and `age`. I got the names for the chosen attributes by manually mapping the index of the 4 highest scores to the index of the attribute names.

```
[  111.52  1411.887   17.605   53.108 2175.565  127.669    5.393
   181.304]
[[ 148.    0.   33.6  50. ]
 [  85.    0.   26.6  31. ]
 [ 183.    0.   23.3  32. ]
 [  89.   94.   28.1  21. ]
 [ 137.  168.   43.1  33. ]]
```

Listing 8.2: Output of univariate feature selection.

8.3 Recursive Feature Elimination

The Recursive Feature Elimination (or RFE) works by recursively removing attributes and building a model on those attributes that remain. It uses the model accuracy to identify which

[2] http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest

attributes (and combination of attributes) contribute the most to predicting the target attribute. You can learn more about the `RFE` class[3] in the scikit-learn documentation. The example below uses RFE with the logistic regression algorithm to select the top 3 features. The choice of algorithm does not matter too much as long as it is skillful and consistent.

```
# Feature Extraction with RFE
from pandas import read_csv
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
# load data
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
model = LogisticRegression()
rfe = RFE(model, 3)
fit = rfe.fit(X, Y)
print("Num Features: %d") % fit.n_features_
print("Selected Features: %s") % fit.support_
print("Feature Ranking: %s") % fit.ranking_
```

Listing 8.3: Example of RFE feature selection.

You can see that RFE chose the top 3 features as `preg`, `mass` and `pedi`. These are marked `True` in the `support_` array and marked with a choice `1` in the `ranking_` array. Again, you can manually map the feature indexes to the indexes of attribute names.

```
Num Features: 3
Selected Features: [ True False False False False True True False]
Feature Ranking: [1 2 3 5 6 1 1 4]
```

Listing 8.4: Output of RFE feature selection.

8.4 Principal Component Analysis

Principal Component Analysis (or PCA) uses linear algebra to transform the dataset into a compressed form. Generally this is called a data reduction technique. A property of PCA is that you can choose the number of dimensions or principal components in the transformed result. In the example below, we use PCA and select 3 principal components. Learn more about the `PCA` class in scikit-learn by reviewing the API[4].

```
# Feature Extraction with PCA
import numpy
from pandas import read_csv
from sklearn.decomposition import PCA
# load data
url = "https://goo.gl/vhm1eU"
```

[3] http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html#sklearn.feature_selection.RFE

[4] http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

```
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
pca = PCA(n_components=3)
fit = pca.fit(X)
# summarize components
print("Explained Variance: %s") % fit.explained_variance_ratio_
print(fit.components_)
```

Listing 8.5: Example of PCA feature extraction.

You can see that the transformed dataset (3 principal components) bare little resemblance to the source data.

```
Explained Variance: [ 0.88854663 0.06159078 0.02579012]
[[ -2.02176587e-03 9.78115765e-02 1.60930503e-02 6.07566861e-02
    9.93110844e-01 1.40108085e-02 5.37167919e-04 -3.56474430e-03]
 [  2.26488861e-02 9.72210040e-01 1.41909330e-01 -5.78614699e-02
   -9.46266913e-02 4.69729766e-02 8.16804621e-04 1.40168181e-01]
 [ -2.24649003e-02 1.43428710e-01 -9.22467192e-01 -3.07013055e-01
    2.09773019e-02 -1.32444542e-01 -6.39983017e-04 -1.25454310e-01]]
```

Listing 8.6: Output of PCA feature extraction.

8.5 Feature Importance

Bagged decision trees like Random Forest and Extra Trees can be used to estimate the importance of features. In the example below we construct a `ExtraTreesClassifier` classifier for the Pima Indians onset of diabetes dataset. You can learn more about the `ExtraTreesClassifier` class[5] in the scikit-learn API.

```
# Feature Importance with Extra Trees Classifier
from pandas import read_csv
from sklearn.ensemble import ExtraTreesClassifier
# load data
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# feature extraction
model = ExtraTreesClassifier()
model.fit(X, Y)
print(model.feature_importances_)
```

Listing 8.7: Example of feature importance.

[5] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

You can see that we are given an importance score for each attribute where the larger the score, the more important the attribute. The scores suggest at the importance of `plas`, `age` and `mass`.

```
[ 0.11070069 0.2213717  0.08824115 0.08068703 0.07281761 0.14548537 0.12654214 0.15415431]
```

Listing 8.8: Output of feature importance.

8.6 Summary

In this chapter you discovered feature selection for preparing machine learning data in Python with scikit-learn. You learned about 4 different automatic feature selection techniques:

- Univariate Selection.
- Recursive Feature Elimination.
- Principle Component Analysis.
- Feature Importance.

8.6.1 Next

Now it is time to start looking at how to evaluate machine learning algorithms on your dataset. In the next lesson you will discover resampling methods that can be used to estimate the performance of a machine learning algorithm on unseen data.

Chapter 9

Evaluate the Performance of Machine Learning Algorithms with Resampling

You need to know how well your algorithms perform on unseen data. The best way to evaluate the performance of an algorithm would be to make predictions for new data to which you already know the answers. The second best way is to use clever techniques from statistics called resampling methods that allow you to make accurate estimates for how well your algorithm will perform on new data. In this chapter you will discover how you can estimate the accuracy of your machine learning algorithms using resampling methods in Python and scikit-learn on the Pima Indians dataset. Let's get started.

9.1 Evaluate Machine Learning Algorithms

Why can't you train your machine learning algorithm on your dataset and use predictions from this same dataset to evaluate machine learning algorithms? The simple answer is overfitting.

Imagine an algorithm that remembers every observation it is shown during training. If you evaluated your machine learning algorithm on the same dataset used to train the algorithm, then an algorithm like this would have a perfect score on the training dataset. But the predictions it made on new data would be terrible. We must evaluate our machine learning algorithms on data that is not used to train the algorithm.

The evaluation is an estimate that we can use to talk about how well we think the algorithm may actually do in practice. It is not a guarantee of performance. Once we estimate the performance of our algorithm, we can then re-train the final algorithm on the entire training dataset and get it ready for operational use. Next up we are going to look at four different techniques that we can use to split up our training dataset and create useful estimates of performance for our machine learning algorithms:

- Train and Test Sets.
- k-fold Cross Validation.
- Leave One Out Cross Validation.
- Repeated Random Test-Train Splits.

9.2 Split into Train and Test Sets

The simplest method that we can use to evaluate the performance of a machine learning algorithm is to use different training and testing datasets. We can take our original dataset and split it into two parts. Train the algorithm on the first part, make predictions on the second part and evaluate the predictions against the expected results. The size of the split can depend on the size and specifics of your dataset, although it is common to use 67% of the data for training and the remaining 33% for testing.

This algorithm evaluation technique is very fast. It is ideal for large datasets (millions of records) where there is strong evidence that both splits of the data are representative of the underlying problem. Because of the speed, it is useful to use this approach when the algorithm you are investigating is slow to train. A downside of this technique is that it can have a high variance. This means that differences in the training and test dataset can result in meaningful differences in the estimate of accuracy. In the example below we split the Pima Indians dataset into 67%/33% splits for training and test and evaluate the accuracy of a Logistic Regression model.

```
# Evaluate using a train and a test set
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
test_size = 0.33
seed = 7
X_train, X_test, Y_train, Y_test = cross_validation.train_test_split(X, Y,
    test_size=test_size, random_state=seed)
model = LogisticRegression()
model.fit(X_train, Y_train)
result = model.score(X_test, Y_test)
print("Accuracy: %.3f%%") % (result*100.0)
```

Listing 9.1: Example of evaluating an algorithm with a train and test set.

We can see that the estimated accuracy for the model was approximately 75%. Note that in addition to specifying the size of the split, we also specify the random seed. Because the split of the data is random, we want to ensure that the results are reproducible. By specifying the random seed we ensure that we get the same random numbers each time we run the code and in turn the same split of data. This is important if we want to compare this result to the estimated accuracy of another machine learning algorithm or the same algorithm with a different configuration. To ensure the comparison was apples-for-apples, we must ensure that they are trained and tested on exactly the same data.

```
Accuracy: 75.591%
```

Listing 9.2: Output of evaluating an algorithm with a train and test set.

9.3 K-fold Cross Validation

Cross validation is an approach that you can use to estimate the performance of a machine learning algorithm with less variance than a single train-test set split. It works by splitting the dataset into k-parts (e.g. $k = 5$ or $k = 10$). Each split of the data is called a fold. The algorithm is trained on $k - 1$ folds with one held back and tested on the held back fold. This is repeated so that each fold of the dataset is given a chance to be the held back test set. After running cross validation you end up with k different performance scores that you can summarize using a mean and a standard deviation.

The result is a more reliable estimate of the performance of the algorithm on new data. It is more accurate because the algorithm is trained and evaluated multiple times on different data. The choice of k must allow the size of each test partition to be large enough to be a reasonable sample of the problem, whilst allowing enough repetitions of the train-test evaluation of the algorithm to provide a fair estimate of the algorithms performance on unseen data. For modest sized datasets in the thousands or tens of thousands of records, k values of 3, 5 and 10 are common. In the example below we use 10-fold cross validation.

```
# Evaluate using Cross Validation
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LogisticRegression()
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print("Accuracy: %.3f%% (%.3f%%)") % (results.mean()*100.0, results.std()*100.0)
```

Listing 9.3: Example of evaluating an algorithm with k-fold Cross Validation.

You can see that we report both the mean and the standard deviation of the performance measure. When summarizing performance measures, it is a good practice to summarize the distribution of the measures, in this case assuming a Gaussian distribution of performance (a very reasonable assumption) and recording the mean and standard deviation.

```
Accuracy: 76.951% (4.841%)
```

Listing 9.4: Output of evaluating an algorithm with k-fold Cross Validation.

9.4 Leave One Out Cross Validation

You can configure cross validation so that the size of the fold is 1 (k is set to the number of observations in your dataset). This variation of cross validation is called leave-one-out cross validation. The result is a large number of performance measures that can be summarized in

an effort to give a more reasonable estimate of the accuracy of your model on unseen data. A downside is that it can be a computationally more expensive procedure than k-fold cross validation. In the example below we use leave-one-out cross validation.

```
# Evaluate using Leave One Out Cross Validation
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
loocv = cross_validation.LeaveOneOut(n=num_instances)
model = LogisticRegression()
results = cross_validation.cross_val_score(model, X, Y, cv=loocv)
print("Accuracy: %.3f%% (%.3f%%)") % (results.mean()*100.0, results.std()*100.0)
```

Listing 9.5: Example of evaluating an algorithm with Leave One Out Cross Validation.

You can see in the standard deviation that the score has more variance than the k-fold cross validation results described above.

```
Accuracy: 76.823% (42.196%)
```

Listing 9.6: Output of evaluating an algorithm with Leave One Out Cross Validation.

9.5 Repeated Random Test-Train Splits

Another variation on k-fold cross validation is to create a random split of the data like the train/test split described above, but repeat the process of splitting and evaluation of the algorithm multiple times, like cross validation. This has the speed of using a train/test split and the reduction in variance in the estimated performance of k-fold cross validation. You can also repeat the process many more times as needed to improve the accuracy. A down side is that repetitions may include much of the same data in the train or the test split from run to run, introducing redundancy into the evaluation. The example below splits the data into a 67%/33% train/test split and repeats the process 10 times.

```
# Evaluate using Shuffle Split Cross Validation
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_samples = 10
test_size = 0.33
num_instances = len(X)
```

```
seed = 7
kfold = cross_validation.ShuffleSplit(n=num_instances, n_iter=num_samples,
    test_size=test_size, random_state=seed)
model = LogisticRegression()
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print("Accuracy: %.3f%% (%.3f%%)") % (results.mean()*100.0, results.std()*100.0)
```

Listing 9.7: Example of evaluating an algorithm with Shuffle Split Cross Validation.

We can see that in this case the distribution of the performance measure is on par with k-fold cross validation above.

```
Accuracy: 76.496% (1.698%)
```

Listing 9.8: Output of evaluating an algorithm with Shuffle Split Cross Validation.

9.6 What Techniques to Use When

This section lists some tips to consider what resampling technique to use in different circumstances.

- Generally k-fold cross validation is the *gold standard* for evaluating the performance of a machine learning algorithm on unseen data with k set to 3, 5, or 10.
- Using a train/test split is good for speed when using a slow algorithm and produces performance estimates with lower bias when using large datasets.
- Techniques like leave-one-out cross validation and repeated random splits can be useful intermediates when trying to balance variance in the estimated performance, model training speed and dataset size.

The best advice is to experiment and find a technique for your problem that is fast and produces reasonable estimates of performance that you can use to make decisions. If in doubt, use 10-fold cross validation.

9.7 Summary

In this chapter you discovered statistical techniques that you can use to estimate the performance of your machine learning algorithms, called resampling. Specifically, you learned about:

- Train and Test Sets.
- Cross Validation.
- Leave One Out Cross Validation.
- Repeated Random Test-Train Splits.

9.7.1 Next

In the next section you will learn how you can evaluate the performance of classification and regression algorithms using a suite of different metrics and built in evaluation reports.

Chapter 10

Machine Learning Algorithm Performance Metrics

The metrics that you choose to evaluate your machine learning algorithms are very important. Choice of metrics influences how the performance of machine learning algorithms is measured and compared. They influence how you weight the importance of different characteristics in the results and your ultimate choice of which algorithm to choose. In this chapter you will discover how to select and use different machine learning performance metrics in Python with scikit-learn. Let's get started.

10.1 Algorithm Evaluation Metrics

In this lesson, various different algorithm evaluation metrics are demonstrated for both classification and regression type machine learning problems. In each recipe, the dataset is downloaded directly from the UCI Machine Learning repository.

- For classification metrics, the Pima Indians onset of diabetes dataset is used as demonstration. This is a binary classification problem where all of the input variables are numeric.
- For regression metrics, the Boston House Price dataset is used as demonstration. this is a regression problem where all of the input variables are also numeric.

All recipes evaluate the same algorithms, Logistic Regression for classification and Linear Regression for the regression problems. A 10-fold cross validation test harness is used to demonstrate each metric, because this is the most likely scenario you will use when employing different algorithm evaluation metrics.

A caveat in these recipes is the `cross_validation.cross_val_score` function[1] used to report the performance in each recipe. It does allow the use of different scoring metrics that will be discussed, but all scores are reported so that they can be sorted in ascending order (largest score is best). Some evaluation metrics (like mean squared error) are naturally descending scores (the smallest score is best) and as such are reported as negative by the

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

`cross_validation.cross_val_score()` function. This is important to note, because some scores will be reported as negative that by definition can never be negative. I will remind you about this caveat as we work through the lesson.

You can learn more about machine learning algorithm performance metrics supported by scikit-learn on the page *Model evaluation: quantifying the quality of predictions*[2]. Let's get on with the evaluation metrics.

10.2 Classification Metrics

Classification problems are perhaps the most common type of machine learning problem and as such there are a myriad of metrics that can be used to evaluate predictions for these problems. In this section we will review how to use the following metrics:

- Classification Accuracy.
- Logarithmic Loss.
- Area Under ROC Curve.
- Confusion Matrix.
- Classification Report.

10.2.1 Classification Accuracy

Classification accuracy is the number of correct predictions made as a ratio of all predictions made. This is the most common evaluation metric for classification problems, it is also the most misused. It is really only suitable when there are an equal number of observations in each class (which is rarely the case) and that all predictions and prediction errors are equally important, which is often not the case. Below is an example of calculating classification accuracy.

```
# Cross Validation Classification Accuracy
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LogisticRegression()
scoring = 'accuracy'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print("Accuracy: %.3f (%.3f)") % (results.mean(), results.std())
```

Listing 10.1: Example of evaluating an algorithm by classification accuracy.

[2] http://scikit-learn.org/stable/modules/model_evaluation.html

You can see that the ratio is reported. This can be converted into a percentage by multiplying the value by 100, giving an accuracy score of approximately 77% accurate.

```
Accuracy: 0.770 (0.048)
```

Listing 10.2: Output of evaluating an algorithm by classification accuracy.

10.2.2 Logarithmic Loss

Logarithmic loss (or logloss) is a performance metric for evaluating the predictions of probabilities of membership to a given class. The scalar probability between 0 and 1 can be seen as a measure of confidence for a prediction by an algorithm. Predictions that are correct or incorrect are rewarded or punished proportionally to the confidence of the prediction. Below is an example of calculating logloss for Logistic regression predictions on the Pima Indians onset of diabetes dataset.

```
# Cross Validation Classification LogLoss
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LogisticRegression()
scoring = 'log_loss'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print("Logloss: %.3f (%.3f)") % (results.mean(), results.std())
```

Listing 10.3: Example of evaluating an algorithm by logloss.

Smaller logloss is better with 0 representing a perfect logloss. As mentioned above, the measure is inverted to be ascending when using the `cross_val_score()` function.

```
Logloss: -0.493 (0.047)
```

Listing 10.4: Output of evaluating an algorithm by logloss.

10.2.3 Area Under ROC Curve

Area under ROC Curve (or AUC for short) is a performance metric for binary classification problems. The AUC represents a model's ability to discriminate between positive and negative classes. An area of 1.0 represents a model that made all predictions perfectly. An area of 0.5 represents a model that is as good as random. ROC can be broken down into sensitivity and specificity. A binary classification problem is really a trade-off between sensitivity and specificity.

- Sensitivity is the true positive rate also called the recall. It is the number of instances from the positive (first) class that actually predicted correctly.
- Specificity is also called the true negative rate. Is the number of instances from the negative (second) class that were actually predicted correctly.

The example below provides a demonstration of calculating AUC.

```
# Cross Validation Classification ROC AUC
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LogisticRegression()
scoring = 'roc_auc'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print("AUC: %.3f (%.3f)") % (results.mean(), results.std())
```

Listing 10.5: Example of evaluating an algorithm by AUC.

You can see the AUC is relatively close to 1 and greater than 0.5, suggesting some skill in the predictions

```
AUC: 0.824 (0.041)
```

Listing 10.6: Output of evaluating an algorithm by AUC.

10.2.4 Confusion Matrix

The confusion matrix is a handy presentation of the accuracy of a model with two or more classes. The table presents predictions on the x-axis and accuracy outcomes on the y-axis. The cells of the table are the number of predictions made by a machine learning algorithm. For example, a machine learning algorithm can predict 0 or 1 and each prediction may actually have been a 0 or 1. Predictions for 0 that were actually 0 appear in the cell for $prediction = 0$ and $actual = 0$, whereas predictions for 0 that were actually 1 appear in the cell for $prediction = 0$ and $actual = 1$. And so on. Below is an example of calculating a confusion matrix for a set of predictions by a Logistic Regression on the Pima Indians onset of diabetes dataset.

```
# Cross Validation Classification Confusion Matrix
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
```

```
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
test_size = 0.33
seed = 7
X_train, X_test, Y_train, Y_test = cross_validation.train_test_split(X, Y,
    test_size=test_size, random_state=seed)
model = LogisticRegression()
model.fit(X_train, Y_train)
predicted = model.predict(X_test)
matrix = confusion_matrix(Y_test, predicted)
print(matrix)
```

Listing 10.7: Example of evaluating an algorithm by confusion matrix.

Although the array is printed without headings, you can see that the majority of the predictions fall on the diagonal line of the matrix (which are correct predictions).

```
[[141  21]
 [ 41  51]]
```

Listing 10.8: Output of evaluating an algorithm by confusion matrix.

10.2.5 Classification Report

The scikit-learn library provides a convenience report when working on classification problems to give you a quick idea of the accuracy of a model using a number of measures. The `classification_report()` function displays the precision, recall, F1-score and support for each class. The example below demonstrates the report on the binary classification problem.

```
# Cross Validation Classification Report
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
test_size = 0.33
seed = 7
X_train, X_test, Y_train, Y_test = cross_validation.train_test_split(X, Y,
    test_size=test_size, random_state=seed)
model = LogisticRegression()
model.fit(X_train, Y_train)
predicted = model.predict(X_test)
report = classification_report(Y_test, predicted)
print(report)
```

Listing 10.9: Example of evaluating an algorithm by classification report.

You can see good prediction and recall for the algorithm.

```
             precision    recall  f1-score   support

        0.0       0.77      0.87      0.82       162
        1.0       0.71      0.55      0.62        92

avg / total       0.75      0.76      0.75       254
```

Listing 10.10: Output of evaluating an algorithm by classification report.

10.3 Regression Metrics

In this section will review 3 of the most common metrics for evaluating predictions on regression machine learning problems:

- Mean Absolute Error.
- Mean Squared Error.
- R^2.

10.3.1 Mean Absolute Error

The Mean Absolute Error (or MAE) is the sum of the absolute differences between predictions and actual values. It gives an idea of how wrong the predictions were. The measure gives an idea of the magnitude of the error, but no idea of the direction (e.g. over or under predicting). The example below demonstrates calculating mean absolute error on the Boston house price dataset.

```
# Cross Validation Regression MAE
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LinearRegression
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LinearRegression()
scoring = 'mean_absolute_error'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print("MAE: %.3f (%.3f)") % (results.mean(), results.std())
```

Listing 10.11: Example of evaluating an algorithm by Mean Absolute Error.

A value of 0 indicates no error or perfect predictions. Like logloss, this metric is inverted by the `cross_val_score()` function.

```
MAE: -4.005 (2.084)
```

Listing 10.12: Output of evaluating an algorithm by Mean Absolute Error.

10.3.2 Mean Squared Error

The Mean Squared Error (or MSE) is much like the mean absolute error in that it provides a gross idea of the magnitude of error. Taking the square root of the mean squared error converts the units back to the original units of the output variable and can be meaningful for description and presentation. This is called the Root Mean Squared Error (or RMSE). The example below provides a demonstration of calculating mean squared error.

```
# Cross Validation Regression MSE
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LinearRegression
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LinearRegression()
scoring = 'mean_squared_error'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print("MSE: %.3f (%.3f)") % (results.mean(), results.std())
```

Listing 10.13: Example of evaluating an algorithm by Mean Squared Error.

This metric too is inverted so that the results are increasing. Remember to take the absolute value before taking the square root if you are interested in calculating the RMSE.

```
MSE: -34.705 (45.574)
```

Listing 10.14: Output of evaluating an algorithm by Mean Squared Error.

10.3.3 R^2 Metric

The R^2 (or R Squared) metric provides an indication of the goodness of fit of a set of predictions to the actual values. In statistical literature this measure is called the coefficient of determination. This is a value between 0 and 1 for no-fit and perfect fit respectively. The example below provides a demonstration of calculating the mean R^2 for a set of predictions.

```
# Cross Validation Regression R^2
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LinearRegression
url = "https://goo.gl/sXleFv"
```

```
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LinearRegression()
scoring = 'r2'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print("R^2: %.3f (%.3f)") % (results.mean(), results.std())
```

Listing 10.15: Example of evaluating an algorithm by R Squared.

You can see the predictions have a reasonable fit to the actual values with a value closer to zero and less than 0.5.

```
R^2: 0.203 (0.595)
```

Listing 10.16: Output of evaluating an algorithm by R Squared.

10.4 Summary

In this chapter you discovered metrics that you can use to evaluate your machine learning algorithms.

You learned about three classification metrics: Accuracy, Logarithmic Loss and Area Under ROC Curve. You also learned about two convenience methods for classification prediction results: the Confusion Matrix and the Classification Report. Finally, you also learned about three metrics for regression problems: Mean Absolute Error, Mean Squared Error and R^2.

10.4.1 Next

You now know how to evaluate the performance of machine learning algorithms using a variety of different metrics and how to use those metrics to estimate the performance of algorithms on new unseen data using resampling. In the next lesson you will start looking at machine learning algorithms themselves, starting with classification techniques.

Chapter 11

Spot-Check Classification Algorithms

Spot-checking is a way of discovering which algorithms perform well on your machine learning problem. You cannot know which algorithms are best suited to your problem beforehand. You must trial a number of methods and focus attention on those that prove themselves the most promising. In this chapter you will discover six machine learning algorithms that you can use when spot-checking your classification problem in Python with scikit-learn. After completing this lesson you will know:

1. How to spot-check machine learning algorithms on a classification problem.
2. How to spot-check two linear classification algorithms.
3. How to spot-check four nonlinear classification algorithms.

Let's get started.

11.1 Algorithm Spot-Checking

You cannot know which algorithm will work best on your dataset beforehand. You must use trial and error to discover a shortlist of algorithms that do well on your problem that you can then double down on and tune further. I call this process spot-checking.

The question is not: *What algorithm should I use on my dataset?* Instead it is: *What algorithms should I spot-check on my dataset?* You can guess at what algorithms might do well on your dataset, and this can be a good starting point. I recommend trying a mixture of algorithms and see what is good at picking out the structure in your data. Below are some suggestions when spot-checking algorithms on your dataset:

- Try a mixture of algorithm representations (e.g. instances and trees).
- Try a mixture of learning algorithms (e.g. different algorithms for learning the same type of representation).
- Try a mixture of modeling types (e.g. linear and nonlinear functions or parametric and nonparametric).

Let's get specific. In the next section, we will look at algorithms that you can use to spot-check on your next classification machine learning project in Python.

11.2 Algorithms Overview

We are going to take a look at six classification algorithms that you can spot-check on your dataset. Starting with two linear machine learning algorithms:

- Logistic Regression.
- Linear Discriminant Analysis.

Then looking at four nonlinear machine learning algorithms:

- k-Nearest Neighbors.
- Naive Bayes.
- Classification and Regression Trees.
- Support Vector Machines.

Each recipe is demonstrated on the Pima Indians onset of Diabetes dataset. A test harness using 10-fold cross validation is used to demonstrate how to spot-check each machine learning algorithm and mean accuracy measures are used to indicate algorithm performance. The recipes assume that you know about each machine learning algorithm and how to use them. We will not go into the API or parameterization of each algorithm.

11.3 Linear Machine Learning Algorithms

This section demonstrates minimal recipes for how to use two linear machine learning algorithms: logistic regression and linear discriminant analysis.

11.3.1 Logistic Regression

Logistic regression assumes a Gaussian distribution for the numeric input variables and can model binary classification problems. You can construct a logistic regression model using the `LogisticRegression` class[1].

```
# Logistic Regression Classification
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
```

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

```
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LogisticRegression()
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 11.1: Example of the logistic regression algorithm.

Running the example prints the mean estimated accuracy.

```
0.76951469583
```

Listing 11.2: Output of the logistic regression algorithm.

11.3.2 Linear Discriminant Analysis

Linear Discriminant Analysis or LDA is a statistical technique for binary and multiclass classification. It too assumes a Gaussian distribution for the numerical input variables. You can construct an LDA model using the `LinearDiscriminantAnalysis` class[2].

```
# LDA Classification
import pandas
from sklearn import cross_validation
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LinearDiscriminantAnalysis()
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 11.3: Example of the LDA algorithm.

Running the example prints the mean estimated accuracy.

```
0.773462064252
```

Listing 11.4: Output of the LDA algorithm.

11.4 Nonlinear Machine Learning Algorithms

This section demonstrates minimal recipes for how to use 4 nonlinear machine learning algorithms.

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

11.4.1 K-Nearest Neighbors

The k-Nearest Neighbors algorithm (or KNN) uses a distance metric to find the k most similar instances in the training data for a new instance and takes the mean outcome of the neighbors as the prediction. You can construct a KNN model using the `KNeighborsClassifier` class[3].

```
# KNN Classification
import pandas
from sklearn import cross_validation
from sklearn.neighbors import KNeighborsClassifier
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
random_state = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds,
    random_state=random_state)
model = KNeighborsClassifier()
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 11.5: Example of the KNN algorithm.

Running the example prints the mean estimated accuracy.

```
0.726555023923
```

Listing 11.6: Output of the KNN algorithm.

11.4.2 Naive Bayes

Naive Bayes calculates the probability of each class and the conditional probability of each class given each input value. These probabilities are estimated for new data and multiplied together, assuming that they are all independent (a simple or naive assumption). When working with real-valued data, a Gaussian distribution is assumed to easily estimate the probabilities for input variables using the Gaussian Probability Density Function. You can construct a Naive Bayes model using the `GaussianNB` class[4].

```
# Gaussian Naive Bayes Classification
import pandas
from sklearn import cross_validation
from sklearn.naive_bayes import GaussianNB
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
```

[3] http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

[4] http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html

```
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = GaussianNB()
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 11.7: Example of the Naive Bayes algorithm.

Running the example prints the mean estimated accuracy.

```
0.75517771702
```

Listing 11.8: Output of the Naive Bayes algorithm.

11.4.3 Classification and Regression Trees

Classification and Regression Trees (CART or just decision trees) construct a binary tree from the training data. Split points are chosen greedily by evaluating each attribute and each value of each attribute in the training data in order to minimize a cost function (like the Gini index). You can construct a CART model using the `DecisionTreeClassifier` class[5].

```
# CART Classification
import pandas
from sklearn import cross_validation
from sklearn.tree import DecisionTreeClassifier
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = DecisionTreeClassifier()
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 11.9: Example of the CART algorithm.

Running the example prints the mean estimated accuracy.

```
0.692600820232
```

Listing 11.10: Output of the CART algorithm.

[5] http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

11.4.4 Support Vector Machines

Support Vector Machines (or SVM) seek a line that best separates two classes. Those data instances that are closest to the line that best separates the classes are called support vectors and influence where the line is placed. SVM has been extended to support multiple classes. Of particular importance is the use of different kernel functions via the kernel parameter. A powerful Radial Basis Function is used by default. You can construct an SVM model using the `SVC` class[6].

```
# SVM Classification
import pandas
from sklearn import cross_validation
from sklearn.svm import SVC
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = SVC()
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 11.11: Example of the SVM algorithm.

Running the example prints the mean estimated accuracy.

```
0.651025290499
```

Listing 11.12: Output of the SVM algorithm.

11.5 Summary

In this chapter you discovered 6 machine learning algorithms that you can use to spot-check on your classification problem in Python using scikit-learn. Specifically, you learned how to spot-check two linear machine learning algorithms: Logistic Regression and Linear Discriminant Analysis. You also learned how to spot-check four nonlinear algorithms: k-Nearest Neighbors, Naive Bayes, Classification and Regression Trees and Support Vector Machines.

11.5.1 Next

In the next lesson you will discover how you can use spot-checking on regression machine learning problems and practice with seven different regression algorithms.

[6] http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Chapter 12

Spot-Check Regression Algorithms

Spot-checking is a way of discovering which algorithms perform well on your machine learning problem. You cannot know which algorithms are best suited to your problem beforehand. You must trial a number of methods and focus attention on those that prove themselves the most promising. In this chapter you will discover six machine learning algorithms that you can use when spot-checking your regression problem in Python with scikit-learn. After completing this lesson you will know:

1. How to spot-check machine learning algorithms on a regression problem.
2. How to spot-check four linear regression algorithms.
3. How to spot-check three nonlinear regression algorithms.

Let's get started.

12.1 Algorithms Overview

In this lesson we are going to take a look at seven regression algorithms that you can spot-check on your dataset. Starting with four linear machine learning algorithms:

- Linear Regression.
- Ridge Regression.
- LASSO Linear Regression.
- Elastic Net Regression.

Then looking at three nonlinear machine learning algorithms:

- k-Nearest Neighbors.
- Classification and Regression Trees.
- Support Vector Machines.

Each recipe is demonstrated on the Boston House Price dataset. This is a regression problem where all attributes are numeric. A test harness with 10-fold cross validation is used to demonstrate how to spot-check each machine learning algorithm and mean squared error measures are used to indicate algorithm performance. Note that mean squared error values are inverted (negative). This is a quirk of the `cross_val_score()` function used that requires all algorithm metrics to be sorted in ascending order (larger value is better). The recipes assume that you know about each machine learning algorithm and how to use them. We will not go into the API or parameterization of each algorithm.

12.2 Linear Machine Learning Algorithms

This section provides examples of how to use four different linear machine learning algorithms for regression in Python with scikit-learn.

12.2.1 Linear Regression

Linear regression assumes that the input variables have a Gaussian distribution. It is also assumed that input variables are relevant to the output variable and that they are not highly correlated with each other (a problem called collinearity). You can construct a linear regression model using the `LinearRegression` class[1].

```
# Linear Regression
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LinearRegression
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = LinearRegression()
scoring = 'mean_squared_error'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print(results.mean())
```

Listing 12.1: Example of the linear regression algorithm.

Running the example provides a estimate of mean squared error.

```
-34.7052559445
```

Listing 12.2: Output of the linear regression algorithm.

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

12.2.2 Ridge Regression

Ridge regression is an extension of linear regression where the loss function is modified to minimize the complexity of the model measured as the sum squared value of the coefficient values (also called the L2-norm). You can construct a ridge regression model by using the `Ridge` class[2].

```
# Ridge Regression
import pandas
from sklearn import cross_validation
from sklearn.linear_model import Ridge
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = Ridge()
scoring = 'mean_squared_error'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print(results.mean())
```

Listing 12.3: Example of the ridge regression algorithm.

Running the example provides an estimate of the mean squared error.

```
-34.0782462093
```

Listing 12.4: Output of the ridge regression algorithm.

12.2.3 LASSO Regression

The Least Absolute Shrinkage and Selection Operator (or LASSO for short) is a modification of linear regression, like ridge regression, where the loss function is modified to minimize the complexity of the model measured as the sum absolute value of the coefficient values (also called the L1-norm). You can construct a LASSO model by using the `Lasso` class[3].

```
# Lasso Regression
import pandas
from sklearn import cross_validation
from sklearn.linear_model import Lasso
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
```

[2] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html
[3] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

```
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = Lasso()
scoring = 'mean_squared_error'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print(results.mean())
```

Listing 12.5: Example of the LASSO regression algorithm.

Running the example provides an estimate of the mean squared error.

```
-34.4640845883
```

Listing 12.6: Output of the LASSO regression algorithm.

12.2.4 ElasticNet Regression

ElasticNet is a form of regularization regression that combines the properties of both Ridge Regression and LASSO regression. It seeks to minimize the complexity of the regression model (magnitude and number of regression coefficients) by penalizing the model using both the L2-norm (sum squared coefficient values) and the L1-norm (sum absolute coefficient values). You can construct an ElasticNet model using the `ElasticNet` class[4].

```
# ElasticNet Regression
import pandas
from sklearn import cross_validation
from sklearn.linear_model import ElasticNet
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = ElasticNet()
scoring = 'mean_squared_error'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print(results.mean())
```

Listing 12.7: Example of the ElasticNet regression algorithm.

Running the example provides an estimate of the mean squared error.

```
-31.1645737142
```

Listing 12.8: Output of the ElasticNet regression algorithm.

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

12.3 Nonlinear Machine Learning Algorithms

This section provides examples of how to use three different nonlinear machine learning algorithms for regression in Python with scikit-learn.

12.3.1 K-Nearest Neighbors

The k-Nearest Neighbors algorithm (or KNN) locates the k most similar instances in the training dataset for a new data instance. From the k neighbors, a mean or median output variable is taken as the prediction. Of note is the distance metric used (the `metric` argument). The Minkowski distance is used by default, which is a generalization of both the Euclidean distance (used when all inputs have the same scale) and Manhattan distance (for when the scales of the input variables differ). You can construct a KNN model for regression using the `KNeighborsRegressor` class[5].

```
# KNN Regression
import pandas
from sklearn import cross_validation
from sklearn.neighbors import KNeighborsRegressor
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = KNeighborsRegressor()
scoring = 'mean_squared_error'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print(results.mean())
```

Listing 12.9: Example of the KNN regression algorithm.

Running the example provides an estimate of the mean squared error.

```
-107.28683898
```

Listing 12.10: Output of the KNN regression algorithm.

12.3.2 Classification and Regression Trees

Decision trees or the Classification and Regression Trees (CART as they are known) use the training data to select the best points to split the data in order to minimize a cost metric. The default cost metric for regression decision trees is the mean squared error, specified in the `criterion` parameter. You can create a CART model for regression using the `DecisionTreeRegressor` class[6].

[5] http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

[6] http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

```
# Decision Tree Regression
import pandas
from sklearn import cross_validation
from sklearn.tree import DecisionTreeRegressor
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = DecisionTreeRegressor()
scoring = 'mean_squared_error'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print(results.mean())
```

Listing 12.11: Example of the CART regression algorithm.

Running the example provides an estimate of the mean squared error.

```
-35.4906027451
```

Listing 12.12: Output of the CART regression algorithm.

12.3.3 Support Vector Machines

Support Vector Machines (SVM) were developed for binary classification. The technique has been extended for the prediction real-valued problems called Support Vector Regression (SVR). Like the classification example, SVR is built upon the LIBSVM library. You can create an SVM model for regression using the `SVR` class[7].

```
# SVM Regression
import pandas
from sklearn import cross_validation
from sklearn.svm import SVR
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataframe = pandas.read_csv(url, delim_whitespace=True, names=names)
array = dataframe.values
X = array[:,0:13]
Y = array[:,13]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = SVR()
scoring = 'mean_squared_error'
results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
```

[7] http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

```
print(results.mean())
```

Listing 12.13: Example of the SVM regression algorithm.

Running the example provides an estimate of the mean squared error.

```
-91.0478243332
```

Listing 12.14: Output of the SVM regression algorithm.

12.4 Summary

In this chapter you discovered how to spot-check machine learning algorithms for regression problems in Python using scikit-learn. Specifically, you learned about four linear machine learning algorithms: Linear Regression, Ridge Regression, LASSO Linear Regression and Elastic Net Regression. You also learned about three nonlinear algorithms: k-Nearest Neighbors, Classification and Regression Trees and Support Vector Machines.

12.4.1 Next

Now that you know how to use classification and regression algorithms you need to know how to compare the results of different algorithms to each other. In the next lesson you will discover how to design simple experiments to directly compare machine learning algorithms to each other on your dataset.

Chapter 13

Compare Machine Learning Algorithms

It is important to compare the performance of multiple different machine learning algorithms consistently. In this chapter you will discover how you can create a test harness to compare multiple different machine learning algorithms in Python with scikit-learn. You can use this test harness as a template on your own machine learning problems and add more and different algorithms to compare. After completing this lesson you will know:

1. How to formulate an experiment to directly compare machine learning algorithms.
2. A reusable template for evaluating the performance of multiple algorithms on one dataset.
3. How to report and visualize the results when comparing algorithm performance.

Let's get started.

13.1 Choose The Best Machine Learning Model

When you work on a machine learning project, you often end up with multiple good models to choose from. Each model will have different performance characteristics. Using resampling methods like cross validation, you can get an estimate for how accurate each model may be on unseen data. You need to be able to use these estimates to choose one or two best models from the suite of models that you have created.

When you have a new dataset, it is a good idea to visualize the data using different techniques in order to look at the data from different perspectives. The same idea applies to model selection. You should use a number of different ways of looking at the estimated accuracy of your machine learning algorithms in order to choose the one or two algorithm to finalize. A way to do this is to use visualization methods to show the average accuracy, variance and other properties of the distribution of model accuracies. In the next section you will discover exactly how you can do that in Python with scikit-learn.

13.2 Compare Machine Learning Algorithms Consistently

The key to a fair comparison of machine learning algorithms is ensuring that each algorithm is evaluated in the same way on the same data. You can achieve this by forcing each algorithm

to be evaluated on a consistent test harness. In the example below six different classification algorithms are compared on a single dataset:

- Logistic Regression.
- Linear Discriminant Analysis.
- k-Nearest Neighbors.
- Classification and Regression Trees.
- Naive Bayes.
- Support Vector Machines.

The dataset is the Pima Indians onset of diabetes problem. The problem has two classes and eight numeric input variables of varying scales. The 10-fold cross validation procedure is used to evaluate each algorithm, importantly configured with the same random seed to ensure that the same splits to the training data are performed and that each algorithm is evaluated in precisely the same way. Each algorithm is given a short name, useful for summarizing results afterward.

```
# Compare Algorithms
import pandas
import matplotlib.pyplot as plt
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
# load dataset
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# prepare configuration for cross validation test harness
num_folds = 10
num_instances = len(X)
seed = 7
# prepare models
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
# evaluate each model in turn
results = []
names = []
scoring = 'accuracy'
for name, model in models:
```

```
  kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
  cv_results = cross_validation.cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
  results.append(cv_results)
  names.append(name)
  msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
  print(msg)
# boxplot algorithm comparison
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Listing 13.1: Example of comparing multiple algorithms.

Running the example provides a list of each algorithm short name, the mean accuracy and the standard deviation accuracy.

```
LR: 0.769515 (0.048411)
LDA: 0.773462 (0.051592)
KNN: 0.726555 (0.061821)
CART: 0.695232 (0.062517)
NB: 0.755178 (0.042766)
SVM: 0.651025 (0.072141)
```

Listing 13.2: Output of comparing multiple algorithms.

The example also provides a box and whisker plot showing the spread of the accuracy scores across each cross validation fold for each algorithm.

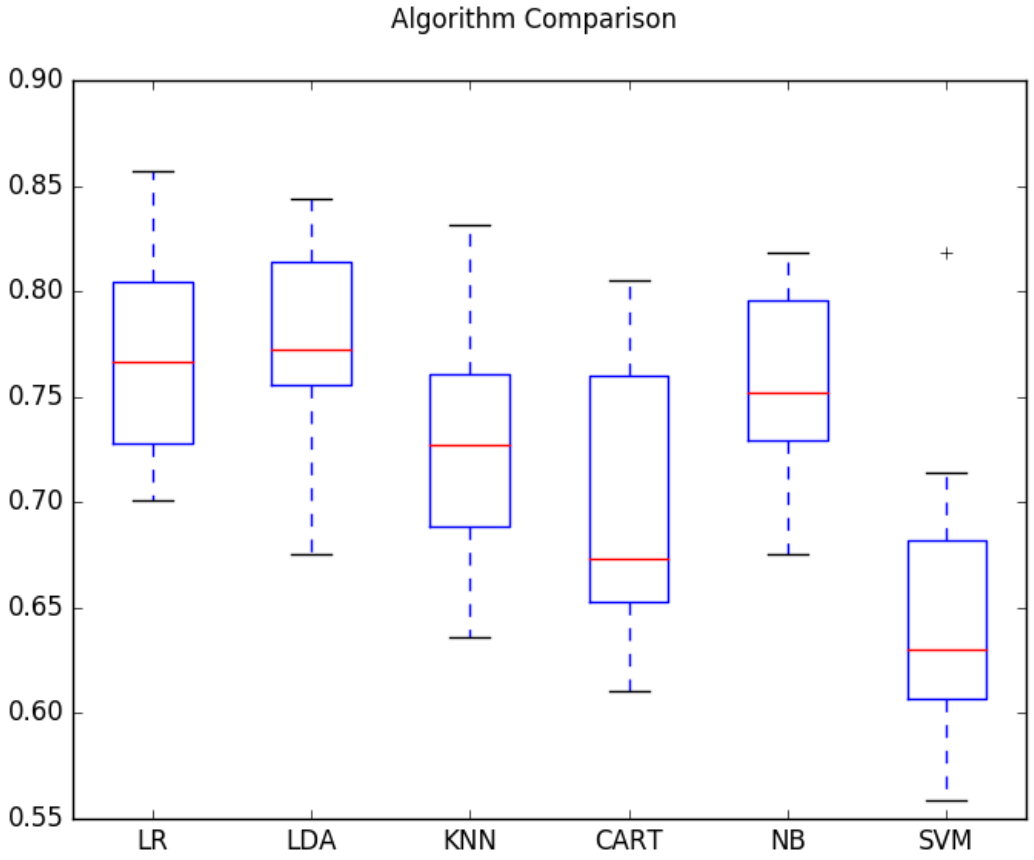

Figure 13.1: Box and Whisker Plots Comparing Algorithm Performance

From these results, it would suggest that both logistic regression and linear discriminant analysis are perhaps worthy of further study on this problem.

13.3 Summary

In this chapter you discovered how to evaluate multiple different machine learning algorithms on a dataset in Python with scikit-learn. You learned how to both use the same test harness to evaluate the algorithms and how to summarize the results both numerically and using a box and whisker plot. You can use this recipe as a template for evaluating multiple algorithms on your own problems.

13.3.1 Next

In this lesson you learned how to compare the performance of machine learning algorithms to each other. But what if you need to prepare your data as part of the comparison process. In the next lesson you will discover Pipelines in scikit-learn and how they overcome the common problems of data leakage when comparing machine learning algorithms.

Chapter 14

Automate Machine Learning Workflows with Pipelines

There are standard workflows in a machine learning project that can be automated. In Python scikit-learn, Pipelines help to clearly define and automate these workflows. In this chapter you will discover Pipelines in scikit-learn and how you can automate common machine learning workflows. After completing this lesson you will know:

1. How to use pipelines to minimize data leakage.
2. How to construct a data preparation and modeling pipeline.
3. How to construct a feature extraction and modeling pipeline.

Let's get started.

14.1 Automating Machine Learning Workflows

There are standard workflows in applied machine learning. Standard because they overcome common problems like data leakage in your test harness. Python scikit-learn provides a Pipeline utility to help automate machine learning workflows. Pipelines work by allowing for a linear sequence of data transforms to be chained together culminating in a modeling process that can be evaluated.

The goal is to ensure that all of the steps in the pipeline are constrained to the data available for the evaluation, such as the training dataset or each fold of the cross validation procedure. You can learn more about Pipelines in scikit-learn by reading the Pipeline section[1] of the user guide. You can also review the API documentation for the Pipeline and FeatureUnion classes and the `pipeline` module[2].

14.2 Data Preparation and Modeling Pipeline

An easy trap to fall into in applied machine learning is leaking data from your training dataset to your test dataset. To avoid this trap you need a robust test harness with strong separation of

[1] http://scikit-learn.org/stable/modules/pipeline.html
[2] http://scikit-learn.org/stable/modules/classes.html#module-sklearn.pipeline

training and testing. This includes data preparation. Data preparation is one easy way to leak knowledge of the whole training dataset to the algorithm. For example, preparing your data using normalization or standardization on the entire training dataset before learning would not be a valid test because the training dataset would have been influenced by the scale of the data in the test set.

Pipelines help you prevent data leakage in your test harness by ensuring that data preparation like standardization is constrained to each fold of your cross validation procedure. The example below demonstrates this important data preparation and model evaluation workflow on the Pima Indians onset of diabetes dataset. The pipeline is defined with two steps:

1. Standardize the data.
2. Learn a Linear Discriminant Analysis model.

The pipeline is then evaluated using 10-fold cross validation.

```
# Create a pipeline that standardizes the data then creates a model
from pandas import read_csv
from sklearn.cross_validation import KFold
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
# load data
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# create pipeline
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('lda', LinearDiscriminantAnalysis()))
model = Pipeline(estimators)
# evaluate pipeline
num_folds = 10
num_instances = len(X)
seed = 7
kfold = KFold(n=num_instances, n_folds=num_folds, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 14.1: Example of a Pipeline to standardize and model data.

Notice how we create a Python list of steps that are provided to the Pipeline for process the data. Also notice how the Pipeline itself is treated like an estimator and is evaluated in its entirety by the k-fold cross validation procedure. Running the example provides a summary of accuracy of the setup on the dataset.

```
0.773462064252
```

Listing 14.2: Output of a Pipeline to standardize and model data.

14.3 Feature Extraction and Modeling Pipeline

Feature extraction is another procedure that is susceptible to data leakage. Like data preparation, feature extraction procedures must be restricted to the data in your training dataset. The pipeline provides a handy tool called the `FeatureUnion` which allows the results of multiple feature selection and extraction procedures to be combined into a larger dataset on which a model can be trained. Importantly, all the feature extraction and the feature union occurs within each fold of the cross validation procedure. The example below demonstrates the pipeline defined with four steps:

1. Feature Extraction with Principal Component Analysis (3 features).
2. Feature Extraction with Statistical Selection (6 features).
3. Feature Union.
4. Learn a Logistic Regression Model.

The pipeline is then evaluated using 10-fold cross validation.

```
# Create a pipeline that extracts features from the data then creates a model
from pandas import read_csv
from sklearn.cross_validation import KFold
from sklearn.cross_validation import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest
# load data
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
# create feature union
features = []
features.append(('pca', PCA(n_components=3)))
features.append(('select_best', SelectKBest(k=6)))
feature_union = FeatureUnion(features)
# create pipeline
estimators = []
estimators.append(('feature_union', feature_union))
estimators.append(('logistic', LogisticRegression()))
model = Pipeline(estimators)
# evaluate pipeline
num_folds = 10
num_instances = len(X)
seed = 7
kfold = KFold(n=num_instances, n_folds=num_folds, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 14.3: Example of a Pipeline extract and combine features before modeling.

Notice how the FeatureUnion is it's own Pipeline that in turn is a single step in the final Pipeline used to feed Logistic Regression. This might get you thinking about how you can start embedding pipelines within pipelines. Running the example provides a summary of accuracy of the setup on the dataset.

```
0.776042378674
```

Listing 14.4: Output of a Pipeline extract and combine features before modeling.

14.4 Summary

In this chapter you discovered the difficulties of data leakage in applied machine learning. You discovered the Pipeline utilities in Python scikit-learn and how they can be used to automate standard applied machine learning workflows. You learned how to use Pipelines in two important use cases:

- Data preparation and modeling constrained to each fold of the cross validation procedure.
- Feature extraction and feature union constrained to each fold of the cross validation procedure.

14.4.1 Next

This completes the lessons on how to evaluate machine learning algorithms. In the next lesson you will take your first look at how to improve algorithm performance on your problems by using ensemble methods.

Chapter 15

Improve Performance with Ensembles

Ensembles can give you a boost in accuracy on your dataset. In this chapter you will discover how you can create some of the most powerful types of ensembles in Python using scikit-learn. This lesson will step you through Boosting, Bagging and Majority Voting and show you how you can continue to ratchet up the accuracy of the models on your own datasets. After completing this lesson you will know:

1. How to use bagging ensemble methods such as bagged decision trees, random forest and extra trees.
2. How to use boosting ensemble methods such as AdaBoost and stochastic gradient boosting.
3. How to use voting ensemble methods to combine the predictions from multiple algorithms.

Let's get started.

15.1 Combine Models Into Ensemble Predictions

The three most popular methods for combining the predictions from different models are:

- **Bagging**. Building multiple models (typically of the same type) from different subsamples of the training dataset.
- **Boosting**. Building multiple models (typically of the same type) each of which learns to fix the prediction errors of a prior model in the sequence of models.
- **Voting**. Building multiple models (typically of differing types) and simple statistics (like calculating the mean) are used to combine predictions.

This assumes you are generally familiar with machine learning algorithms and ensemble methods and will not go into the details of how the algorithms work or their parameters. The Pima Indians onset of Diabetes dataset is used to demonstrate each algorithm. Each ensemble algorithm is demonstrated using 10-fold cross validation and the classification accuracy performance metric.

15.2 Bagging Algorithms

Bootstrap Aggregation (or Bagging) involves taking multiple samples from your training dataset (with replacement) and training a model for each sample. The final output prediction is averaged across the predictions of all of the sub-models. The three bagging models covered in this section are as follows:

- Bagged Decision Trees.
- Random Forest.
- Extra Trees.

15.2.1 Bagged Decision Trees

Bagging performs best with algorithms that have high variance. A popular example are decision trees, often constructed without pruning. In the example below is an example of using the `BaggingClassifier` with the Classification and Regression Trees algorithm (`DecisionTreeClassifier`[1]). A total of 100 trees are created.

```
# Bagged Decision Trees for Classification
import pandas
from sklearn import cross_validation
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
cart = DecisionTreeClassifier()
num_trees = 100
model = BaggingClassifier(base_estimator=cart, n_estimators=num_trees, random_state=seed)
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 15.1: Example of Bagged Decision Trees Ensemble Algorithm.

Running the example, we get a robust estimate of model accuracy.

```
0.770745044429
```

Listing 15.2: Output of Bagged Decision Trees Ensemble Algorithm.

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

15.2.2 Random Forest

Random Forests is an extension of bagged decision trees. Samples of the training dataset are taken with replacement, but the trees are constructed in a way that reduces the correlation between individual classifiers. Specifically, rather than greedily choosing the best split point in the construction of each tree, only a random subset of features are considered for each split. You can construct a Random Forest model for classification using the `RandomForestClassifier` class[2]. The example below demonstrates using Random Forest for classification with 100 trees and split points chosen from a random selection of 3 features.

```
# Random Forest Classification
import pandas
from sklearn import cross_validation
from sklearn.ensemble import RandomForestClassifier
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
num_trees = 100
max_features = 3
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = RandomForestClassifier(n_estimators=num_trees, max_features=max_features)
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 15.3: Example of Random Forest Ensemble Algorithm.

Running the example provides a mean estimate of classification accuracy.

```
0.770727956254
```

Listing 15.4: Output of Random Forest Ensemble Algorithm.

15.2.3 Extra Trees

Extra Trees are another modification of bagging where random trees are constructed from samples of the training dataset. You can construct an Extra Trees model for classification using the `ExtraTreesClassifier` class[3]. The example below provides a demonstration of extra trees with the number of trees set to 100 and splits chosen from 7 random features.

```
# Extra Trees Classification
import pandas
from sklearn import cross_validation
from sklearn.ensemble import ExtraTreesClassifier
url = "https://goo.gl/vhm1eU"
```

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

[3] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

```
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
num_trees = 100
max_features = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = ExtraTreesClassifier(n_estimators=num_trees, max_features=max_features)
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 15.5: Example of Extra Trees Ensemble Algorithm.

Running the example provides a mean estimate of classification accuracy.

```
0.760269993165
```

Listing 15.6: Output of Extra Trees Ensemble Algorithm.

15.3 Boosting Algorithms

Boosting ensemble algorithms creates a sequence of models that attempt to correct the mistakes of the models before them in the sequence. Once created, the models make predictions which may be weighted by their demonstrated accuracy and the results are combined to create a final output prediction. The two most common boosting ensemble machine learning algorithms are:

- AdaBoost.
- Stochastic Gradient Boosting.

15.3.1 AdaBoost

AdaBoost was perhaps the first successful boosting ensemble algorithm. It generally works by weighting instances in the dataset by how easy or difficult they are to classify, allowing the algorithm to pay or less attention to them in the construction of subsequent models. You can construct an AdaBoost model for classification using the `AdaBoostClassifier` class[4]. The example below demonstrates the construction of 30 decision trees in sequence using the AdaBoost algorithm.

```
# AdaBoost Classification
import pandas
from sklearn import cross_validation
from sklearn.ensemble import AdaBoostClassifier
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
```

[4] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

```
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
num_trees = 30
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = AdaBoostClassifier(n_estimators=num_trees, random_state=seed)
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 15.7: Example of AdaBoost Ensemble Algorithm.

Running the example provides a mean estimate of classification accuracy.

```
0.76045796309
```

Listing 15.8: Output of AdaBoost Ensemble Algorithm.

15.3.2 Stochastic Gradient Boosting

Stochastic Gradient Boosting (also called Gradient Boosting Machines) are one of the most sophisticated ensemble techniques. It is also a technique that is proving to be perhaps one of the best techniques available for improving performance via ensembles. You can construct a Gradient Boosting model for classification using the `GradientBoostingClassifier` class[5]. The example below demonstrates Stochastic Gradient Boosting for classification with 100 trees.

```
# Stochastic Gradient Boosting Classification
import pandas
from sklearn import cross_validation
from sklearn.ensemble import GradientBoostingClassifier
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
num_trees = 100
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
model = GradientBoostingClassifier(n_estimators=num_trees, random_state=seed)
results = cross_validation.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 15.9: Example of Stochastic Gradient Boosting Ensemble Algorithm.

Running the example provides a mean estimate of classification accuracy.

```
0.764285714286
```

Listing 15.10: Output of Stochastic Gradient Boosting Ensemble Algorithm.

[5] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

15.4 Voting Ensemble

Voting is one of the simplest ways of combining the predictions from multiple machine learning algorithms. It works by first creating two or more standalone models from your training dataset. A Voting Classifier can then be used to wrap your models and average the predictions of the sub-models when asked to make predictions for new data. The predictions of the sub-models can be weighted, but specifying the weights for classifiers manually or even heuristically is difficult. More advanced methods can learn how to best weight the predictions from sub-models, but this is called stacking (stacked aggregation) and is currently not provided in scikit-learn.

You can create a voting ensemble model for classification using the `VotingClassifier` class[6]. The code below provides an example of combining the predictions of logistic regression, classification and regression trees and support vector machines together for a classification problem.

```
# Voting Ensemble for Classification
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
num_folds = 10
num_instances = len(X)
seed = 7
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
# create the sub models
estimators = []
model1 = LogisticRegression()
estimators.append(('logistic', model1))
model2 = DecisionTreeClassifier()
estimators.append(('cart', model2))
model3 = SVC()
estimators.append(('svm', model3))
# create the ensemble model
ensemble = VotingClassifier(estimators)
results = cross_validation.cross_val_score(ensemble, X, Y, cv=kfold)
print(results.mean())
```

Listing 15.11: Example of the Voting Ensemble Algorithm.

Running the example provides a mean estimate of classification accuracy.

```
0.729049897471
```

Listing 15.12: Output of the Voting Ensemble Algorithm.

[6] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html

15.5 Summary

In this chapter you discovered ensemble machine learning algorithms for improving the performance of models on your problems. You learned about:

- Bagging Ensembles including Bagged Decision Trees, Random Forest and Extra Trees.
- Boosting Ensembles including AdaBoost and Stochastic Gradient Boosting.
- Voting Ensembles for averaging the predictions for any arbitrary models.

15.5.1 Next

In the next section you will discover another technique that you can use to improve the performance of algorithms on your dataset called algorithm tuning.

Chapter 16

Improve Performance with Algorithm Tuning

Machine learning models are parameterized so that their behavior can be tuned for a given problem. Models can have many parameters and finding the best combination of parameters can be treated as a search problem. In this chapter you will discover how to tune the parameters of machine learning algorithms in Python using the scikit-learn. After completing this lesson you will know:

1. The importance of algorithm parameter tuning to improve algorithm performance.
2. How to use a grid search algorithm tuning strategy.
3. How to use a random search algorithm tuning strategy.

Let's get started.

16.1 Machine Learning Algorithm Parameters

Algorithm tuning is a final step in the process of applied machine learning before finalizing your model. It is sometimes called hyperparameter optimization where the algorithm parameters are referred to as hyperparameters, whereas the coefficients found by the machine learning algorithm itself are referred to as parameters. Optimization suggests the search-nature of the problem. Phrased as a search problem, you can use different search strategies to find a good and robust parameter or set of parameters for an algorithm on a given problem. Python scikit-learn provides two simple methods for algorithm parameter tuning:

- Grid Search Parameter Tuning.
- Random Search Parameter Tuning.

16.2 Grid Search Parameter Tuning

Grid search is an approach to parameter tuning that will methodically build and evaluate a model for each combination of algorithm parameters specified in a grid. You can perform a grid

search using the `GridSearchCV` class[1]. The example below evaluates different `alpha` values for the Ridge Regression algorithm on the standard diabetes dataset. This is a one-dimensional grid search.

```
# Grid Search for Algorithm Tuning
import pandas
import numpy
from sklearn import cross_validation
from sklearn.linear_model import Ridge
from sklearn.grid_search import GridSearchCV
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
alphas = numpy.array([1,0.1,0.01,0.001,0.0001,0])
param_grid = dict(alpha=alphas)
model = Ridge()
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid.fit(X, Y)
print(grid.best_score_)
print(grid.best_estimator_.alpha)
```

Listing 16.1: Example of a grid search for algorithm parameters.

Running the example lists out the optimal score achieved and the set of parameters in the grid that achieved that score. In this case the `alpha` value of 1.0.

```
0.279617559313
1.0
```

Listing 16.2: Output of a grid search for algorithm parameters.

16.3 Random Search Parameter Tuning

Random search is an approach to parameter tuning that will sample algorithm parameters from a random distribution (i.e. uniform) for a fixed number of iterations. A model is constructed and evaluated for each combination of parameters chosen. You can perform a random search for algorithm parameters using the `RandomizedSearchCV` class[2]. The example below evaluates different random `alpha` values between 0 and 1 for the Ridge Regression algorithm on the standard diabetes dataset. A total of 100 iterations are performed with uniformly random `alpha` values selected in the range between 0 and 1 (the range that `alpha` values can take).

```
# Randomized for Algorithm Tuning
import pandas
import numpy
from scipy.stats import uniform
from sklearn.linear_model import Ridge
from sklearn.grid_search import RandomizedSearchCV
```

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

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

```
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
param_grid = {'alpha': uniform()}
seed = 7
model = Ridge()
iterations = 100
rsearch = RandomizedSearchCV(estimator=model, param_distributions=param_grid,
    n_iter=iterations, random_state=seed)
rsearch.fit(X, Y)
print(rsearch.best_score_)
print(rsearch.best_estimator_.alpha)
```

Listing 16.3: Example of a random search for algorithm parameters.

Running the example produces results much like those in the grid search example above. An optimal `alpha` value near 1.0 is discovered.

```
0.279617354112
0.989527376274
```

Listing 16.4: Output of a random search for algorithm parameters.

16.4 Summary

Algorithm parameter tuning is an important step for improving algorithm performance right before presenting results or preparing a system for production. In this chapter you discovered algorithm parameter tuning and two methods that you can use right now in Python and scikit-learn to improve your algorithm results:

- Grid Search Parameter Tuning
- Random Search Parameter Tuning

16.4.1 Next

This lesson concludes the coverage of techniques that you can use to improve the performance of algorithms on your dataset. In the next and final lesson you will discover how you can finalize your model for using it on unseen data.

Chapter 17

Save and Load Machine Learning Models

Finding an accurate machine learning model is not the end of the project. In this chapter you will discover how to save and load your machine learning model in Python using scikit-learn. This allows you to save your model to file and load it later in order to make predictions. After completing this lesson you will know:

1. The importance of serializing models for reuse.
2. How to use `pickle` to serialize and deserialize machine learning models.
3. How to use `Joblib` to serialize and deserialize machine learning models.

Let's get started.

17.1 Finalize Your Model with pickle

Pickle is the standard way of serializing objects in Python. You can use the `pickle`[1] operation to serialize your machine learning algorithms and save the serialized format to a file. Later you can load this file to deserialize your model and use it to make new predictions. The example below demonstrates how you can train a logistic regression model on the Pima Indians onset of diabetes dataset, save the model to file and load it to make predictions on the unseen test set.

```
# Save Model Using Pickle
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
import pickle
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
test_size = 0.33
```

[1] https://docs.python.org/2/library/pickle.html

```
seed = 7
X_train, X_test, Y_train, Y_test = cross_validation.train_test_split(X, Y,
    test_size=test_size, random_state=seed)
# Fit the model on 33%
model = LogisticRegression()
model.fit(X_train, Y_train)
# save the model to disk
filename = 'finalized_model.sav'
pickle.dump(model, open(filename, 'wb'))

# some time later...

# load the model from disk
loaded_model = pickle.load(open(filename, 'rb'))
result = loaded_model.score(X_test, Y_test)
print(result)
```

Listing 17.1: Example of using pickle to serialize and deserialize a model.

Running the example saves the model to `finalized_model.sav` in your local working directory. Load the saved model and evaluating it provides an estimate of accuracy of the model on unseen data.

```
0.755905511811
```

Listing 17.2: Output of using pickle to serialize and deserialize a model.

17.2 Finalize Your Model with Joblib

The `Joblib`[2] library is part of the SciPy ecosystem and provides utilities for pipelining Python jobs. It provides utilities for saving and loading Python objects that make use of NumPy data structures, efficiently[3]. This can be useful for some machine learning algorithms that require a lot of parameters or store the entire dataset (e.g. k-Nearest Neighbors). The example below demonstrates how you can train a logistic regression model on the Pima Indians onset of diabetes dataset, save the model to file using Joblib and load it to make predictions on the unseen test set.

```
# Save Model Using Joblib
import pandas
from sklearn import cross_validation
from sklearn.linear_model import LogisticRegression
from sklearn.externals import joblib
url = "https://goo.gl/vhm1eU"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
test_size = 0.33
seed = 7
```

[2] https://pypi.python.org/pypi/joblib
[3] https://pythonhosted.org/joblib/generated/joblib.dump.html

```
X_train, X_test, Y_train, Y_test = cross_validation.train_test_split(X, Y,
    test_size=test_size, random_state=seed)
# Fit the model on 33%
model = LogisticRegression()
model.fit(X_train, Y_train)
# save the model to disk
filename = 'finalized_model.sav'
joblib.dump(model, filename)

# some time later...

# load the model from disk
loaded_model = joblib.load(filename)
result = loaded_model.score(X_test, Y_test)
print(result)
```

Listing 17.3: Example of using Joblib to serialize and deserialize a model.

Running the example saves the model to file as `finalized_model.sav` and also creates one file for each NumPy array in the model (four additional files). After the model is loaded an estimate of the accuracy of the model on unseen data is reported.

```
0.755905511811
```

Listing 17.4: Output of using Joblib to serialize and deserialize a model.

17.3 Tips for Finalizing Your Model

This section lists some important considerations when finalizing your machine learning models.

- **Python Version**. Take note of the Python version. You almost certainly require the same major (and maybe minor) version of Python used to serialize the model when you later load it and deserialize it.
- **Library Versions**. The version of all major libraries used in your machine learning project almost certainly need to be the same when deserializing a saved model. This is not limited to the version of NumPy and the version of scikit-learn.
- **Manual Serialization**. You might like to manually output the parameters of your learned model so that you can use them directly in scikit-learn or another platform in the future. Often the techniques used internally by machine learning algorithms to make predictions are a lot simpler than those used to learn the parameters can may be easy to implement in custom code that you have control over.

Take note of the version so that you can re-create the environment if for some reason you cannot reload your model on another machine or another platform at a later time.

17.4 Summary

In this chapter you discovered how to persist your machine learning algorithms in Python with scikit-learn. You learned two techniques that you can use:

- The `pickle` API for serializing standard Python objects.
- The `Joblib` API for efficiently serializing Python objects with NumPy arrays.

17.4.1 Next

This concludes your lessons on machine learning in Python with SciPy and scikit-learn. Next in Part III you will tie together everything you have learned and work through end-to-end applied machine learning projects.

Part III

Projects

Chapter 18

Predictive Modeling Project Template

Applied machine learning is an empirical skill. You cannot get better at it by reading books and articles. You have to practice. In this lesson you will discover the simple six-step machine learning project template that you can use to jump-start your project in Python. After completing this lesson you will know:

1. How to structure an end-to-end predictive modeling project.
2. How to map the tasks you learned about in Part II onto a project.
3. How to best use the structured project template to ensure an accurate result for your dataset.

Let's get started.

18.1 Practice Machine Learning With Projects

Working through machine learning problems from end-to-end is critically important. You can read about machine learning. You can also try out small one-off recipes. But applied machine learning will not come alive for you until you work through a dataset from beginning to end.

Working through a project forces you to think about how the model will be used, to challenge your assumptions and to get good at all parts of a project, not just your favorite parts. The best way to practice predictive modeling machine learning projects is to use standardized datasets from the UCI Machine Learning Repository. Once you have a practice dataset and a bunch of Python recipes, how do you put it all together and work through the problem end-to-end?

18.1.1 Use A Structured Step-By-Step Process

Any predictive modeling machine learning project can be broken down into six common tasks:

1. Define Problem.
2. Summarize Data.
3. Prepare Data.

4. Evaluate Algorithms.
5. Improve Results.
6. Present Results.

Tasks can be combined or broken down further, but this is the general structure. To work through predictive modeling machine learning problems in Python, you need to map Python onto this process. The tasks may need to be adapted or renamed slightly to suit the Python way of doing things (e.g. Pandas for data loading and scikit-learn for modeling). The next section provides exactly this mapping and elaborates each task and the types of sub-tasks and libraries that you can use.

18.2 Machine Learning Project Template in Python

This section presents a project template that you can use to work through machine learning problems in Python end-to-end.

18.2.1 Template Summary

Below is the project template that you can use in your machine learning projects in Python.

```
# Python Project Template

# 1. Prepare Problem
# a) Load libraries
# b) Load dataset

# 2. Summarize Data
# a) Descriptive statistics
# b) Data visualizations

# 3. Prepare Data
# a) Data Cleaning
# b) Feature Selection
# c) Data Transforms

# 4. Evaluate Algorithms
# a) Split-out validation dataset
# b) Test options and evaluation metric
# c) Spot-Check Algorithms
# d) Compare Algorithms

# 5. Improve Accuracy
# a) Algorithm Tuning
# b) Ensembles

# 6. Finalize Model
# a) Predictions on validation dataset
# b) Create standalone model on entire training dataset
# c) Save model for later use
```

Listing 18.1: Predictive modeling machine learning project template.

18.2.2 How To Use The Project Template

1. Create a new file for your project (e.g. `project_name.py`).
2. Copy the project template.
3. Paste it into your empty project file.
4. Start to fill it in, using recipes from this book and others.

18.3 Machine Learning Project Template Steps

This section gives you additional details on each of the steps of the template.

18.3.1 Prepare Problem

This step is about loading everything you need to start working on your problem. This includes:

- Python modules, classes and functions that you intend to use.
- Loading your dataset from CSV.

This is also the home of any global configuration you might need to do. It is also the place where you might need to make a reduced sample of your dataset if it is too large to work with. Ideally, your dataset should be small enough to build a model or create a visualization within a minute, ideally 30 seconds. You can always scale up well performing models later.

18.3.2 Summarize Data

This step is about better understanding the data that you have available. This includes understanding your data using:

- Descriptive statistics such as summaries.
- Data visualizations such as plots with Matplotlib, ideally using convenience functions from Pandas.

Take your time and use the results to prompt a lot of questions, assumptions and hypotheses that you can investigate later with specialized models.

18.3.3 Prepare Data

This step is about preparing the data in such a way that it best exposes the structure of the problem and the relationships between your input attributes with the output variable. This includes tasks such as:

- Cleaning data by removing duplicates, marking missing values and even imputing missing values.

- Feature selection where redundant features may be removed and new features developed.
- Data transforms where attributes are scaled or redistributed in order to best expose the structure of the problem later to learning algorithms.

Start simple. Revisit this step often and cycle with the next step until you converge on a subset of algorithms and a presentation of the data that results in accurate or accurate-enough models to proceed.

18.3.4 Evaluate Algorithms

This step is about finding a subset of machine learning algorithms that are good at exploiting the structure of your data (e.g. have better than average skill). This involves steps such as:

- Separating out a validation dataset to use for later confirmation of the skill of your developed model.
- Defining test options using scikit-learn such as cross validation and the evaluation metric to use.
- Spot-checking a suite of linear and nonlinear machine learning algorithms.
- Comparing the estimated accuracy of algorithms.

On a given problem you will likely spend most of your time on this and the previous step until you converge on a set of 3-to-5 well performing machine learning algorithms.

18.3.5 Improve Accuracy

Once you have a shortlist of machine learning algorithms, you need to get the most out of them. There are two different ways to improve the accuracy of your models:

- Search for a combination of parameters for each algorithm using scikit-learn that yields the best results.
- Combine the prediction of multiple models into an ensemble prediction using ensemble techniques.

The line between this and the previous step can blur when a project becomes concrete. There may be a little algorithm tuning in the previous step. And in the case of ensembles, you may bring more than a shortlist of algorithms forward to combine their predictions.

18.3.6 Finalize Model

Once you have found a model that you believe can make accurate predictions on unseen data, you are ready to finalize it. Finalizing a model may involve sub-tasks such as:

- Using an optimal model tuned by scikit-learn to make predictions on unseen data.
- Creating a standalone model using the parameters tuned by scikit-learn.

- Saving an optimal model to file for later use.

Once you make it this far you are ready to present results to stakeholders and/or deploy your model to start making predictions on unseen data.

18.4 Tips For Using The Template Well

This section lists tips that you can use to make the most of the machine learning project template in Python.

- **Fast First Pass**. Make a first-pass through the project steps as fast as possible. This will give you confidence that you have all the parts that you need and a baseline from which to improve.
- **Cycles**. The process in not linear but cyclic. You will loop between steps, and probably spend most of your time in tight loops between steps 3-4 or 3-4-5 until you achieve a level of accuracy that is sufficient or you run out of time.
- **Attempt Every Step**. It is easy to skip steps, especially if you are not confident or familiar with the tasks of that step. Try and do something at each step in the process, even if it does not improve accuracy. You can always build upon it later. Don't skip steps, just reduce their contribution.
- **Ratchet Accuracy**. The goal of the project is model accuracy. Every step contributes towards this goal. Treat changes that you make as experiments that increase accuracy as the golden path in the process and reorganize other steps around them. Accuracy is a ratchet that can only move in one direction (better, not worse).
- **Adapt As Needed**. Modify the steps as you need on a project, especially as you become more experienced with the template. Blur the edges of tasks, such as steps 4-5 to best serve model accuracy.

18.5 Summary

In this lesson you discovered a machine learning project template in Python. It laid out the steps of a predictive modeling machine learning project with the goal of maximizing model accuracy. You can copy-and-paste the template and use it to jump-start your current or next machine learning project in Python.

18.5.1 Next Step

Now that you know how to structure a predictive modeling machine learning project in Python, you need to put this knowledge to use. In the next lesson you will work through a simple case study problem end-to-end. This is a famous case study and the *hello world* of machine learning projects.

Chapter 19

Your First Machine Learning Project in Python Step-By-Step

You need to see how all of the pieces of a predictive modeling machine learning project actually fit together. In this lesson you will complete your first machine learning project using Python. In this step-by-step tutorial project you will:

- Download and install Python SciPy and get the most useful package for machine learning in Python.
- Load a dataset and understand it's structure using statistical summaries and data visualization.
- Create 6 machine learning models, pick the best and build confidence that the accuracy is reliable.

If you are a machine learning beginner and looking to finally get started using Python, this tutorial was designed for you. Let's get started!

19.1 The Hello World of Machine Learning

The best small project to start with on a new tool is the classification of iris flowers. This is a good dataset for your first project because it is so well understood.

- Attributes are numeric so you have to figure out how to load and handle data.
- It is a classification problem, allowing you to practice with an easier type of supervised learning algorithm.
- It is a multiclass classification problem (multi-nominal) that may require some specialized handling.
- It only has 4 attributes and 150 rows, meaning it is small and easily fits into memory (and a screen or single sheet of paper).
- All of the numeric attributes are in the same units and the same scale not requiring any special scaling or transforms to get started.

In this tutorial we are going to work through a small machine learning project end-to-end. Here is an overview of what we are going to cover:

1. Loading the dataset.
2. Summarizing the dataset.
3. Visualizing the dataset.
4. Evaluating some algorithms.
5. Making some predictions.

Take your time and work through each step. Try to type in the commands yourself or copy-and-paste the commands to speed things up. Start your Python interactive environment and let's get started with your *hello world* machine learning project in Python.

19.2 Load The Data

In this step we are going to load the libraries and the iris data CSV file from URL.

19.2.1 Import libraries

First, let's import all of the modules, functions and objects we are going to use in this tutorial.

```
# Load libraries
import pandas
from pandas.tools.plotting import scatter_matrix
import matplotlib.pyplot as plt
from sklearn import cross_validation
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
```

Listing 19.1: Load libraries.

Everything should load without error. If you have an error, stop. You need a working SciPy environment before continuing. See the advice in Chapter 2 about setting up your environment.

19.2.2 Load Dataset

We can load the data directly from the UCI Machine Learning repository. We are using Pandas to load the data. We will also use Pandas next to explore the data both with descriptive statistics and data visualization. Note that we are specifying the names of each column when loading the data. This will help later when we explore the data.

```
# Load dataset
url = "https://goo.gl/mLmoIz"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
dataset = pandas.read_csv(url, names=names)
```

Listing 19.2: Load the Iris dataset.

The dataset should load without incident. If you do have network problems, you can download the `iris.data` file[1] into your working directory and load it using the same method, changing URL to the local file name.

19.3 Summarize the Dataset

Now it is time to take a look at the data. In this step we are going to take a look at the data a few different ways:

- Dimensions of the dataset.
- Peek at the data itself.
- Statistical summary of all attributes.
- Breakdown of the data by the class variable.

Don't worry, each look at the data is one command. These are useful commands that you can use again and again on future projects.

19.3.1 Dimensions of Dataset

We can get a quick idea of how many instances (rows) and how many attributes (columns) the data contains with the shape property.

```
# shape
print(dataset.shape)
```

Listing 19.3: Print the shape of the dataset.

You should see 150 instances and 5 attributes:

```
(150, 5)
```

Listing 19.4: Output of shape of the dataset.

19.3.2 Peek at the Data

It is also always a good idea to actually eyeball your data.

```
# head
print(dataset.head(20))
```

Listing 19.5: Print the first few rows of the dataset.

[1] https://goo.gl/mLmoIz

You should see the first 20 rows of the data:

```
    sepal-length sepal-width petal-length petal-width     class
0            5.1         3.5          1.4         0.2  Iris-setosa
1            4.9         3.0          1.4         0.2  Iris-setosa
2            4.7         3.2          1.3         0.2  Iris-setosa
3            4.6         3.1          1.5         0.2  Iris-setosa
4            5.0         3.6          1.4         0.2  Iris-setosa
5            5.4         3.9          1.7         0.4  Iris-setosa
6            4.6         3.4          1.4         0.3  Iris-setosa
7            5.0         3.4          1.5         0.2  Iris-setosa
8            4.4         2.9          1.4         0.2  Iris-setosa
9            4.9         3.1          1.5         0.1  Iris-setosa
10           5.4         3.7          1.5         0.2  Iris-setosa
11           4.8         3.4          1.6         0.2  Iris-setosa
12           4.8         3.0          1.4         0.1  Iris-setosa
13           4.3         3.0          1.1         0.1  Iris-setosa
14           5.8         4.0          1.2         0.2  Iris-setosa
15           5.7         4.4          1.5         0.4  Iris-setosa
16           5.4         3.9          1.3         0.4  Iris-setosa
17           5.1         3.5          1.4         0.3  Iris-setosa
18           5.7         3.8          1.7         0.3  Iris-setosa
19           5.1         3.8          1.5         0.3  Iris-setosa
```

Listing 19.6: Output of the first few rows of the dataset.

19.3.3 Statistical Summary

Now we can take a look at a summary of each attribute. This includes the count, mean, the min and max values as well as some percentiles.

```
# descriptions
print(dataset.describe())
```

Listing 19.7: Print the statistical descriptions of the dataset.

We can see that all of the numerical values have the same scale (centimeters) and similar ranges between 0 and 8 centimeters.

```
       sepal-length sepal-width petal-length petal-width
count    150.000000  150.000000   150.000000  150.000000
mean       5.843333    3.054000     3.758667    1.198667
std        0.828066    0.433594     1.764420    0.763161
min        4.300000    2.000000     1.000000    0.100000
25%        5.100000    2.800000     1.600000    0.300000
50%        5.800000    3.000000     4.350000    1.300000
75%        6.400000    3.300000     5.100000    1.800000
max        7.900000    4.400000     6.900000    2.500000
```

Listing 19.8: Output of the statistical descriptions of the dataset.

19.3.4 Class Distribution

Let's now take a look at the number of instances (rows) that belong to each class. We can view this as an absolute count.

```
# class distribution
print(dataset.groupby('class').size())
```

Listing 19.9: Print the class distribution in the dataset.

We can see that each class has the same number of instances (50 or 33% of the dataset).

```
class
Iris-setosa      50
Iris-versicolor  50
Iris-virginica   50
```

Listing 19.10: Output of the class distribution in the dataset.

19.4 Data Visualization

We now have a basic idea about the data. We need to extend this with some visualizations. We are going to look at two types of plots:

- Univariate plots to better understand each attribute.
- Multivariate plots to better understand the relationships between attributes.

19.4.1 Univariate Plots

We will start with some univariate plots, that is, plots of each individual variable. Given that the input variables are numeric, we can create box and whisker plots of each.

```
# box and whisker plots
dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)
plt.show()
```

Listing 19.11: Visualize the dataset using box and whisker plots.

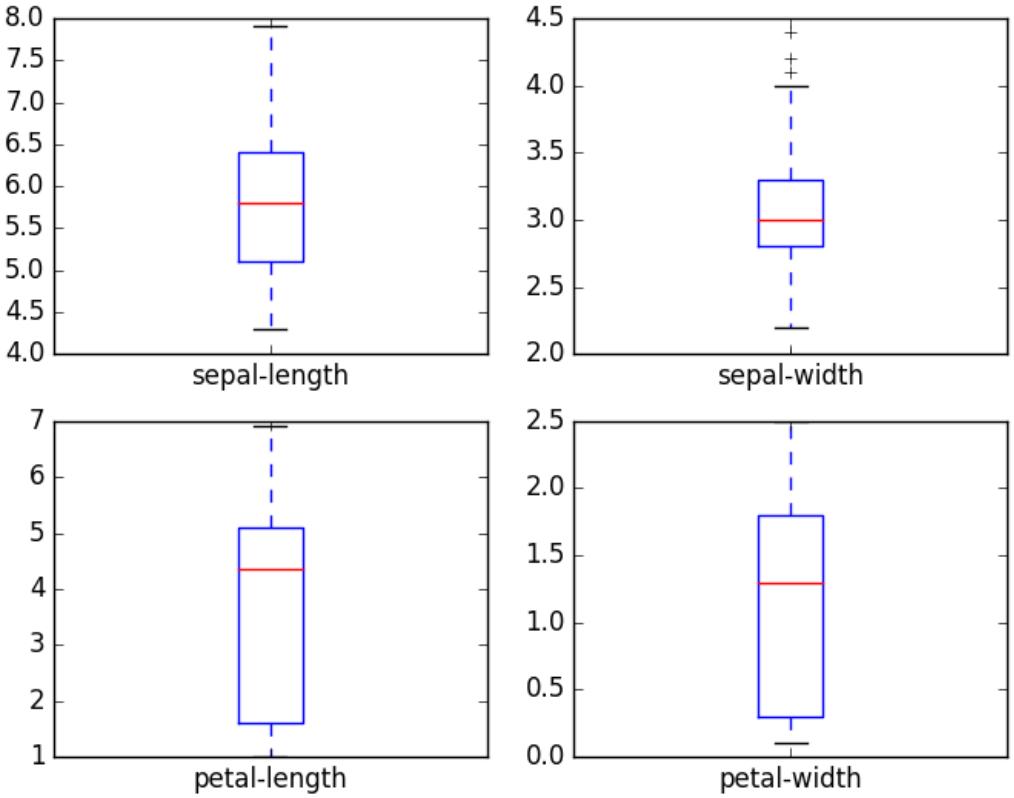

Figure 19.1: Box and Whisker Plots of Each Attribute.

We can also create a histogram of each input variable to get an idea of the distribution.

```
# histograms
dataset.hist()
plt.show()
```

Listing 19.12: Visualize the dataset using histogram plots.

It looks like perhaps two of the input variables have a Gaussian distribution. This is useful to note as we can use algorithms that can exploit this assumption.

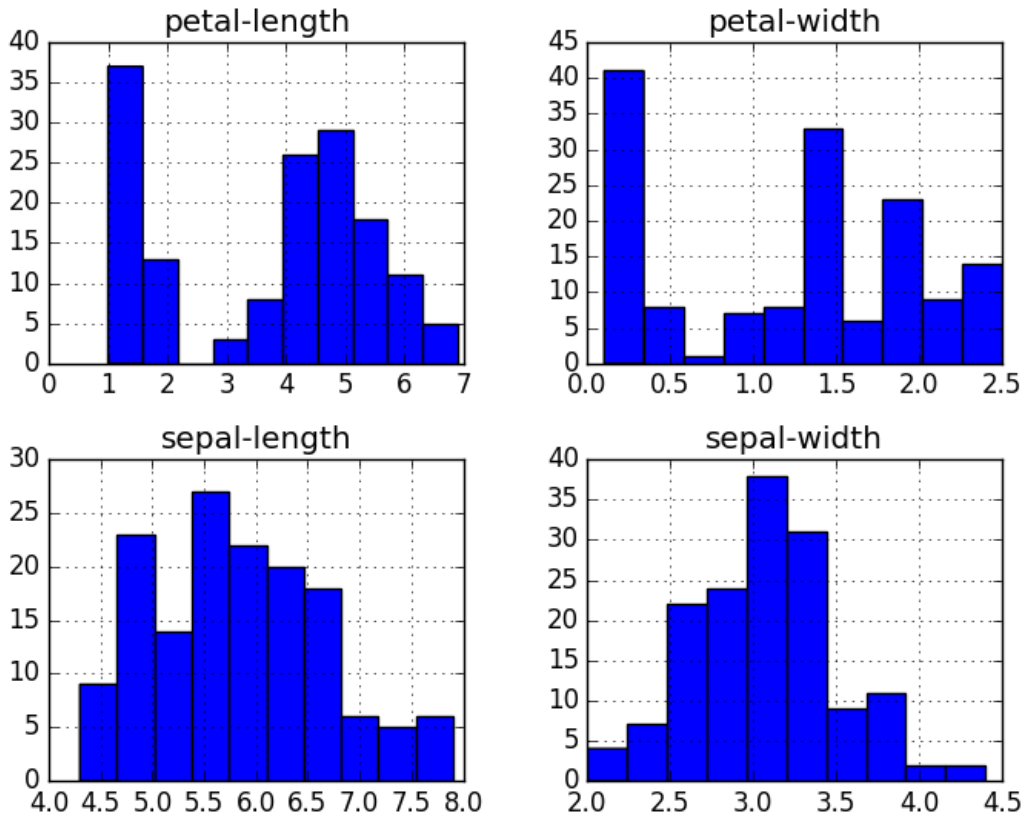

Figure 19.2: Histogram Plots of Each Attribute.

19.4.2 Multivariate Plots

Now we can look at the interactions between the variables. Let's look at scatter plots of all pairs of attributes. This can be helpful to spot structured relationships between input variables.

```
# scatter plot matrix
scatter_matrix(dataset)
plt.show()
```

Listing 19.13: Visualize the dataset using scatter plots.

Note the diagonal grouping of some pairs of attributes. This suggests a high correlation and a predictable relationship.

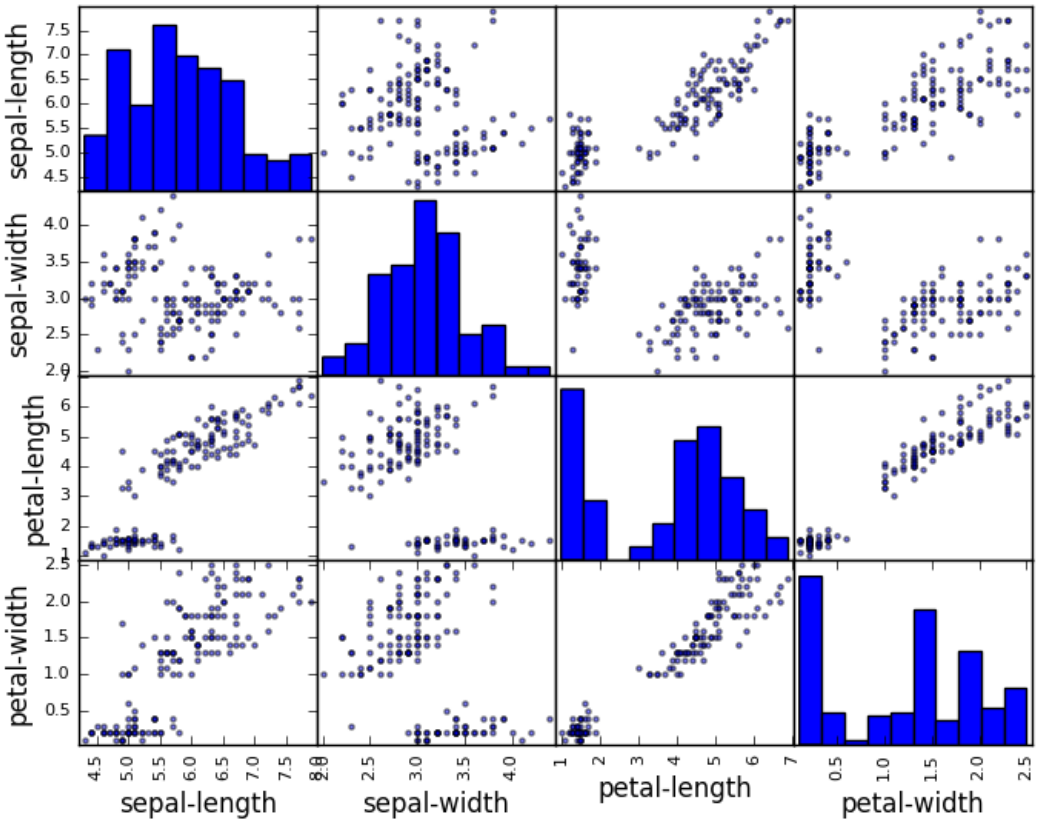

Figure 19.3: Scatter Plots of Each Pairwise Set of Attribute.

19.5 Evaluate Some Algorithms

Now it is time to create some models of the data and estimate their accuracy on unseen data. Here is what we are going to cover in this step:

1. Separate out a validation dataset.
2. Setup the test harness to use 10-fold cross validation.
3. Build 5 different models to predict species from flower measurements
4. Select the best model.

19.5.1 Create a Validation Dataset

We need to know whether or not the model that we created is any good. Later, we will use statistical methods to estimate the accuracy of the models that we create on unseen data. We also want a more concrete estimate of the accuracy of the best model on unseen data by evaluating it on actual unseen data. That is, we are going to hold back some data that the algorithms will not get to see and we will use this data to get a second and independent idea of

how accurate the best model might actually be. We will split the loaded dataset into two, 80% of which we will use to train our models and 20% that we will hold back as a validation dataset.

```
# Split-out validation dataset
array = dataset.values
X = array[:,0:4]
Y = array[:,4]
validation_size = 0.20
seed = 7
X_train, X_validation, Y_train, Y_validation = cross_validation.train_test_split(X, Y,
    test_size=validation_size, random_state=seed)
```

Listing 19.14: Separate data into Train and Validation Datasets.

You now have training data in the `X_train` and `Y_train` for preparing models and a `X_validation` and `Y_validation` sets that we can use later.

19.5.2 Test Harness

We will use 10-fold cross validation to estimate accuracy. This will split our dataset into 10 parts, train on 9 and test on 1 and repeat for all combinations of train-test splits.

```
# Test options and evaluation metric
num_folds = 10
num_instances = len(X_train)
seed = 7
scoring = 'accuracy'
```

Listing 19.15: Setup Algorithm Evaluation Test Harness.

We are using the metric of `accuracy` to evaluate models. This is a ratio of the number of correctly predicted instances divided by the total number of instances in the dataset multiplied by 100 to give a percentage (e.g. 95% accurate). We will be using the scoring variable when we run build and evaluate each model next.

19.5.3 Build Models

We don't know which algorithms would be good on this problem or what configurations to use. We get an idea from the plots that some of the classes are partially linearly separable in some dimensions, so we are expecting generally good results. Let's evaluate six different algorithms:

- Logistic Regression (LR).
- Linear Discriminant Analysis (LDA).
- k-Nearest Neighbors (KNN).
- Classification and Regression Trees (CART).
- Gaussian Naive Bayes (NB).
- Support Vector Machines (SVM).

This list is a good mixture of simple linear (LR and LDA), nonlinear (KNN, CART, NB and SVM) algorithms. We reset the random number seed before each run to ensure that the evaluation of each algorithm is performed using exactly the same data splits. It ensures the results are directly comparable. Let's build and evaluate our five models:

```
# Spot-Check Algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
# evaluate each model in turn
results = []
names = []
for name, model in models:
  kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
  cv_results = cross_validation.cross_val_score(model, X_train, Y_train, cv=kfold,
      scoring=scoring)
  results.append(cv_results)
  names.append(name)
  msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
  print(msg)
```

Listing 19.16: Evaluate a suite of algorithms on the dataset.

19.5.4 Select The Best Model

We now have 6 models and accuracy estimations for each. We need to compare the models to each other and select the most accurate. Running the example above, we get the following raw results:

```
LR: 0.966667 (0.040825)
LDA: 0.975000 (0.038188)
KNN: 0.983333 (0.033333)
CART: 0.975000 (0.038188)
NB: 0.975000 (0.053359)
SVM: 0.981667 (0.025000)
```

Listing 19.17: Output of evaluating a suite of algorithms.

We can see that it looks like KNN has the largest estimated accuracy score. We can also create a plot of the model evaluation results and compare the spread and the mean accuracy of each model. There is a population of accuracy measures for each algorithm because each algorithm was evaluated 10 times (10 fold cross validation).

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Listing 19.18: Plot the distribution of scores for each algorithm.

You can see that the box and whisker plots are squashed at the top of the range, with many samples achieving 100% accuracy.

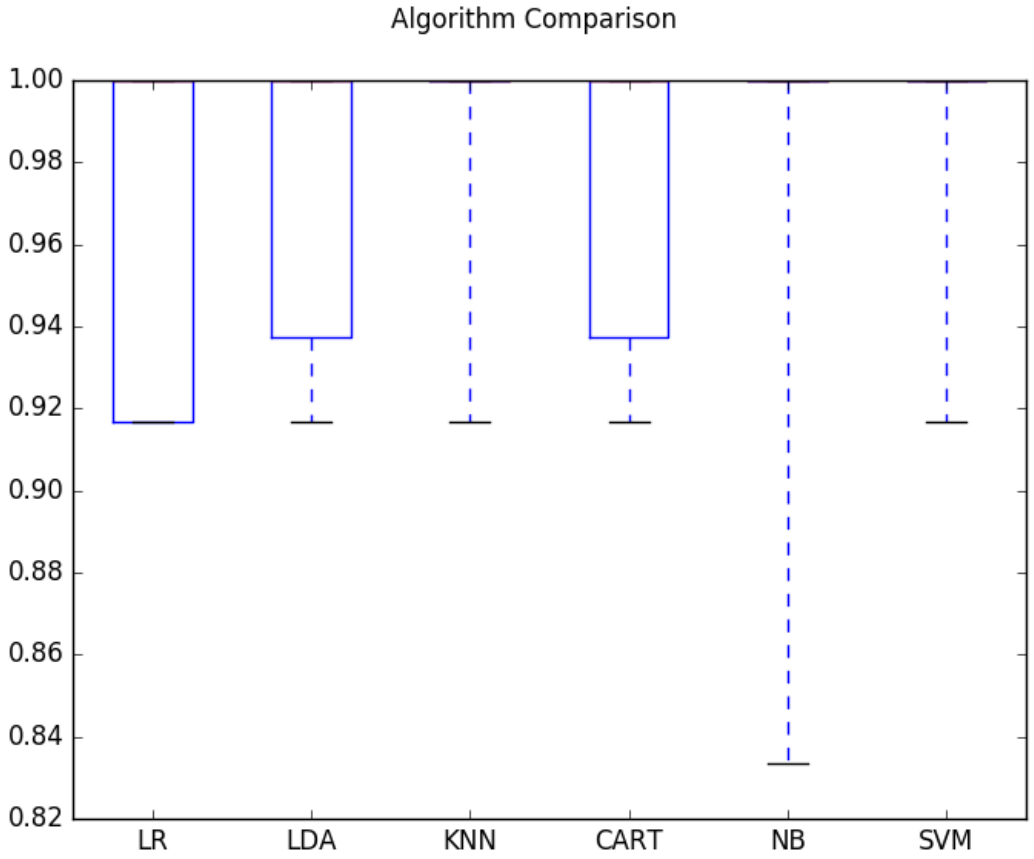

Figure 19.4: Box and Whisker Plots Comparing Algorithm Performance.

19.6 Make Predictions

The KNN algorithm was the most accurate model that we tested. Now we want to get an idea of the accuracy of the model on our validation dataset. This will give us an independent final check on the accuracy of the best model. It is important to keep a validation set just in case you made a slip during training, such as overfitting to the training set or a data leak. Both will result in an overly optimistic result. We can run the KNN model directly on the validation set and summarize the results as a final accuracy score, a confusion matrix and a classification report.

```
# Make predictions on validation dataset
knn = KNeighborsClassifier()
knn.fit(X_train, Y_train)
predictions = knn.predict(X_validation)
```

```
print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

Listing 19.19: Make Predictions on the Validation Dataset.

We can see that the accuracy is 0.9 or 90%. The confusion matrix provides an indication of the three errors made. Finally the classification report provides a breakdown of each class by precision, recall, f1-score and support showing excellent results (granted the validation dataset was small).

```
0.9

[[ 7  0  0]
 [ 0 11  1]
 [ 0  2  9]]

             precision   recall f1-score  support

Iris-setosa      1.00     1.00     1.00        7
Iris-versicolor 0.85     0.92     0.88       12
Iris-virginica  0.90     0.82     0.86       11

avg / total      0.90     0.90     0.90       30
```

Listing 19.20: Output of Making Predictions on the Validation Dataset.

19.7 Summary

In this lesson you discovered step-by-step how to complete your first machine learning project in Python. You discovered that completing a small end-to-end project from loading the data to making predictions is the best way to get familiar with the platform.

19.7.1 Next Step

You have applied the lessons from Part II on a simple problem and completed your first machine learning project. Next you will take things one step further and work through a regression predictive modeling problem. It will be a slightly more complex project and involve data transforms, algorithm tuning and use of ensemble methods to improve results.

Chapter 20

Regression Machine Learning Case Study Project

How do you work through a predictive modeling machine learning problem end-to-end? In this lesson you will work through a case study regression predictive modeling problem in Python including each step of the applied machine learning process. After completing this project, you will know:

- How to work through a regression predictive modeling problem end-to-end.
- How to use data transforms to improve model performance.
- How to use algorithm tuning to improve model performance.
- How to use ensemble methods and tuning of ensemble methods to improve model performance.

Let's get started.

20.1 Problem Definition

For this project we will investigate the Boston House Price dataset. Each record in the database describes a Boston suburb or town. The data was drawn from the Boston Standard Metropolitan Statistical Area (SMSA) in 1970. The attributes are defined as follows (taken from the UCI Machine Learning Repository[1]):

1. `CRIM`: per capita crime rate by town
2. `ZN`: proportion of residential land zoned for lots over 25,000 sq.ft.
3. `INDUS`: proportion of non-retail business acres per town
4. `CHAS`: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. `NOX`: nitric oxides concentration (parts per 10 million)

[1] https://archive.ics.uci.edu/ml/datasets/Housing

6. RM: average number of rooms per dwelling
7. AGE: proportion of owner-occupied units built prior to 1940
8. DIS: weighted distances to five Boston employment centers
9. RAD: index of accessibility to radial highways
10. TAX: full-value property-tax rate per $10,000
11. PTRATIO: pupil-teacher ratio by town
12. B: $1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
13. LSTAT: % lower status of the population
14. MEDV: Median value of owner-occupied homes in $1000s

We can see that the input attributes have a mixture of units.

20.2 Load the Dataset

Let's start off by loading the libraries required for this project.

```
# Load libraries
import pandas
import numpy
import matplotlib.pyplot as plt
from pandas.tools.plotting import scatter_matrix
from sklearn.preprocessing import StandardScaler
from sklearn import cross_validation
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.metrics import mean_squared_error
```

Listing 20.1: Load libraries.

We can now load the dataset directly from the UCI Machine Learning repository website.

```
# Load dataset
url = "https://goo.gl/sXleFv"
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
    'B', 'LSTAT', 'MEDV']
dataset = pandas.read_csv(url, delim_whitespace=True, names=names)
```

Listing 20.2: Load the dataset.

You can see that we are specifying the short names for each attribute so that we can reference them clearly later. You can also see that attributes are delimited by whitespace rather than commas in this file and we indicate this to `pandas.read_csv()` function via the `delim_whitespace` argument. We now have our data loaded.

20.3 Analyze Data

We can now take a closer look at our loaded data.

20.3.1 Descriptive Statistics

Let's start off by confirming the dimensions of the dataset, e.g. the number of rows and columns.

```
# shape
print(dataset.shape)
```

Listing 20.3: Print the shape of the dataset.

We have 506 instances to work with and can confirm the data has 14 attributes including the output attribute `MEDV`.

```
(506, 14)
```

Listing 20.4: Output of shape of the dataset.

Let's also look at the data types of each attribute.

```
# types
print(dataset.dtypes)
```

Listing 20.5: Print the data types of each attribute.

We can see that all of the attributes are numeric, mostly real values (float) and some have been interpreted as integers (int).

```
CRIM      float64
ZN        float64
INDUS     float64
CHAS        int64
NOX       float64
RM        float64
AGE       float64
DIS       float64
RAD         int64
TAX       float64
PTRATIO   float64
B         float64
LSTAT     float64
MEDV      float64
```

Listing 20.6: Output of the data types for each attribute.

Let's now take a peek at the first 20 rows of the data.

```
# head
print(dataset.head(20))
```

Listing 20.7: Print the first few rows of the dataset.

We can confirm that the scales for the attributes are all over the place because of the differing units. We may benefit from some transforms later on.

```
      CRIM    ZN  INDUS  CHAS    NOX     RM    AGE     DIS  RAD  TAX  PTRATIO      B  LSTAT  MEDV
0  0.00632  18.0   2.31     0  0.538  6.575   65.2  4.0900    1  296     15.3  396.90   4.98  24.0
1  0.02731   0.0   7.07     0  0.469  6.421   78.9  4.9671    2  242     17.8  396.90   9.14  21.6
2  0.02729   0.0   7.07     0  0.469  7.185   61.1  4.9671    2  242     17.8  392.83   4.03  34.7
3  0.03237   0.0   2.18     0  0.458  6.998   45.8  6.0622    3  222     18.7  394.63   2.94  33.4
4  0.06905   0.0   2.18     0  0.458  7.147   54.2  6.0622    3  222     18.7  396.90   5.33  36.2
5  0.02985   0.0   2.18     0  0.458  6.430   58.7  6.0622    3  222     18.7  394.12   5.21  28.7
6  0.08829  12.5   7.87     0  0.524  6.012   66.6  5.5605    5  311     15.2  395.60  12.43  22.9
7  0.14455  12.5   7.87     0  0.524  6.172   96.1  5.9505    5  311     15.2  396.90  19.15  27.1
8  0.21124  12.5   7.87     0  0.524  5.631  100.0  6.0821    5  311     15.2  386.63  29.93  16.5
9  0.17004  12.5   7.87     0  0.524  6.004   85.9  6.5921    5  311     15.2  386.71  17.10  18.9
10 0.22489  12.5   7.87     0  0.524  6.377   94.3  6.3467    5  311     15.2  392.52  20.45  15.0
11 0.11747  12.5   7.87     0  0.524  6.009   82.9  6.2267    5  311     15.2  396.90  13.27  18.9
12 0.09378  12.5   7.87     0  0.524  5.889   39.0  5.4509    5  311     15.2  390.50  15.71  21.7
13 0.62976   0.0   8.14     0  0.538  5.949   61.8  4.7075    4  307     21.0  396.90   8.26  20.4
14 0.63796   0.0   8.14     0  0.538  6.096   84.5  4.4619    4  307     21.0  380.02  10.26  18.2
15 0.62739   0.0   8.14     0  0.538  5.834   56.5  4.4986    4  307     21.0  395.62   8.47  19.9
16 1.05393   0.0   8.14     0  0.538  5.935   29.3  4.4986    4  307     21.0  386.85   6.58  23.1
17 0.78420   0.0   8.14     0  0.538  5.990   81.7  4.2579    4  307     21.0  386.75  14.67  17.5
18 0.80271   0.0   8.14     0  0.538  5.456   36.6  3.7965    4  307     21.0  288.99  11.69  20.2
19 0.72580   0.0   8.14     0  0.538  5.727   69.5  3.7965    4  307     21.0  390.95  11.28  18.2
```

Listing 20.8: Output of the first few rows of the dataset.

Let's summarize the distribution of each attribute.

```
# descriptions
pandas.set_option('precision', 1)
print(dataset.describe())
```

Listing 20.9: Print the statistical descriptions of the dataset.

We now have a better feeling for how different the attributes are. The min and max values as well are the means vary a lot. We are likely going to get better results by rescaling the data in some way.

```
             CRIM     ZN  INDUS     CHAS    NOX     RM    AGE    DIS    RAD    TAX  PTRATIO      B
                LSTAT  MEDV
count  5.1e+02  506.0  506.0  5.1e+02  506.0  506.0  506.0  506.0  506.0  506.0    506.0  506.0
    506.0  506.0
mean   3.6e+00   11.4   11.1  6.9e-02    0.6    6.3   68.6    3.8    9.5  408.2     18.5  356.7
    12.7   22.5
std    8.6e+00   23.3    6.9  2.5e-01    0.1    0.7   28.1    2.1    8.7  168.5      2.2   91.3
    7.1    9.2
min    6.3e-03    0.0    0.5  0.0e+00    0.4    3.6    2.9    1.1    1.0  187.0     12.6    0.3
    1.7    5.0
25%    8.2e-02    0.0    5.2  0.0e+00    0.4    5.9   45.0    2.1    4.0  279.0     17.4  375.4
    6.9   17.0
```

```
50%   2.6e-01   0.0   9.7  0.0e+00   0.5   6.2   77.5   3.2   5.0  330.0    19.1  391.4
    11.4   21.2
75%   3.7e+00  12.5  18.1  0.0e+00   0.6   6.6   94.1   5.2  24.0  666.0    20.2  396.2
    17.0   25.0
max   8.9e+01 100.0  27.7  1.0e+00   0.9   8.8  100.0  12.1  24.0  711.0    22.0  396.9
    38.0   50.0
```

Listing 20.10: Output of the statistical descriptions of the dataset.

Now, let's now take a look at the correlation between all of the numeric attributes.

```
# correlation
pandas.set_option('precision', 2)
print(dataset.corr(method='pearson'))
```

Listing 20.11: Print the correlations between the attributes.

This is interesting. We can see that many of the attributes have a strong correlation (e.g. > 0.70 or < -0.70). For example:

- `NOX` and `INDUS` with 0.77.
- `DIS` and `INDUS` with -0.71.
- `TAX` and `INDUS` with 0.72.
- `AGE` and `NOX` with 0.73.
- `DIS` and `NOX` with -0.78.

It also looks like `LSTAT` has a good negative correlation with the output variable `MEDV` with a value of -0.74.

```
        CRIM    ZN  INDUS     CHAS   NOX    RM   AGE   DIS      RAD   TAX  PTRATIO    B  LSTAT
            MEDV
CRIM    1.00 -0.20  0.41 -5.59e-02 0.42 -0.22 0.35 -0.38 6.26e-01 0.58    0.29 -0.39  0.46
    -0.39
ZN      -0.20 1.00  -0.53 -4.27e-02 -0.52 0.31 -0.57 0.66 -3.12e-01 -0.31 -0.39 0.18 -0.41
    0.36
INDUS   0.41 -0.53  1.00  6.29e-02 0.76 -0.39 0.64 -0.71 5.95e-01 0.72    0.38 -0.36  0.60
    -0.48
CHAS    -0.06 -0.04  0.06  1.00e+00 0.09  0.09  0.09 -0.10 -7.37e-03 -0.04 -0.12  0.05  -0.05
    0.18
NOX     0.42 -0.52  0.76  9.12e-02 1.00 -0.30 0.73 -0.77 6.11e-01 0.67    0.19 -0.38  0.59
    -0.43
RM      -0.22 0.31  -0.39  9.13e-02 -0.30 1.00 -0.24 0.21 -2.10e-01 -0.29 -0.36 0.13  -0.61
    0.70
AGE     0.35 -0.57  0.64  8.65e-02 0.73 -0.24 1.00 -0.75 4.56e-01 0.51    0.26 -0.27  0.60
    -0.38
DIS     -0.38 0.66  -0.71 -9.92e-02 -0.77 0.21 -0.75 1.00 -4.95e-01 -0.53 -0.23 0.29  -0.50
    0.25
RAD     0.63 -0.31  0.60 -7.37e-03 0.61 -0.21 0.46 -0.49 1.00e+00 0.91    0.46 -0.44  0.49
    -0.38
TAX     0.58 -0.31  0.72 -3.56e-02 0.67 -0.29 0.51 -0.53 9.10e-01 1.00    0.46 -0.44  0.54
    -0.47
PTRATIO 0.29 -0.39  0.38 -1.22e-01 0.19 -0.36 0.26 -0.23 4.65e-01 0.46    1.00 -0.18  0.37
    -0.51
```

```
B       -0.39  0.18  -0.36  4.88e-02 -0.38  0.13  -0.27  0.29  -4.44e-01 -0.44  -0.18  1.00   -0.37
    0.33
LSTAT    0.46 -0.41   0.60 -5.39e-02  0.59 -0.61   0.60 -0.50  4.89e-01  0.54    0.37 -0.37   1.00
    -0.74
MEDV    -0.39  0.36  -0.48  1.75e-01 -0.43  0.70  -0.38  0.25  -3.82e-01 -0.47  -0.51  0.33   -0.74
    1.00
```

Listing 20.12: Output of the statistical descriptions of the dataset.

20.4 Data Visualizations

20.4.1 Unimodal Data Visualizations

Let's look at visualizations of individual attributes. It is often useful to look at your data using multiple different visualizations in order to spark ideas. Let's look at histograms of each attribute to get a sense of the data distributions.

```
# histograms
dataset.hist(sharex=False, sharey=False, xlabelsize=1, ylabelsize=1)
plt.show()
```

Listing 20.13: Visualize the dataset using histogram plots.

We can see that some attributes may have an exponential distribution, such as `CRIM`, `ZN`, `AGE` and `B`. We can see that others may have a bimodal distribution such as `RAD` and `TAX`.

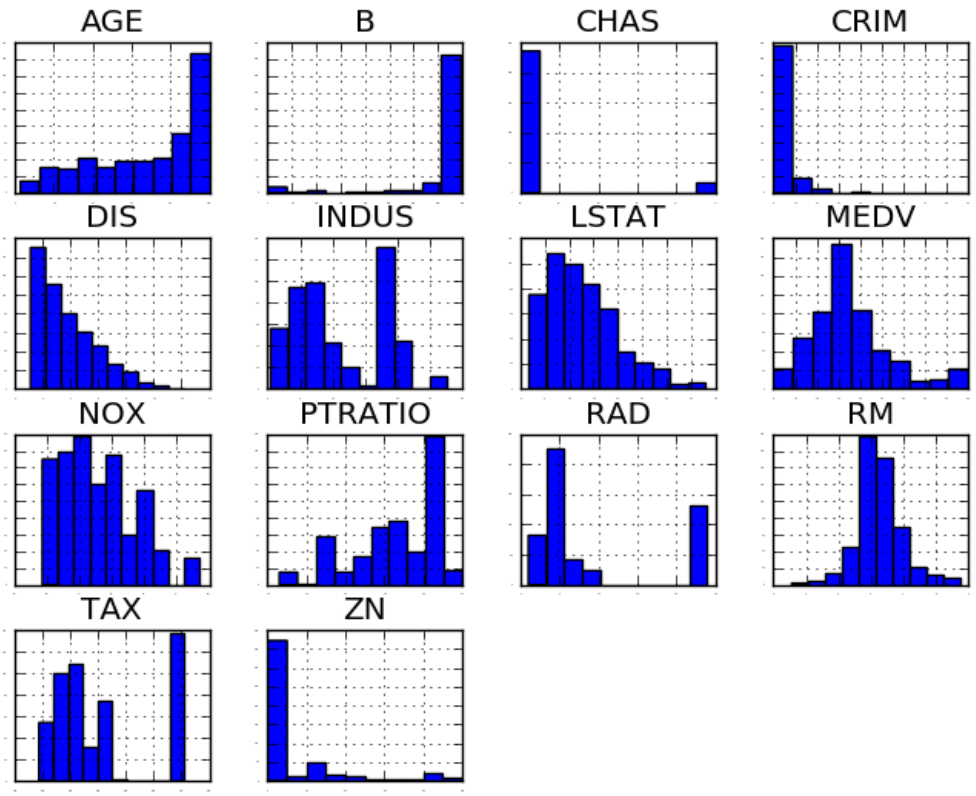

Figure 20.1: Histogram Plots of Each Attribute.

Let's look at the same distributions using density plots that smooth them out a bit.

```
# density
dataset.plot(kind='density', subplots=True, layout=(4,4), sharex=False, legend=False,
    fontsize=1)
plt.show()
```

Listing 20.14: Visualize the dataset using density plots.

This perhaps adds more evidence to our suspicion about possible exponential and bimodal distributions. It also looks like `NOX`, `RM` and `LSTAT` may be skewed Gaussian distributions, which might be helpful later with transforms.

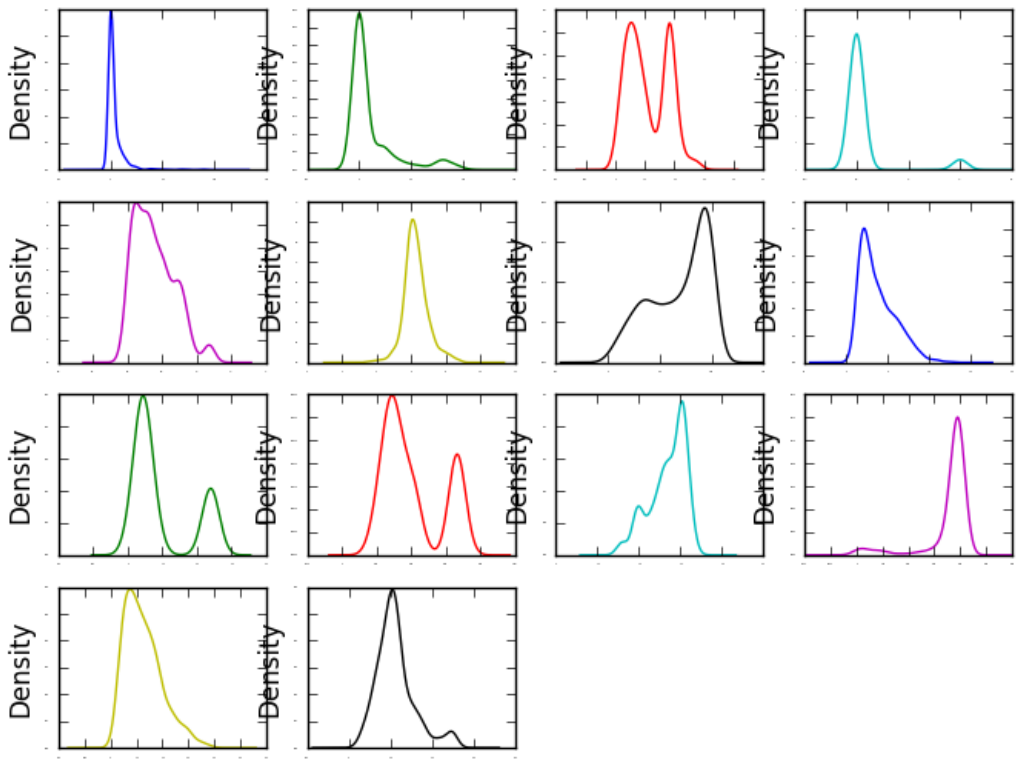

Figure 20.2: Density Plots of Each Attribute.

Let's look at the data with box and whisker plots of each attribute.

```
# box and whisker plots
dataset.plot(kind='box', subplots=True, layout=(4,4), sharex=False, sharey=False,
    fontsize=8)
plt.show()
```

Listing 20.15: Visualize the dataset using box and whisker plots.

This helps point out the skew in many distributions so much so that data looks like outliers (e.g. beyond the whisker of the plots).

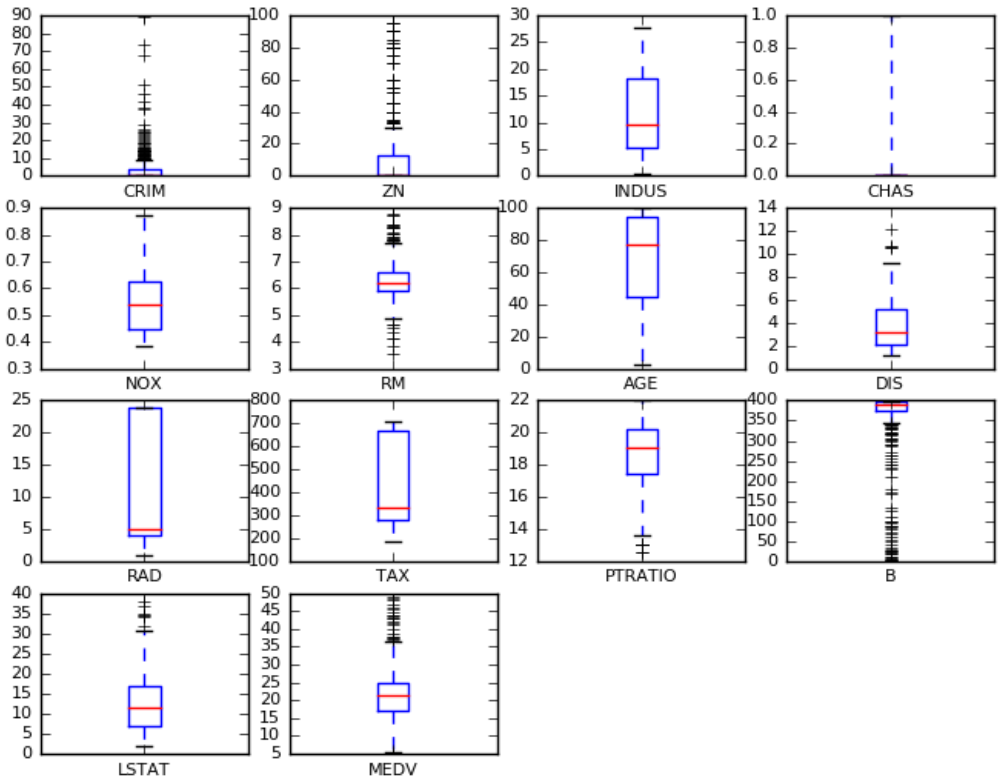

Figure 20.3: Box and Whisker Plots of Each Attribute.

20.4.2 Multimodal Data Visualizations

Let's look at some visualizations of the interactions between variables. The best place to start is a scatter plot matrix.

```
# scatter plot matrix
scatter_matrix(dataset)
plt.show()
```

Listing 20.16: Visualize the dataset using scatter plots.

We can see that some of the higher correlated attributes do show good structure in their relationship. Not linear, but nice predictable curved relationships.

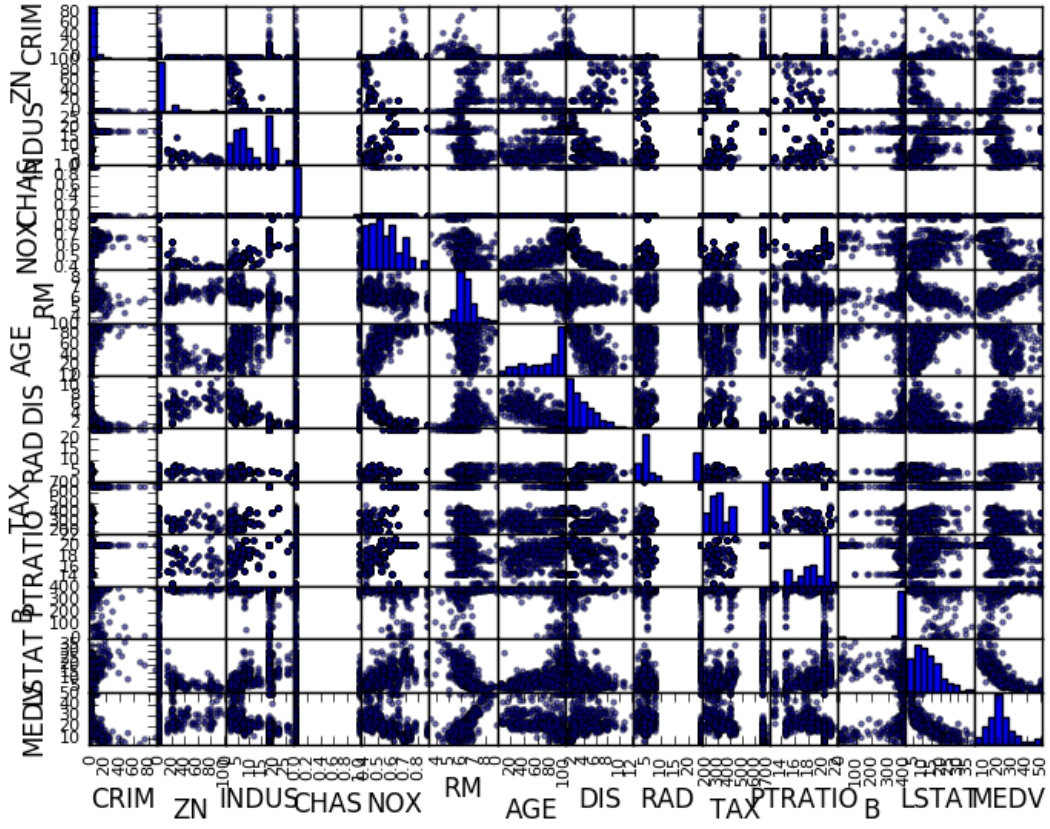

Figure 20.4: Scatter Plot Matrix of Dataset Attributes.

Let's also visualize the correlations between the attributes.

```
# correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(dataset.corr(), vmin=-1, vmax=1, interpolation='none')
fig.colorbar(cax)
ticks = numpy.arange(0,14,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names)
ax.set_yticklabels(names)
plt.show()
```

Listing 20.17: Visualize the correlations between attributes.

The dark red color shows positive correlation whereas the dark blue color shows negative correlation. We can also see some dark red and dark blue that suggest candidates for removal to better improve accuracy of models later on.

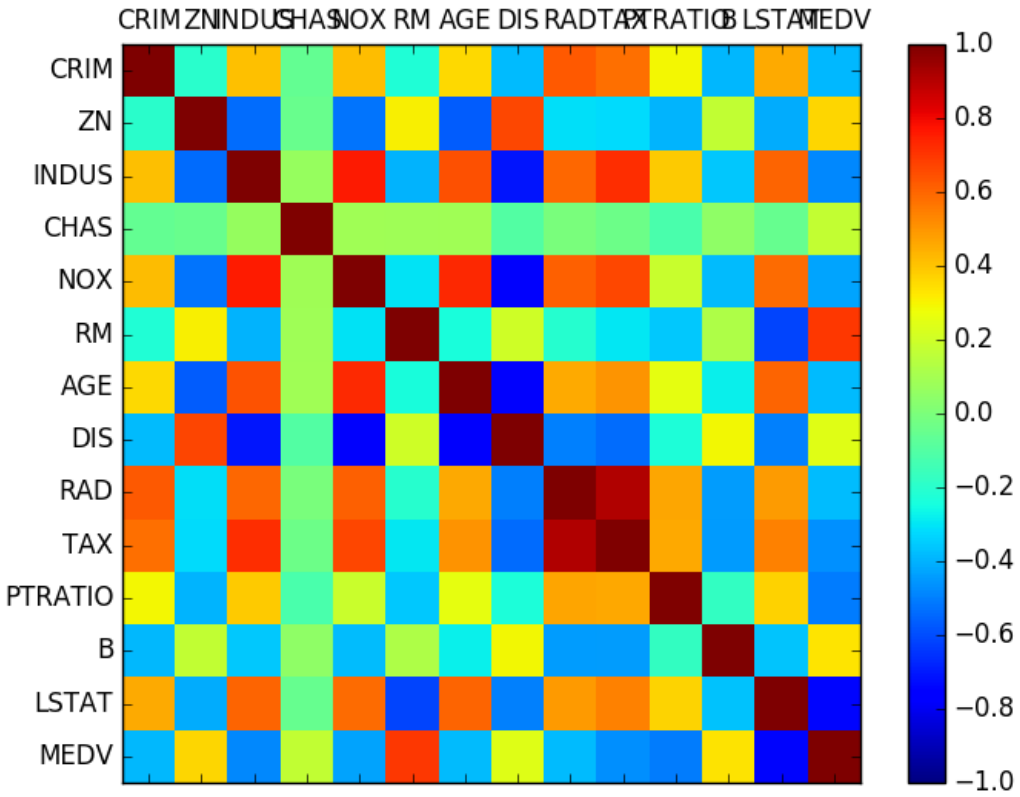

Figure 20.5: Correlation Matrix of Dataset Attributes.

20.4.3 Summary of Ideas

There is a lot of structure in this dataset. We need to think about transforms that we could use later to better expose the structure which in turn may improve modeling accuracy. So far it would be worth trying:

- Feature selection and removing the most correlated attributes.
- Normalizing the dataset to reduce the effect of differing scales.
- Standardizing the dataset to reduce the effects of differing distributions.

With lots of additional time I would also explore the possibility of binning (discretization) of the data. This can often improve accuracy for decision tree algorithms.

20.5 Validation Dataset

It is a good idea to use a validation hold-out set. This is a sample of the data that we hold back from our analysis and modeling. We use it right at the end of our project to confirm the accuracy of our final model. It is a smoke test that we can use to see if we messed up and to

give us confidence on our estimates of accuracy on unseen data. We will use 80% of the dataset for modeling and hold back 20% for validation.

```
# Split-out validation dataset
array = dataset.values
X = array[:,0:13]
Y = array[:,13]
validation_size = 0.20
seed = 7
X_train, X_validation, Y_train, Y_validation = cross_validation.train_test_split(X, Y,
    test_size=validation_size, random_state=seed)
```

Listing 20.18: Separate Data into a Training and Validation Datasets.

20.6 Evaluate Algorithms: Baseline

We have no idea what algorithms will do well on this problem. Gut feel suggests regression algorithms like Linear Regression and ElasticNet may do well. It is also possible that decision trees and even SVM may do well. I have no idea. Let's design our test harness. We will use 10-fold cross validation. The dataset is not too small and this is a good standard test harness configuration. We will evaluate algorithms using the Mean Squared Error (MSE) metric. MSE will give a gross idea of how wrong all predictions are (0 is perfect).

```
# Test options and evaluation metric
num_folds = 10
num_instances = len(X_train)
seed = 7
scoring = 'mean_squared_error'
```

Listing 20.19: Configure Algorithm Evaluation Test Harness.

Let's create a baseline of performance on this problem and spot-check a number of different algorithms. We will select a suite of different algorithms capable of working on this regression problem. The six algorithms selected include:

- **Linear Algorithms**: Linear Regression (LR), Lasso Regression (LASSO) and ElasticNet (EN).
- **Nonlinear Algorithms**: Classification and Regression Trees (CART), Support Vector Regression (SVR) and k-Nearest Neighbors (KNN).

```
# Spot-Check Algorithms
models = []
models.append(('LR', LinearRegression()))
models.append(('LASSO', Lasso()))
models.append(('EN', ElasticNet()))
models.append(('KNN', KNeighborsRegressor()))
models.append(('CART', DecisionTreeRegressor()))
models.append(('SVR', SVR()))
```

Listing 20.20: Create the List of Algorithms to Evaluate.

The algorithms all use default tuning parameters. Let's compare the algorithms. We will display the mean and standard deviation of MSE for each algorithm as we calculate it and collect the results for use later.

```
# evaluate each model in turn
results = []
names = []
for name, model in models:
  kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
  cv_results = cross_validation.cross_val_score(model, X_train, Y_train, cv=kfold,
      scoring=scoring)
  results.append(cv_results)
  names.append(name)
  msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
  print(msg)
```

Listing 20.21: Evaluate the List of Algorithms.

It looks like LR has the lowest MSE, followed closely by CART.

```
LR: -21.379856 (9.414264)
LASSO: -26.423561 (11.651110)
EN: -27.502259 (12.305022)
KNN: -41.896488 (13.901688)
CART: -23.608957 (12.033061)
SVR: -85.518342 (31.994798)
```

Listing 20.22: Results from Evaluating Algorithms.

Let's take a look at the distribution of scores across all cross validation folds by algorithm.

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Listing 20.23: Visualzie the Differences in Algorithm Performance.

We can see similar distributions for the regression algorithms and perhaps a tighter distribution of scores for CART.

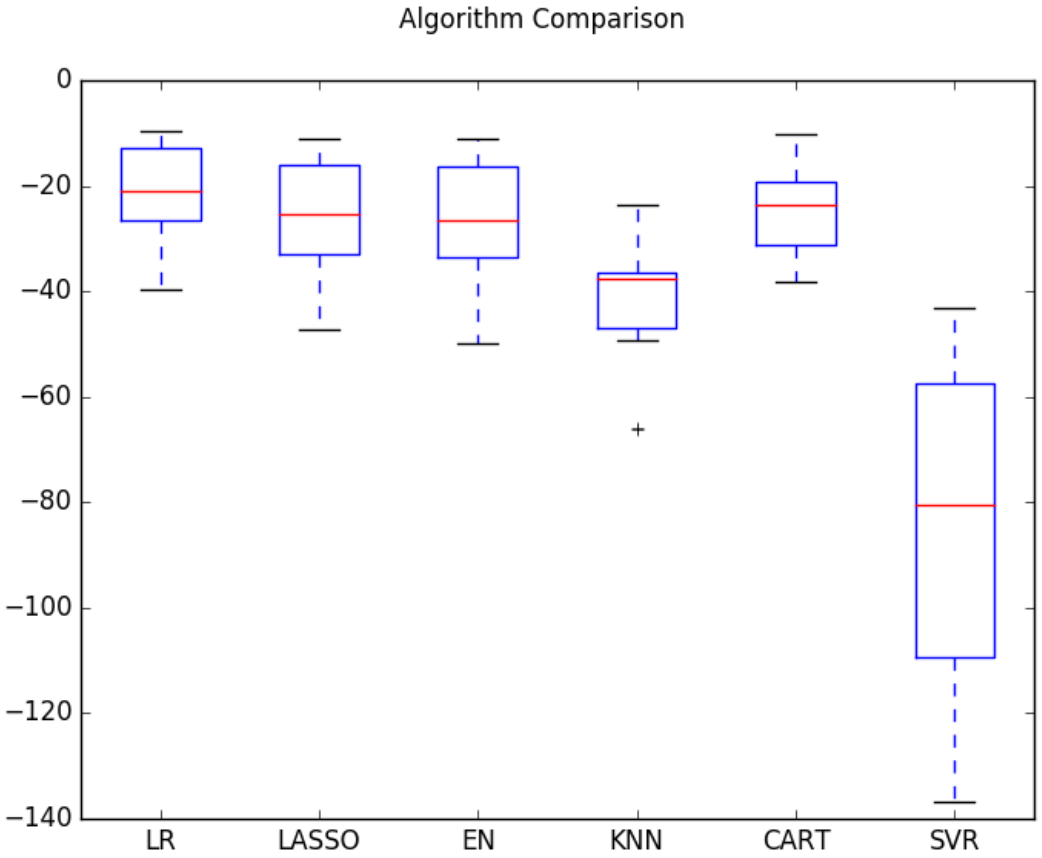

Figure 20.6: Compare Algorithm Performance.

The differing scales of the data is probably hurting the skill of all of the algorithms and perhaps more so for SVR and KNN. In the next section we will look at running the same algorithms using a standardized copy of the data.

20.7 Evaluate Algorithms: Standardization

We suspect that the differing scales of the raw data may be negatively impacting the skill of some of the algorithms. Let's evaluate the same algorithms with a standardized copy of the dataset. This is where the data is transformed such that each attribute has a mean value of zero and a standard deviation of 1. We also need to avoid data leakage when we transform the data. A good way to avoid leakage is to use pipelines that standardize the data and build the model for each fold in the cross validation test harness. That way we can get a fair estimation of how each model with standardized data might perform on unseen data.

```
# Standardize the dataset
pipelines = []
pipelines.append(('ScaledLR', Pipeline([('Scaler', StandardScaler()),('LR',
    LinearRegression())])))
pipelines.append(('ScaledLASSO', Pipeline([('Scaler', StandardScaler()),('LASSO',
    Lasso())])))
```

```
pipelines.append(('ScaledEN', Pipeline([('Scaler', StandardScaler()),('EN',
    ElasticNet())])))
pipelines.append(('ScaledKNN', Pipeline([('Scaler', StandardScaler()),('KNN',
    KNeighborsRegressor())])))
pipelines.append(('ScaledCART', Pipeline([('Scaler', StandardScaler()),('CART',
    DecisionTreeRegressor())])))
pipelines.append(('ScaledSVR', Pipeline([('Scaler', StandardScaler()),('SVR', SVR())])))
results = []
names = []
for name, model in pipelines:
  kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
  cv_results = cross_validation.cross_val_score(model, X_train, Y_train, cv=kfold,
      scoring=scoring)
  results.append(cv_results)
  names.append(name)
  msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
  print(msg)
```

Listing 20.24: Evaluate Algorithms On Standardized Dataset.

Running the example provides a list of mean squared errors. We can see that scaling did have an effect on KNN, driving the error lower than the other models.

```
ScaledLR: -21.379856 (9.414264)
ScaledLASSO: -26.607314 (8.978761)
ScaledEN: -27.932372 (10.587490)
ScaledKNN: -20.107620 (12.376949)
ScaledCART: -23.360362 (9.671240)
ScaledSVR: -29.633086 (17.009186)
```

Listing 20.25: Results from Evaluating Algorithms On Standardized Dataset.

Let's take a look at the distribution of the scores across the cross validation folds.

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Scaled Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Listing 20.26: Visualzie the Differences in Algorithm Performance on Standardized Dataset.

We can see that KNN has both a tight distribution of error and has the lowest score.

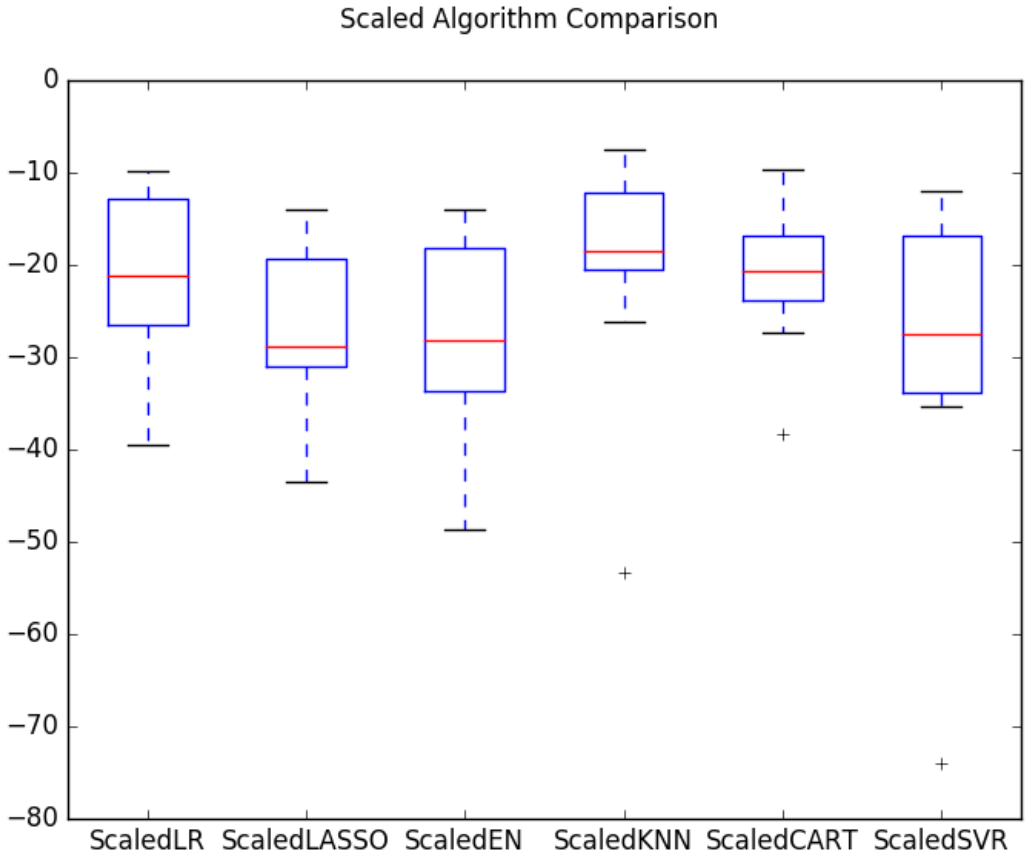

Figure 20.7: Compare Algorithm Performance on the Standardized dataset.

20.8 Improve Results With Tuning

We know from the results in the previous section that KNN achieves good results on a scaled version of the dataset. But can it do better. The default value for the number of neighbors in KNN is 7. We can use a grid search to try a set of different numbers of neighbors and see if we can improve the score. The below example tries odd k values from 1 to 21, an arbitrary range covering a known good value of 7. Each k value (`n_neighbors`) is evaluated using 10-fold cross validation on a standardized copy of the training dataset.

```
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
k_values = numpy.array([1,3,5,7,9,11,13,15,17,19,21])
param_grid = dict(n_neighbors=k_values)
model = KNeighborsRegressor()
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, cv=kfold)
grid_result = grid.fit(rescaledX, Y_train)
```

Listing 20.27: Tune the Parameters of the KNN Algorithm on the Standardized Dataset.

We can display the mean and standard deviation scores as well as the best performing value for k below.

```
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

Listing 20.28: Print Output From Tuning the KNN Algorithm.

You can see that the best for k (`n_neighbors`) is 3 providing a mean squared error of -18.172137, the best so far.

```
Best: -18.172137 using {'n_neighbors': 3}

-20.169640 (14.986904) with: {'n_neighbors': 1}
-18.109304 (12.880861) with: {'n_neighbors': 3}
-20.063115 (12.138331) with: {'n_neighbors': 5}
-20.514297 (12.278136) with: {'n_neighbors': 7}
-20.319536 (11.554509) with: {'n_neighbors': 9}
-20.963145 (11.540907) with: {'n_neighbors': 11}
-21.099040 (11.870962) with: {'n_neighbors': 13}
-21.506843 (11.468311) with: {'n_neighbors': 15}
-22.739137 (11.499596) with: {'n_neighbors': 17}
-23.829011 (11.277558) with: {'n_neighbors': 19}
-24.320892 (11.849667) with: {'n_neighbors': 21}
```

Listing 20.29: Output From Tuning the KNN Algorithm.

20.9 Ensemble Methods

Another way that we can improve the performance of algorithms on this problem is by using ensemble methods. In this section we will evaluate four different ensemble machine learning algorithms, two boosting and two bagging methods:

- **Boosting Methods**: AdaBoost (AB) and Gradient Boosting (GBM).
- **Bagging Methods**: Random Forests (RF) and Extra Trees (ET).

We will use the same test harness as before, 10-fold cross validation and pipelines that standardize the training data for each fold.

```
ensembles = []
ensembles.append(('ScaledAB', Pipeline([('Scaler', StandardScaler()),('AB',
    AdaBoostRegressor())])))
ensembles.append(('ScaledGBM', Pipeline([('Scaler', StandardScaler()),('GBM',
    GradientBoostingRegressor())])))
ensembles.append(('ScaledRF', Pipeline([('Scaler', StandardScaler()),('RF',
    RandomForestRegressor())])))
ensembles.append(('ScaledET', Pipeline([('Scaler', StandardScaler()),('ET',
    ExtraTreesRegressor())])))
results = []
names = []
for name, model in ensembles:
  kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
  cv_results = cross_validation.cross_val_score(model, X_train, Y_train, cv=kfold,
      scoring=scoring)
```

```
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

Listing 20.30: Evaluate Ensemble Algorithms on the Standardized Dataset.

Running the example calculates the mean squared error for each method using the default parameters. We can see that we're generally getting better scores than our linear and nonlinear algorithms in previous sections.

```
ScaledAB: -14.964638 (6.069505)
ScaledGBM: -9.999626 (4.391458)
ScaledRF: -13.676055 (6.968407)
ScaledET: -11.497637 (7.164636)
```

Listing 20.31: Output from Evaluating Ensemble Algorithms.

We can also plot the distribution of scores across the cross validation folds.

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Scaled Ensemble Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Listing 20.32: Visualzie the Differences in Ensemble Algorithm Performance on Standardized Dataset.

It looks like Gradient Boosting has a better mean score, it also looks like Extra Trees has a similar distribution and perhaps a better median score.

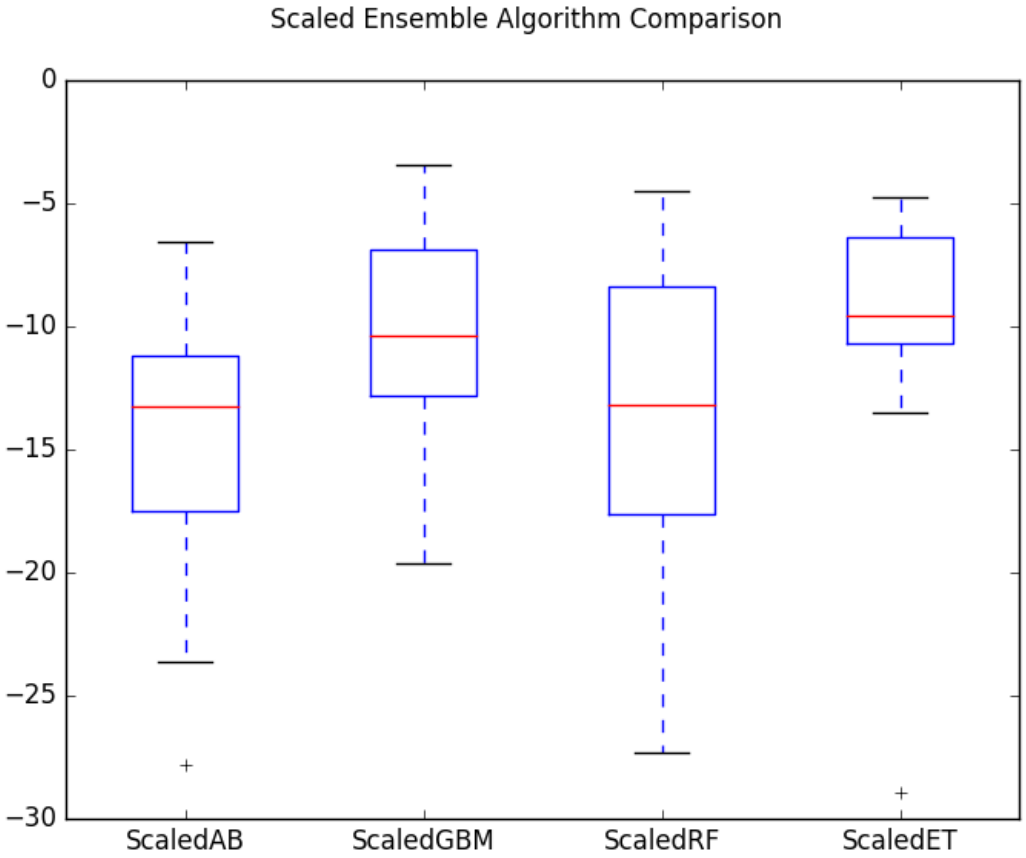

Figure 20.8: Compare the Performance of Ensemble Algorithms.

We can probably do better, given that the ensemble techniques used the default parameters. In the next section we will look at tuning the Gradient Boosting to further lift the performance.

20.10 Tune Ensemble Methods

The default number of boosting stages to perform (`n_estimators`) is 100. This is a good candidate parameter of Gradient Boosting to tune. Often, the larger the number of boosting stages, the better the performance but the longer the training time. In this section we will look at tuning the number of stages for gradient boosting. Below we define a parameter grid `n_estimators` values from 50 to 400 in increments of 50. Each setting is evaluated using 10-fold cross validation.

```
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
param_grid = dict(n_estimators=numpy.array([50,100,150,200,250,300,350,400]))
model = GradientBoostingRegressor(random_state=seed)
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, cv=kfold)
grid_result = grid.fit(rescaledX, Y_train)
```

Listing 20.33: Tune GBM on Scaled Dataset.

As before, we can summarize the best configuration and get an idea of how performance changed with each different configuration.

```
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

Listing 20.34: Print Performance of Tuned GBM on Scaled Dataset.

We can see that the best configuration was `n_estimators`=400 resulting in a mean squared error of -9.356471, about 0.65 units better than the untuned method.

```
Best: -9.356471 using {'n_estimators': 400}

-10.794196 (4.711473) with: {'n_estimators': 50}
-10.023378 (4.430026) with: {'n_estimators': 100}
-9.677657 (4.264829) with: {'n_estimators': 150}
-9.523680 (4.259064) with: {'n_estimators': 200}
-9.432755 (4.250884) with: {'n_estimators': 250}
-9.414258 (4.262219) with: {'n_estimators': 300}
-9.353381 (4.242264) with: {'n_estimators': 350}
-9.339880 (4.255717) with: {'n_estimators': 400}
```

Listing 20.35: Output Performance of Tuned GBM on Scaled Dataset.

Next we can finalize the model and prepare it for general use.

20.11 Finalize Model

In this section we will finalize the gradient boosting model and evaluate it on our hold out validation dataset. First we need to prepare the model and train it on the entire training dataset. This includes standardizing the training dataset before training.

```
# prepare the model
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
model = GradientBoostingRegressor(random_state=seed, n_estimators=400)
model.fit(rescaledX, Y_train)
```

Listing 20.36: Construct the Finalized Model.

We can then scale the inputs for the validation dataset and generate predictions.

```
# transform the validation dataset
rescaledValidationX = scaler.transform(X_validation)
predictions = model.predict(rescaledValidationX)
print(mean_squared_error(Y_validation, predictions))
```

Listing 20.37: Evaluate the Finalized Model.

We can see that the estimated mean squared error is 11.8, close to our estimate of -9.3.

```
11.8752520792
```

Listing 20.38: Output of Evaluating the Finalized Model.

20.12 Summary

In this chapter you worked through a regression predictive modeling machine learning problem from end-to-end using Python. Specifically, the steps covered were:

- Problem Definition (Boston house price data).
- Loading the Dataset.
- Analyze Data (some skewed distributions and correlated attributes).
- Evaluate Algorithms (Linear Regression looked good).
- Evaluate Algorithms with Standardization (KNN looked good).
- Algorithm Tuning (`K=3` for KNN was best).
- Ensemble Methods (Bagging and Boosting, Gradient Boosting looked good).
- Tuning Ensemble Methods (getting the most from Gradient Boosting).
- Finalize Model (use all training data and confirm using validation dataset).

Working through this case study showed you how the recipes for specific machine learning tasks can be pulled together into a complete project. Working through this case study is good practice at applied machine learning using Python and scikit-learn.

20.12.1 Next Step

You have now completed two predictive modeling machine learning projects end-to-end. The first was a multiclass classification problem and this second project was a regression problem. Next is the third and final case study on a binary classification problem.

Chapter 21

Binary Classification Machine Learning Case Study Project

How do you work through a predictive modeling machine learning problem end-to-end? In this lesson you will work through a case study classification predictive modeling problem in Python including each step of the applied machine learning process. After completing this project, you will know:

- How to work through a classification predictive modeling problem end-to-end.
- How to use data transforms to improve model performance.
- How to use algorithm tuning to improve model performance.
- How to use ensemble methods and tuning of ensemble methods to improve model performance.

Let's get started.

21.1 Problem Definition

The focus of this project will be the Sonar Mines vs Rocks dataset[1]. The problem is to predict metal or rock objects from sonar return data. Each pattern is a set of 60 numbers in the range 0.0 to 1.0. Each number represents the energy within a particular frequency band, integrated over a certain period of time. The label associated with each record contains the letter `R` if the object is a rock and `M` if it is a mine (metal cylinder). The numbers in the labels are in increasing order of aspect angle, but they do not encode the angle directly.

21.2 Load the Dataset

Let's start off by loading the libraries required for this project.

[1] https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)

```
# Load libraries
import pandas
import numpy
import matplotlib.pyplot as plt
from pandas.tools.plotting import scatter_matrix
from sklearn.preprocessing import StandardScaler
from sklearn import cross_validation
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
```

Listing 21.1: Load libraries.

We can now load the dataset directly from the UCI Machine Learning repository website.

```
url = "https://goo.gl/NXoJfR"
dataset = pandas.read_csv(url, header=None)
```

Listing 21.2: Load the dataset.

You can see that we are not specifying the names of the attributes this time. This is because other than the class attribute (the last column), the variables do not have meaningful names. We also indicate that there is no header information, this is to avoid file loading code taking the first record as the column names. Now that we have the dataset loaded we can take a look at it.

21.3 Analyze Data

Let's take a closer look at our loaded data.

21.3.1 Descriptive Statistics

We will start off by confirming the dimensions of the dataset, e.g. the number of rows and columns.

```
# shape
print(dataset.shape)
```

Listing 21.3: Print the shape of the dataset.

We have 208 instances to work with and can confirm the data has 61 attributes including the class attribute.

```
(208, 61)
```

Listing 21.4: Output of shape of the dataset.

Let's also look at the data types of each attribute.

```
# types
pandas.set_option('display.max_rows', 500)
print(dataset.dtypes)
```

Listing 21.5: Print the data types of each attribute.

We can see that all of the attributes are numeric (float) and that the class value has been read in as an object.

```
0     float64
1     float64
2     float64
3     float64
4     float64
5     float64
6     float64
7     float64
8     float64
9     float64
10    float64
...
49    float64
50    float64
51    float64
52    float64
53    float64
54    float64
55    float64
56    float64
57    float64
58    float64
59    float64
60     object
```

Listing 21.6: Output of the data types for each attribute.

Let's now take a peek at the first 20 rows of the data.

```
# head
pandas.set_option('display.width', 100)
print(dataset.head(20))
```

Listing 21.7: Print the first few rows of the dataset.

This does not show all of the columns, but we can see all of the data has the same scale. We can also see that the class attribute (60) has string values.

```
      0      1      2      3      4      5      6      7      8      9  ...     51  \
0  0.0200 0.0371 0.0428 0.0207 0.0954 0.0986 0.1539 0.1601 0.3109 0.2111 ... 0.0027
1  0.0453 0.0523 0.0843 0.0689 0.1183 0.2583 0.2156 0.3481 0.3337 0.2872 ... 0.0084
2  0.0262 0.0582 0.1099 0.1083 0.0974 0.2280 0.2431 0.3771 0.5598 0.6194 ... 0.0232
3  0.0100 0.0171 0.0623 0.0205 0.0205 0.0368 0.1098 0.1276 0.0598 0.1264 ... 0.0121
```

```
4   0.0762 0.0666 0.0481 0.0394 0.0590 0.0649 0.1209 0.2467 0.3564 0.4459 ... 0.0031
5   0.0286 0.0453 0.0277 0.0174 0.0384 0.0990 0.1201 0.1833 0.2105 0.3039 ... 0.0045
6   0.0317 0.0956 0.1321 0.1408 0.1674 0.1710 0.0731 0.1401 0.2083 0.3513 ... 0.0201
7   0.0519 0.0548 0.0842 0.0319 0.1158 0.0922 0.1027 0.0613 0.1465 0.2838 ... 0.0081
8   0.0223 0.0375 0.0484 0.0475 0.0647 0.0591 0.0753 0.0098 0.0684 0.1487 ... 0.0145
9   0.0164 0.0173 0.0347 0.0070 0.0187 0.0671 0.1056 0.0697 0.0962 0.0251 ... 0.0090
10  0.0039 0.0063 0.0152 0.0336 0.0310 0.0284 0.0396 0.0272 0.0323 0.0452 ... 0.0062
11  0.0123 0.0309 0.0169 0.0313 0.0358 0.0102 0.0182 0.0579 0.1122 0.0835 ... 0.0133
12  0.0079 0.0086 0.0055 0.0250 0.0344 0.0546 0.0528 0.0958 0.1009 0.1240 ... 0.0176
13  0.0090 0.0062 0.0253 0.0489 0.1197 0.1589 0.1392 0.0987 0.0955 0.1895 ... 0.0059
14  0.0124 0.0433 0.0604 0.0449 0.0597 0.0355 0.0531 0.0343 0.1052 0.2120 ... 0.0083
15  0.0298 0.0615 0.0650 0.0921 0.1615 0.2294 0.2176 0.2033 0.1459 0.0852 ... 0.0031
16  0.0352 0.0116 0.0191 0.0469 0.0737 0.1185 0.1683 0.1541 0.1466 0.2912 ... 0.0346
17  0.0192 0.0607 0.0378 0.0774 0.1388 0.0809 0.0568 0.0219 0.1037 0.1186 ... 0.0331
18  0.0270 0.0092 0.0145 0.0278 0.0412 0.0757 0.1026 0.1138 0.0794 0.1520 ... 0.0084
19  0.0126 0.0149 0.0641 0.1732 0.2565 0.2559 0.2947 0.4110 0.4983 0.5920 ... 0.0092

       52     53     54     55     56     57     58     59 60
0   0.0065 0.0159 0.0072 0.0167 0.0180 0.0084 0.0090 0.0032  R
1   0.0089 0.0048 0.0094 0.0191 0.0140 0.0049 0.0052 0.0044  R
2   0.0166 0.0095 0.0180 0.0244 0.0316 0.0164 0.0095 0.0078  R
3   0.0036 0.0150 0.0085 0.0073 0.0050 0.0044 0.0040 0.0117  R
4   0.0054 0.0105 0.0110 0.0015 0.0072 0.0048 0.0107 0.0094  R
5   0.0014 0.0038 0.0013 0.0089 0.0057 0.0027 0.0051 0.0062  R
6   0.0248 0.0131 0.0070 0.0138 0.0092 0.0143 0.0036 0.0103  R
7   0.0120 0.0045 0.0121 0.0097 0.0085 0.0047 0.0048 0.0053  R
8   0.0128 0.0145 0.0058 0.0049 0.0065 0.0093 0.0059 0.0022  R
9   0.0223 0.0179 0.0084 0.0068 0.0032 0.0035 0.0056 0.0040  R
10  0.0120 0.0052 0.0056 0.0093 0.0042 0.0003 0.0053 0.0036  R
11  0.0265 0.0224 0.0074 0.0118 0.0026 0.0092 0.0009 0.0044  R
12  0.0127 0.0088 0.0098 0.0019 0.0059 0.0058 0.0059 0.0032  R
13  0.0095 0.0194 0.0080 0.0152 0.0158 0.0053 0.0189 0.0102  R
14  0.0057 0.0174 0.0188 0.0054 0.0114 0.0196 0.0147 0.0062  R
15  0.0153 0.0071 0.0212 0.0076 0.0152 0.0049 0.0200 0.0073  R
16  0.0158 0.0154 0.0109 0.0048 0.0095 0.0015 0.0073 0.0067  R
17  0.0131 0.0120 0.0108 0.0024 0.0045 0.0037 0.0112 0.0075  R
18  0.0010 0.0018 0.0068 0.0039 0.0120 0.0132 0.0070 0.0088  R
19  0.0035 0.0098 0.0121 0.0006 0.0181 0.0094 0.0116 0.0063  R
```

Listing 21.8: Output of the first few rows of the dataset.

Let's summarize the distribution of each attribute.

```
# descriptions, change precision to 3 places
pandas.set_option('precision', 3)
print(dataset.describe())
```

Listing 21.9: Print the statistical descriptions of the dataset.

Again, as we expect, the data has the same range, but interestingly differing mean values. There may be some benefit from standardizing the data.

```
            0          1          2          3          4          5          6          7          8          9  \
count 208.000 2.080e+02 208.000 208.000 208.000 208.000 208.000 208.000 208.000 208.000
mean    0.029 3.844e-02   0.044   0.054   0.075   0.105   0.122   0.135   0.178   0.208
std     0.023 3.296e-02   0.038   0.047   0.056   0.059   0.062   0.085   0.118   0.134
min     0.002 6.000e-04   0.002   0.006   0.007   0.010   0.003   0.005   0.007   0.011
```

```
25%     0.013  1.645e-02   0.019   0.024   0.038   0.067   0.081   0.080   0.097   0.111
50%     0.023  3.080e-02   0.034   0.044   0.062   0.092   0.107   0.112   0.152   0.182
75%     0.036  4.795e-02   0.058   0.065   0.100   0.134   0.154   0.170   0.233   0.269
max     0.137  2.339e-01   0.306   0.426   0.401   0.382   0.373   0.459   0.683   0.711

        ...          50        51        52      53        54        55        56  \
count   ...     208.000  2.080e+02 2.080e+02 208.000 2.080e+02 2.080e+02 2.080e+02
mean    ...       0.016  1.342e-02 1.071e-02   0.011 9.290e-03 8.222e-03 7.820e-03
std     ...       0.012  9.634e-03 7.060e-03   0.007 7.088e-03 5.736e-03 5.785e-03
min     ...       0.000  8.000e-04 5.000e-04   0.001 6.000e-04 4.000e-04 3.000e-04
25%     ...       0.008  7.275e-03 5.075e-03   0.005 4.150e-03 4.400e-03 3.700e-03
50%     ...       0.014  1.140e-02 9.550e-03   0.009 7.500e-03 6.850e-03 5.950e-03
75%     ...       0.021  1.673e-02 1.490e-02   0.015 1.210e-02 1.058e-02 1.043e-02
max     ...       0.100  7.090e-02 3.900e-02   0.035 4.470e-02 3.940e-02 3.550e-02

             57        58        59
count 2.080e+02 2.080e+02 2.080e+02
mean  7.949e-03 7.941e-03 6.507e-03
std   6.470e-03 6.181e-03 5.031e-03
min   3.000e-04 1.000e-04 6.000e-04
25%   3.600e-03 3.675e-03 3.100e-03
50%   5.800e-03 6.400e-03 5.300e-03
75%   1.035e-02 1.033e-02 8.525e-03
max   4.400e-02 3.640e-02 4.390e-02
```

Listing 21.10: Output of the statistical descriptions of the dataset.

Let's take a quick look at the breakdown of class values.

```
# class distribution
print(dataset.groupby(60).size())
```

Listing 21.11: Print the class breakdown of the dataset.

We can see that the classes are reasonably balanced between `M` (mines) and `R` (rocks).

```
M    111
R     97
```

Listing 21.12: Output of the class breakdown of the dataset.

21.3.2 Unimodal Data Visualizations

Let's look at visualizations of individual attributes. It is often useful to look at your data using multiple different visualizations in order to spark ideas. Let's look at histograms of each attribute to get a sense of the data distributions.

```
# histograms
dataset.hist(sharex=False, sharey=False, xlabelsize=1, ylabelsize=1)
plt.show()
```

Listing 21.13: Visualize the dataset with Histogram Plots.

We can see that there are a lot of Gaussian-like distributions and perhaps some exponential-like distributions for other attributes.

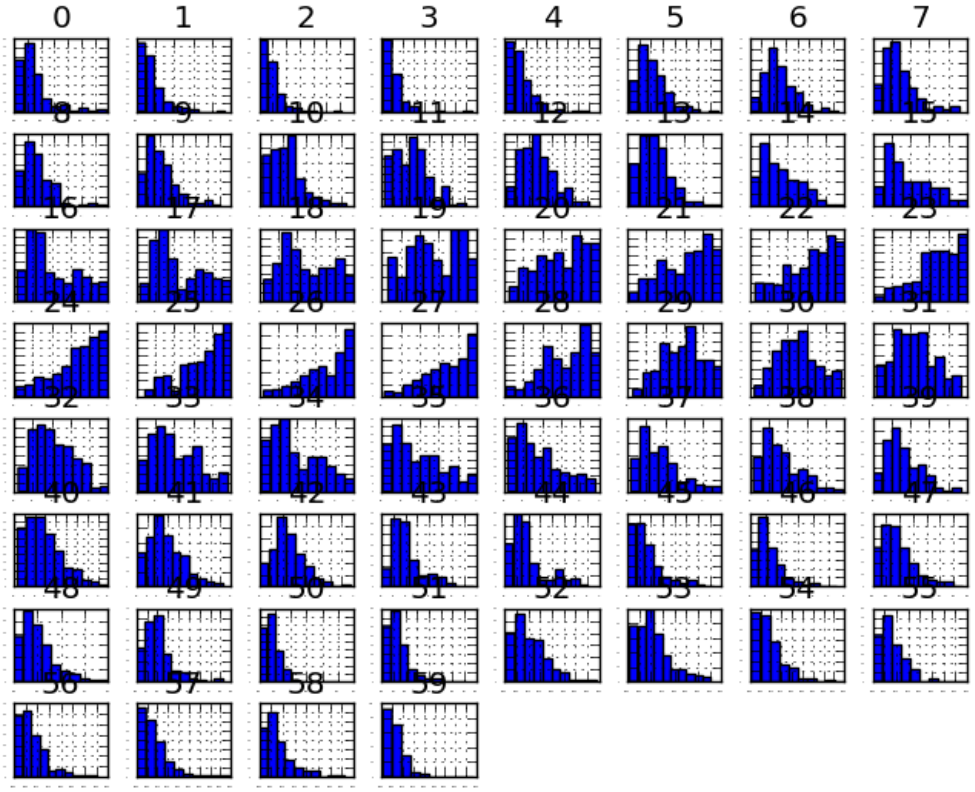

Figure 21.1: Histogram Plots of Attributes from the Dataset.

Let's take a look at the same perspective of the data using density plots.

```
# density
dataset.plot(kind='density', subplots=True, layout=(8,8), sharex=False, legend=False,
    fontsize=1)
plt.show()
```

Listing 21.14: Visualize the dataset with Density Plots.

This is useful, you can see that many of the attributes have a skewed distribution. A power transform like a Box-Cox transform that can correct for the skew in distributions might be useful.

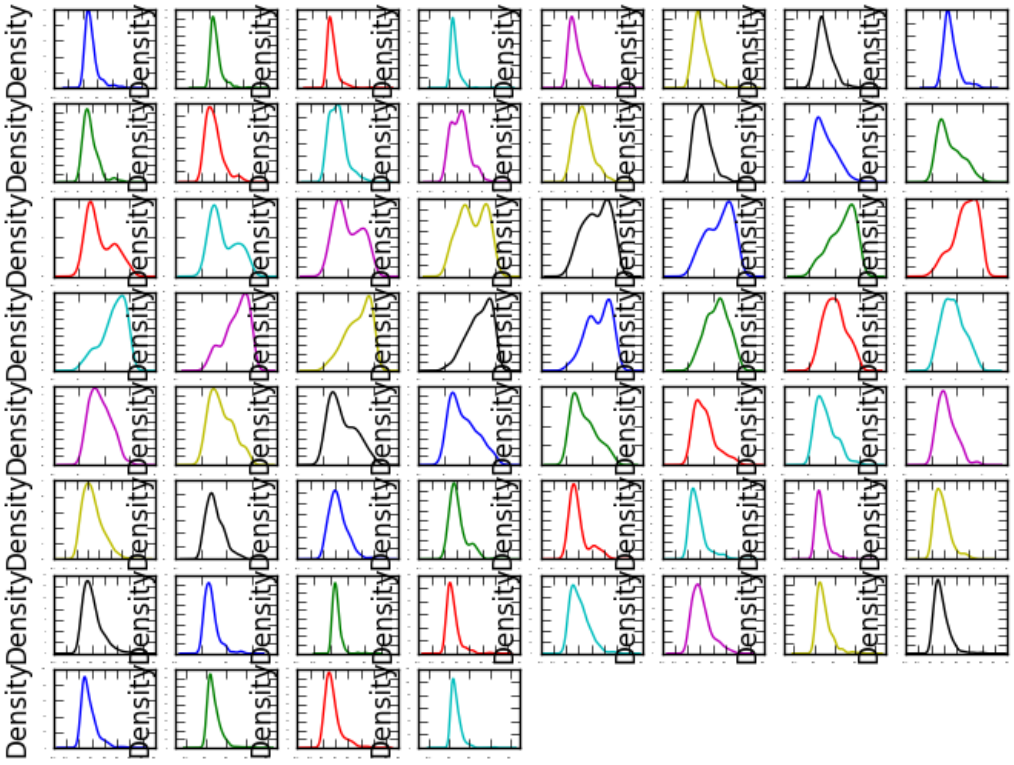

Figure 21.2: Density Plots of Attributes from the Dataset.

It is always good to look at box and whisker plots of numeric attributes to get an idea of the spread of values.

```
# box and whisker plots
dataset.plot(kind='box', subplots=True, layout=(8,8), sharex=False, sharey=False,
    fontsize=1)
plt.show()
```

Listing 21.15: Visualize the dataset with Box and Whisker Plots.

We can see that attributes do have quite different spreads. Given the scales are the same, it may suggest some benefit in standardizing the data for modeling to get all of the means lined up.

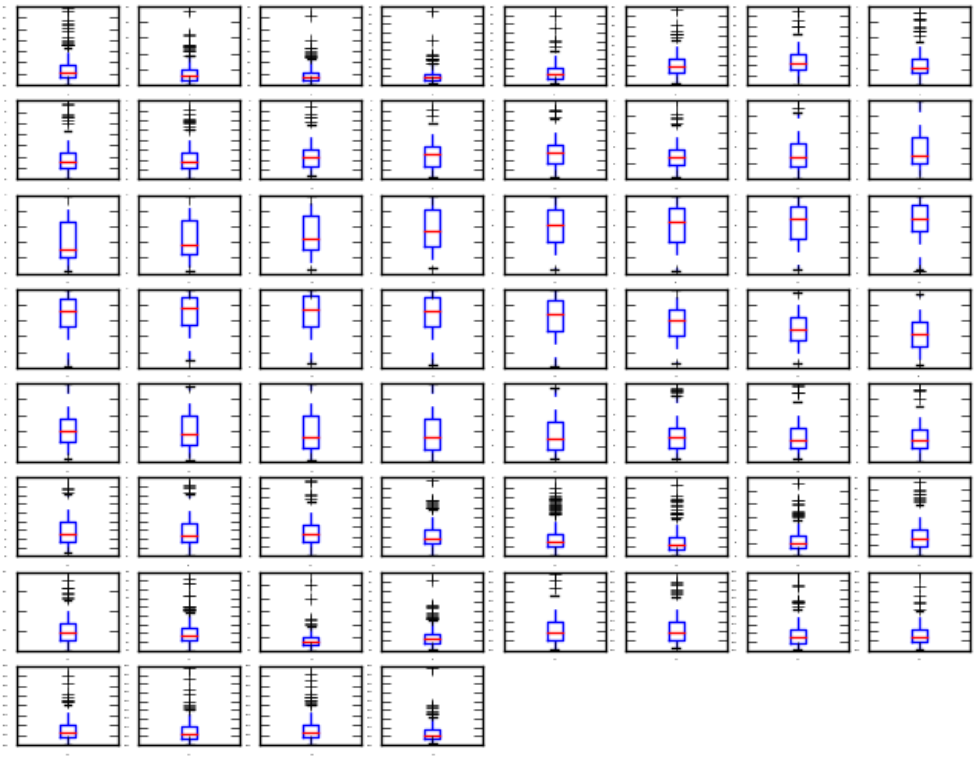

Figure 21.3: Box and Whisker Plots of Attributes from the Dataset.

21.3.3 Multimodal Data Visualizations

Let's visualize the correlations between the attributes.

```
# correlation matrix
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(dataset.corr(), vmin=-1, vmax=1, interpolation='none')
fig.colorbar(cax)
plt.show()
```

Listing 21.16: Visualize the correlations between attributes.

It looks like there is also some structure in the order of the attributes. The red around the diagonal suggests that attributes that are next to each other are generally more correlated with each other. The blue patches also suggest some moderate negative correlation the further attributes are away from each other in the ordering. This makes sense if the order of the attributes refers to the angle of sensors for the sonar chirp.

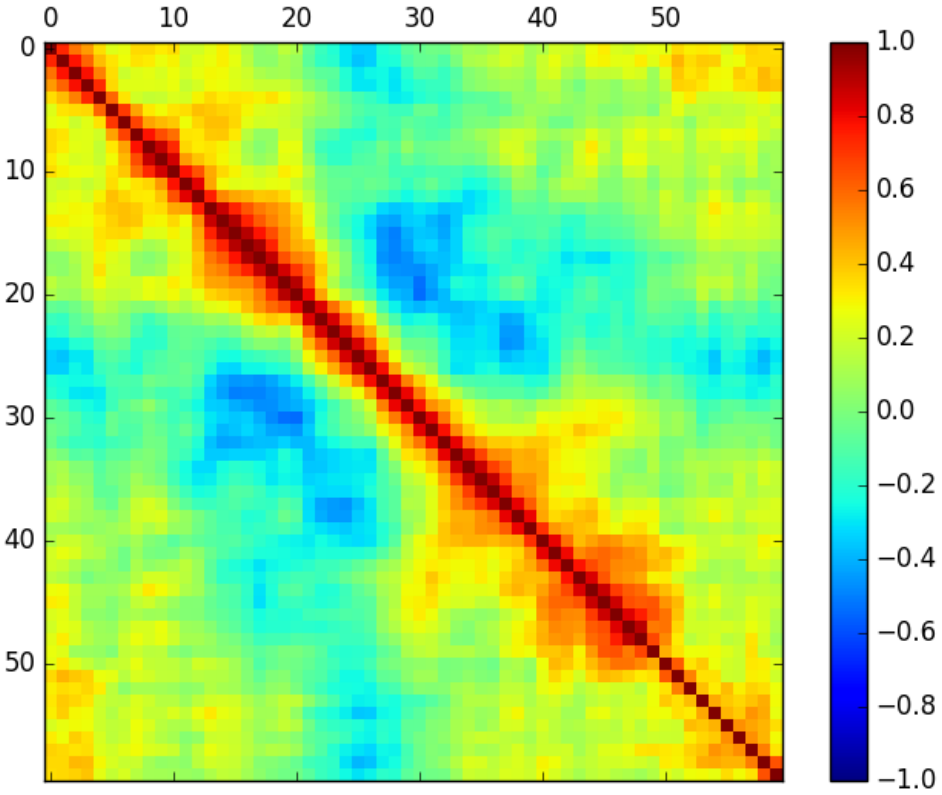

Figure 21.4: Plot of Correlations Between Attributes from the Dataset.

21.4 Validation Dataset

It is a good idea to use a validation hold-out set. This is a sample of the data that we hold back from our analysis and modeling. We use it right at the end of our project to confirm the accuracy of our final model. It is a smoke test that we can use to see if we messed up and to give us confidence on our estimates of accuracy on unseen data. We will use 80% of the dataset for modeling and hold back 20% for validation.

```
# Split-out validation dataset
array = dataset.values
X = array[:,0:60].astype(float)
Y = array[:,60]
validation_size = 0.20
seed = 7
X_train, X_validation, Y_train, Y_validation = cross_validation.train_test_split(X, Y,
    test_size=validation_size, random_state=seed)
```

Listing 21.17: Create Separate Training and Validation Datasets.

21.5 Evaluate Algorithms: Baseline

We don't know what algorithms will do well on this dataset. Gut feel suggests distance based algorithms like k-Nearest Neighbors and Support Vector Machines may do well. Let's design our test harness. We will use 10-fold cross validation. The dataset is not too small and this is a good standard test harness configuration. We will evaluate algorithms using the `accuracy` metric. This is a gross metric that will give a quick idea of how correct a given model is. More useful on binary classification problems like this one.

```
# Test options and evaluation metric
num_folds = 10
num_instances = len(X_train)
seed = 7
scoring = 'accuracy'
```

Listing 21.18: Prepare the Test Harness for Evaluating Algorithms.

Let's create a baseline of performance on this problem and spot-check a number of different algorithms. We will select a suite of different algorithms capable of working on this classification problem. The six algorithms selected include:

- **Linear Algorithms**: Logistic Regression (LR) and Linear Discriminant Analysis (LDA).
- **Nonlinear Algorithms**: Classification and Regression Trees (CART), Support Vector Machines (SVM), Gaussian Naive Bayes (NB) and k-Nearest Neighbors (KNN).

```
# Spot-Check Algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
```

Listing 21.19: Prepare Algorithms to Evaluate.

The algorithms all use default tuning parameters. Let's compare the algorithms. We will display the mean and standard deviation of accuracy for each algorithm as we calculate it and collect the results for use later.

```
results = []
names = []
for name, model in models:
  kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
  cv_results = cross_validation.cross_val_score(model, X_train, Y_train, cv=kfold,
      scoring=scoring)
  results.append(cv_results)
  names.append(name)
  msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
  print(msg)
```

Listing 21.20: Evaluate Algorithms Using the Test Harness.

Running the example provides the output below. The results suggest That both Logistic Regression and k-Nearest Neighbors may be worth further study.

```
LR: 0.782721 (0.093796)
LDA: 0.746324 (0.117854)
KNN: 0.808088 (0.067507)
CART: 0.740809 (0.118120)
NB: 0.648897 (0.141868)
SVM: 0.608824 (0.118656)
```

Listing 21.21: Output of Evaluating Algorithms.

These are just mean accuracy values. It is always wise to look at the distribution of accuracy values calculated across cross validation folds. We can do that graphically using box and whisker plots.

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Listing 21.22: Visualization of the Distribution of Algorithm Performance.

The results show a tight distribution for KNN which is encouraging, suggesting low variance. The poor results for SVM are surprising.

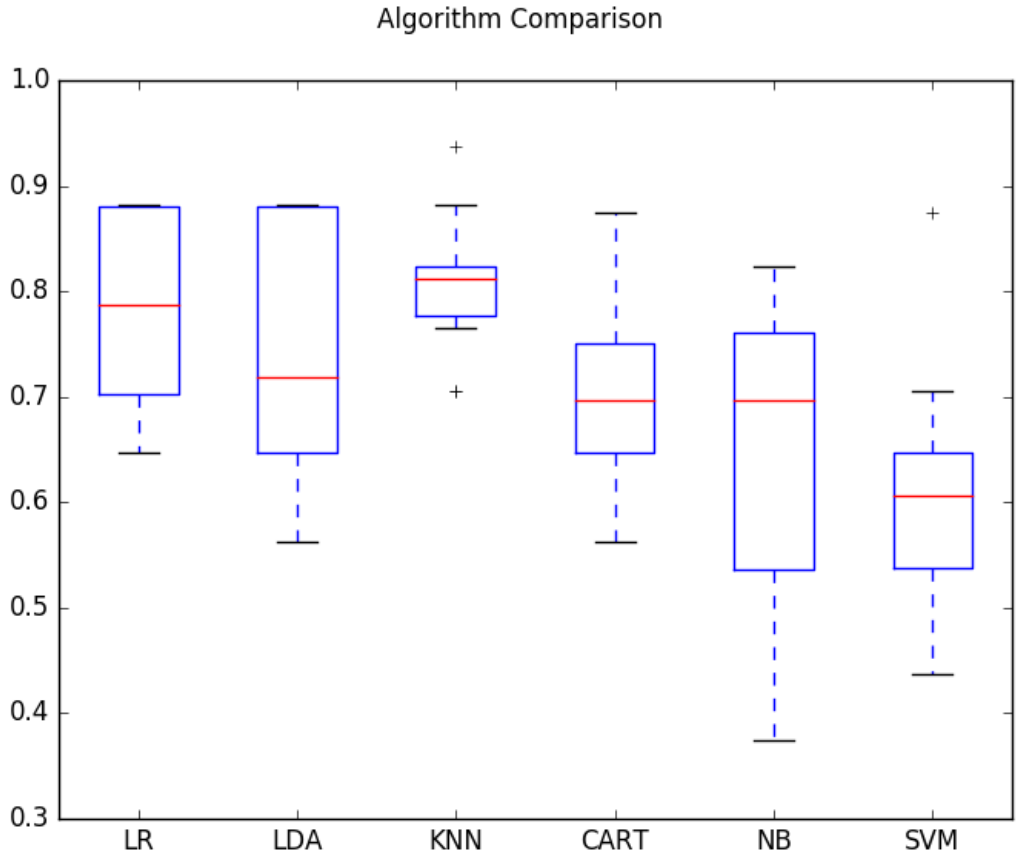

Figure 21.5: Box and Whisker Plots of Algorithm Performance.

It is possible that the varied distribution of the attributes is having an effect on the accuracy of algorithms such as SVM. In the next section we will repeat this spot-check with a standardized copy of the training dataset.

21.6 Evaluate Algorithms: Standardize Data

We suspect that the differing distributions of the raw data may be negatively impacting the skill of some of the algorithms. Let's evaluate the same algorithms with a standardized copy of the dataset. This is where the data is transformed such that each attribute has a mean value of zero and a standard deviation of one. We also need to avoid data leakage when we transform the data. A good way to avoid leakage is to use pipelines that standardize the data and build the model for each fold in the cross validation test harness. That way we can get a fair estimation of how each model with standardized data might perform on unseen data.

```
# Standardize the dataset
pipelines = []
pipelines.append(('ScaledLR', Pipeline([('Scaler', StandardScaler()),('LR',
    LogisticRegression())])))
pipelines.append(('ScaledLDA', Pipeline([('Scaler', StandardScaler()),('LDA',
    LinearDiscriminantAnalysis())])))
```

```
pipelines.append(('ScaledKNN', Pipeline([('Scaler', StandardScaler()),('KNN',
    KNeighborsClassifier())])))
pipelines.append(('ScaledCART', Pipeline([('Scaler', StandardScaler()),('CART',
    DecisionTreeClassifier())])))
pipelines.append(('ScaledNB', Pipeline([('Scaler', StandardScaler()),('NB',
    GaussianNB())])))
pipelines.append(('ScaledSVM', Pipeline([('Scaler', StandardScaler()),('SVM', SVC())])))
results = []
names = []
for name, model in pipelines:
  kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
  cv_results = cross_validation.cross_val_score(model, X_train, Y_train, cv=kfold,
      scoring=scoring)
  results.append(cv_results)
  names.append(name)
  msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
  print(msg)
```

Listing 21.23: Evaluate Algorithms on a Scaled Dataset.

Running the example provides the results listed below. We can see that KNN is still doing well, even better than before. We can also see that the standardization of the data has lifted the skill of SVM to be the most accurate algorithm tested so far.

```
ScaledLR: 0.734191 (0.095885)
ScaledLDA: 0.746324 (0.117854)
ScaledKNN: 0.825735 (0.054511)
ScaledCART: 0.711765 (0.107567)
ScaledNB: 0.648897 (0.141868)
ScaledSVM: 0.836397 (0.088697)
```

Listing 21.24: Output of Evaluating Algorithms on the Scaled Dataset.

Again, we should plot the distribution of the accuracy scores using box and whisker plots.

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Scaled Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Listing 21.25: Visualization of the Distribution of Algorithm Performance on the Scaled Dataset.

The results suggest digging deeper into the SVM and KNN algorithms. It is very likely that configuration beyond the default may yield even more accurate models.

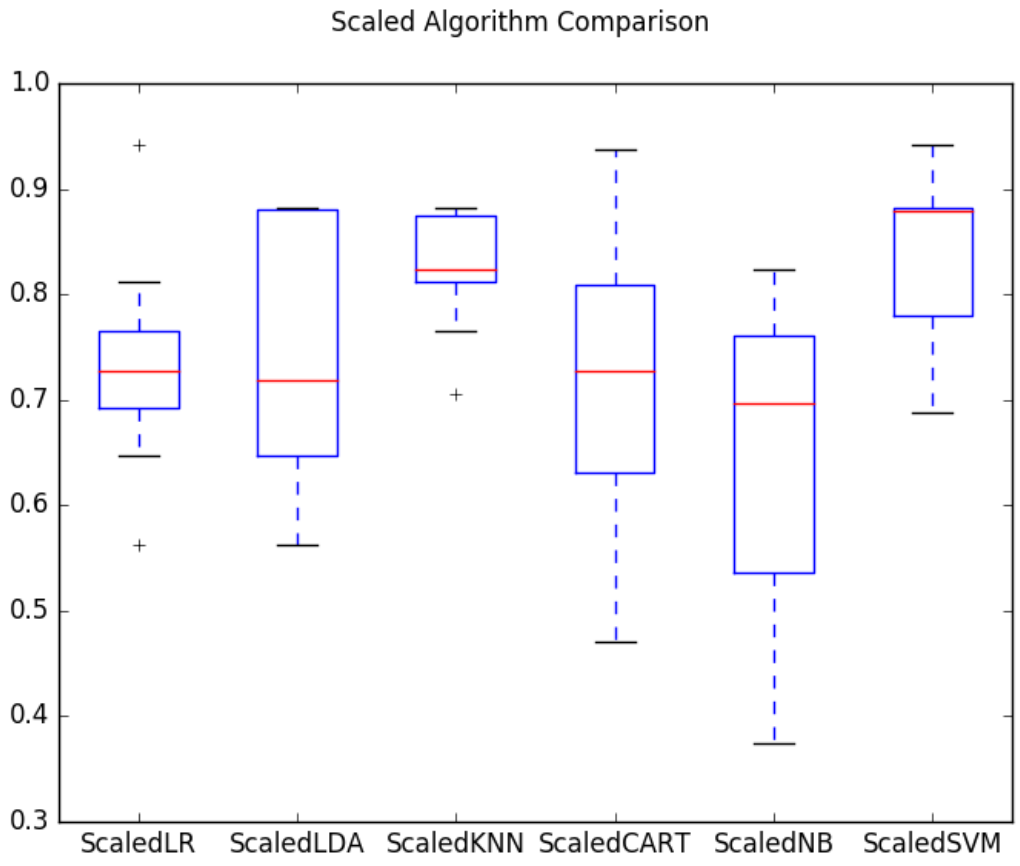

Figure 21.6: Box and Whisker Plots of Algorithm Performance on the Standardized Dataset.

21.7 Algorithm Tuning

In this section we investigate tuning the parameters for two algorithms that show promise from the spot-checking in the previous section: KNN and SVM.

21.7.1 Tuning KNN

We can start off by tuning the number of neighbors for KNN. The default number of neighbors is 7. Below we try all odd values of k from 1 to 21, covering the default value of 7. Each k value is evaluated using 10-fold cross validation on the training standardized dataset.

```
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
neighbors = [1,3,5,7,9,11,13,15,17,19,21]
param_grid = dict(n_neighbors=neighbors)
model = KNeighborsClassifier()
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, cv=kfold)
grid_result = grid.fit(rescaledX, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
```

```
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

Listing 21.26: Tune the KNN Algorithm on the Scaled Dataset.

We can print out configuration that resulted in the highest accuracy as well as the accuracy of all values tried. Running the example we see the results below.

```
Best: 0.849398 using {'n_neighbors': 1}

0.850000 (0.059686) with: {'n_neighbors': 1}
0.837132 (0.066014) with: {'n_neighbors': 3}
0.837500 (0.037377) with: {'n_neighbors': 5}
0.763971 (0.089374) with: {'n_neighbors': 7}
0.751471 (0.087051) with: {'n_neighbors': 9}
0.733456 (0.104831) with: {'n_neighbors': 11}
0.733088 (0.105806) with: {'n_neighbors': 13}
0.727941 (0.076148) with: {'n_neighbors': 15}
0.709926 (0.079287) with: {'n_neighbors': 17}
0.722059 (0.085088) with: {'n_neighbors': 19}
0.710294 (0.109505) with: {'n_neighbors': 21}
```

Listing 21.27: Results of Tuning KNN on the Scaled Dataset.

We can see that the optimal configuration is `K=1`. This is interesting as the algorithm will make predictions using the most similar instance in the training dataset alone.

21.7.2 Tuning SVM

We can tune two key parameters of the SVM algorithm, the value of C (how much to relax the margin) and the type of `kernel`. The default for SVM (the SVC class) is to use the Radial Basis Function (RBF) kernel with a C value set to 1.0. Like with KNN, we will perform a grid search using 10-fold cross validation with a standardized copy of the training dataset. We will try a number of simpler `kernel` types and C values with less bias and more bias (less than and more than 1.0 respectively).

```
# Tune scaled SVM
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
c_values = [0.1, 0.3, 0.5, 0.7, 0.9, 1.0, 1.3, 1.5, 1.7, 2.0]
kernel_values = ['linear', 'poly', 'rbf', 'sigmoid']
param_grid = dict(C=c_values, kernel=kernel_values)
model = SVC()
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring=scoring, cv=kfold)
grid_result = grid.fit(rescaledX, Y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

Listing 21.28: Tune the SVM Algorithm on the Scaled Dataset.

Running the example prints out the best configuration, the accuracy as well as the accuracies for all configuration combinations.

```
Best: 0.867470 using {'kernel': 'rbf', 'C': 1.5}
```

```
0.758456 (0.099483) with: {'kernel': 'linear', 'C': 0.1}
0.529412 (0.118825) with: {'kernel': 'poly', 'C': 0.1}
0.573162 (0.130930) with: {'kernel': 'rbf', 'C': 0.1}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 0.1}
0.746324 (0.109507) with: {'kernel': 'linear', 'C': 0.3}
0.642647 (0.132187) with: {'kernel': 'poly', 'C': 0.3}
0.765809 (0.091692) with: {'kernel': 'rbf', 'C': 0.3}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 0.3}
0.740074 (0.082636) with: {'kernel': 'linear', 'C': 0.5}
0.680147 (0.098595) with: {'kernel': 'poly', 'C': 0.5}
0.788235 (0.064190) with: {'kernel': 'rbf', 'C': 0.5}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 0.5}
0.746691 (0.084198) with: {'kernel': 'linear', 'C': 0.7}
0.740074 (0.127908) with: {'kernel': 'poly', 'C': 0.7}
0.812500 (0.085513) with: {'kernel': 'rbf', 'C': 0.7}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 0.7}
0.758824 (0.096520) with: {'kernel': 'linear', 'C': 0.9}
0.770221 (0.102510) with: {'kernel': 'poly', 'C': 0.9}
0.836397 (0.088697) with: {'kernel': 'rbf', 'C': 0.9}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 0.9}
0.752574 (0.098883) with: {'kernel': 'linear', 'C': 1.0}
0.788235 (0.108418) with: {'kernel': 'poly', 'C': 1.0}
0.836397 (0.088697) with: {'kernel': 'rbf', 'C': 1.0}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 1.0}
0.769853 (0.106086) with: {'kernel': 'linear', 'C': 1.3}
0.818382 (0.107151) with: {'kernel': 'poly', 'C': 1.3}
0.848162 (0.080414) with: {'kernel': 'rbf', 'C': 1.3}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 1.3}
0.758088 (0.092026) with: {'kernel': 'linear', 'C': 1.5}
0.830147 (0.110255) with: {'kernel': 'poly', 'C': 1.5}
0.866176 (0.091458) with: {'kernel': 'rbf', 'C': 1.5}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 1.5}
0.746324 (0.090414) with: {'kernel': 'linear', 'C': 1.7}
0.830515 (0.116706) with: {'kernel': 'poly', 'C': 1.7}
0.860294 (0.088281) with: {'kernel': 'rbf', 'C': 1.7}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 1.7}
0.758456 (0.094064) with: {'kernel': 'linear', 'C': 2.0}
0.830882 (0.108950) with: {'kernel': 'poly', 'C': 2.0}
0.866176 (0.095166) with: {'kernel': 'rbf', 'C': 2.0}
0.409559 (0.073625) with: {'kernel': 'sigmoid', 'C': 2.0}
```

Listing 21.29: Results of Tuning SVM on the Scaled Dataset.

We can see the most accurate configuration was SVM with an RBF `kernel` and a C value of 1.5. The accuracy 86.7470% is seemingly better than what KNN could achieve.

21.8 Ensemble Methods

Another way that we can improve the performance of algorithms on this problem is by using ensemble methods. In this section we will evaluate four different ensemble machine learning algorithms, two boosting and two bagging methods:

- **Boosting Methods**: AdaBoost (AB) and Gradient Boosting (GBM).

- **Bagging Methods**: Random Forests (RF) and Extra Trees (ET).

We will use the same test harness as before, 10-fold cross validation. No data standardization is used in this case because all four ensemble algorithms are based on decision trees that are less sensitive to data distributions.

```
# ensembles
ensembles = []
ensembles.append(('AB', AdaBoostClassifier()))
ensembles.append(('GBM', GradientBoostingClassifier()))
ensembles.append(('RF', RandomForestClassifier()))
ensembles.append(('ET', ExtraTreesClassifier()))
results = []
names = []
for name, model in ensembles:
  kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds, random_state=seed)
  cv_results = cross_validation.cross_val_score(model, X_train, Y_train, cv=kfold,
      scoring=scoring)
  results.append(cv_results)
  names.append(name)
  msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
  print(msg)
```

Listing 21.30: Evaluate Ensemble Algorithms.

Running the example provides the following accuracy scores.

```
AB: 0.819853 (0.058293)
GBM: 0.829044 (0.143517)
RF: 0.765074 (0.107129)
ET: 0.794485 (0.087874)
```

Listing 21.31: Output of Evaluate Ensemble Algorithms.

We can see that both boosting techniques provide strong accuracy scores in the low 80s (%) with default configurations. We can plot the distribution of accuracy scores across the cross validation folds.

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle('Ensemble Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

Listing 21.32: Visualize the Distribution of Ensemble Algorithm Performance.

The results suggest GBM may be worthy of further study, with a strong mean and a spread that skews up towards high 90s (%) in accuracy.

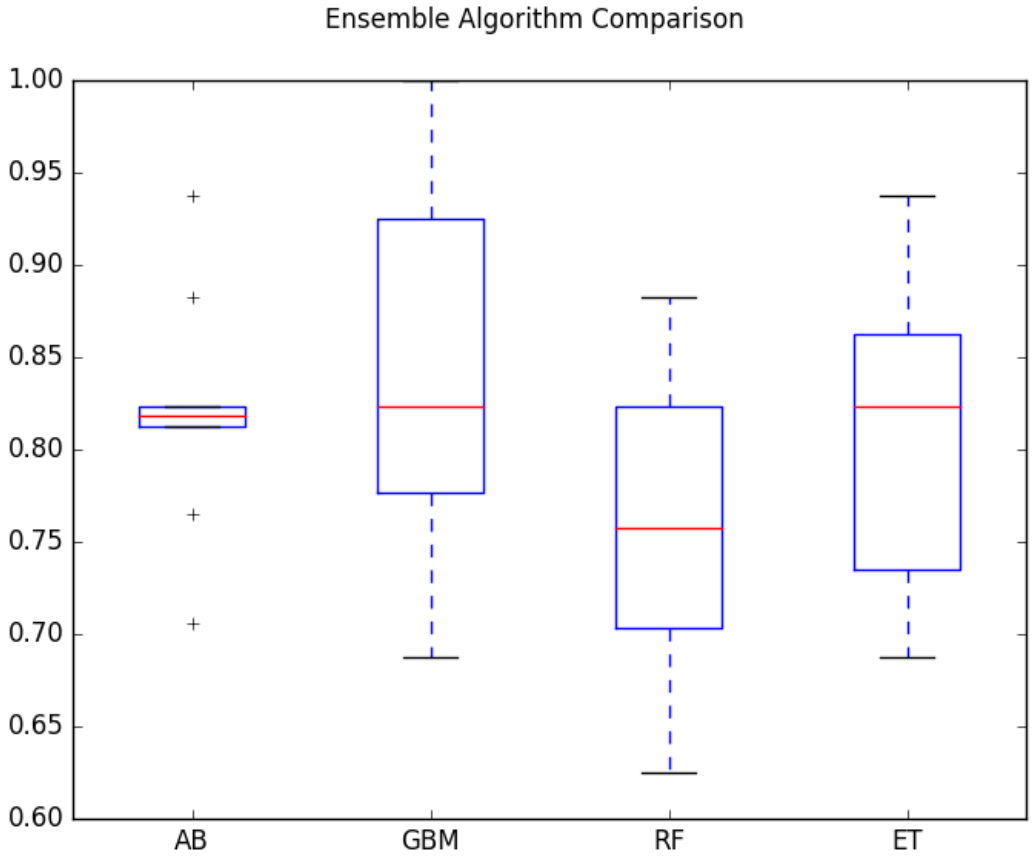

Figure 21.7: Box and Whisker Plots of Ensemble Performance.

21.9 Finalize Model

The SVM showed the most promise as a low complexity and stable model for this problem. In this section we will finalize the model by training it on the entire training dataset and make predictions for the hold-out validation dataset to confirm our findings. A part of the findings was that SVM performs better when the dataset is standardized so that all attributes have a mean value of zero and a standard deviation of one. We can calculate this from the entire training dataset and apply the same transform to the input attributes from the validation dataset.

```
# prepare the model
scaler = StandardScaler().fit(X_train)
rescaledX = scaler.transform(X_train)
model = SVC(C=1.5)
model.fit(rescaledX, Y_train)
# estimate accuracy on validation dataset
rescaledValidationX = scaler.transform(X_validation)
predictions = model.predict(rescaledValidationX)
print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

Listing 21.33: Evaluate SVM on the Validation Dataset.

We can see that we achieve an accuracy of nearly 86% on the held-out validation dataset. A score that matches closely to our expectations estimated above during the tuning of SVM.

```
0.857142857143

[[23  4]
 [ 2 13]]

             precision   recall f1-score  support

          M       0.92     0.85     0.88       27
          R       0.76     0.87     0.81       15

avg / total       0.86     0.86     0.86       42
```

Listing 21.34: Output of Evaluating SVM on the Validation Dataset.

21.10 Summary

In this chapter you worked through a classification predictive modeling machine learning problem from end-to-end using Python. Specifically, the steps covered were:

- Problem Definition (Sonar return data).
- Loading the Dataset.
- Analyze Data (same scale but different distributions of data).
- Evaluate Algorithms (KNN looked good).
- Evaluate Algorithms with Standardization (KNN and SVM looked good).
- Algorithm Tuning (`K=1` for KNN was good, SVM with an RBF kernel and `C=1.5` was best).
- Ensemble Methods (Bagging and Boosting, not quite as good as SVM).
- Finalize Model (use all training data and confirm using validation dataset).

Working through this case study showed you how the recipes for specific machine learning tasks can be pulled together into a complete project. Working through this case study is good practice at applied machine learning using Python.

21.10.1 Next Step

This was the third and final predictive modeling project case study. Well done! You now have experience and skills in working through predictive modeling machine learning projects end-to-end. In the next section you will discover ideas for additional small case study projects that you could work on for further practice.

Chapter 22

More Predictive Modeling Projects

You can now work through predictive modeling machine learning projects using Python. Now what? In this chapter, we look at ways that you can practice and refine your new found skills.

22.1 Build And Maintain Recipes

Throughout this book you have worked through many machine learning lessons using Python. Taken together, this is the start of your own private code base that you can use to jump-start your current or next machine learning project. These recipes are a beginning, not an end. The larger and more sophisticated that your catalog of machine learning recipes becomes, the faster you can get started on new projects and the more accurate the models that you can develop.

As you apply your machine learning skills using the Python platform, you will develop experience and skills with new and different techniques with Python. You can pull out or abstract snippets and recipes as you go along and add them to your own collection of recipes, building upon the code that you can use on future machine learning projects. With time, you will amass your own mature and highly-tailored catalog of machine learning code for Python.

22.2 Small Projects on Small Datasets

Keep practicing your skills using Python. Datasets from the UCI Machine Learning Repository[1] were used throughout this book to demonstrate how to achieve specific tasks in a machine learning project. They were also used in the longer case study projects. They are standardized, relatively clean, well understood and excellent for you to use as practice datasets.

You can use the datasets on the UCI Machine Learning repository as the focus of small (5-to-10 hours of effort) focused machine learning projects using the Python platform. Once completed, you can write-up your findings and share them online as part of your expanding portfolio of machine learning projects.

These can be used by you later as a repository of knowledge on which you can build and further develop your skills. They can also be used to demonstrate to managers or future employers that you are capable of delivering results on predictive modeling machine learning projects using the Python platform. Here is a process that you can use to practice machine learning on Python:

[1] http://archive.ics.uci.edu/ml/

1. Browse the list of free datasets on the repository and download some that look interesting to you.

2. Use the project template and recipes in this book to work through the dataset and develop an accurate model.

3. Write up your work-flow and findings in a way that you can refer to them later or perhaps share it publicly on a website.

Keep the project short, limit your projects to 5-to-10 hours, say a week worth of nights and spare time.

22.3 Competitive Machine Learning

Use competitive machine learning to push your skills. Working on small projects in the previous section is a good way to practice the fundamentals. At some point the problems will become easy for you. You also need to be pushed out of your comfort zone to help you grow your skills further.

An excellent way to develop your machine learning skills with Python further is to start participating in competitions. In a competition, the organizer provides you with a training dataset, a test dataset on which you are to make predictions, a performance measure and a time limit. You and your competitors then work to create the most accurate model possible. Winners often get prize money.

These competitions often last weeks to months and can be a lot of fun. They also offer a great opportunity to test your skills with machine learning tools on datasets that often require a lot of cleaning and preparation. The premier website for machine learning competitions is Kaggle[2].

Competitions are stratified into different classes such as research, recruitment and 101 for beginners. A good place to start would be the beginner competitions as they are often less challenging and have a lot of help in the form of tutorials to get you started.

22.4 Summary

In this chapter you have discovered three areas where you could practice your new found machine learning skills with Python. They were:

1. To continue to build up and maintain your catalog of machine learning recipes starting with the catalog of recipes provided as a bonus with this book.

2. To continue to work on the standard machine learning datasets on the UCI Machine Learning Repository.

3. To start work through the larger datasets from competitive machine learning and even start participating in machine learning competitions.

[2] https://www.kaggle.com

22.4.1 Next Step

This concludes Part III of this book on machine learning projects. Up next we finish off the book with a summary of how far you have come and where you can look if you need additional help with Python.

Part IV

Conclusions

Chapter 23

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come.

1. You started off with an interest in machine learning and a strong desire to be able to practice and apply machine learning using Python.
2. You downloaded, installed and started Python, perhaps for the first time, and started to get familiar with the syntax of the language and the SciPy ecosystem.
3. Slowly and steadily over the course of a number of lessons you learned how the standard tasks of a predictive modeling machine learning project map onto the Python platform.
4. Building upon the recipes for common machine learning tasks you worked through your first machine learning problems end-to-end using Python.
5. Using a standard template, the recipes and experience you have gathered, you are now capable of working through new and different predictive modeling machine learning problems on your own.

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 work through machine learning problems end-to-end using Python. This is a platform that is used by a majority of working data scientist professionals. The sky is the limit for you.

I want to take a moment and sincerely thank you for letting me help you start your machine learning journey with Python. I hope you keep learning and have fun as you continue to master machine learning.

Chapter 24

Getting More Help

This is just the beginning of your machine learning journey with Python. As you start to work on your own machine learning projects you may need help. This chapter points out some of the best sources of Python and machine learning help that you can find.

24.1 General Advice

Generally the documentation for Python and the SciPy stack is excellent. There are a good mix of user guides and API documentation. I would advise you to read the API documentation for the classes and functions you are using, now that you know which classes and functions to use. The API documentation will give you a fuller understanding on the deeper configuration that you can explore to serve your interests of better performing models.

Another invaluable resources are Question and Answer sites like StackOverflow[1]. You can search for error messages and problems that you are having and find sample code and ideas that will directly help you. Also make use of the *Related* posts on the right-hand side of the screen as they can guide you to related and perhaps just as helpful posts.

24.2 Help With Python

Python is a fully featured programming language. As such, the more you learn about it, the better you can use it. If you are relatively new to the Python platform here are some valuable resources for going one step deeper:

- Google Python Class.
  https://developers.google.com/edu/python/
- Python HOWTOs, invaluable for learning idioms and such (Python 2).
  https://docs.python.org/2/howto/index.html
- Python Standard Library Reference (Python 2).
  https://docs.python.org/2/library/index.html

[1] http://StackOverflow.com

24.3 Help With SciPy and NumPy

It is a great idea to become familiar with the broader SciPy ecosystem and for that I would recommend the SciPy Lecture Notes listed below. I think the NumPy documentation is excellent (and deep) but you probably don't need it unless you are doing something exotic.

- SciPy Lecture Notes.
  http://www.scipy-lectures.org/
- NumPy User Guide.
  http://docs.scipy.org/doc/numpy/user/

24.4 Help With Matplotlib

Get good at plotting. Confident to the point that you can copy-and-paste a recipe to plot data at the drop of a hat. It is an invaluable skill. I think you will be best served looking at lots of Matplotlib examples for different plot types and preparing recipes for yourself to use later. Making pretty plots is a different topic entirely and for that I would recommend studying the API carefully. I see plots as disposable tools for learning more about a problem.

- Matplotlib gallery of plot types and sample code.
  http://matplotlib.org/gallery.html
- Matplotlib Beginners Guide.
  http://matplotlib.org/users/beginner.html
- Matplotlib API Reference.
  http://matplotlib.org/api/index.html

24.5 Help With Pandas

There is a lot of documentation for Pandas, I think because it is such a flexible tool. Generally, the cookbook examples will be most useful to you here as they will give you ideas on different ways to slice and dice your data.

- Pandas documentation page (user guide). Note the table of contents of the left hand side, it's very extensive.
  http://pandas.pydata.org/pandas-docs/stable/
- Pandas cookbook providing many short and sweet examples.
  http://pandas.pydata.org/pandas-docs/stable/cookbook.html
- Pandas API Reference.
  http://pandas.pydata.org/pandas-docs/stable/api.html

24.6 Help With scikit-learn

There is excellent documentation on the scikit-learn website. I highly recommend spending time reading the API for each class and function that you use in order to get the most from them.

- The scikit-learn API Reference.
  http://scikit-learn.org/stable/modules/classes.html
- The scikit-learn User Guide.
  http://scikit-learn.org/stable/user_guide.html
- The scikit-learn Example Gallery.
  http://scikit-learn.org/stable/auto_examples/index.html