Machine Learning Algorithms From Scratch

With Python

Jason Brownlee

MACHINE LEARNING MASTERY

Machine Learning Algorithms From Scratch

Edition: v1.2

Contents

Welcome iii

I Data Preparation 1

1 Load Data From CSV 2
1.1 Description . . . 2
1.2 Tutorial . . . 3
1.3 Extensions . . . 8
1.4 Review . . . 8

2 Scale Machine Learning Data 9
2.1 Description . . . 9
2.2 Tutorial . . . 9
2.3 Extensions . . . 18
2.4 Review . . . 18

3 Algorithm Evaluation Methods 19
3.1 Description . . . 19
3.2 Tutorial . . . 19
3.3 Extensions . . . 23
3.4 Review . . . 24

4 Evaluation Metrics 25
4.1 Description . . . 25
4.2 Tutorial . . . 26
4.3 Extensions . . . 32
4.4 Review . . . 33

5 Baseline Models 34
5.1 Description . . . 34
5.2 Tutorial . . . 35
5.3 Extensions . . . 39
5.4 Review . . . 39

II Linear Algorithms **40**

6 Algorithm Test Harnesses **41**
6.1 Description . . . 41
6.2 Tutorial . . . 42
6.3 Extensions . . . 47
6.4 Review . . . 47

7 Simple Linear Regression **48**
7.1 Description . . . 48
7.2 Tutorial . . . 49
7.3 Extensions . . . 58
7.4 Review . . . 58

8 Multivariate Linear Regression **60**
8.1 Description . . . 60
8.2 Tutorial . . . 61
8.3 Extensions . . . 68
8.4 Review . . . 68

9 Logistic Regression **70**
9.1 Description . . . 70
9.2 Tutorial . . . 71
9.3 Extensions . . . 79
9.4 Review . . . 79

10 Perceptron **80**
10.1 Description . . . 80
10.2 Tutorial . . . 81
10.3 Extensions . . . 88
10.4 Review . . . 88

III Nonlinear Algorithms **90**

11 Classification and Regression Trees **91**
11.1 Descriptions . . . 91
11.2 Tutorial . . . 92
11.3 Extensions . . . 108
11.4 Review . . . 108

12 Naive Bayes **110**
12.1 Descriptions . . . 110
12.2 Tutorial . . . 111
12.3 Extensions . . . 124
12.4 Review . . . 124

13 k-Nearest Neighbors **126**
13.1 Description . . . 126
13.2 Tutorial . . . 127
13.3 Extensions . . . 138
13.4 Review . . . 139

14 Learning Vector Quantization **140**
14.1 Description . . . 140
14.2 Tutorial . . . 141
14.3 Extensions . . . 152
14.4 Review . . . 152

15 Back-Propagation **153**
15.1 Description . . . 153
15.2 Tutorial . . . 154
15.3 Extensions . . . 172
15.4 Review . . . 172

IV Ensemble Algorithms **174**

16 Bootstrap Aggregation **175**
16.1 Descriptions . . . 175
16.2 Tutorial . . . 176
16.3 Extensions . . . 183
16.4 Review . . . 183

17 Random Forest **184**
17.1 Description . . . 184
17.2 Tutorial . . . 185
17.3 Extensions . . . 191
17.4 Review . . . 191

18 Stacked Generalization **193**
18.1 Description . . . 193
18.2 Tutorial . . . 194
18.3 Extensions . . . 202
18.4 Review . . . 203

V Conclusions **204**

19 How Far You Have Come **205**

20 Getting More Help **206**
20.1 Machine Learning Books . . . 206
20.2 Forums and Q&A Websites . . . 206
20.3 Contact the Author . . . 207

VI Appendix **208**

A Standard Datasets **209**
A.1 Overview . . . 209
A.2 Swedish Auto Insurance Dataset . . . 210
A.3 Wine Quality Dataset . . . 211
A.4 Pima Indians Diabetes Dataset . . . 212
A.5 Sonar Dataset . . . 213
A.6 Banknote Dataset . . . 213
A.7 Iris Flower Dataset . . . 214
A.8 Abalone Dataset . . . 215
A.9 Ionosphere Dataset . . . 215
A.10 Wheat Seeds Dataset . . . 216

B Python Crash Course **217**
B.1 Assignment . . . 217
B.2 Flow Control . . . 219
B.3 Data Structures . . . 220

Welcome

Welcome to Machine Learning Algorithms From Scratch. This is your guide to learning the details of machine learning algorithms by implementing them from scratch in Python. You will discover how to load data, evaluate models and implement a suite of top machine learning algorithms using step-by-step tutorials and sample code.

Machine learning algorithms do have a lot of math and theory under the covers, but you do not need to know why algorithms work to be able to implement them and apply them to achieve real and valuable results. From an applied perspective, machine learning is a shallow field and a motivated developer can quickly pick it up and start making very real and impactful contributions. This is my goal for you and this book is your ticket to that outcome.

Implement Machine Learning Algorithms

Most developers that I know (myself included) learn best by implementing. It is our preferred learning style and it is the reason that I created this book. This section lists the benefits of implementing machine learning algorithms from scratch, some benefits from extending your own implementations as well as some limitations to this approach to learning.

Implementation Benefits

Machine learning is all about algorithms and there are so many algorithms that it can feel overwhelming. Really, there are probably only 10 algorithms that if understood will unlock the field for you. An approach that you can use to get a handle on machine learning algorithms is to implement them from scratch. This will give you a deep understanding of how the algorithm works and all of the micro-decision points within the method that can be parameterized or modified to tune it to a specific problem. The benefits of implementing algorithms from scratch are:

- **Understanding**: You will gain a deep appreciation for how the algorithm works. You will begin to understand how the mathematical description of the method relates to vectors and matrices of numbers that your code operates on. You will also know how all of the parameters are used, their effects and even have insights into how it could be further parameterized to specialize it for a problem.
- **Starting Point**: Your implementation will provide the basis for more advanced extensions and even an operational system that uses the algorithm. Your deep knowledge of the algorithm and your implementation can give you advantages of knowing the space and time complexity of your own code over using an opaque off-the-shelf library.

- **Ownership**: The implementation is your own giving you confidence with the method and ownership over how it is realized as a system. It is no longer just a machine learning algorithm, but a method that is now in your toolbox.

Implementation Extensions

Once you have implemented an algorithm you can explore making improvements to the implementation. Some examples of improvements you could explore include:

- **Experimentation**: You can expose many of the micro-decisions you made in the algorithms implementation as parameters and perform studies on variations of those parameters. This can lead to new insights and disambiguation of code that you can share and promote.
- **Optimization**: You can explore opportunities to make the implementation more efficient by using tools, libraries, different languages, different data structures, patterns and internal algorithms. Knowledge you have of algorithms and data structures for classical computer science can be very beneficial in this type of work.
- **Specialization**: You may explore ways of making the algorithm more specific to a problem. This can be required when creating production systems and is a valuable skill. Making an algorithm more problem specific can also lead to increases in efficiency (such as running time) and efficacy (such as accuracy or other performance measures).
- **Generalization**: Opportunities can be created by making a specific algorithm more general. Programmers (like mathematicians) are uniquely skilled in abstraction and you may be able to see how the algorithm could be applied to more general cases of a class of problem or other problems entirely.

Implementation Limitations

Developers and engineers often learn best by implementing, but implementing machine learning algorithms is not the place to start for everyone.

- **Slow for Beginners**. Often, practitioners will make more progress and progress faster by learning how to apply machine learning algorithms to predictive modeling problems. Implementing algorithms is a second step for learning how to get more out of each algorithm by discovering how they work and how the parameters affect their behavior.
- **Speed and Correctness**. Algorithms that you use to solve business problems need to be fast and correct. And this can be very hard to do for beginners. The implementations developed for learning purposes are almost certainly going to be too slow or too fragile for use in operations (that includes all examples in this book). Use implementations for learning and efficient code libraries for production systems.

Book Organization

This book is divided into 6 main parts:

1. **Introduction**. Welcomes you to the book and clearly lays out what to expect and your learning outcomes (you are here).

2. **Data Preparation**. Tutorials for loading and preparing data, evaluating model predictions, estimating model skill and developing a baseline for model performance.

3. **Linear Algorithms**. Tutorials on linear machine learning algorithms such as linear regression, multivariate linear regression, logistic regression and the Perceptron algorithm.

4. **Nonlinear Algorithms**. Tutorials on nonlinear machine learning algorithms such as Naive Bayes, k-Nearest Neighbors, Learning Vector Quantization, Back-propagation and Decision Trees.

5. **Ensemble Algorithms**. Tutorials on ensemble machine learning algorithms such as Bootstrap Aggregation, Random Forest and Stacked Generalization.

6. **Conclusions**. A review of how far you have come and resources for getting help and further reading.

There are a few ways you can read this book. You can dip into the tutorials as your need or interests motivate you. Alternatively, you can work through the book end-to-end and take advantage of how the tutorials build in complexity and range. I recommend the latter approach. To get the very most from this book, I recommend taking each tutorial and building upon them. Attempt to improve the results, apply the method to a similar but different problem, and so on. I share a number of extension ideas for you to consider in each tutorial.

Write up what you tried or learned and share it on your blog, social media or send me an email at `jason@MachineLearningMastery.com`. This book is what you make of it and by putting in a little extra, you can quickly become a true force in machine learning algorithms.

Tutorial Structure

A tutorial-based approach is used throughout the book. It is conversational rather than formal and focuses on the ideas, the code needed to implement those ideas, and the results to expect. All tutorials in this book follow a carefully designed 6-part structure. This structure can be summarized as follows:

1. Overview

2. Description

3. Tutorial

4. Case Study

5. Extensions

6. Review

Overview

This is a short section that summarizes the tutorial. You will discover exactly what you will know after completing the tutorial.

Description

This section describes both the technique and problem that you will be applying it to. It will not describe the theory behind why the technique works. Instead, it focuses on the salient details regarding how the technique works. This includes points relevant to implementing it from scratch.

Also included in this section is a summary of the problem that the technique will be evaluated on, if relevant. Not all tutorials will have a problem description: only those tutorials on a given machine learning algorithm. Only standard well-known machine learning problems are used as they are freely accessible and known best results are available for comparison.

Tutorial

The tutorial is the steps to complete to go from just an idea to a fully working implementation. Each tutorial was designed with three principles in mind:

1. **Procedural**: Tutorials are procedural, meaning that they are presented as a recipe of discrete steps intended to be completed in order.

2. **Standalone**: Tutorials are standalone, meaning that all code needed to run the example is available within the tutorial, even if this involves repetition.

3. **Consistent**: Tutorials are consistent, meaning that the same routines developed earlier in the book are used again and again to load data and evaluate algorithms.

A summary of the steps in the tutorial is provided followed by the numbered steps of each part of the procedure. Code was designed to be modular and broken down into many small functions that can be understood and tested as standalone units. A procedural rather than object-oriented approach was used in all code examples. Clever Python tricks and advanced use of lambdas and list comprehensions were kept to a minimum in favor of `for`-loops. This was done intentionally for 3 reasons:

- To be kind to Python novices.
- To be understandable as almost pseudocode, a large benefit of Python.
- To be readily adaptable for use in other other languages and environments.

If you have advanced Python skills and you can see more efficient ways to structure the code, please share your ideas with me and we can put them on the ML Mastery blog. I would love to see what you come up with.

Disclaimer: All the code in this book is for education and demonstration only. Code is not intended for use in production systems or operational environments.

Case Study

Each algorithm tutorial ends with a complete code listing of a fully working case study on a real-world predictive modeling problem. This is to show you how to use the technique in practice, often leveraging techniques introduced earlier in the book such as data loading, data preparation and algorithm evaluation. This is to ensure that even if small copy-paste errors were made during the execution of the tutorial or steps were skipped, that you always have a reference version of the tutorial to run and use as a template for your own work. A sample output is also provided from executing the example, again so that you have reference for comparison.

Extensions

This section lists ideas to extend the example in the tutorial. This may include additional implementation concerns to make the technique more robust or generally applicable. It may also include usage heuristics for the technique to ensure you can get more out of its application to new problems. Please try some of the extension ideas. Even email me and share your experiences; we can put them to the ML Mastery blog.

Review

Tutorials end with a summary of the principles and skills that you learned. This helps to reinforce and remind you of your progress through the book and keep you highly motivated.

Requirements For This Book

Python

You do not need to be a Python expert, but it would be helpful if you have or know how to install and setup a Python environment. You are expected to know some basic Python syntax. If you are a programmer from another language like Java or C#, there is a Python crash course in the appendix to bring you up to speed quickly.

The tutorials assume that you have Python 2.7 installed and working. I have not tested the code with a Python 3 environment, but my students tell me that the code works with little or no modification. You may have a Python environment on your workstation or laptop, it may be in a VM or a Docker instance that you run, or it may be a server instance that you can configure in the cloud.

Machine Learning

You do not need to be a machine learning expert, but it would be helpful if you knew how to navigate a small machine learning problem. Concepts and techniques are described at the beginning of each tutorial, but only briefly enough to give you context. Additional resources are listed for you to learn more on each concept introduced. There are resources to go into these topics in more detail at the end of the book, but some knowledge of these areas might make things easier for you.

Your Outcomes From Reading This Book

This book will lead you from being a developer who is interested in machine learning algorithms to a machine learning developer that knows how to implement a suite of machine learning algorithms and techniques from scratch in Python. Specifically, you will know:

- How to load from CSV files and prepare data for modeling.
- How to select algorithm evaluation metrics and resampling techniques for a test harness.
- How to develop a baseline expectation of performance for a given problem.
- How to implement and apply a suite of linear machine learning algorithms.
- How to implement and apply a suite of advanced nonlinear machine learning algorithms.
- How to implement and apply ensemble machine learning algorithms to improve performance.

From this outcome you will:

- Know how top machine learning algorithms work internally.
- Know how to better configure machine learning algorithms in order to get the most out of them.
- Know the myriad of micro-decisions that a machine learning library has hidden from you in practice.
- Know how you might begin to develop your own custom machine learning algorithm implementations.

What This Book is Not

This book solves a specific problem of getting you, a developer, up to speed on how to implement top machine learning algorithms from scratch in Python. This book was not intended to be everything to everyone and it is very important to calibrate your expectations. Specifically:

- **This is not a machine learning textbook**. We will not be getting into the theory or mathematical description of machine learning algorithms as this is not required to implement algorithms from scratch. You are also expected to have some familiarity with machine learning basics, or be able to pick them up yourself.
- **This is not an applications book**. We will be using real-world problems as case studies, but we will not linger on the best practices for working through machine learning problems end-to-end.
- **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 already be familiar with Python or a developer who can pick up a new C-like language relatively quickly.

Resources are provided in the final chapter if you are interested in focusing on one of these related areas.

Summary

It is a special time right now. The interest and information available about applied machine learning is so great. The pace of change of machine learning feels like it has never been so fast, spurred by the amazing results that the methods are showing in such a broad range of fields.

This is the start of your journey into expanding your understanding of machine learning algorithms and I am excited for you. Take your time, have fun and I'm so excited to see where you can take this amazing new technology.

Next

In the next section you will start with your first tutorial on how to load machine learning data.

Part I

Data Preparation

Chapter 1

Load Data From CSV

You must know how to load data before you can use it to train a machine learning model. When starting out, it is a good idea to stick with small in-memory datasets using standard file formats like comma separated value (.csv). In this tutorial you will discover how to load your data in Python from scratch, including:

- How to load a CSV file.
- How to convert strings from a file to floating point numbers.
- How to convert class values from a file to integers.

Let's get started.

1.1 Description

1.1.1 Comma Separated Values

The standard file format for small datasets is Comma Separated Values or CSV. In its simplest form, CSV files are comprised of rows of data. Each row is divided into columns using a comma (`,`). In this tutorial, we are going to practice loading two different, standard machine learning datasets in CSV format.

1.1.2 Pima Indians Diabetes Dataset

In this tutorial we will use the Pima Indians Diabetes Dataset. This dataset involves the prediction of the onset of diabetes within 5 years. The baseline performance on the problem is approximately 65%. You can learn more about it in Appendix A, Section A.4. Download the dataset and save it into your current working directory with the filename `pima-indians-diabetes.csv`.

1.1.3 Iris Flower Species Dataset

In this tutorial we will also use the Iris Flower Species Dataset. This dataset involves the prediction of iris flower species. The baseline performance on the problem is approximately 26%. You can learn more about it in Appendix A, Section A.7. Download the dataset and save it into your current working directory with the filename `iris.csv`.

1.2 Tutorial

This tutorial is divided into 3 parts:

1. Load a file.
2. Load a file and convert Strings to Floats.
3. Load a file and convert Strings to Integers.

These steps will provide the foundations you need to handle loading your own data.

1.2.1 Load CSV File

The first step is to load the CSV file. We will use the `csv` module that is a part of the standard library. The `reader()` function in the csv module takes a file as an argument.

We will create a function called `load_csv()` to wrap this behavior that will take a filename and return our dataset. We will represent the loaded dataset as a list of lists. The first list is a list of observations or rows, and the second list is the list of column values for a given row. Below is the complete function for loading a CSV file.

```
# Load a CSV file
def load_csv(filename):
  file = open(filename, "r")
  lines = reader(file)
  dataset = list(lines)
  return dataset
```

Listing 1.1: Function for loading a CSV.

We can test this function by loading the Pima Indians dataset. Taking a peek at the first 5 rows of the raw data file we can see the following:

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
```

Listing 1.2: Peek at Pima Indians Diabetes dataset.

The data is numeric and separated by commas and we can expect that the whole file meets this expectation. Let's use the new function and load the dataset. Once loaded we can report some simple details such as the number of rows and columns loaded. Putting all of this together, we get the following:

```
# Example of loading Pima Indians CSV dataset
from csv import reader

# Load a CSV file
def load_csv(filename):
  file = open(filename, "r")
  lines = reader(file)
  dataset = list(lines)
  return dataset
```

```
# Load dataset
filename = 'pima-indians-diabetes.csv'
dataset = load_csv(filename)
print('Loaded data file {0} with {1} rows and {2} columns').format(filename, len(dataset),
    len(dataset[0]))
```

Listing 1.3: Example of Loading the Pima Indians Diabetes Dataset CSV File.

Running this example we see:

```
Loaded data file pima-indians-diabetes.csv with 768 rows and 9 columns
```

Listing 1.4: Sample output from loading the Pima Indians Diabetes dataset CSV file.

A limitation of this function is that it will load empty lines from data files and add them to our list of rows. We can overcome this by adding rows of data one at a time to our dataset and skipping empty rows. Below is the updated example with this new improved version of the `load_csv()` function.

```
# Example of loading Pima Indians CSV dataset
from csv import reader

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Load dataset
filename = 'pima-indians-diabetes.csv'
dataset = load_csv(filename)
print('Loaded data file {0} with {1} rows and {2} columns').format(filename, len(dataset),
    len(dataset[0]))
```

Listing 1.5: Improved Example of Loading the Pima Indians Diabetes Dataset CSV File.

Running this example we see:

```
Loaded data file pima-indians-diabetes.csv with 768 rows and 9 columns
```

Listing 1.6: Sample Output From Loading the Pima Indians Diabetes Dataset CSV File.

1.2.2 Convert String to Floats

Most, if not all machine learning algorithms prefer to work with numbers. Specifically, floating point numbers are preferred. Our code for loading a CSV file returns a dataset as a list of lists, but each value is a string. We can see this if we print out one record from the dataset:

```
print(dataset[0])
```

Listing 1.7: Display One Record From a Dataset.

This produces output like:

```
['6', '148', '72', '35', '0', '33.6', '0.627', '50', '1']
```

Listing 1.8: Sample Output From Displaying One Row of Data.

We can write a small function to convert specific columns of our loaded dataset to floating point values. Below is this function called `str_column_to_float()`. It will convert a given column in the dataset to floating point values, careful to strip any whitespace from the value before making the conversion.

```
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())
```

Listing 1.9: Function For Converting String Data To Floats.

We can test this function by combining it with our load CSV function above, and convert all of the numeric data in the Pima Indians dataset to floating point values. The complete example is below.

```
# Example of converting string variables to float
from csv import reader

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Load pima-indians-diabetes dataset
filename = 'pima-indians-diabetes.csv'
dataset = load_csv(filename)
print('Loaded data file {0} with {1} rows and {2} columns').format(filename, len(dataset),
    len(dataset[0]))
print(dataset[0])
# convert string columns to float
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
print(dataset[0])
```

Listing 1.10: Example of Converting String Values to Floats in the Pima Indians Diabetes Dataset.

Running this example we see the first row of the dataset printed both before and after the conversion. We can see that the values in each column have been converted from strings to numbers.

```
Loaded data file pima-indians-diabetes.csv with 768 rows and 9 columns
['6', '148', '72', '35', '0', '33.6', '0.627', '50', '1']
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0, 1.0]
```

Listing 1.11: Sample Output From Converting String Values to Floats.

1.2.3 Convert String to Integers

The iris flowers dataset is like the Pima Indians dataset, in that the columns contain numeric data. The difference is the final column, traditionally used to hold the outcome or value to be predicted for a given row. The final column in the iris flowers data is the iris flower species as a string. For example, below are the first 5 rows of the raw dataset.

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
```

Listing 1.12: Peek at Iris Flower Species dataset.

Some machine learning algorithms prefer all values to be numeric, including the outcome or predicted value. We can convert the class value in the iris flowers dataset to an integer by creating a map.

1. First, we locate all of the unique class values, which happen to be: `Iris-setosa`, `Iris-versicolor` and `Iris-virginica`.
2. Next, we assign an integer value to each, such as: 0, 1 and 2.
3. Finally, we replace all occurrences of class string values with their corresponding integer values.

Below is a function to do just that called `str_column_to_int()`. Like the previously introduced `str_column_to_float()` it operates on a single column in the dataset.

```
# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup
```

Listing 1.13: Function To Integer Encode String Class Values.

We can test this new function in addition to the previous two functions for loading a CSV file and converting columns to floating point values. It also returns the dictionary mapping of class values to integer values, in case any users downstream want to convert predictions back to string values again. The example below loads the iris dataset then converts the first 3 columns to floats and the final column to integer values.

```
# Example of integer encoding string class values
from csv import reader

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Load iris dataset
filename = 'iris.csv'
dataset = load_csv(filename)
print('Loaded data file {0} with {1} rows and {2} columns').format(filename, len(dataset),
    len(dataset[0]))
print(dataset[0])
# convert string columns to float
for i in range(4):
  str_column_to_float(dataset, i)
# convert class column to int
lookup = str_column_to_int(dataset, 4)
print(dataset[0])
print(lookup)
```

Listing 1.14: Example of Integer Encoding Class Values in the Iris Dataset.

Running this example produces the output below. We can see the first row of the dataset before and after the data type conversions. We can also see the dictionary mapping of class values to integers.

```
Loaded data file iris.csv with 150 rows and 5 columns
['5.1', '3.5', '1.4', '0.2', 'Iris-setosa']
[5.1, 3.5, 1.4, 0.2, 1]
{'Iris-virginica': 0, 'Iris-setosa': 1, 'Iris-versicolor': 2}
```

Listing 1.15: Sample Output From Integer Encoding Class Values.

1.3 Extensions

You learned how to load CSV files and perform basic data conversions. Data loading can be a difficult task given the variety of data cleaning and conversion that may be required from problem to problem. There are many extensions that you could make to make these examples more robust to new and different data files. Below are just a few ideas you can consider researching and implementing yourself:

- Detect and remove empty lines at the top or bottom of the file.
- Detect and handle missing values in a column.
- Detect and handle rows that do not match expectations for the rest of the file.
- Support for other delimiters such as pipe (`|`) or white space.
- Support more efficient data structures such as arrays.

Two libraries you may wish to use in practice for loading CSV data are NumPy and Pandas. NumPy offers the `loadtxt()`[1] function for loading data files as NumPy arrays. Pandas offers the `read_csv()`[2] function that offers a lot of flexibility regarding data types, file headers and more.

1.4 Review

In this tutorial, you discovered how you can load your machine learning data from scratch in Python. Specifically, you learned:

- How to load a CSV file into memory.
- How to convert string values to floating point values.
- How to convert a string class value into an integer encoding.

1.4.1 Further Reading

- Section 13.1, CSV File Reading and Writing, *The Python Standard Library*
  https://docs.python.org/2/library/csv.html
- CSV file format in RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV)
  https://tools.ietf.org/html/rfc4180

1.4.2 Next

In the next tutorial, you will discover how to rescale your machine learning data for algorithms that weight input values.

[1] http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html
[2] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

Chapter 2

Scale Machine Learning Data

Many machine learning algorithms expect data to be scaled consistently. There are two popular methods that you should consider when scaling your data for machine learning. In this tutorial, you will discover how you can rescale your data for machine learning. After reading this tutorial you will know:

- How to normalize your data from scratch.
- How to standardize your data from scratch.
- When to normalize as opposed to standardize data.

Let's get started.

2.1 Description

Many machine learning algorithms expect the scale of the input and even the output data to be equivalent. It can help in methods that weight inputs in order to make a prediction, such as in linear regression and logistic regression. It is practically required in methods that combine weighted inputs in complex ways such as in artificial neural networks and deep learning.

2.1.1 Pima Indians Diabetes Dataset

In this tutorial we will use the Pima Indians Diabetes Dataset. This dataset involves the prediction of the onset of diabetes within 5 years. The baseline performance on the problem is approximately 65%. You can learn more about it in Appendix A, Section A.4. Download the dataset and save it into your current working directory with the filename `pima-indians-diabetes.csv`.

2.2 Tutorial

This tutorial is divided into 3 parts:

1. Normalize Data.
2. Standardize Data.

3. When to Normalize and Standardize.

These steps will provide the foundations you need to handle scaling your own data.

2.2.1 Normalize Data

Normalization can refer to different techniques depending on context. Here, we use normalization to refer to rescaling an input variable to the range between 0 and 1. Normalization requires that you know the minimum and maximum values for each attribute.

This can be estimated from training data or specified directly if you have deep knowledge of the problem domain. You can easily estimate the minimum and maximum values for each attribute in a dataset by enumerating through the values. The snippet of code below defines the `dataset_minmax()` function that calculates the min and max value for each attribute in a dataset, then returns an array of these minimum and maximum values.

```
# Find the min and max values for each column
def dataset_minmax(dataset):
  minmax = list()
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    value_min = min(col_values)
    value_max = max(col_values)
    minmax.append([value_min, value_max])
  return minmax
```

Listing 2.1: Function To Calculate the Min and Max Values For a Dataset.

We can contrive a small dataset for testing as follows:

```
x1  x2
50  20
20  90
```

Listing 2.2: Small Contrived Dataset.

With this contrived dataset, we can test our function for calculating the min and max for each column.

```
# Find the min and max values for each column
def dataset_minmax(dataset):
  minmax = list()
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    value_min = min(col_values)
    value_max = max(col_values)
    minmax.append([value_min, value_max])
  return minmax

# Contrive small dataset
dataset = [[50, 30], [20, 90]]
print(dataset)
# Calculate min and max for each column
minmax = dataset_minmax(dataset)
print(minmax)
```

Listing 2.3: Example Calculating the Min and Max Values of a Contrived Dataset.

Running the example produces the following output. First, the dataset is printed in a list-of-lists format, then the min and max for each column is printed in the format `column1: min,max and column2: min,max`. For example:

```
[[50, 30], [20, 90]]
[[20, 50], [30, 90]]
```

Listing 2.4: Output of Example Calculating the Min and Max Values.

Once we have estimates of the maximum and minimum allowed values for each column, we can now normalize the raw data to the range 0 and 1. The calculation to normalize a single value for a column is:

$$\text{scaled value} = \frac{value - min}{max - min} \tag{2.1}$$

Below is an implementation of this in a function called `normalize_dataset()` that normalizes values in each column of a provided dataset.

```
# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])
```

Listing 2.5: Function To Normalize a Dataset.

We can tie this function together with the `dataset_minmax()` function and normalize the contrived dataset.

```
# Find the min and max values for each column
def dataset_minmax(dataset):
  minmax = list()
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    value_min = min(col_values)
    value_max = max(col_values)
    minmax.append([value_min, value_max])
  return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Contrive small dataset
dataset = [[50, 30], [20, 90]]
print(dataset)
# Calculate min and max for each column
minmax = dataset_minmax(dataset)
print(minmax)
# Normalize columns
normalize_dataset(dataset, minmax)
print(dataset)
```

Listing 2.6: Example of Normalize the Contrived Dataset.

Running this example prints the output below, including the normalized dataset.

```
[[50, 30], [20, 90]]
[[20, 50], [30, 90]]
[[1, 0], [0, 1]]
```

Listing 2.7: Example Output of Normalizing the Contrived Dataset.

We can combine this code with code for loading a CSV dataset and load and normalize the Pima Indians Diabetes dataset. The example first loads the dataset and converts the values for each column from string to floating point values. The minimum and maximum values for each column are estimated from the dataset, and finally, the values in the dataset are normalized.

```
# Example of normalizing the diabetes dataset
from csv import reader

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Find the min and max values for each column
def dataset_minmax(dataset):
  minmax = list()
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    value_min = min(col_values)
    value_max = max(col_values)
    minmax.append([value_min, value_max])
  return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Load pima-indians-diabetes dataset
filename = 'pima-indians-diabetes.csv'
dataset = load_csv(filename)
print('Loaded data file {0} with {1} rows and {2} columns').format(filename, len(dataset),
    len(dataset[0]))
# convert string columns to float
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
print(dataset[0])
```

```
# Calculate min and max for each column
minmax = dataset_minmax(dataset)
# Normalize columns
normalize_dataset(dataset, minmax)
print(dataset[0])
```

Listing 2.8: Example of Normalize the Diabetes Dataset.

Running the example produces the output below. The first record from the dataset is printed before and after normalization, showing the effect of the scaling.

```
Loaded data file pima-indians-diabetes.csv with 768 rows and 9 columns
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0, 1.0]
[0.35294117647058826, 0.7437185929648241, 0.5901639344262295, 0.35353535353535354, 0.0,
    0.5007451564828614, 0.23441502988898377, 0.48333333333333334, 1.0]
```

Listing 2.9: Example Output of Normalizing the Diabetes Dataset.

2.2.2 Standardize Data

Standardization is a rescaling technique that refers to centering the distribution of the data on the value 0 and the standard deviation to the value 1. Together, the mean and the standard deviation can be used to summarize a normal distribution, also called the Gaussian distribution or bell curve.

It requires that the mean and standard deviation of the values for each column be known prior to scaling. As with normalizing above, we can estimate these values from training data, or use domain knowledge to specify their values. Let's start with creating functions to estimate the mean and standard deviation statistics for each column from a dataset. The mean describes the middle or central tendency for a collection of numbers. The mean for a column is calculated as the sum of all values for a column divided by the total number of values.

$$\text{mean} = \frac{\sum_{i=1}^{n} values_i}{count(values)} \tag{2.2}$$

The function below named `column_means()` calculates the mean values for each column in the dataset.

```
# calculate column means
def column_means(dataset):
  means = [0 for i in range(len(dataset[0]))]
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    means[i] = sum(col_values) / float(len(dataset))
  return means
```

Listing 2.10: Function To Calculate Means For Each Column in a Dataset.

The standard deviation describes the average spread of values from the mean. It can be calculated as the square root of the sum of the squared difference between each value and the mean and dividing by the number of values minus 1.

$$\text{standard deviation} = \sqrt{\frac{\sum_{i=1}^{n}(value_i - mean)^2}{count(values) - 1}} \tag{2.3}$$

The function below named `column_stdevs()` calculates the standard deviation of values for each column in the dataset and assumes the means have already been calculated.

```
# calculate column standard deviations
def column_stdevs(dataset, means):
  stdevs = [0 for i in range(len(dataset[0]))]
  for i in range(len(dataset[0])):
    variance = [pow(row[i]-means[i], 2) for row in dataset]
    stdevs[i] = sum(variance)
  stdevs = [sqrt(x/(float(len(dataset)-1))) for x in stdevs]
  return stdevs
```

Listing 2.11: Function To Calculate Standard Deviations For Each Column in a Dataset.

Again, we can contrive a small dataset to demonstrate the estimate of the mean and standard deviation from a dataset.

```
x1  x2
50  30
20  90
30  50
```

Listing 2.12: Small Contrived Dataset To Test Standardization.

Using an excel spreadsheet, we can estimate the mean and standard deviation for each column as follows:

```
       x1     x2
mean   33.3   56.6
stdev  15.27  30.55
```

Listing 2.13: Expected Descriptive Statistics For Contrived Dataset.

Using the contrived dataset, we can estimate the summary statistics.

```
# Example of calculating stats on a contrived dataset
from math import sqrt

# calculate column means
def column_means(dataset):
  means = [0 for i in range(len(dataset[0]))]
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    means[i] = sum(col_values) / float(len(dataset))
  return means

# calculate column standard deviations
def column_stdevs(dataset, means):
  stdevs = [0 for i in range(len(dataset[0]))]
  for i in range(len(dataset[0])):
    variance = [pow(row[i]-means[i], 2) for row in dataset]
    stdevs[i] = sum(variance)
  stdevs = [sqrt(x/(float(len(dataset)-1))) for x in stdevs]
  return stdevs

# Standardize dataset
dataset = [[50, 30], [20, 90], [30, 50]]
print(dataset)
```

```
# Estimate mean and standard deviation
means = column_means(dataset)
stdevs = column_stdevs(dataset, means)
print(means)
print(stdevs)
```

Listing 2.14: Example of Calculating Statistics from the Contrived Dataset.

Executing the example provides the following output, matching the numbers calculated in the spreadsheet.

```
[[50, 30], [20, 90], [30, 50]]
[33.333333333333336, 56.666666666666664]
[15.275252316519467, 30.550504633038933]
```

Listing 2.15: Example Output From Calculating Statistics from the Contrived Dataset.

Once the summary statistics are calculated, we can easily standardize the values in each column. The calculation to standardize a given value is as follows:

$$\text{standardized\_value}_i = \frac{\sum_{i=1}^{n}(value_i - mean)}{stdev} \tag{2.4}$$

Below is a function named `standardize_dataset()` that implements this equation

```
# standardize dataset
def standardize_dataset(dataset, means, stdevs):
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - means[i]) / stdevs[i]
```

Listing 2.16: Function To Standardize a Dataset.

Combining this with the functions to estimate the mean and standard deviation summary statistics, we can standardize our contrived dataset.

```
# Example of standardizing a contrived dataset
from math import sqrt

# calculate column means
def column_means(dataset):
  means = [0 for i in range(len(dataset[0]))]
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    means[i] = sum(col_values) / float(len(dataset))
  return means

# calculate column standard deviations
def column_stdevs(dataset, means):
  stdevs = [0 for i in range(len(dataset[0]))]
  for i in range(len(dataset[0])):
    variance = [pow(row[i]-means[i], 2) for row in dataset]
    stdevs[i] = sum(variance)
  stdevs = [sqrt(x/(float(len(dataset)-1))) for x in stdevs]
  return stdevs

# standardize dataset
def standardize_dataset(dataset, means, stdevs):
```

```
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - means[i]) / stdevs[i]

# Standardize dataset
dataset = [[50, 30], [20, 90], [30, 50]]
print(dataset)
# Estimate mean and standard deviation
means = column_means(dataset)
stdevs = column_stdevs(dataset, means)
print(means)
print(stdevs)
# standardize dataset
standardize_dataset(dataset, means, stdevs)
print(dataset)
```

Listing 2.17: Example of Standardizing the Contrived Dataset.

Executing this example produces the following output, showing standardized values for the contrived dataset.

```
[[50, 30], [20, 90], [30, 50]]
[33.333333333333336, 56.666666666666664]
[15.275252316519467, 30.550504633038933]
[[1.0910894511799618, -0.8728715609439694], [-0.8728715609439697, 1.091089451179962],
    [-0.21821789023599253, -0.2182178902359923]]
```

Listing 2.18: Example Output From Standardizing the Contrived Dataset.

Again, we can demonstrate the standardization of a machine learning dataset. The example below demonstrates how to load and standardize the Pima Indians diabetes dataset, assumed to be in the current working directory as in the previous normalization example.

```
# Standardize the Diabetes Dataset
from csv import reader
from math import sqrt

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# calculate column means
def column_means(dataset):
  means = [0 for i in range(len(dataset[0]))]
  for i in range(len(dataset[0])):
```

```
    col_values = [row[i] for row in dataset]
    means[i] = sum(col_values) / float(len(dataset))
  return means

# calculate column standard deviations
def column_stdevs(dataset, means):
  stdevs = [0 for i in range(len(dataset[0]))]
  for i in range(len(dataset[0])):
    variance = [pow(row[i]-means[i], 2) for row in dataset]
    stdevs[i] = sum(variance)
  stdevs = [sqrt(x/(float(len(dataset)-1))) for x in stdevs]
  return stdevs

# standardize dataset
def standardize_dataset(dataset, means, stdevs):
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - means[i]) / stdevs[i]

# Load pima-indians-diabetes dataset
filename = 'pima-indians-diabetes.csv'
dataset = load_csv(filename)
print('Loaded data file {0} with {1} rows and {2} columns').format(filename, len(dataset),
    len(dataset[0]))
# convert string columns to float
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
print(dataset[0])
# Estimate mean and standard deviation
means = column_means(dataset)
stdevs = column_stdevs(dataset, means)
# standardize dataset
standardize_dataset(dataset, means, stdevs)
print(dataset[0])
```

Listing 2.19: Standardize the Diabetes Dataset.

Running the example prints the first row of the dataset, first in a raw format as loaded, and then standardized which allows us to see the difference for comparison.

```
Loaded data file pima-indians-diabetes.csv with 768 rows and 9 columns
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0, 1.0]
[0.6395304921176576, 0.8477713205896718, 0.14954329852954296, 0.9066790623472505,
    -0.692439324724129, 0.2038799072674717, 0.468186870229798, 1.4250667195933604,
    1.3650063669598067]
```

Listing 2.20: Example Output From Standardizing the Diabetes Dataset.

2.2.3 When to Normalize and Standardize

Standardization is a scaling technique that assumes your data conforms to a normal distribution. If a given data attribute is normal or close to normal, this is probably the scaling method to use. It is good practice to record the summary statistics used in the standardization process so that you can apply them when standardizing data in the future that you may want to use with your model. Normalization is a scaling technique that does not assume any specific distribution.

If your data is not normally distributed, consider normalizing it prior to applying your machine learning algorithm. It is good practice to record the minimum and maximum values for each column used in the normalization process, again, in case you need to normalize new data in the future to be used with your model.

2.3 Extensions

There are many other data transforms you could apply. The idea of data transforms is to best expose the structure of your problem in your data to the learning algorithm. It may not be clear what transforms are required upfront. A combination of trial and error and exploratory data analysis (plots and stats) can help tease out what may work. Below are some additional transforms you may want to consider researching and implementing:

- Normalization that permits a configurable range, such as -1 to 1 and more.
- Standardization that permits a configurable spread, such as 1, 2 or more standard deviations from the mean.
- Exponential transforms such as logarithm, square root and exponents.
- Power transforms such as Box-Cox for fixing the skew in normally distributed data.

2.4 Review

In this tutorial, you discovered how to rescale your data for machine learning from scratch. Specifically, you learned:

- How to normalize data from scratch.
- How to standardize data from scratch.
- When to use normalization or standardization on your data.

2.4.1 Further Reading

- Chapter 3 Data Pre-processing, page 27, *Applied Predictive Modeling*, 2013
  http://amzn.to/2e3lNXF

2.4.2 Next

In the next tutorial, you will discover how to estimate the skill of a predictive modeling algorithm on unseen data.

Chapter 3

Algorithm Evaluation Methods

The goal of predictive modeling is to create models that make good predictions on new data. We don't have access to this new data at the time of training, so we must use statistical methods to estimate the performance of a model on new data. This class of methods is called resampling methods, as they are resampling your available training data. In this tutorial, you will discover how to implement resampling methods from scratch in Python. After completing this tutorial, you will know:

- How to implement a train and test split of your data.
- How to implement a k-fold cross-validation split of your data.

Let's get started.

3.1 Description

The goal of resampling methods is to make the best use of your training data in order to accurately estimate the performance of a model on new unseen data. Accurate estimates of performance can then be used to help you choose which set of model parameters to use or which model to select.

Once you have chosen a model, you can train for final model on the entire training dataset and start using it to make predictions. There are two common resampling methods that you can use:

- A train and test split of your data.
- k-fold cross-validation.

In this tutorial, we will look at using each and when to use one method over the other.

3.2 Tutorial

This tutorial is divided into 3 parts:

1. Train and Test Split.

2. k-fold Cross-Validation Split.

3. How to Choose a Resampling Method.

These steps will provide the foundations you need to handle resampling your dataset to estimate algorithm performance on new data.

3.2.1 Train and Test Split

The train and test split is the easiest resampling method. As such, it is the most widely used. The train and test split involves separating a dataset into two parts:

1. Training Dataset.

2. Test Dataset.

The training dataset is used by the machine learning algorithm to train the model. The test dataset is held back and is used to evaluate the performance of the model. The rows assigned to each dataset are randomly selected. This is an attempt to ensure that the training and evaluation of a model is objective.

If multiple algorithms are compared or multiple configurations of the same algorithm are compared, the same train and test split of the dataset should be used. This is to ensure that the comparison of performance is consistent or apples-to-apples. We can achieve this by seeding the random number generator the same way before splitting the data, or by holding the same split of the dataset for use by multiple algorithms. We can implement the train and test split of a dataset in a single function.

Below is a function named `train_test_split()` to split a dataset into a train and test split. It accepts two arguments: the dataset to split as a list of lists and an optional split percentage. A default split percentage of 0.6 or 60% is used. This will assign 60% of the dataset to the training dataset and leave the remaining 40% to the test dataset. A 60/40 for train/test is a good default split of the data.

The function first calculates how many rows the training set requires from the provided dataset. A copy of the original dataset is made. Random rows are selected and removed from the copied dataset and added to the train dataset until the train dataset contains the target number of rows. The rows that remain in the copy of the dataset are then returned as the test dataset. The `randrange()` function from the random model is used to generate a random integer in the range between 0 and the size of the list.

```
# Split a dataset into a train and test set
def train_test_split(dataset, split=0.60):
  train = list()
  train_size = split * len(dataset)
  dataset_copy = list(dataset)
  while len(train) < train_size:
    index = randrange(len(dataset_copy))
    train.append(dataset_copy.pop(index))
  return train, dataset_copy
```

Listing 3.1: Function To Split a Dataset.

We can test this function using a contrived dataset of 10 rows, each with a single column. The complete example is listed below.

```
# Example of Splitting a Contrived Dataset into Train and Test
from random import seed
from random import randrange

# Split a dataset into a train and test set
def train_test_split(dataset, split=0.60):
  train = list()
  train_size = split * len(dataset)
  dataset_copy = list(dataset)
  while len(train) < train_size:
    index = randrange(len(dataset_copy))
    train.append(dataset_copy.pop(index))
  return train, dataset_copy

# test train/test split
seed(1)
dataset = [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]
train, test = train_test_split(dataset)
print(train)
print(test)
```

Listing 3.2: Example of Splitting a Contrived Dataset into Train and Test Splits.

The example fixes the random seed before splitting the training dataset. This is to ensure the exact same split of the data is made every time the code is executed. This is handy if we want to use the same split many times to evaluate and compare the performance of different algorithms. Running the example produces the output below. The data in the train and test set is printed, showing that 6/10 or 60% of the records were assigned to the training dataset and 4/10 or 40% of the records were assigned to the test set.

```
[[2], [9], [8], [3], [5], [6]]
[[1], [4], [7], [10]]
```

Listing 3.3: Example Output from Splitting a Dataset.

3.2.2 k-fold Cross-Validation Split

A limitation of using the train and test split method is that you get a noisy estimate of algorithm performance. The k-fold cross-validation method (also called just cross-validation) is a resampling method that provides a more accurate estimate of algorithm performance.

It does this by first splitting the data into k groups. The algorithm is then trained and evaluated k times and the performance summarized by taking the mean performance score. Each group of data is called a fold, hence the name k-fold cross-validation. It works by first training the algorithm on the k-1 groups of the data and evaluating it on the kth hold-out group as the test set. This is repeated so that each of the k groups is given an opportunity to be held out and used as the test set. As such, the value of k should be divisible by the number of rows in your training dataset, to ensure each of the k groups has the same number of rows.

You should choose a value for k that splits the data into groups with enough rows that each group is still representative of the original dataset. A good default to use is k=3 for a small

dataset or k=10 for a larger dataset. A quick way to check if the fold sizes are representative is to calculate summary statistics such as mean and standard deviation and see how much the values differ from the same statistics on the whole dataset. We can reuse what we learned in the previous section in creating a train and test split here in implementing k-fold cross-validation.

Instead of two groups, we must return k-folds or k groups of data. Below is a function named `cross_validation_split()` that implements the cross-validation split of data. As before, we create a copy of the dataset from which to draw randomly chosen rows. We calculate the size of each fold as the size of the dataset divided by the number of folds required.

$$\text{fold size} = \frac{count(rows)}{count(folds)} \tag{3.1}$$

If the dataset does not cleanly divide by the number of folds, there may be some remainder rows and they will not be used in the split. We then create a list of rows with the required size and add them to a list of folds which is then returned at the end.

```
# Split a dataset into $k$ folds
def cross_validation_split(dataset, folds=3):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / folds)
  for i in range(folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split
```

Listing 3.4: Function Create A Cross-Validation Split.

We can test this resampling method on the same small contrived dataset as above. Each row has only a single column value, but we can imagine how this might scale to a standard machine learning dataset. The complete example is listed below. As before, we fix the seed for the random number generator to ensure that each time the code is executed that the same rows are used in the same folds. A k value of 4 is used for demonstration purposes. We would expect that the 10 rows divided into 4 folds will result in 2 rows per fold, with a remainder of 2 that will not be used in the split.

```
# Example of Creating a Cross Validation Split
from random import seed
from random import randrange

# Split a dataset into k folds
def cross_validation_split(dataset, folds=3):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / folds)
  for i in range(folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
```

```
  return dataset_split

# test cross validation split
seed(1)
dataset = [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]
folds = cross_validation_split(dataset, 4)
print(folds)
```

Listing 3.5: Example of a Cross-Validation Split of a Contrived Dataset.

Running the example produces the output below. The list of the folds is printed, showing that indeed as expected there are two rows per fold.

```
[[[2], [9]], [[8], [3]], [[5], [6]], [[7], [10]]]
```

Listing 3.6: Example Output from Creating a Cross-Validation Split.

3.2.3 How to Choose a Resampling Method

The gold standard for estimating the performance of machine learning algorithms on new data is k-fold cross-validation. When well-configured, k-fold cross-validation gives a robust estimate of performance compared to other methods such as the train and test split. The downside of cross-validation is that it can be time-consuming to run, requiring k different models to be trained and evaluated. This is a problem if you have a very large dataset or if you are evaluating a model that takes a long time to train.

The train and test split resampling method is the most widely used. This is because it is easy to understand and implement, and because it gives a quick estimate of algorithm performance. Only a single model is constructed and evaluated. Although the train and test split method can give a noisy or unreliable estimate of the performance of a model on new data, this becomes less of a problem if you have a very large dataset.

Large datasets are those in the hundreds of thousands or millions of records, large enough that splitting it in half results in two datasets that have nearly equivalent statistical properties. In such cases, there may be little need to use k-fold cross-validation as an evaluation of the algorithm and a train and test split may be just as reliable.

3.3 Extensions

In this tutorial, we have looked at the two most common resampling methods. There are other methods you may want to investigate and implement as extensions to this tutorial. For example:

- **Repeated Train and Test**. This is where the train and test split is used, but the process is repeated many times.
- **LOOCV or Leave One Out Cross-Validation**. This is a form of k-fold cross-validation where the value of k is fixed at 1.
- **Stratification**. In classification problems, this is where the balance of class values in each group is forced to match the original dataset.

3.4 Review

In this tutorial, you discovered how to implement resampling methods in Python from scratch. Specifically, you learned:

- How to implement the train and test split method.
- How to implement the k-fold cross-validation method.
- When to use each method.

3.4.1 Further Reading

- Section 9.6, Generate pseudo-random numbers, *The Python Standard Library* https://docs.python.org/2/library/random.html
- Section 5.1. Cross Validation, page 176, *An Introduction to Statistical Learning*, 2014. http://amzn.to/2eeTyQX
- Section 18.4. Evaluating and Choosing the Best Hypothesis, page 708, *Artificial Intelligence: A Modern Approach*, 2010. http://amzn.to/2e3lFqP
- Section 4.4 Resampling Techniques, page 69, *Applied Predictive Modeling*, 2013 http://amzn.to/2e3lNXF
- Section 5.3, Cross-validation, page 149, *Data Mining: Practical Machine Learning Tools and Techniques*, second edition, 2005. http://amzn.to/2fj3SYY

3.4.2 Next

In the next tutorial, you will discover how to evaluate the predictions made by predictive modeling algorithms.

Chapter 4

Evaluation Metrics

After you make predictions, you need to know if they are any good. There are standard measures that we can use to summarize how good a set of predictions actually is. Knowing how good a set of predictions is allows you to make estimates about the skill of a given machine learning model of your problem. In this tutorial, you will discover how to implement four standard prediction evaluation metrics from scratch in Python.

After reading this tutorial, you will know:

- How to implement classification accuracy.
- How to implement and interpret a confusion matrix.
- How to implement mean absolute error for regression.
- How to implement root mean squared error for regression.

Let's get started.

4.1 Description

You must estimate the quality of a set of predictions when training a machine learning model. Performance metrics like classification accuracy and root mean squared error can give you a clear objective idea of how good a set of predictions is, and in turn how good the model is that generated them. This is important as it allows you to tell the difference and select among:

- Different transforms of the data used to train the same machine learning model.
- Different machine learning models trained on the same data.
- Different configurations for a machine learning model trained on the same data.

As such, performance metrics are a required building block in implementing machine learning algorithms from scratch.

4.2 Tutorial

This tutorial is divided into 4 parts:

1. Classification Accuracy.
2. Confusion Matrix.
3. Mean Absolute Error.
4. Root Mean Squared Error.

These steps will provide the foundations you need to handle evaluating predictions made by machine learning algorithms.

4.2.1 Classification Accuracy

A quick way to evaluate a set of predictions on a classification problem is by using accuracy. Classification accuracy is a ratio of the number of correct predictions out of all predictions that were made. It is often presented as a percentage between 0% for the worst possible accuracy and 100% for the best possible accuracy.

$$\text{accuracy} = \frac{\text{correct predictions}}{\text{total predictions}} \times 100 \tag{4.1}$$

We can implement this in a function that takes the expected outcomes and the predictions as arguments. Below is this function named `accuracy_metric()` that returns classification accuracy as a percentage. Notice that we use `==` to compare the equality actual to predicted values. This allows us to compare integers or strings, two main data types that we may choose to use when loading classification data.

```
# Calculate accuracy percentage between two lists
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0
```

Listing 4.1: Function To Calculate Classification Accuracy.

We can contrive a small dataset to test this function. Below are a set of 10 actual and predicted integer values. There are two mistakes in the set of predictions.

```
actuall  predicted
0        0
0        1
0        0
0        0
0        0
1        1
1        0
1        1
1        1
```

```
1        1
```

Listing 4.2: Example of a Set of Contrived Predictions and Expected Values.

Below is a complete example with this dataset to test the `accuracy_metric()` function.

```
# Example of calculating classification accuracy

# Calculate accuracy percentage between two lists
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Test accuracy
actual = [0,0,0,0,0,1,1,1,1,1]
predicted = [0,1,0,0,0,1,0,1,1,1]
accuracy = accuracy_metric(actual, predicted)
print(accuracy)
```

Listing 4.3: Example of Calculating Classification Accuracy.

Running this example produces the expected accuracy of 80% or 8/10.

```
80.0
```

Listing 4.4: Example Output From Calculating Classification Accuracy.

Accuracy is a good metric to use when you have a small number of class values, such as 2, also called a binary classification problem. Accuracy starts to lose it's meaning when you have more class values and you may need to review a different perspective on the results, such as a confusion matrix.

4.2.2 Confusion Matrix

A confusion matrix provides a summary of all of the predictions made compared to the expected actual values. The results are presented in a matrix with counts in each cell. The counts of actual class values are summarized horizontally, whereas the counts of predictions for each class values are presented vertically. A perfect set of predictions is shown as a diagonal line from the top left to the bottom right of the matrix.

The value of a confusion matrix for classification problems is that you can clearly see which predictions were wrong and the type of mistake that was made. Let's create a function to calculate a confusion matrix.

We can start off by defining the function to calculate the confusion matrix given a list of actual class values and a list of predictions. The function is listed below and is named `confusion_matrix()`. It first makes a list of all of the unique class values and assigns each class value a unique integer or index into the confusion matrix.

The confusion matrix is always square, with the number of class values indicating the number of rows and columns required. Here, the first index into the matrix is the row for actual values and the second is the column for predicted values. After the square confusion matrix is created and initialized to zero counts in each cell, it is a matter of looping through all predictions and

incrementing the count in each cell. The function returns two objects. The first is the set of unique class values, so that they can be displayed when the confusion matrix is drawn. The second is the confusion matrix itself with the counts in each cell.

```
# calculate a confusion matrix
def confusion_matrix(actual, predicted):
  unique = set(actual)
  matrix = [list() for x in range(len(unique))]
  for i in range(len(unique)):
    matrix[i] = [0 for x in range(len(unique))]
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for i in range(len(actual)):
    x = lookup[actual[i]]
    y = lookup[predicted[i]]
    matrix[x][y] += 1
  return unique, matrix
```

Listing 4.5: Function To Calculate a Confusion Matrix.

Let's make this concrete with an example. Below is another contrived dataset, this time with 3 mistakes.

```
actual   predicted
0        0
0        1
0        1
0        0
0        0
1        1
1        0
1        1
1        1
1        1
```

Listing 4.6: Example of a Set of Contrived Predictions and Expected Values.

We can calculate and print the confusion matrix for this dataset as follows:

```
# Example of Calculating a Confusion Matrix

# calculate a confusion matrix
def confusion_matrix(actual, predicted):
  unique = set(actual)
  matrix = [list() for x in range(len(unique))]
  for i in range(len(unique)):
    matrix[i] = [0 for x in range(len(unique))]
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for i in range(len(actual)):
    x = lookup[actual[i]]
    y = lookup[predicted[i]]
    matrix[x][y] += 1
  return unique, matrix
```

```
# Test confusion matrix with integers
actual = [0,0,0,0,0,1,1,1,1,1]
predicted = [0,1,1,0,0,1,0,1,1,1]
unique, matrix = confusion_matrix(actual, predicted)
print(unique)
print(matrix)
```

Listing 4.7: Example of Calculating a Confusion Matrix.

Running the example produces the output below. The example first prints the list of unique values and then the confusion matrix.

```
set([0, 1])
[[3, 2], [1, 4]]
```

Listing 4.8: Example Output From Calculating a Confusion Matrix.

It's hard to interpret the results this way. It would help if we could display the matrix as intended with rows and columns. Below is a function to correctly display the matrix. The function is named `print_confusion_matrix()`. It names the columns as P for Predictions and the rows as A for Actual. Each column and row are named for the class value to which it corresponds.

The matrix is laid out with the expectation that each class label is a single character or single digit integer and that the counts are also single digit integers. You could extend it to handle large class labels or prediction counts as an exercise.

```
# pretty print a confusion matrix
def print_confusion_matrix(unique, matrix):
  print('(P)' + ' '.join(str(x) for x in unique))
  print('(A)---')
  for i, x in enumerate(unique):
    print("%s| %s" % (x, ' '.join(str(x) for x in matrix[i])))
```

Listing 4.9: Function To Pretty Print a Confusion Matrix.

We can piece together all of the functions and display a human readable confusion matrix

```
# Example of Calculating and Displaying a Pretty Confusion Matrix

# calculate a confusion matrix
def confusion_matrix(actual, predicted):
  unique = set(actual)
  matrix = [list() for x in range(len(unique))]
  for i in range(len(unique)):
    matrix[i] = [0 for x in range(len(unique))]
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for i in range(len(actual)):
    x = lookup[actual[i]]
    y = lookup[predicted[i]]
    matrix[x][y] += 1
  return unique, matrix

# pretty print a confusion matrix
def print_confusion_matrix(unique, matrix):
  print('(P)' + ' '.join(str(x) for x in unique))
```

```
  print('(A)---')
  for i, x in enumerate(unique):
    print("%s| %s" % (x, ' '.join(str(x) for x in matrix[i])))

# Test confusion matrix with integers
actual = [0,0,0,0,0,1,1,1,1,1]
predicted = [0,1,1,0,0,1,0,1,1,1]
unique, matrix = confusion_matrix(actual, predicted)
print_confusion_matrix(unique, matrix)
```

Listing 4.10: Example of Calculating and Displaying a Pretty Confusion Matrix.

Running the example produces the output below. We can see the class labels of 0 and 1 across the top and bottom. Looking down the diagonal of the matrix from the top left to bottom right, we can see that 3 predictions of 0 were correct and 4 predictions of 1 were correct.

Looking in the other cells, we can see 2 + 1 or 3 prediction errors. We can see that 2 predictions were made as a 1 that were in fact actually a 0 class value. And we can see 1 prediction that was a 0 that was in fact actually a 1.

```
(P)0 1
(A)---
0| 3 2
1| 1 4
```

Listing 4.11: Example Output From Printing a Pretty Confusion Matrix.

A confusion matrix is always a good idea to use in addition to classification accuracy to help interpret the predictions.

4.2.3 Mean Absolute Error

Regression problems are those where a real value is predicted. An easy metric to consider is the error in the predicted values as compared to the expected values. The Mean Absolute Error or MAE for short is a good first error metric to use. It is calculated as the average of the absolute error values, where *absolute* means made positive so that they can be added together.

$$MAE = \frac{\sum_{i=1}^{n} abs(predicted_i - actual_i)}{\text{total predictions}} \tag{4.2}$$

Below is a function named `mae_metric()` that implements this metric. As above, it expects a list of actual outcome values and a list of predictions. We use the built-in `abs()` Python function to calculate the absolute error values that are summed together.

```
def mae_metric(actual, predicted):
  sum_error = 0.0
  for i in range(len(actual)):
    sum_error += abs(predicted[i] - actual[i])
```

Listing 4.12: Function To Calculate Mean Absolute Error.

We can contrive a small regression dataset to test this function.

```
actual   predicted
0.1      0.11
0.2      0.19
```

```
0.3      0.29
0.4      0.41
0.5      0.5
```

Listing 4.13: Small Set of Contrived Regression Predictions and Actual Values.

Only one prediction (0.5) is correct, whereas all other predictions are wrong by 0.01. Therefore, we would expect the mean absolute error (or the average positive error) for these predictions to be a little less than 0.01. Below is an example that tests the `mae_metric()` function with the contrived dataset.

```
# Example of Calculating Mean Absolute Error

# Calculate mean absolute error
def mae_metric(actual, predicted):
  sum_error = 0.0
  for i in range(len(actual)):
    sum_error += abs(predicted[i] - actual[i])
  return sum_error / float(len(actual))

# Test RMSE
actual = [0.1, 0.2, 0.3, 0.4, 0.5]
predicted = [0.11, 0.19, 0.29, 0.41, 0.5]
mae = mae_metric(actual, predicted)
print(mae)
```

Listing 4.14: Example of Calculating Mean Absolute Error.

Running this example prints the output below. We can see that as expected, the MAE was 0.008, a small value slightly lower than 0.01.

```
0.008
```

Listing 4.15: Example Output From Calculating the Mean Absolute Error.

4.2.4 Root Mean Squared Error

Another popular way to calculate the error in a set of regression predictions is to use the Root Mean Squared Error. Shortened as RMSE, the metric is sometimes called Mean Squared Error or MSE, dropping the Root part from the calculation and the name. RMSE is calculated as the square root of the mean of the squared differences between actual outcomes and predictions. Squaring each error forces the values to be positive, and the square root of the mean squared error returns the error metric back to the original units for comparison.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(predicted_i - actual_i)^2}{\text{total predictions}}} \tag{4.3}$$

Below is an implementation of this in a function named `rmse_metric()`. It uses the `sqrt()` function from the math module and uses the `**` operator to raise the error to the 2nd power.

```
# Calculate root mean squared error
def rmse_metric(actual, predicted):
  sum_error = 0.0
  for i in range(len(actual)):
```

```
    prediction_error = predicted[i] - actual[i]
    sum_error += (prediction_error ** 2)
  mean_error = sum_error / float(len(actual))
  return sqrt(mean_error)
```

Listing 4.16: Function To Calculate Root Mean Squared Error.

We can test this metric on the same dataset used to test the calculation of Mean Absolute Error above. Below is a complete example. Again, we would expect an error value to be generally close to 0.01.

```
# Example of Calculating the Root Mean Squared Error
from math import sqrt

# Calculate root mean squared error
def rmse_metric(actual, predicted):
  sum_error = 0.0
  for i in range(len(actual)):
    prediction_error = predicted[i] - actual[i]
    sum_error += (prediction_error ** 2)
  mean_error = sum_error / float(len(actual))
  return sqrt(mean_error)

# Test RMSE
actual = [0.1, 0.2, 0.3, 0.4, 0.5]
predicted = [0.11, 0.19, 0.29, 0.41, 0.5]
rmse = rmse_metric(actual, predicted)
print(rmse)
```

Listing 4.17: Example of Calculating Root Mean Squared Error.

Running the example, we see the results below. The result is slightly higher at 0.0089. RMSE values are always slightly higher than MSE values, which becomes more pronounced as the prediction errors increase. This is a benefit of using RMSE over MSE in that it penalizes larger errors with worse scores.

```
0.00894427191
```

Listing 4.18: Example Output From Calculating the Root Mean Squared Error.

4.3 Extensions

You have only seen a small sample of the most widely used performance metrics. There are many other performance metrics that you may require. Below is a list of 5 additional performance metrics that you may wish to implement to extend this tutorial

- Precision for classification.
- Recall for classification.
- F1 for classification.
- Area Under ROC Curve or AUC for classification.
- Goodness of Fit or R^2 (R squared) for regression.

4.4 Review

In this tutorial, you discovered how to implement algorithm prediction performance metrics from scratch in Python. Specifically, you learned:

- How to implement and interpret classification accuracy.
- How to implement and interpret the confusion matrix for classification problems.
- How to implement and interpret mean absolute error for regression.
- How to implement and interpret root mean squared error for regression.

4.4.1 Further Reading

- Section 9.2, Mathematical functions, *The Python Standard Library* https://docs.python.org/2/library/math.html

4.4.2 Next

In the next tutorial, you will discover how to develop a baseline of performance on your predictive modeling problem.

Chapter 5

Baseline Models

It is important to establish baseline performance on a predictive modeling problem. A baseline provides a point of comparison for the more advanced methods that you evaluate later. In this tutorial, you will discover how to implement baseline machine learning algorithms from scratch in Python.

After completing this tutorial, you will know:

- How to implement the random prediction algorithm.
- How to implement the zero rule prediction algorithm.

Let's get started.

5.1 Description

There are many machine learning algorithms to choose from. Hundreds in fact. You must know whether the predictions for a given algorithm are good or not. But how do you know? The answer is to use a baseline prediction algorithm. A baseline prediction algorithm provides a set of predictions that you can evaluate as you would any predictions for your problem, such as classification accuracy or RMSE.

The scores from these algorithms provide the required point of comparison when evaluating all other machine learning algorithms on your problem. Once established, you can comment on how much better a given algorithm is as compared to the naive baseline algorithm, providing context on just how good a given method actually is. The two most commonly used baseline algorithms are:

- Random Prediction Algorithm.
- Zero Rule Algorithm.

When starting on a new problem that is more sticky than a conventional classification or regression problem, it is a good idea to first devise a random prediction algorithm that is specific to your prediction problem. Later you can improve upon this and devise a zero rule algorithm. Let's implement these algorithms and see how they work.

5.2 Tutorial

This tutorial is divided into 2 parts:

1. Random Prediction Algorithm.
2. Zero Rule Algorithm.

These steps will provide the foundations you need to handle implementing and calculating baseline performance for your machine learning algorithms.

5.2.1 Random Prediction Algorithm

The random prediction algorithm predicts a random outcome as observed in the training data. It is perhaps the simplest algorithm to implement. It requires that you store all of the distinct outcome values in the training data, which could be large on regression problems with lots of distinct values.

Because random numbers are used to make decisions, it is a good idea to fix the random number seed prior to using the algorithm. This is to ensure that we get the same set of random numbers, and in turn the same decisions each time the algorithm is run. Below is an implementation of the Random Prediction Algorithm in a function named `random_algorithm()`. The function takes both a training dataset that includes output values and a test dataset for which output values must be predicted. The function will work for both classification and regression problems. It assumes that the output value in the training data is the final column for each row.

First, the set of unique output values is collected from the training data. Then a randomly selected output value from the set is selected for each row in the test set.

```
# Generate random predictions
def random_algorithm(train, test):
  output_values = [row[-1] for row in train]
  unique = list(set(output_values))
  predicted = list()
  for row in test:
    index = randrange(len(unique))
    predicted.append(unique[index])
  return predicted
```

Listing 5.1: Function To Make Random Predictions.

We can test this function with a small dataset that only contains the output column for simplicity. The output values in the training dataset are either `0` or `1`, meaning that the set of predictions the algorithm will choose from is `(0, 1)`. The test set also contains a single column, with no data as the predictions are not known.

```
# Example of Making Random Predictions
from random import seed
from random import randrange

# Generate random predictions
def random_algorithm(train, test):
  output_values = [row[-1] for row in train]
```

```
  unique = list(set(output_values))
  predicted = list()
  for row in test:
    index = randrange(len(unique))
    predicted.append(unique[index])
  return predicted

seed(1)
train = [[0], [1], [0], [1], [0], [1]]
test = [[None], [None], [None], [None]]
predictions = random_algorithm(train, test)
print(predictions)
```

Listing 5.2: Example of Making Random Predictions.

Running the example calculates random predictions for the test dataset and prints those predictions.

```
[0, 1, 1, 0]
```

Listing 5.3: Example Output From Making Random Predictions.

The random prediction algorithm is easy to implement and fast to run, but we could do better as a baseline.

5.2.2 Zero Rule Algorithm

The Zero Rule Algorithm is a better baseline than the random algorithm. It uses more information about a given problem to create one rule in order to make predictions. This rule is different depending on the problem type. Let's start with classification problems, predicting a class label.

Classification

For classification problems, the one rule is to predict the class value that is most common in the training dataset. This means that if a training dataset has 90 instances of class `0` and 10 instances of class `1` that it will predict 0 and achieve a baseline accuracy of 90/100 or 90%.

This is much better than the random prediction algorithm that would only achieve 82% accuracy on average. For details on how this is estimate for random search is calculated, see below:

$$\begin{aligned} &= ((0.9 \times 0.9) + (0.1 \times 0.1)) \times 100 \\ &= 82\% \end{aligned} \tag{5.1}$$

Below is a function named `zero_rule_algorithm_classification()` that implements this for the classification case.

```
# zero rule algorithm for classification
def zero_rule_algorithm_classification(train, test):
  output_values = [row[-1] for row in train]
  prediction = max(set(output_values), key=output_values.count)
  predicted = [prediction for i in range(len(test))]
  return predicted
```

Listing 5.4: Function To Make Zero Rule Classification Predictions.

The function makes use of the `max()` function with the key attribute, which is a little clever. Given a list of class values observed in the training data, the `max()` function takes a set of unique class values and calls the count on the list of class values for each class value in the set. The result is that it returns the class value that has the highest count of observed values in the list of class values observed in the training dataset.

If all class values have the same count, then we will choose the first class value observed in the dataset. Once we select a class value, it is used to make a prediction for each row in the test dataset. Below is a worked example with a contrived dataset that contains 4 examples of class `0` and 2 examples of class `1`. We would expect the algorithm to choose the class value `0` as the prediction for each row in the test dataset.

```
# Example of Zero Rule Classification Predictions
from random import seed
from random import randrange

# zero rule algorithm for classification
def zero_rule_algorithm_classification(train, test):
  output_values = [row[-1] for row in train]
  prediction = max(set(output_values), key=output_values.count)
  predicted = [prediction for i in range(len(test))]
  return predicted

seed(1)
train = [['0'], ['0'], ['0'], ['0'], ['1'], ['1']]
test = [[None], [None], [None], [None]]
predictions = zero_rule_algorithm_classification(train, test)
print(predictions)
```

Listing 5.5: Example of Zero Rule Classification Predictions.

Running this example makes the predictions and prints them to screen. As expected, the class value of 0 was chosen and predicted.

```
['0', '0', '0', '0']
```

Listing 5.6: Example Output From Making Zero Rule Classification Predictions.

Regression

Regression problems require the prediction of a real value. A good default prediction for real values is to predict the central tendency. This could be the mean or the median. A good default is to use the mean (also called the average) of the output value observed in the training data.

This is likely to have a lower error than random prediction which will return any observed output value. Below is a function to do that named `zero_rule_algorithm_regression()`. It works by calculating the mean value for the observed output values.

$$mean = \frac{\sum_{i=1}^{n} value_i}{count(values)} \tag{5.2}$$

Once calculated, the mean is then predicted for each row in the training data.

```
# zero rule algorithm for regression
def zero_rule_algorithm_regression(train, test):
  output_values = [row[-1] for row in train]
  prediction = sum(output_values) / float(len(output_values))
  predicted = [prediction for i in range(len(test))]
  return predicted
```

Listing 5.7: Function To Make Zero Rule Regression Predictions.

This function can be tested with a simple example. We can contrive a small dataset where the mean value is known to be 15.

```
10
15
12
15
18
20

mean = (10 + 15 + 12 + 15 + 18 + 20) / 6
mean = 90 / 6
mean = 15
```

Listing 5.8: Contrived Regression Dataset And Expected Mean.

Below is the complete example. We would expect that the mean value of 15 will be predicted for each of the 4 rows in the test dataset.

```
# Example of Zero Rule Regression Predictions
from random import seed
from random import randrange

# zero rule algorithm for regression
def zero_rule_algorithm_regression(train, test):
  output_values = [row[-1] for row in train]
  prediction = sum(output_values) / float(len(output_values))
  predicted = [prediction for i in range(len(test))]
  return predicted

seed(1)
train = [[10], [15], [12], [15], [18], [20]]
test = [[None], [None], [None], [None]]
predictions = zero_rule_algorithm_regression(train, test)
print(predictions)
```

Listing 5.9: Example of Zero Rule Regression Predictions.

Running the example calculates the predicted output values that are printed. As expected, the mean value of 15 is predicted for each row in the test dataset.

```
[15.0, 15.0, 15.0, 15.0, 15.0, 15.0]
```

Listing 5.10: Example Output From Making Zero Rule Regression Predictions.

5.3 Extensions

Below are a few extensions to the baseline algorithms that you may wish to investigate and implement as an extension to this tutorial.

- Alternate Central Tendency where the median, mode or other central tendency calculations are predicted instead of the mean.
- Moving Average for time series problems where the mean of the last n records is predicted.

5.4 Review

In this tutorial, you discovered the importance of calculating a baseline of performance on your machine learning problem.

You now know:

- How to implement a random prediction algorithm for classification and regression problems.
- How to implement a zero rule algorithm for classification and regression problems.

5.4.1 Next

This tutorial ends Part 1 on data preparation. Next, you will start Part 2 on linear algorithms. In the next tutorial, you will discover how to implement a test harness to evaluate predictive modeling algorithms consistently.

Part II

Linear Algorithms

Chapter 6

Algorithm Test Harnesses

We cannot know which algorithm will be best for a given problem. Therefore, we need to design a test harness that we can use to evaluate different machine learning algorithms. In this tutorial, you will discover how to develop a machine learning algorithm test harness from scratch in Python. After completing this tutorial, you will know:

- How to implement a train-test algorithm test harness.
- How to implement a k-fold cross-validation algorithm test harness.

Let's get started.

6.1 Description

A test harness provides a consistent way to evaluate machine learning algorithms on a dataset. It involves 3 elements:

1. The resampling method to split-up the dataset.
2. The machine learning algorithm to evaluate.
3. The performance measure by which to evaluate predictions.

The loading and preparation of a dataset is a prerequisite step that must have been completed prior to using the test harness. The test harness must allow for different machine learning algorithms to be evaluated, whilst the dataset, resampling method and performance measures are kept constant. In this tutorial, we are going to demonstrate the test harnesses with a real dataset.

6.1.1 Pima Indians Diabetes Dataset

In this tutorial we will use the Pima Indians Diabetes Dataset. This dataset involves the prediction of the onset of diabetes within 5 years. The baseline performance on the problem is approximately 65%. You can learn more about it in Appendix A, Section A.4. Download the dataset and save it into your current working directory with the filename `pima-indians-diabetes.csv`.

6.2 Tutorial

This tutorial is broken down into two main sections:

1. Train-Test Algorithm Test Harness.
2. Cross-Validation Algorithm Test Harness.

These test harnesses will give you the foundation that you need to evaluate a suite of machine learning algorithms on a given predictive modeling problem.

6.2.1 Train-Test Algorithm Test Harness

The train-test split is a simple resampling method that can be used to evaluate a machine learning algorithm. As such, it is a good starting point for developing a test harness. We can assume the prior development of a function to split a dataset into train and test sets and a function to evaluate the accuracy of a set of predictions.

We need a function that can take a dataset and an algorithm and return a performance score. Below is a function named `evaluate_algorithm()` that achieves this. It takes 3 fixed arguments including the dataset, the algorithm function and the split percentage for the train-test split.

First, the dataset is split into train and test elements. Next, a copy of the test set is made and each output value is cleared by setting it to the `None` value to prevent the algorithm from cheating accidentally.

The algorithm provided as a parameter is a function that expects the train and test datasets on which to prepare and then make predictions. The algorithm may require additional configuration parameters. This is handled by using the variable arguments `*args` in the `evaluate_algorithm()` function and passing them on to the algorithm function.

The algorithm function is expected to return a list of predictions, one for each row in the training dataset. These are compared to the actual output values from the unmodified test dataset by the `accuracy_metric()` function. Finally, the accuracy is returned.

```
# Evaluate an algorithm using a train/test split
def evaluate_algorithm(dataset, algorithm, split, *args):
  train, test = train_test_split(dataset, split)
  test_set = list()
  for row in test:
    row_copy = list(row)
    row_copy[-1] = None
    test_set.append(row_copy)
  predicted = algorithm(train, test_set, *args)
  actual = [row[-1] for row in test]
  accuracy = accuracy_metric(actual, predicted)
  return accuracy
```

Listing 6.1: Function To Evaluate An Algorithm Using a Train/Test Split.

The evaluation function does make some strong assumptions, but they can easily be changed if needed. Specifically, it assumes that the last row in the dataset is always the output value. A different column could be used. The use of the `accuracy_metric()` assumes that the problem is a classification problem, but this could be changed to mean squared error for regression problems.

Let's piece this together with a worked example. We will use the Pima Indians Diabetes dataset and evaluate the Zero Rule algorithm.

```
# Example of a Train-Test Test Harness
from random import seed
from random import randrange
from csv import reader

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Split a dataset into a train and test set
def train_test_split(dataset, split):
  train = list()
  train_size = split * len(dataset)
  dataset_copy = list(dataset)
  while len(train) < train_size:
    index = randrange(len(dataset_copy))
    train.append(dataset_copy.pop(index))
  return train, dataset_copy

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a train/test split
def evaluate_algorithm(dataset, algorithm, split, *args):
  train, test = train_test_split(dataset, split)
  test_set = list()
  for row in test:
    row_copy = list(row)
    row_copy[-1] = None
    test_set.append(row_copy)
  predicted = algorithm(train, test_set, *args)
  actual = [row[-1] for row in test]
  accuracy = accuracy_metric(actual, predicted)
  return accuracy

# zero rule algorithm for classification
```

```
def zero_rule_algorithm_classification(train, test):
  output_values = [row[-1] for row in train]
  prediction = max(set(output_values), key=output_values.count)
  predicted = [prediction for i in range(len(test))]
  return predicted

# Test the train/test harness
seed(1)
# load and prepare data
filename = 'pima-indians-diabetes.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
# evaluate algorithm
split = 0.6
accuracy = evaluate_algorithm(dataset, zero_rule_algorithm_classification, split)
print('Accuracy: %.3f%%' % (accuracy))
```

Listing 6.2: Example of Train/Test Algorithm Test Harness on the Diabetes Dataset.

The dataset was split into 60% for training the model and 40% for evaluating it. Notice how the name of the Zero Rule algorithm `zero_rule_algorithm_classification` was passed as an argument to the `evaluate_algorithm()` function. You can see how this test harness may be used again and again with different algorithms. Running the example above prints out the accuracy of the model.

```
Accuracy: 69.055%
```

Listing 6.3: Example Output From Using the Train/Test Split.

6.2.2 Cross-Validation Algorithm Test Harness

Cross-validation is a resampling technique that provides more reliable estimates of algorithm performance on unseen data. It requires the creation and evaluation of k models on different subsets of your data, and as such is more computationally expensive. Nevertheless, it is the gold standard for evaluating machine learning algorithms.

As in the previous section, we need to create a function that ties together the resampling method, the evaluation of the algorithm on the dataset and the performance calculation method. Unlike above, the algorithm must be evaluated on different subsets of the dataset many times. This means we need additional loops within our `evaluate_algorithm()` function.

Below is a function that implements algorithm evaluation with cross-validation. First, the dataset is split into `n_folds` groups called folds. Next, we loop giving each fold an opportunity to be held out of training and used to evaluate the algorithm. A copy of the list of folds is created and the held out fold is removed from this list. Then the list of folds is flattened into one long list of rows to match the algorithms expectation of a training dataset. This is done using the `sum()` function.

Once the training dataset is prepared the rest of the function within this loop is as above. A copy of the test dataset (the fold) is made and the output values are cleared to avoid accidental cheating by algorithms. The algorithm is prepared on the train dataset and makes predictions on the test dataset. The predictions are evaluated and stored in a list. Unlike the train-test algorithm test harness, a list of scores is returned, one for each cross-validation fold.

```
# Evaluate an algorithm using a cross-validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores
```

Listing 6.4: Function To Evaluate An Algorithm Using k-fold Cross-Validation.

Although slightly more complex in code and slower to run, this function provides a more robust estimate of algorithm performance. We can tie all of this together with a complete example on the diabetes dataset with the Zero Rule algorithm.

```
# Example of a Cross Validation Test Harness
from random import seed
from random import randrange
from csv import reader

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
```

```
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# zero rule algorithm for classification
def zero_rule_algorithm_classification(train, test):
  output_values = [row[-1] for row in train]
  prediction = max(set(output_values), key=output_values.count)
  predicted = [prediction for i in range(len(test))]
  return predicted

# Test cross validation test harness
seed(1)
# load and prepare data
filename = 'pima-indians-diabetes.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
# evaluate algorithm
n_folds = 5
scores = evaluate_algorithm(dataset, zero_rule_algorithm_classification, n_folds)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/len(scores)))
```

Listing 6.5: Example of the k-fold Cross-Validation Algorithm Test Harness on the Diabetes Dataset.

A total of 5 cross-validation folds were used to evaluate the Zero Rule Algorithm. As such, 5 scores were returned from the `evaluate_algorithm()` algorithm. Running this example both prints these list of scores calculated and prints the mean score.

```
Scores: [60.130718954248366, 62.745098039215684, 64.70588235294117, 71.89542483660131,
    66.01307189542483]
Mean Accuracy: 65.098%
```

Listing 6.6: Example Output From Using the Cross-Validation Test Harness.

You now have two different test harnesses that you can use to evaluate your own machine learning algorithms.

6.3 Extensions

This section lists extensions to this tutorial that you may wish to consider.

- **Parameterized Evaluation**. Pass in the function used to evaluate predictions, allowing you to seamlessly work with regression problems.
- **Parameterized Resampling**. Pass in the function used to calculate resampling splits, allowing you to easily switch between the train-test and cross-validation methods.
- **Standard Deviation Scores**. Calculate the standard deviation to get an idea of the spread of scores when evaluating algorithms using cross-validation.

6.4 Review

In this tutorial, you discovered how to create a test harness from scratch to evaluate your machine learning algorithms. Specifically, you now know:

- How to implement and use a train-test algorithm test harness.
- How to implement and use a cross-validation algorithm test harness.

6.4.1 Next

In the next tutorial, you will discover how to implement and apply the simple linear regression algorithm.

Chapter 7

Simple Linear Regression

Linear regression is a prediction method that is more than 200 years old. Simple linear regression is a great first machine learning algorithm to implement as it requires you to estimate properties from your training dataset, but is simple enough for beginners to understand. In this tutorial, you will discover how to implement the simple linear regression algorithm from scratch in Python.

After completing this tutorial you will know:

- How to estimate statistical quantities from training data.
- How to estimate linear regression coefficients from data.
- How to make predictions using linear regression for new data.

Let's get started.

7.1 Description

This section is divided into two parts: a description of the simple linear regression technique and a description of the dataset to which we will later apply it.

7.1.1 Simple Linear Regression

Linear regression assumes a linear or straight line relationship between the input variables (`X`) and the single output variable (`y`). More specifically, that output (`y`) can be calculated from a linear combination of the input variables (`X`). When there is a single input variable, the method is referred to as a simple linear regression.

In simple linear regression we can use statistics on the training data to estimate the coefficients required by the model to make predictions on new data. The line for a simple linear regression model can be written as:

$$y = b0 + b1 \times x \tag{7.1}$$

Where `b0` and `b1` are the coefficients we must estimate from the training data. Once the coefficients are known, we can use this equation to estimate output values for `y` given new input

examples of `x`. It requires that you calculate statistical properties from the data such as mean, variance and covariance.

All the algebra has been taken care of and we are left with some arithmetic to implement to estimate the simple linear regression coefficients. Briefly, we can estimate the coefficients as follows:

$$B1 = \frac{\sum_{i=1}^{n}((x_i - mean(x)) \times (y_i - mean(y)))}{\sum_{i=1}^{n}(x_i - mean(x))^2}$$
$$B0 = mean(y) - B1 \times mean(x) \tag{7.2}$$

Where the `i` refers to the value of the `i`th value of the input `x` or output `y`. Don't worry if this is not clear right now, these are the functions we will implement in the tutorial.

7.1.2 Swedish Auto Insurance Dataset

In this tutorial we will use the Swedish Auto Insurance Dataset. This dataset involves the prediction of total claim payments. The baseline RMSE on the problem is approximately 72.251 thousand Kronor. You can learn more about it in Appendix A, Section A.2. Download the dataset and save it into your current working directory with the filename `insurance.csv`. Note: you may need to convert the European comma (`,`) to the decimal dot (`.`). You will also need change the file from white-space-separated variables to CSV format.

7.2 Tutorial

This tutorial is broken down into five parts:

1. Calculate Mean and Variance.
2. Calculate Covariance.
3. Estimate Coefficients.
4. Make Predictions.
5. Swedish Auto Insurance Case Study.

These steps will give you the foundation you need to implement and train simple linear regression models for your own prediction problems.

7.2.1 Calculate Mean and Variance

The first step is to estimate the mean and the variance of both the input and output variables from the training data. The mean of a list of numbers can be calculated as:

$$mean(x) = \frac{\sum_{i=1} x_i}{count(x)} \tag{7.3}$$

Below is a function named `mean()` that implements this behavior for a list of numbers.

```
# Calculate the mean value of a list of numbers
def mean(values):
  return sum(values) / float(len(values))
```

Listing 7.1: Function To Calculate the Mean of a List of Numbers.

The variance is the sum squared difference for each value from the mean value. Variance for a list of numbers can be calculated as:

$$variance = \sum_{i=1}^{n}(x_i - mean(x))^2 \tag{7.4}$$

Below is a function named `variance()` that calculates the variance of a list of numbers. It requires the mean of the list to be provided as an argument, just so we don't have to calculate it more than once.

```
# Calculate the variance of a list of numbers
def variance(values, mean):
  return sum([(x-mean)**2 for x in values])
```

Listing 7.2: Function To Calculate the Variance of a List of Numbers.

We can put these two functions together and test them on a small, contrived dataset. Below is a small dataset of `x` and `y` values.

```
x,  y
1,  1
2,  3
4,  3
3,  2
5,  5
```

Listing 7.3: Small Contrived Dataset For Testing.

We can plot this dataset on a scatter plot graph as follows:

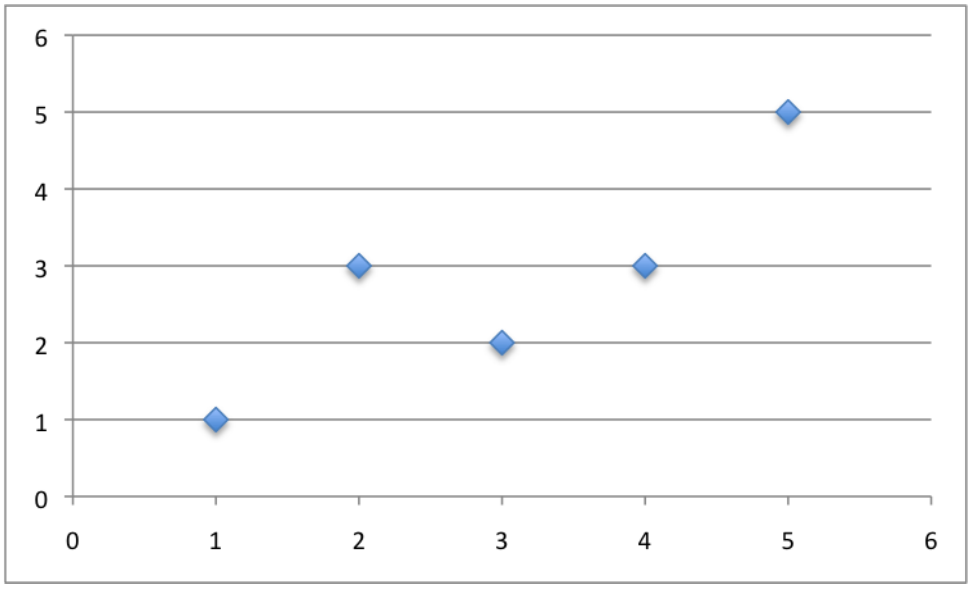

Figure 7.1: Plot of the Small Contrived Dataset.

We can calculate the mean and variance for both the x and y values in the example below.

```
# Example of Estimating Mean and Variance

# Calculate the mean value of a list of numbers
def mean(values):
  return sum(values) / float(len(values))

# Calculate the variance of a list of numbers
def variance(values, mean):
  return sum([(x-mean)**2 for x in values])

# calculate mean and variance
dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]
x = [row[0] for row in dataset]
y = [row[1] for row in dataset]
mean_x, mean_y = mean(x), mean(y)
var_x, var_y = variance(x, mean_x), variance(y, mean_y)
print('x stats: mean=%.3f variance=%.3f' % (mean_x, var_x))
print('y stats: mean=%.3f variance=%.3f' % (mean_y, var_y))
```

Listing 7.4: Example to Calculate Mean and Variance on the Contrived Dataset.

Running this example prints out the mean and variance for both columns.

```
x stats: mean=3.000 variance=10.000
y stats: mean=2.800 variance=8.800
```

Listing 7.5: Example Output of Mean and Variance on the Contrived Dataset.

This is our first step; next we need to put these values to use in calculating the covariance.

7.2.2 Calculate Covariance

The covariance of two groups of numbers describes how those numbers change together. Covariance is a generalization of correlation. Correlation describes the relationship between two groups of numbers, whereas covariance can describe the relationship between two or more groups of numbers. Additionally, covariance can be normalized to produce a correlation value. Nevertheless, we can calculate the covariance between two variables as follows:

$$covariance = \sum_{i=1}^{n}(x_i - mean(x)) \times (y_i - mean(y)) \tag{7.5}$$

Below is a function named `covariance()` that implements this statistic. It builds upon the previous step and takes the lists of `x` and `y` values as well as the mean of these values as arguments.

```
# Calculate covariance between x and y
def covariance(x, mean_x, y, mean_y):
  covar = 0.0
  for i in range(len(x)):
    covar += (x[i] - mean_x) * (y[i] - mean_y)
  return covar
```

Listing 7.6: Function To Calculate the Covariance.

We can test the calculation of the covariance on the same small contrived dataset as in the previous section. Putting it all together we get the example below.

```
# Example of Calculating Covariance

# Calculate the mean value of a list of numbers
def mean(values):
  return sum(values) / float(len(values))

# Calculate covariance between x and y
def covariance(x, mean_x, y, mean_y):
  covar = 0.0
  for i in range(len(x)):
    covar += (x[i] - mean_x) * (y[i] - mean_y)
  return covar

# calculate covariance
dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]
x = [row[0] for row in dataset]
y = [row[1] for row in dataset]
mean_x, mean_y = mean(x), mean(y)
covar = covariance(x, mean_x, y, mean_y)
print('Covariance: %.3f' % (covar))
```

Listing 7.7: Example to Calculate Covariance on the Contrived Dataset.

Running this example prints the covariance for the `x` and `y` variables.

```
Covariance: 8.000
```

Listing 7.8: Example Output of Calculating Covariance on the Contrived Dataset.

We now have all the pieces in place to calculate the coefficients for our model.

7.2.3 Estimate Coefficients

We must estimate the values for two coefficients in simple linear regression. The first is `B1` which can be estimated as:

$$B1 = \frac{\sum_{i=1}^{n}(x_i - mean(x) \times (y_i - mean(y))}{\sum_{i=1}^{n}(x_i - mean(x))^2} \tag{7.6}$$

We have learned some things above and can simplify this arithmetic to:

$$B1 = \frac{covariance(x, y)}{variance(x)} \tag{7.7}$$

We already have functions to calculate `covariance()` and `variance()`. Next, we need to estimate a value for `B0`, also called the intercept as it controls the starting point of the line where it intersects the y-axis.

$$B0 = mean(y) - B1 \times mean(x) \tag{7.8}$$

Again, we know how to estimate `B1` and we have a function to estimate `mean()`. We can put all of this together into a function named `coefficients()` that takes the dataset as an argument and returns the coefficients.

```
# Calculate coefficients
def coefficients(dataset):
  x = [row[0] for row in dataset]
  y = [row[1] for row in dataset]
  x_mean, y_mean = mean(x), mean(y)
  b1 = covariance(x, x_mean, y, y_mean) / variance(x, x_mean)
  b0 = y_mean - b1 * x_mean
  return [b0, b1]
```

Listing 7.9: Function To Calculate the Coefficients.

We can put this together with all of the functions from the previous two steps and test out the calculation of coefficients.

```
# Example of Calculating Coefficients

# Calculate the mean value of a list of numbers
def mean(values):
  return sum(values) / float(len(values))

# Calculate covariance between x and y
def covariance(x, mean_x, y, mean_y):
  covar = 0.0
  for i in range(len(x)):
    covar += (x[i] - mean_x) * (y[i] - mean_y)
  return covar

# Calculate the variance of a list of numbers
def variance(values, mean):
  return sum([(x-mean)**2 for x in values])

# Calculate coefficients
def coefficients(dataset):
  x = [row[0] for row in dataset]
  y = [row[1] for row in dataset]
  x_mean, y_mean = mean(x), mean(y)
  b1 = covariance(x, x_mean, y, y_mean) / variance(x, x_mean)
  b0 = y_mean - b1 * x_mean
  return [b0, b1]

# calculate coefficients
dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]
b0, b1 = coefficients(dataset)
print('Coefficients: B0=%.3f, B1=%.3f' % (b0, b1))
```

Listing 7.10: Example to Calculate Coefficients on the Contrived Dataset.

Running this example calculates and prints the coefficients.

```
Coefficients: B0=0.400, B1=0.800
```

Listing 7.11: Example Output of Calculating Coefficients on the Contrived Dataset.

Now that we know how to estimate the coefficients, the next step is to use them.

7.2.4 Make Predictions

The simple linear regression model is a line defined by coefficients estimated from training data. Once the coefficients are estimated, we can use them to make predictions. The equation to make predictions with a simple linear regression model is as follows:

$$y = b0 + b1 \times x \tag{7.9}$$

Below is a function named `simple_linear_regression()` that implements the prediction equation to make predictions on a test dataset. It also ties together the estimation of the coefficients on training data from the steps above. The coefficients prepared from the training data are used to make predictions on the test data, which are then returned.

```
def simple_linear_regression(train, test):
  predictions = list()
  b0, b1 = coefficients(train)
  for row in test:
    yhat = b0 + b1 * row[0]
    predictions.append(yhat)
  return predictions
```

Listing 7.12: Function To Run Simple Linear Regression.

Let's pull together everything we have learned and make predictions for our simple contrived dataset. As part of this example, we will also add in a function to manage the evaluation of the predictions called `evaluate_algorithm()` and another function to estimate the Root Mean Squared Error of the predictions called `rmse_metric()`. The full example is listed below.

```
# Example of Standalone Simple Linear Regression
from math import sqrt

# Calculate root mean squared error
def rmse_metric(actual, predicted):
  sum_error = 0.0
  for i in range(len(actual)):
    prediction_error = predicted[i] - actual[i]
    sum_error += (prediction_error ** 2)
  mean_error = sum_error / float(len(actual))
  return sqrt(mean_error)

# Evaluate regression algorithm on training dataset
def evaluate_algorithm(dataset, algorithm):
  test_set = list()
  for row in dataset:
    row_copy = list(row)
    row_copy[-1] = None
    test_set.append(row_copy)
  predicted = algorithm(dataset, test_set)
  print(predicted)
  actual = [row[-1] for row in dataset]
  rmse = rmse_metric(actual, predicted)
  return rmse

# Calculate the mean value of a list of numbers
def mean(values):
  return sum(values) / float(len(values))
```

```
# Calculate covariance between x and y
def covariance(x, mean_x, y, mean_y):
  covar = 0.0
  for i in range(len(x)):
    covar += (x[i] - mean_x) * (y[i] - mean_y)
  return covar

# Calculate the variance of a list of numbers
def variance(values, mean):
  return sum([(x-mean)**2 for x in values])

# Calculate coefficients
def coefficients(dataset):
  x = [row[0] for row in dataset]
  y = [row[1] for row in dataset]
  x_mean, y_mean = mean(x), mean(y)
  b1 = covariance(x, x_mean, y, y_mean) / variance(x, x_mean)
  b0 = y_mean - b1 * x_mean
  return [b0, b1]

# Simple linear regression algorithm
def simple_linear_regression(train, test):
  predictions = list()
  b0, b1 = coefficients(train)
  for row in test:
    yhat = b0 + b1 * row[0]
    predictions.append(yhat)
  return predictions

# Test simple linear regression
dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]
rmse = evaluate_algorithm(dataset, simple_linear_regression)
print('RMSE: %.3f' % (rmse))
```

Listing 7.13: Example of Simple Linear Regression on the Contrived Dataset.

Running this example displays the following output that first lists the predictions and the RMSE of these predictions.

```
[1.1999999999999995, 1.9999999999999996, 3.5999999999999996, 2.8, 4.399999999999999]
RMSE: 0.693
```

Listing 7.14: Example Output Simple Linear Regression on the Contrived Dataset.

Finally, we can plot the predictions as a line and compare it to the original dataset.

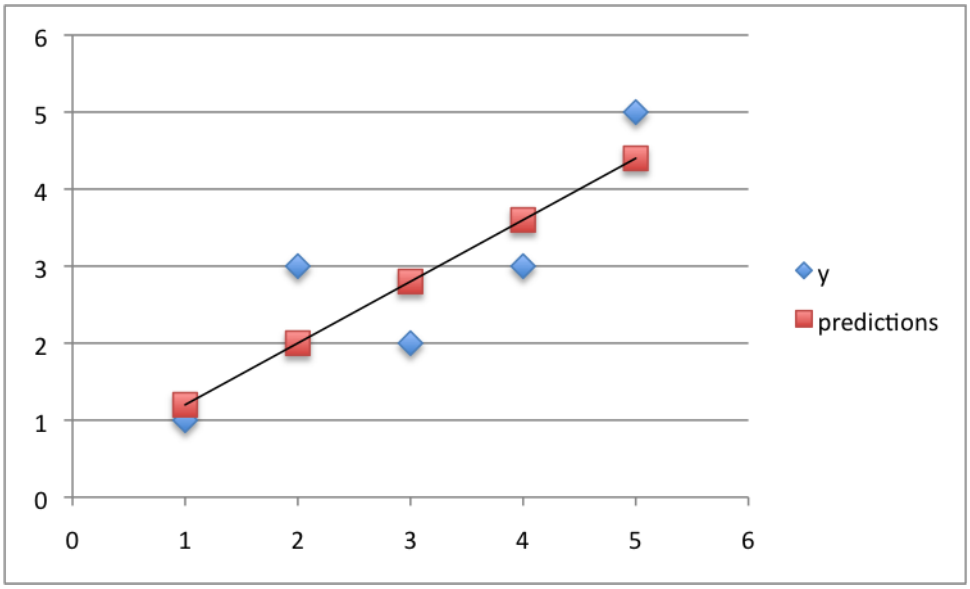

Figure 7.2: Plot of the Simple Linear Regression Predictions on the Contrived Dataset.

7.2.5 Swedish Auto Insurance Case Study

We now know how to implement a simple linear regression model. Let's apply it to the Swedish insurance dataset. This section assumes that you have downloaded the dataset to the file `insurance.csv` and it is available in the current working directory. We will add some convenience functions to the simple linear regression from the previous steps.

Specifically a function to load the CSV file called `load_csv()`, a function to convert a loaded dataset to numbers called `str_column_to_float()`, a function to evaluate an algorithm using a train and test set called `train_test_split()` a function to calculate RMSE called `rmse_metric()` and a function to evaluate an algorithm called `evaluate_algorithm()`.

The complete example is listed below. A training dataset of 60% of the data is used to prepare the model and predictions are made on the remaining 40%.

```
# Example of Simple Linear Regression on the Swedish Insurance Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
```

```
  for row in dataset:
    row[column] = float(row[column].strip())

# Split a dataset into a train and test set
def train_test_split(dataset, split):
  train = list()
  train_size = split * len(dataset)
  dataset_copy = list(dataset)
  while len(train) < train_size:
    index = randrange(len(dataset_copy))
    train.append(dataset_copy.pop(index))
  return train, dataset_copy

# Calculate root mean squared error
def rmse_metric(actual, predicted):
  sum_error = 0.0
  for i in range(len(actual)):
    prediction_error = predicted[i] - actual[i]
    sum_error += (prediction_error ** 2)
  mean_error = sum_error / float(len(actual))
  return sqrt(mean_error)

# Evaluate an algorithm using a train/test split
def evaluate_algorithm(dataset, algorithm, split, *args):
  train, test = train_test_split(dataset, split)
  test_set = list()
  for row in test:
    row_copy = list(row)
    row_copy[-1] = None
    test_set.append(row_copy)
  predicted = algorithm(train, test_set, *args)
  actual = [row[-1] for row in test]
  rmse = rmse_metric(actual, predicted)
  return rmse

# Calculate the mean value of a list of numbers
def mean(values):
  return sum(values) / float(len(values))

# Calculate covariance between x and y
def covariance(x, mean_x, y, mean_y):
  covar = 0.0
  for i in range(len(x)):
    covar += (x[i] - mean_x) * (y[i] - mean_y)
  return covar

# Calculate the variance of a list of numbers
def variance(values, mean):
  return sum([(x-mean)**2 for x in values])

# Calculate coefficients
def coefficients(dataset):
  x = [row[0] for row in dataset]
  y = [row[1] for row in dataset]
  x_mean, y_mean = mean(x), mean(y)
  b1 = covariance(x, x_mean, y, y_mean) / variance(x, x_mean)
```

```
  b0 = y_mean - b1 * x_mean
  return [b0, b1]

# Simple linear regression algorithm
def simple_linear_regression(train, test):
  predictions = list()
  b0, b1 = coefficients(train)
  for row in test:
    yhat = b0 + b1 * row[0]
    predictions.append(yhat)
  return predictions

# Simple linear regression on insurance dataset
seed(1)
# load and prepare data
filename = 'insurance.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
# evaluate algorithm
split = 0.6
rmse = evaluate_algorithm(dataset, simple_linear_regression, split)
print('RMSE: %.3f' % (rmse))
```

Listing 7.15: Example of Simple Linear Regression on the Insurance Dataset.

Running the algorithm prints the RMSE for the trained model on the training dataset. A score of about 38 (thousands of Kronor) was achieved, which is much better than the baseline performance of 72 (thousands of Kronor) on the same problem.

```
RMSE: 38.339
```

Listing 7.16: Example Output Simple Linear Regression on the Insurance Dataset.

7.3 Extensions

The best extension to this tutorial is to try out the algorithm on more problems. Small datasets with just an input (x) and output (y) columns are popular for demonstration in statistical books and courses. Many of these datasets are available online. Seek out some more small datasets and make predictions using simple linear regression.

7.4 Review

In this tutorial, you discovered how to implement the simple linear regression algorithm from scratch in Python. Specifically, you learned:

- How to estimate statistics from a training dataset like mean, variance and covariance.
- How to estimate model coefficients and use them to make predictions.
- How to use simple linear regression to make predictions on a real dataset.

7.4.1 Further Reading

- Section 3.1 Simple Linear Regression, page 61, *An Introduction to Statistical Learning*, 2014.
  http://amzn.to/2eeTyQX
- Section 18.6. Regression and Classification with Linear Models, page 717, *Artificial Intelligence: A Modern Approach*, 2010.
  http://amzn.to/2e3lFqP

7.4.2 Next

In the next tutorial, you will discover how to implement and apply the multivariate linear regression algorithm.

Chapter 8

Multivariate Linear Regression

The core of many machine learning algorithms is optimization. Optimization algorithms are used by machine learning algorithms to find a good set of model parameters given a training dataset. The most common optimization algorithm used in machine learning is stochastic gradient descent. In this tutorial, you will discover how to implement stochastic gradient descent to optimize a linear regression algorithm from scratch with Python.

After completing this tutorial, you will know:

- How to estimate linear regression coefficients using stochastic gradient descent.
- How to make predictions for multivariate linear regression.
- How to implement linear regression with stochastic gradient descent to make predictions on new data.

Let's get started.

8.1 Description

In this section, we will describe linear regression, the stochastic gradient descent technique and the Wine Quality dataset used in this tutorial.

8.1.1 Multivariate Linear Regression

Linear regression is a technique for predicting a real value. Confusingly, these problems where a real value is to be predicted are called regression problems. Linear regression is a technique where a straight line is used to model the relationship between input and output values. In more than two dimensions, this straight line may be thought of as a plane or hyperplane.

Predictions are made as a combination of the input values to predict the output value. Each input attribute (`x`) is weighted using a coefficient (`b`), and the goal of the learning algorithm is to discover a set of coefficients that results in good predictions (`y`).

$$y = b0 + b1 \times x1 + b2 \times x2 + ... \tag{8.1}$$

Coefficients can be found using stochastic gradient descent.

8.1.2 Stochastic Gradient Descent

Gradient Descent is the process of minimizing a function following the slope or gradient of that function. In machine learning, we can use a technique that evaluates and updates the coefficients every iteration called stochastic gradient descent to minimize the error of a model on our training data.

The way this optimization algorithm works is that each training instance is shown to the model one at a time. The model makes a prediction for a training instance, the error is calculated and the model is updated in order to reduce the error for the next prediction. This process is repeated for a fixed number of iterations.

This procedure can be used to find the set of coefficients in a model that result in the smallest error for the model on the training data. Each iteration, the coefficients (`b`) in machine learning language are updated using the equation:

$$b = b - \text{learning rate} \times \text{error} \times x \tag{8.2}$$

Where `b` is the coefficient or weight being optimized, `learning rate` is a learning rate that you must configure (e.g. 0.01), `error` is the prediction error for the model on the training data attributed to the weight, and `x` is the input value.

8.1.3 Wine Quality Dataset

In this tutorial we will use the Wine Quality Dataset. This dataset involves the prediction of white Wine Quality. The baseline RMSE on the problem is approximately 0.148 quality points. You can learn more about it in Appendix A, Section A.3. Download the dataset and save it into your current working directory with the filename `winequality-white.csv`. You must remove the header information from the file and convert the semicolon character (`;`) separator to the comma character (`,`) to meet CSV format.

8.2 Tutorial

This tutorial is broken down into 3 parts:

1. Making Predictions.
2. Estimating Coefficients.
3. Wine Quality Case Study.

This will provide the foundation you need to implement and apply linear regression with stochastic gradient descent on your own predictive modeling problems.

8.2.1 Making Predictions

The first step is to develop a function that can make predictions. This will be needed both in the evaluation of candidate coefficient values in stochastic gradient descent and after the model is finalized and we wish to start making predictions on test data or new data. Below is a function named `predict()` that predicts an output value for a row given a set of coefficients.

The first coefficient in is always the intercept, also called the bias or b0 as it is standalone and not responsible for a specific input value.

```
# Make a prediction with coefficients
def predict(row, coefficients):
  yhat = coefficients[0]
  for i in range(len(row)-1):
    yhat += coefficients[i + 1] * row[i]
  return yhat
```

Listing 8.1: Function To Make Predictions with Coefficients.

We can contrive a small dataset to test our prediction function.

```
x, y
1, 1
2, 3
4, 3
3, 2
5, 5
```

Listing 8.2: Small Contrived Dataset for Testing.

Below is a plot of this dataset.

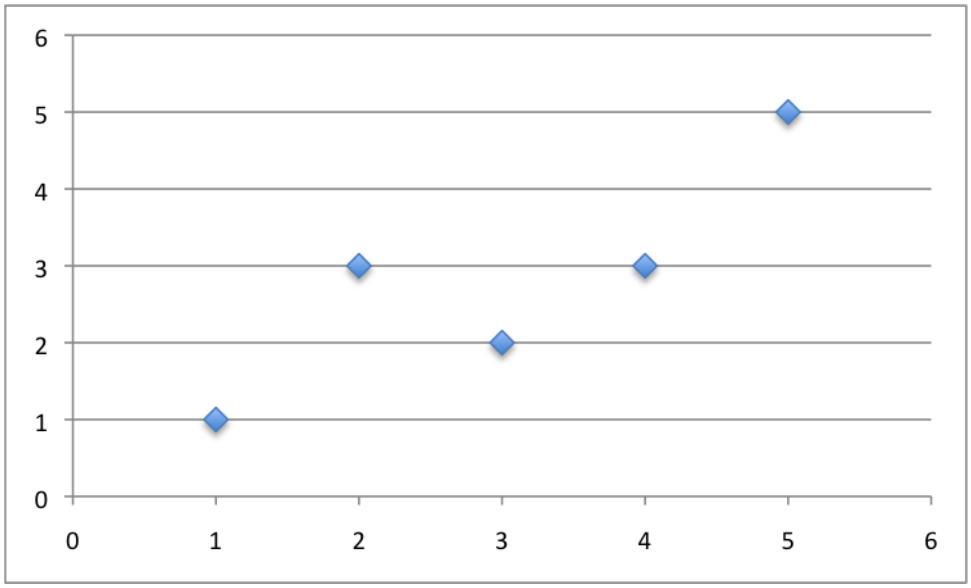

Figure 8.1: Plot of the Small Contrived Dataset.

We can also use previously prepared coefficients to make predictions for this dataset. Putting this all together we can test our `predict()` function below.

```
# Example of making a prediction with coefficients

# Make a prediction
def predict(row, coefficients):
  yhat = coefficients[0]
  for i in range(len(row)-1):
    yhat += coefficients[i + 1] * row[i]
```

```
  return yhat

dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]
coef = [0.4, 0.8]
for row in dataset:
  yhat = predict(row, coef)
  print("Expected=%.3f, Predicted=%.3f" % (row[-1], yhat))
```

Listing 8.3: Example of Making Predictions on the Contrived Dataset.

There is a single input value (`x`) and two coefficient values (`b0` and `b1`). The prediction equation we have modeled for this problem is:

$$y = b0 + b1 \times x \tag{8.3}$$

Or, with the specific coefficient values we chose by hand as:

$$y = 0.4 + 0.8 \times x \tag{8.4}$$

Running this function we get predictions that are reasonably close to the expected output (`y`) values.

```
Expected=1.000, Predicted=1.200
Expected=3.000, Predicted=2.000
Expected=3.000, Predicted=3.600
Expected=2.000, Predicted=2.800
Expected=5.000, Predicted=4.400
```

Listing 8.4: Example Output of Predictions on Contrived Dataset.

Now we are ready to implement stochastic gradient descent to optimize our coefficient values.

8.2.2 Estimating Coefficients

We can estimate the coefficient values for our training data using stochastic gradient descent. Stochastic gradient descent requires two parameters:

- **Learning Rate**: Used to limit the amount that each coefficient is corrected each time it is updated.
- **Epochs**: The number of times to run through the training data while updating the coefficients.

These, along with the training data will be the arguments to the function. There are 3 loops we need to perform in the function:

1. Loop over each epoch.
2. Loop over each row in the training data for an epoch.
3. Loop over each coefficient and update it for a row in an epoch.

As you can see, we update each coefficient for each row in the training data, each epoch. Coefficients are updated based on the error the model made. The error is calculated as the difference between the prediction made with the candidate coefficients and the expected output value.

$$error = prediction - expected \tag{8.5}$$

There is one coefficient to weight each input attribute, and these are updated in a consistent way, for example:

$$b1(t+1) = b1(t) - \text{learning rate} \times error(t) \times x1(t) \tag{8.6}$$

The special coefficient at the beginning of the list, also called the intercept or the bias, is updated in a similar way, except without an input as it is not associated with a specific input value:

$$b0(t+1) = b0(t) - \text{learning rate} \times error(t) \tag{8.7}$$

Now we can put all of this together. Below is a function named `coefficients_sgd()` that calculates coefficient values for a training dataset using stochastic gradient descent.

```
# Estimate linear regression coefficients using stochastic gradient descent
def coefficients_sgd(train, l_rate, n_epoch):
  coef = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    sum_error = 0
    for row in train:
      yhat = predict(row, coef)
      error = yhat - row[-1]
      sum_error += error**2
      coef[0] = coef[0] - l_rate * error
      for i in range(len(row)-1):
        coef[i + 1] = coef[i + 1] - l_rate * error * row[i]
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
  return coef
```

Listing 8.5: Function To Estimate Coefficients With Stochastic Gradient Descent.

You can see that in addition we keep track of the sum of the squared error (a positive value) each epoch so that we can print out a nice message in the outer loop. We can test this function on the same small contrived dataset from above.

```
# Example of estimating coefficients

# Make a prediction
def predict(row, coefficients):
  yhat = coefficients[0]
  for i in range(len(row)-1):
    yhat += coefficients[i + 1] * row[i]
  return yhat

# Estimate linear regression coefficients using stochastic gradient descent
def coefficients_sgd(train, l_rate, n_epoch):
  coef = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
```

```
    sum_error = 0
    for row in train:
      yhat = predict(row, coef)
      error = yhat - row[-1]
      sum_error += error**2
      coef[0] = coef[0] - l_rate * error
      for i in range(len(row)-1):
        coef[i + 1] = coef[i + 1] - l_rate * error * row[i]
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
  return coef

# Calculate coefficients
dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]
l_rate = 0.001
n_epoch = 50
coef = coefficients_sgd(dataset, l_rate, n_epoch)
print(coef)
```

Listing 8.6: Example of Estimating Coefficients on the Contrived Dataset.

We use a small learning rate of 0.001 and train the model for 50 epochs, or 50 exposures of the coefficients to the entire training dataset. Running the example prints a message each epoch with the sum squared error for that epoch and the final set of coefficients.

```
...
>epoch=45, lrate=0.001, error=2.650
>epoch=46, lrate=0.001, error=2.627
>epoch=47, lrate=0.001, error=2.607
>epoch=48, lrate=0.001, error=2.589
>epoch=49, lrate=0.001, error=2.573
[0.22998234937311363, 0.8017220304137576]
```

Listing 8.7: Example Output of Estimating Coefficients on the Contrived Dataset.

You can see how error continues to drop even in the final epoch. We could probably train for a lot longer (more epochs) or increase the amount we update the coefficients each epoch (higher learning rate). Experiment and see what you come up with. Now, let's apply this algorithm on a real dataset.

8.2.3 Wine Quality Case Study

In this section, we will train a linear regression model using stochastic gradient descent on the Wine Quality dataset. The example assumes that a CSV copy of the dataset is in the current working directory with the filename `winequality-white.csv`.

The dataset is first loaded, the string values converted to numeric and each column is normalized to values in the range of 0 to 1. This is achieved with helper functions `load_csv()` and `str_column_to_float()` to load and prepare the dataset and `dataset_minmax()` and `normalize_dataset()` to normalize it.

We will use k-fold cross-validation to estimate the performance of the learned model on unseen data. This means that we will construct and evaluate k models and estimate the performance as the mean model error. Root mean squared error will be used to evaluate each model. These behaviors are provided in the `cross_validation_split()`, `rmse_metric()` and `evaluate_algorithm()` helper functions.

We will use the `predict()`, `coefficients_sgd()` and `linear_regression_sgd()` functions created above to train the model. Below is the complete example.

```
# Linear Regression With Stochastic Gradient Descent for Wine Quality
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Find the min and max values for each column
def dataset_minmax(dataset):
  minmax = list()
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    value_min = min(col_values)
    value_max = max(col_values)
    minmax.append([value_min, value_max])
  return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate root mean squared error
def rmse_metric(actual, predicted):
```

```
  sum_error = 0.0
  for i in range(len(actual)):
    prediction_error = predicted[i] - actual[i]
    sum_error += (prediction_error ** 2)
  mean_error = sum_error / float(len(actual))
  return sqrt(mean_error)

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    rmse = rmse_metric(actual, predicted)
    scores.append(rmse)
  return scores

# Make a prediction with coefficients
def predict(row, coefficients):
  yhat = coefficients[0]
  for i in range(len(row)-1):
    yhat += coefficients[i + 1] * row[i]
  return yhat

# Estimate linear regression coefficients using stochastic gradient descent
def coefficients_sgd(train, l_rate, n_epoch):
  coef = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    for row in train:
      yhat = predict(row, coef)
      error = yhat - row[-1]
      coef[0] = coef[0] - l_rate * error
      for i in range(len(row)-1):
        coef[i + 1] = coef[i + 1] - l_rate * error * row[i]
      # print(l_rate, n_epoch, error)
  return coef

# Linear Regression Algorithm With Stochastic Gradient Descent
def linear_regression_sgd(train, test, l_rate, n_epoch):
  predictions = list()
  coef = coefficients_sgd(train, l_rate, n_epoch)
  for row in test:
    yhat = predict(row, coef)
    predictions.append(yhat)
  return(predictions)

# Linear Regression on wine quality dataset
```

```
seed(1)
# load and prepare data
filename = 'winequality-white.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
# normalize
minmax = dataset_minmax(dataset)
normalize_dataset(dataset, minmax)
# evaluate algorithm
n_folds = 5
l_rate = 0.01
n_epoch = 50
scores = evaluate_algorithm(dataset, linear_regression_sgd, n_folds, l_rate, n_epoch)
print('Scores: %s' % scores)
print('Mean RMSE: %.3f' % (sum(scores)/float(len(scores))))
```

Listing 8.8: Example of Multivariate Linear Regression on the Wine Quality Dataset.

A k value of 5 was used for cross-validation, giving each fold $\frac{4,898}{5} = 979.6$ or just under 1000 records to be evaluated upon each iteration. A learning rate of 0.01 and 50 training epochs were chosen with a little experimentation. You can try your own configurations and see if you can beat my score. Running this example prints the scores for each of the 5 cross-validation folds then prints the mean RMSE.

We can see that the RMSE (on the normalized dataset) is 0.126, lower than the baseline value of 0.148.

```
Scores: [0.12259834231519767, 0.12733924130891316, 0.12610773846663892, 0.1289950071681572,
    0.1272180783291014]
Mean RMSE: 0.126
```

Listing 8.9: Example Output of Linear Regression on the Wine Quality Dataset.

8.3 Extensions

This section lists a number of extensions to this tutorial that you may wish to consider exploring.

- **Tune The Example**. Tune the learning rate, number of epochs and even the data preparation method to get an improved score on the Wine Quality dataset.
- **Batch Stochastic Gradient Descent**. Change the stochastic gradient descent algorithm to accumulate updates across each epoch and only update the coefficients in a batch at the end of the epoch.
- **Additional Regression Problems**. Apply the technique to other regression problems on the UCI machine learning repository.

8.4 Review

In this tutorial, you discovered how to implement linear regression using stochastic gradient descent from scratch with Python. Specifically, you learned:

- How to make predictions for a multivariate linear regression problem.
- How to optimize a set of coefficients using stochastic gradient descent.
- How to apply the technique to a real regression predictive modeling problem.

8.4.1 Further Reading

- Section 3.2 Multiple Linear Regression, page 71, *An Introduction to Statistical Learning*, 2014.
  http://amzn.to/2eeTyQX
- Section 18.6. Regression and Classification with Linear Models, page 717, *Artificial Intelligence: A Modern Approach*, 2010.
  http://amzn.to/2e3lFqP
- Section 6.2 Linear Regression, page 105, *Applied Predictive Modeling*, 2013
  http://amzn.to/2e3lNXF
- Section 4.6, Linear Models, page 119, *Data Mining: Practical Machine Learning Tools and Techniques*, second edition, 2005.
  http://amzn.to/2fj3SYY

8.4.2 Next

In the next tutorial, you will discover how to implement and apply the logistic regression algorithm for classification.

Chapter 9

Logistic Regression

Logistic regression is the go-to linear classification algorithm for two-class problems. It is easy to implement, easy to understand and gets great results on a wide variety of problems, even when the expectations the method has for your data are violated. In this tutorial, you will discover how to implement logistic regression with stochastic gradient descent from scratch with Python. After completing this tutorial, you will know:

- How to make predictions with a logistic regression model.
- How to estimate coefficients using stochastic gradient descent.
- How to apply logistic regression to a real prediction problem.

Let's get started.

9.1 Description

This section will give a brief description of the logistic regression technique, stochastic gradient descent and the Pima Indians diabetes dataset we will use in this tutorial.

9.1.1 Logistic Regression

Logistic regression is named for the function used at the core of the method, the logistic function. Logistic regression uses an equation as the representation, very much like linear regression. Input values (`X`) are combined linearly using weights or coefficient values to predict an output value (`y`). A key difference from linear regression is that the output value being modeled is a binary value (0 or 1) rather than a numeric value.

$$yhat = \frac{e^{b0+b1\times x1}}{1 + e^{b0+b1\times x1}} \tag{9.1}$$

This can be simplified as:

$$yhat = \frac{1.0}{1.0 + e^{-(b0+b1\times x1)}} \tag{9.2}$$

Where `e` is the base of the natural logarithms (Euler's number), `yhat` is the predicted output, `b0` is the bias or intercept term and `b1` is the coefficient for the single input value (`x1`). The

yhat prediction is a real value between 0 and 1 that needs to be rounded to an integer value and mapped to a predicted class value.

Each column in your input data has an associated b coefficient (a constant real value) that must be learned from your training data. The actual representation of the model that you would store in memory or in a file is the coefficients in the equation (the beta value or b's). The coefficients of the logistic regression algorithm must be estimated from your training data.

9.1.2 Stochastic Gradient Descent

Logistic Regression uses gradient descent to update the coefficients. Gradient descent was introduced and described in Section 8.1.2. Each gradient descent iteration, the coefficients (`b`) in machine learning language are updated using the equation:

$$b = b + \text{learning rate} \times (y - yhat) \times yhat \times (1 - yhat) \times x \tag{9.3}$$

Where `b` is the coefficient or weight being optimized, `learning rate` is a learning rate that you must configure (e.g. 0.01), `(y - yhat)` is the prediction error for the model on the training data attributed to the weight, `yhat` is the prediction made by the coefficients and `x` is the input value.

9.1.3 Pima Indians Diabetes Dataset

In this tutorial we will use the Pima Indians Diabetes Dataset. This dataset involves the prediction of the onset of diabetes within 5 years. The baseline performance on the problem is approximately 65%. You can learn more about it in Appendix A, Section A.4. Download the dataset and save it into your current working directory with the filename `pima-indians-diabetes.csv`.

9.2 Tutorial

This tutorial is broken down into 3 parts.

1. Making Predictions.
2. Estimating Coefficients.
3. Pima Indians Diabetes Case Study.

This will provide the foundation you need to implement and apply logistic regression with stochastic gradient descent on your own predictive modeling problems.

9.2.1 Making Predictions

The first step is to develop a function that can make predictions. This will be needed both in the evaluation of candidate coefficient values in stochastic gradient descent and after the model is finalized and we wish to start making predictions on test data or new data. Below is a function named `predict()` that predicts an output value for a row given a set of coefficients. The first coefficient in is always the intercept, also called the bias or b0 as it is standalone and not responsible for a specific input value.

```
# Make a prediction with coefficients
def predict(row, coefficients):
  yhat = coefficients[0]
  for i in range(len(row)-1):
    yhat += coefficients[i + 1] * row[i]
  return 1.0 / (1.0 + exp(-yhat))
```

Listing 9.1: Function To Make Logistic Regression Predictions with Coefficients.

We can contrive a small dataset to test our `predict()` function.

```
X1          X2             Y
2.7810836   2.550537003    0
1.465489372  2.362125076   0
3.396561688  4.400293529   0
1.38807019   1.850220317   0
3.06407232   3.005305973   0
7.627531214  2.759262235   1
5.332441248  2.088626775   1
6.922596716  1.77106367    1
8.675418651  -0.242068655  1
7.673756466  3.508563011   1
```

Listing 9.2: Small Contrived Dataset for Testing Logistic Regression.

Below is a plot of the dataset using different colors to show the different classes for each point.

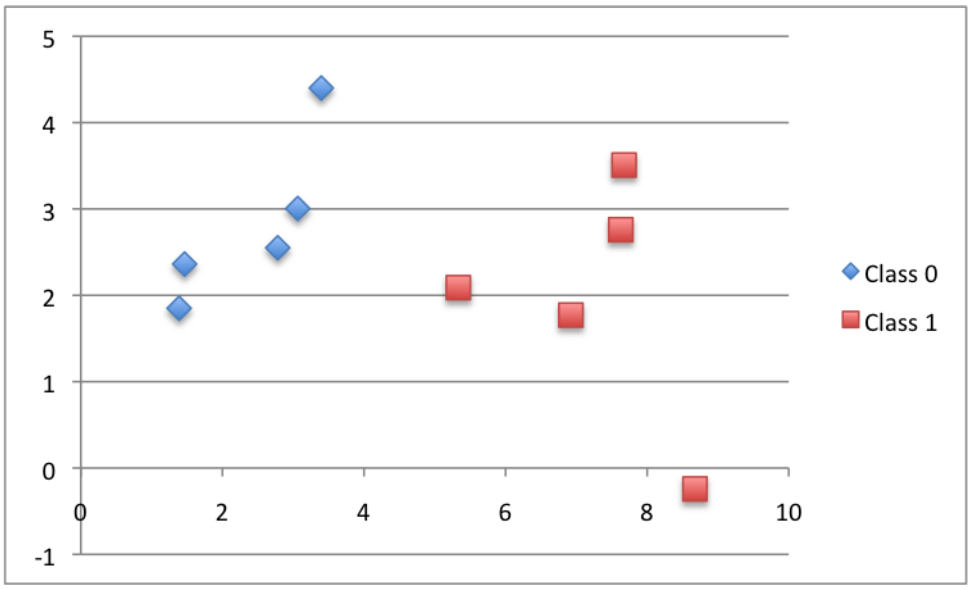

Figure 9.1: Plot of the Small Contrived Dataset for Testing Logistic Regression.

We can also use previously prepared coefficients to make predictions for this dataset. Putting this all together we can test our `predict()` function below.

```
# Example of making a prediction
from math import exp
```

```
# Make a prediction with coefficients
def predict(row, coefficients):
  yhat = coefficients[0]
  for i in range(len(row)-1):
    yhat += coefficients[i + 1] * row[i]
  return 1.0 / (1.0 + exp(-yhat))

# test predictions
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
coef = [-0.406605464, 0.852573316, -1.104746259]
for row in dataset:
  yhat = predict(row, coef)
  print("Expected=%.3f, Predicted=%.3f [%d]" % (row[-1], yhat, round(yhat)))
```

Listing 9.3: Example of Making Predictions on the Contrived Dataset.

There are two inputs values (`X1` and `X2`) and three coefficient values (`b0`, `b1` and `b2`). The prediction equation we have modeled for this problem is:

$$y = \frac{1.0}{1.0 + e^{-(b0+b1 \times X1+b2 \times X2)}} \tag{9.4}$$

Or, with the specific coefficient values we chose by hand as:

$$y = \frac{1.0}{1.0 + e^{-(-0.406605464+0.852573316 \times X1+-1.104746259 \times X2)}} \tag{9.5}$$

Running this function we get predictions that are reasonably close to the expected output (y) values and when rounded make correct predictions of the class.

```
Expected=0.000, Predicted=0.299 [0]
Expected=0.000, Predicted=0.146 [0]
Expected=0.000, Predicted=0.085 [0]
Expected=0.000, Predicted=0.220 [0]
Expected=0.000, Predicted=0.247 [0]
Expected=1.000, Predicted=0.955 [1]
Expected=1.000, Predicted=0.862 [1]
Expected=1.000, Predicted=0.972 [1]
Expected=1.000, Predicted=0.999 [1]
Expected=1.000, Predicted=0.905 [1]
```

Listing 9.4: Example Output From Making Predictions on the Contrived Dataset.

Now we are ready to implement stochastic gradient descent to optimize our coefficient values.

9.2.2 Estimating Coefficients

We can estimate the coefficient values for our training data using stochastic gradient descent. Stochastic gradient descent requires two parameters:

- **Learning Rate**: Used to limit the amount each coefficient is corrected each time it is updated.
- **Epochs**: The number of times to run through the training data while updating the coefficients.

These, along with the training data will be the arguments to the function. There are 3 loops we need to perform in the function:

1. Loop over each epoch.
2. Loop over each row in the training data for an epoch.
3. Loop over each coefficient and update it for a row in an epoch.

As you can see, we update each coefficient for each row in the training data, each epoch. Coefficients are updated based on the error the model made. The error is calculated as the difference between the expected output value and the prediction made with the candidate coefficients. There is one coefficient to weight each input attribute, and these are updated in a consistent way, for example:

$$b1(t+1) = b1(t) + \text{learning rate} \times (y(t) - yhat(t)) \times yhat(t) \times (1 - yhat(t)) \times x1(t) \quad (9.6)$$

The special coefficient at the beginning of the list, also called the intercept, is updated in a similar way, except without an input as it is not associated with a specific input value:

$$b0(t+1) = b0(t) + \text{learning rate} \times (y(t) - yhat(t)) \times yhat(t) \times (1 - yhat(t)) \quad (9.7)$$

Now we can put all of this together. Below is a function named `coefficients_sgd()` that calculates coefficient values for a training dataset using stochastic gradient descent.

```
# Estimate logistic regression coefficients using stochastic gradient descent
def coefficients_sgd(train, l_rate, n_epoch):
  coef = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    sum_error = 0
    for row in train:
      yhat = predict(row, coef)
      error = row[-1] - yhat
      sum_error += error**2
      coef[0] = coef[0] + l_rate * error * yhat * (1.0 - yhat)
      for i in range(len(row)-1):
        coef[i + 1] = coef[i + 1] + l_rate * error * yhat * (1.0 - yhat) * row[i]
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
  return coef
```

Listing 9.5: Function To Estimate Logistic Regression Coefficients.

You can see that in addition we keep track of the sum of the squared error (a positive value) each epoch so that we can print out a nice message each outer loop. We can test this function on the same small contrived dataset from above.

```
# Example of estimating coefficients
from math import exp

# Make a prediction with coefficients
def predict(row, coefficients):
  yhat = coefficients[0]
  for i in range(len(row)-1):
    yhat += coefficients[i + 1] * row[i]
  return 1.0 / (1.0 + exp(-yhat))

# Estimate logistic regression coefficients using stochastic gradient descent
def coefficients_sgd(train, l_rate, n_epoch):
  coef = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    sum_error = 0
    for row in train:
      yhat = predict(row, coef)
      error = row[-1] - yhat
      sum_error += error**2
      coef[0] = coef[0] + l_rate * error * yhat * (1.0 - yhat)
      for i in range(len(row)-1):
        coef[i + 1] = coef[i + 1] + l_rate * error * yhat * (1.0 - yhat) * row[i]
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
  return coef

# Calculate coefficients
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
l_rate = 0.3
n_epoch = 100
coef = coefficients_sgd(dataset, l_rate, n_epoch)
print(coef)
```

Listing 9.6: Example of Estimating Coefficients on the Contrived Dataset.

We use a larger learning rate of 0.3 and train the model for 100 epochs, or 100 exposures of the coefficients to the entire training dataset. Running the example prints a message each epoch with the sum squared error for that epoch and the final set of coefficients.

```
...
>epoch=95, lrate=0.300, error=0.023
>epoch=96, lrate=0.300, error=0.023
>epoch=97, lrate=0.300, error=0.023
>epoch=98, lrate=0.300, error=0.023
```

```
>epoch=99, lrate=0.300, error=0.022
[-0.8596443546618897, 1.5223825112460005, -2.218700210565016]
```

Listing 9.7: Example Output From Estimating Coefficients on the Contrived Dataset.

You can see how error continues to drop even in the final epoch. We could probably train for a lot longer (more epochs) or increase the amount we update the coefficients each epoch (higher learning rate). Experiment and see what you come up with. Now, let's apply this algorithm on a real dataset.

9.2.3 Pima Indians Diabetes Case Study

In this section, we will train a logistic regression model using stochastic gradient descent on the diabetes dataset. The example assumes that a CSV copy of the dataset is in the current working directory with the filename `pima-indians-diabetes.csv`.

The dataset is first loaded, the string values converted to numeric and each column is normalized to values in the range of 0 to 1. This is achieved with the helper functions `load_csv()` and `str_column_to_float()` to load and prepare the dataset and `dataset_minmax()` and `normalize_dataset()` to normalize it.

We will use k-fold cross-validation to estimate the performance of the learned model on unseen data. This means that we will construct and evaluate k models and estimate the performance as the mean model performance. Classification accuracy will be used to evaluate each model. These behaviors are provided in the `cross_validation_split()`, `accuracy_metric()` and `evaluate_algorithm()` helper functions.

We will use the `predict()` and `coefficients_sgd()` functions created above and a new `logistic_regression()` function to train the model. Below is the complete example.

```
# Logistic Regression on Diabetes Dataset
from random import seed
from random import randrange
from csv import reader
from math import exp

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Find the min and max values for each column
def dataset_minmax(dataset):
  minmax = list()
  for i in range(len(dataset[0])):
```

```
    col_values = [row[i] for row in dataset]
    value_min = min(col_values)
    value_max = max(col_values)
    minmax.append([value_min, value_max])
  return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# Make a prediction with coefficients
def predict(row, coefficients):
  yhat = coefficients[0]
  for i in range(len(row)-1):
```

```
    yhat += coefficients[i + 1] * row[i]
  return 1.0 / (1.0 + exp(-yhat))

# Estimate logistic regression coefficients using stochastic gradient descent
def coefficients_sgd(train, l_rate, n_epoch):
  coef = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    for row in train:
      yhat = predict(row, coef)
      error = row[-1] - yhat
      coef[0] = coef[0] + l_rate * error * yhat * (1.0 - yhat)
      for i in range(len(row)-1):
        coef[i + 1] = coef[i + 1] + l_rate * error * yhat * (1.0 - yhat) * row[i]
  return coef

# Linear Regression Algorithm With Stochastic Gradient Descent
def logistic_regression(train, test, l_rate, n_epoch):
  predictions = list()
  coef = coefficients_sgd(train, l_rate, n_epoch)
  for row in test:
    yhat = predict(row, coef)
    yhat = round(yhat)
    predictions.append(yhat)
  return(predictions)

# Test the logistic regression algorithm on the diabetes dataset
seed(1)
# load and prepare data
filename = 'pima-indians-diabetes.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
# normalize
minmax = dataset_minmax(dataset)
normalize_dataset(dataset, minmax)
# evaluate algorithm
n_folds = 5
l_rate = 0.1
n_epoch = 100
scores = evaluate_algorithm(dataset, logistic_regression, n_folds, l_rate, n_epoch)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 9.8: Example of Logistic Regression Applied to the Diabetes Dataset.

A k value of 5 was used for cross-validation, giving each fold $\frac{768}{5} = 153.6$ or just over 150 records to be evaluated upon each iteration. A learning rate of 0.1 and 100 training epochs were chosen with a little experimentation. You can try your own configurations and see if you can beat my score.

Running this example prints the scores for each of the 5 cross-validation folds, then prints the mean classification accuracy. We can see that the accuracy is about 77%, higher than the baseline value of 65%.

```
Scores: [73.20261437908496, 75.81699346405229, 75.81699346405229, 83.66013071895425,
    78.43137254901961]
Mean Accuracy: 77.386%
```

Listing 9.9: Example Output From Logistic Regression on the Diabetes Dataset.

9.3 Extensions

This section lists a number of extensions to this tutorial that you may wish to consider exploring.

- **Tune The Example**. Tune the learning rate, number of epochs and even data preparation method to get an improved score on the dataset.
- **Batch Stochastic Gradient Descent**. Change the stochastic gradient descent algorithm to accumulate updates across each epoch and only update the coefficients in a batch at the end of the epoch.
- **Additional Classification Problems**. Apply the technique to other binary (2 class) classification problems on the UCI machine learning repository.

9.4 Review

In this tutorial, you discovered how to implement logistic regression using stochastic gradient descent from scratch with Python. Specifically, you learned:

- How to make predictions for a multivariate classification problem.
- How to optimize a set of coefficients using stochastic gradient descent.
- How to apply the technique to a real classification predictive modeling problem.

9.4.1 Further Reading

- Section 4.3 Logistic Regression, page 130, *An Introduction to Statistical Learning*, 2014.
http://amzn.to/2eeTyQX
- Section 18.6. Regression and Classification with Linear Models, page 717, *Artificial Intelligence: A Modern Approach*, 2010.
http://amzn.to/2e3lFqP
- Section 12.2 Logistic Regression, page 282, *Applied Predictive Modeling*, 2013
http://amzn.to/2e3lNXF
- Section 4.6, Linear Models, page 119, *Data Mining: Practical Machine Learning Tools and Techniques*, second edition, 2005.
http://amzn.to/2fj3SYY

9.4.2 Next

In the next tutorial, you will discover how to implement and apply the Perceptron algorithm for classification.

Chapter 10

Perceptron

The Perceptron algorithm is the simplest type of artificial neural network. It is a model of a single neuron that can be used for two-class classification problems and provides the foundation for later developing much larger networks. In this tutorial, you will discover how to implement the Perceptron algorithm from scratch with Python. After completing this tutorial, you will know:

- How to train the network weights for the Perceptron.
- How to make predictions with the Perceptron.
- How to implement the Perceptron algorithm for a real-world classification problem.

Let's get started.

10.1 Description

This section provides a brief introduction to the Perceptron algorithm and the Sonar dataset to which we will later apply it.

10.1.1 Perceptron Algorithm

The Perceptron is inspired by the information processing of a single neural cell called a neuron. A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. In a similar way, the Perceptron receives input signals from examples of training data that we weight and combined in a linear equation called the activation.

$$activation = bias + \sum_{i=1}^{n} weight_i \times x_i \tag{10.1}$$

The activation is then transformed into an output value or prediction using a transfer function, such as the step transfer function.

$$prediction = 1.0 \text{ IF activation} \geq 0.0 \text{ ELSE } 0.0 \tag{10.2}$$

In this way, the Perceptron is a classification algorithm for problems with two classes (0 and 1) where a linear equation (like or hyperplane) can be used to separate the two classes. It is

closely related to linear regression and logistic regression that make predictions in a similar way (e.g. a weighted sum of inputs). The weights of the Perceptron algorithm must be estimated from your training data using stochastic gradient descent.

10.1.2 Stochastic Gradient Descent

The Perceptron algorithm uses gradient descent to update the weights. Gradient descent was introduced and described in Section 8.1.2. Each iteration of gradient descent, the weights (`w`) are updated using the equation:

$$w = w + \text{learning rate} \times (expected - predicted) \times x \tag{10.3}$$

Where `w` is weight being optimized, `learning rate` is a learning rate that you must configure (e.g. 0.01), `(expected - predicted)` is the prediction error for the model on the training data attributed to the weight and `x` is the input value.

10.1.3 Sonar Dataset

In this tutorial we will use the Sonar Dataset. This dataset involves the discrimination between mines and rocks. The baseline performance on the problem is approximately 53%. You can learn more about it in Appendix A, Section A.5. Download the dataset and save it into your current working directory with the filename `sonar.all-data.csv`.

10.2 Tutorial

This tutorial is broken down into 3 parts:

1. Making Predictions.
2. Training Network Weights.
3. Sonar Case Study.

These steps will give you the foundation to implement and apply the Perceptron algorithm to your own classification predictive modeling problems.

10.2.1 Making Predictions

The first step is to develop a function that can make predictions. This will be needed both in the evaluation of candidate weight values in stochastic gradient descent, and after the model is finalized and we wish to start making predictions on test data or new data. Below is a function named `predict()` that predicts an output value for a row given a set of weights. The first weight is always the bias as it is standalone and not responsible for a specific input value.

```
# Make a prediction with weights
def predict(row, weights):
  activation = weights[0]
  for i in range(len(row)-1):
    activation += weights[i + 1] * row[i]
```

```
  return 1.0 if activation >= 0.0 else 0.0
```

Listing 10.1: Function To Make Predictions with Perceptron Weights.

We can contrive a small dataset to test our prediction function.

```
X1          X2             Y
2.7810836   2.550537003    0
1.465489372 2.362125076    0
3.396561688 4.400293529    0
1.38807019  1.850220317    0
3.06407232  3.005305973    0
7.627531214 2.759262235    1
5.332441248 2.088626775    1
6.922596716 1.77106367     1
8.675418651 -0.242068655   1
7.673756466 3.508563011    1
```

Listing 10.2: Small Contrived Dataset for Testing Logistic Regression.

Below is a plot of the dataset using different colors to show the different classes for each point.

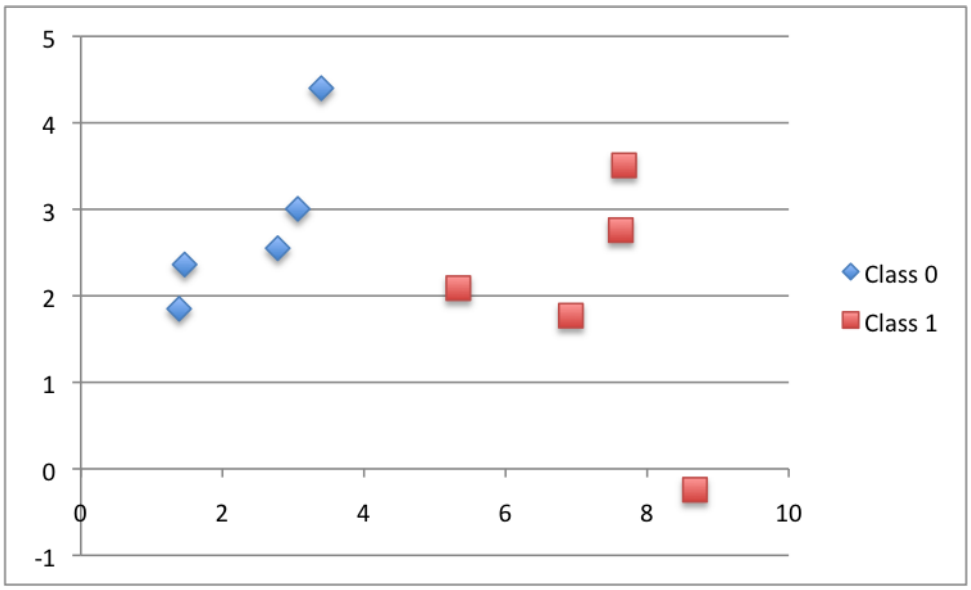

Figure 10.1: Plot of the Small Contrived Dataset for Testing the Perceptron algorithm.

We can also use previously prepared weights to make predictions for this dataset. Putting this all together we can test our `predict()` function below.

```
# Example of making predictions

# Make a prediction with weights
def predict(row, weights):
  activation = weights[0]
  for i in range(len(row)-1):
    activation += weights[i + 1] * row[i]
  return 1.0 if activation >= 0.0 else 0.0
```

```
# test predictions
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
weights = [-0.1, 0.20653640140000007, -0.23418117710000003]
for row in dataset:
  prediction = predict(row, weights)
  print("Expected=%d, Predicted=%d" % (row[-1], prediction))
```

Listing 10.3: Example of Making Predictions on the Contrived Dataset.

There are two inputs values (`X1` and `X2`) and three weight values (`bias`, `w1` and `w2`). The activation equation we have modeled for this problem is:

$$activation = (w1 \times X1) + (w2 \times X2) + bias \tag{10.4}$$

Or, with the specific weight values we chose by hand as:

$$activation = (0.206 \times X1) + (-0.234 \times X2) + -0.1 \tag{10.5}$$

Running this function we get predictions that match the expected output (`y`) values.

```
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
```

Listing 10.4: Example Output From Making Predictions on the Contrived Dataset.

Now we are ready to implement stochastic gradient descent to optimize our weight values.

10.2.2 Training Network Weights

We can estimate the weight values for our training data using stochastic gradient descent. Stochastic gradient descent requires two parameters:

- **Learning Rate**: Used to limit the amount each weight is corrected each time it is updated.
- **Epochs**: The number of times to run through the training data while updating the weight.

These, along with the training data will be the arguments to the function. There are 3 loops we need to perform in the function:

1. Loop over each epoch.
2. Loop over each row in the training data for an epoch.
3. Loop over each weight and update it for a row in an epoch.

As you can see, we update each weight for each row in the training data, each epoch. Weights are updated based on the error the model made. The error is calculated as the difference between the expected output value and the prediction made with the candidate weights.

There is one weight for each input attribute, and these are updated in a consistent way. For example:

$$w(t+1) = w(t) + \text{learning rate} \times (expected(t) - predicted(t)) \times x(t) \tag{10.6}$$

The bias is updated in a similar way, except without an input as it is not associated with a specific input value:

$$bias(t+1) = bias(t) + \text{learning rate} \times (expected(t) - predicted(t)) \tag{10.7}$$

Now we can put all of this together. Below is a function named `train_weights()` that calculates weight values for a training dataset using stochastic gradient descent.

```
# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
  weights = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    sum_error = 0.0
    for row in train:
      prediction = predict(row, weights)
      error = row[-1] - prediction
      sum_error += error**2
      weights[0] = weights[0] + l_rate * error
      for i in range(len(row)-1):
        weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
  return weights
```

Listing 10.5: Function To Estimate Weights for the Perceptron.

You can see that we also keep track of the sum of the squared error (a positive value) each epoch so that we can print out a nice message each outer loop. We can test this function on the same small contrived dataset from above.

```
# Example of training weights

# Make a prediction with weights
def predict(row, weights):
  activation = weights[0]
  for i in range(len(row)-1):
    activation += weights[i + 1] * row[i]
  return 1.0 if activation >= 0.0 else 0.0

# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
  weights = [0.0 for i in range(len(train[0]))]
```

```
  for epoch in range(n_epoch):
    sum_error = 0.0
    for row in train:
      prediction = predict(row, weights)
      error = row[-1] - prediction
      sum_error += error**2
      weights[0] = weights[0] + l_rate * error
      for i in range(len(row)-1):
        weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
  return weights

# Calculate weights
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
l_rate = 0.1
n_epoch = 5
weights = train_weights(dataset, l_rate, n_epoch)
print(weights)
```

Listing 10.6: Example of Estimating Weights on the Contrived Dataset.

We use a learning rate of 0.1 and train the model for only 5 epochs, or 5 exposures of the weights to the entire training dataset. Running the example prints a message each epoch with the sum squared error for that epoch and the final set of weights.

```
>epoch=0, lrate=0.100, error=2.000
>epoch=1, lrate=0.100, error=1.000
>epoch=2, lrate=0.100, error=0.000
>epoch=3, lrate=0.100, error=0.000
>epoch=4, lrate=0.100, error=0.000
[-0.1, 0.20653640140000007, -0.23418117710000003]
```

Listing 10.7: Example Output From Estimating Weights on the Contrived Dataset.

You can see how the problem is learned very quickly by the algorithm. Now, let's apply this algorithm on a real dataset.

10.2.3 Sonar Case Study

In this section, we will train a Perceptron model using stochastic gradient descent on the Sonar dataset. The example assumes that a CSV copy of the dataset is in the current working directory with the file name `sonar.all-data.csv`. The dataset is first loaded, the string values converted to numeric and the output column is converted from strings to the integer values of 0 to 1. This is achieved with helper functions `load_csv()`, `str_column_to_float()` and `str_column_to_int()` to load and prepare the dataset.

We will use k-fold cross-validation to estimate the performance of the learned model on unseen data. This means that we will construct and evaluate k models and estimate the performance as the mean model error. Classification accuracy will be used to evaluate each model. These behaviors are provided in the `cross_validation_split()`, `accuracy_metric()` and `evaluate_algorithm()` helper functions.

We will use the `predict()` and `train_weights()` functions created above to train the model and a new `perceptron()` function to tie them together. Below is the complete example.

```
# Perceptron Algorithm on the Sonar Dataset
from random import seed
from random import randrange
from csv import reader

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
```

```
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# Make a prediction with weights
def predict(row, weights):
  activation = weights[0]
  for i in range(len(row)-1):
    activation += weights[i + 1] * row[i]
  return 1.0 if activation >= 0.0 else 0.0

# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
  weights = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    for row in train:
      prediction = predict(row, weights)
      error = row[-1] - prediction
      weights[0] = weights[0] + l_rate * error
      for i in range(len(row)-1):
        weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
  return weights

# Perceptron Algorithm With Stochastic Gradient Descent
def perceptron(train, test, l_rate, n_epoch):
  predictions = list()
  weights = train_weights(train, l_rate, n_epoch)
  for row in test:
    prediction = predict(row, weights)
    predictions.append(prediction)
  return(predictions)

# Test the Perceptron algorithm on the sonar dataset
seed(1)
# load and prepare data
```

```
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
  str_column_to_float(dataset, i)
# convert string class to integers
str_column_to_int(dataset, len(dataset[0])-1)
# evaluate algorithm
n_folds = 3
l_rate = 0.01
n_epoch = 500
scores = evaluate_algorithm(dataset, perceptron, n_folds, l_rate, n_epoch)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 10.8: Example of the Perceptron Algorithm on the Sonar Dataset.

A k value of 3 was used for cross-validation, giving each fold $\frac{208}{3} = 69.3$ or just under 70 records to be evaluated upon each iteration. A learning rate of 0.1 and 500 training epochs were chosen with a little experimentation. You can try your own configurations and see if you can beat my score.

Running this example prints the scores for each of the 3 cross-validation folds then prints the mean classification accuracy. We can see that the accuracy is about 73%, higher than the baseline value of just over 50%.

```
Scores: [73.91304347826086, 78.26086956521739, 68.11594202898551]
Mean Accuracy: 73.430%
```

Listing 10.9: Example Output of the Perceptron Algorithm on the Sonar Dataset.

10.3 Extensions

This section lists extensions to this tutorial that you may wish to consider exploring.

- **Tune The Example**. Tune the learning rate, number of epochs and even data preparation method to get an improved score on the dataset.
- **Batch Stochastic Gradient Descent**. Change the stochastic gradient descent algorithm to accumulate updates across each epoch and only update the weights in a batch at the end of the epoch.
- **Additional Regression Problems**. Apply the technique to other classification problems on the UCI machine learning repository.

10.4 Review

In this tutorial, you discovered how to implement the Perceptron algorithm using stochastic gradient descent from scratch with Python. Specifically, you learned:

- How to make predictions for a binary classification problem.
- How to optimize a set of weights using stochastic gradient descent.
- How to apply the technique to a real classification predictive modeling problem.

10.4.1 Further Reading

- Section 18.6. Regression and Classification with Linear Models, page 727, *Artificial Intelligence: A Modern Approach*, 2010.
  http://amzn.to/2e3lFqP

- Section 4.6, Linear Models, page 119, *Data Mining: Practical Machine Learning Tools and Techniques*, second edition, 2005.
  http://amzn.to/2fj3SYY

10.4.2 Next

This ends Part 2 on linear algorithms. Next, in Part 3 you will look at nonlinear algorithms. In the next tutorial, you will discover how to implement and apply the decision tree algorithm for classification.

Part III

Nonlinear Algorithms

Chapter 11

Classification and Regression Trees

Decision trees are a powerful prediction method and extremely popular. They are popular because the final model is so easy to understand by practitioners and domain experts alike. The final decision tree can explain exactly why a specific prediction was made, making it very attractive for operational use.

Decision trees also provide the foundation for more advanced ensemble methods such as bagging, random forests and gradient boosting. In this tutorial, you will discover how to implement the Classification And Regression Tree algorithm from scratch with Python. After completing this tutorial, you will know:

- How to calculate and evaluate candidate split points in a data.
- How to arrange splits into a decision tree structure.
- How to apply the classification and regression tree algorithm to a real problem.

Let's get started.

11.1 Descriptions

This section provides a brief introduction to the Classification and Regression Tree algorithm and the Banknote dataset used in this tutorial.

11.1.1 Classification and Regression Trees

Classification and Regression Trees or CART for short is an acronym introduced by Leo Breiman to refer to Decision Tree algorithms that can be used for classification or regression predictive modeling problems. We will focus on using CART for classification in this tutorial. The representation of the CART model is a binary tree. This is the same binary tree from algorithms and data structures, nothing too fancy (each node can have zero, one or two child nodes).

A node represents a single input variable (`X`) and a split point on that variable, assuming the variable is numeric. The leaf nodes (also called terminal nodes) of the tree contain an output variable (`y`) which is used to make a prediction. Once created, a tree can be navigated with a new row of data following each branch with the splits until a final prediction is made.

Creating a binary decision tree is actually a process of dividing up the input space. A greedy approach is used called recursive binary splitting. This is a numerical procedure where all the values are lined up and different split points are tried and tested using a cost function. The split with the best cost (lowest cost because we minimize cost) is selected. All input variables and all possible split points are evaluated and chosen in a greedy manner based on the cost function.

- **Regression**: The cost function that is minimized to choose split points is the sum squared error across all training samples that fall within the rectangle.
- **Classification**: The Gini cost function is used which provides an indication of how pure the nodes are, where node purity refers to how mixed the training data assigned to each node is.

Splitting continues until nodes contain a minimum number of training examples or a maximum tree depth is reached.

11.1.2 Banknote Dataset

In this tutorial we will use the Banknote Dataset. This dataset involves the discrimination between authentic and inauthentic banknotes. The baseline performance on the problem is approximately 50%. You can learn more about it in Appendix A, Section A.6. Download the dataset and save it into your current working directory with the filename `data_banknote_authentication.csv`.

11.2 Tutorial

This tutorial is broken down into 5 parts:

1. Gini Index.
2. Create Split.
3. Build a Tree.
4. Make a Prediction.
5. Banknote Case Study.

These steps will give you the foundation that you need to implement the CART algorithm from scratch and apply it to your own predictive modeling problems.

11.2.1 Gini Index

The Gini index is the name of the cost function used to evaluate splits in the dataset. A split in the dataset involves one input attribute and one value for that attribute. It can be used to divide training patterns into two groups of rows.

A Gini score gives an idea of how good a split is by how mixed the classes are in the two groups created by the split. A perfect separation results in a Gini score of 0, whereas the worst

case split that results in 50/50 classes in each group results in a Gini score of 1.0 (for a 2 class problem).

Calculating Gini is best demonstrated with an example. We have two groups of data with 2 rows in each group. The rows in the first group all belong to class 0 and the rows in the second group belong to class 1, so it's a perfect split. We first need to calculate the proportion of classes in each group.

$$proportion = \frac{count(class\_value)}{count(rows)} \tag{11.1}$$

The proportions for this example would be:

$$\begin{aligned} \text{group\_1\_class\_0} &= \frac{2}{2} = 1 \\ \text{group\_1\_class\_1} &= \frac{0}{2} = 0 \\ \text{group\_2\_class\_0} &= \frac{0}{2} = 0 \\ \text{group\_2\_class\_1} &= \frac{2}{2} = 1 \end{aligned} \tag{11.2}$$

Gini is then calculated as follows:

$$\text{gini\_index} = \sum_{i=1}^{n}(proportion_i \times (1.0 - proportion_i)) \tag{11.3}$$

Across all of the proportions calculated for each group and each class value. In our case, this would be calculated as:

$$\begin{aligned} \text{gini\_index} = &(\text{group\_1\_class\_0} \times (1.0 - \text{group\_1\_class\_0}))+ \\ &(\text{group\_1\_class\_1} \times (1.0 - \text{group\_1\_class\_1}))+ \\ &(\text{group\_2\_class\_0} \times (1.0 - \text{group\_2\_class\_0}))+ \\ &(\text{grogroup\_2\_class\_1} \times (1.0 - \text{group\_2\_class\_1})) \end{aligned} \tag{11.4}$$

Or:

$$\text{gini\_index} = (0 + 0 + 0 + 0) = 0 \tag{11.5}$$

Below is a function named `gini_index()` that calculates the Gini index for a list of groups and a list of known class values. You can see that there are some safety checks in there to avoid a divide by zero for an empty group.

```
# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):
  gini = 0.0
  for class_value in class_values:
    for group in groups:
      size = len(group)
      if size == 0:
        continue
      proportion = [row[-1] for row in group].count(class_value) / float(size)
      gini += (proportion * (1.0 - proportion))
```

```
  return gini
```

Listing 11.1: Function To Calculate the Gini Index of a Dataset split.

We can test this function with our worked example above. We can also test it for the worst case of a 50/50 split in each group. The complete example is listed below.

```
# Example of calculatin Gini index

# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):
  gini = 0.0
  for class_value in class_values:
    for group in groups:
      size = len(group)
      if size == 0:
        continue
      proportion = [row[-1] for row in group].count(class_value) / float(size)
      gini += (proportion * (1.0 - proportion))
  return gini

# test Gini values
print(gini_index([[[1, 1], [1, 0]], [[1, 1], [1, 0]]], [0, 1]))
print(gini_index([[[1, 0], [1, 0]], [[1, 1], [1, 1]]], [0, 1]))
```

Listing 11.2: Example of Calculating Gini Index on a Contrived Dataset.

Running the example prints the two Gini scores, first the score for the worst case at 1.0 followed by the score for the best case at 0.0.

```
1.0
0.0
```

Listing 11.3: Example Output of Calculating Gini Index.

Now that we know how to evaluate the results of a split, let's look at creating splits.

11.2.2 Create Split

A split is comprised of an attribute in the dataset and a value. We can summarize this as the index of an attribute to split and the value by which to split rows on that attribute. This is just a useful shorthand for indexing into rows of data. Creating a split involves three parts, the first we have already looked at which is calculating the Gini score. The remaining two parts are:

1. Splitting a Dataset.
2. Evaluating All Splits.

Let's take a look at each.

Splitting a Dataset

Splitting a dataset means separating a dataset into two lists of rows given the index of an attribute and a split value for that attribute. Once we have the two groups, we can then use our Gini score above to evaluate the cost of the split. Splitting a dataset involves iterating over

each row, checking if the attribute value is below or above the split value and assigning it to the left or right group respectively. Below is a function named `test_split()` that implements this procedure.

```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
      right.append(row)
  return left, right
```

Listing 11.4: Function To Split a Dataset Based on a Split Point.

Not much to it. Note that the right group contains all rows with a value at the index above or equal to the split value.

Evaluating All Splits

With the Gini function above and the test split function we now have everything we need to evaluate splits. Given a dataset, we must check every value on each attribute as a candidate split, evaluate the cost of the split and find the best possible split we could make. Once the best split is found, we can use it as a node in our decision tree.

This is an exhaustive and greedy algorithm. We will use a dictionary to represent a node in the decision tree as we can store data by name. When selecting the best split and using it as a new node for the tree we will store the index of the chosen attribute, the value of that attribute by which to split and the two groups of data split by the chosen split point.

Each group of data is its own small dataset of just those rows assigned to the left or right group by the splitting process. You can imagine how we might split each group again, recursively as we build out our decision tree. Below is a function named `get_split()` that implements this procedure. You can see that it iterates over each attribute (except the class value) and then each value for that attribute, splitting and evaluating splits as it goes. The best split is recorded and then returned after all checks are complete.

```
# Select the best split point for a dataset
def get_split(dataset):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  for index in range(len(dataset[0])-1):
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

Listing 11.5: Function To Find the Best Split Point in a Dataset.

We can contrive a small dataset to test out this function and our whole dataset splitting process.

```
X1          X2          Y
```

```
2.771244718   1.784783929    0
1.728571309   1.169761413    0
3.678319846   2.81281357     0
3.961043357   2.61995032     0
2.999208922   2.209014212    0
7.497545867   3.162953546    1
9.00220326    3.339047188    1
7.444542326   0.476683375    1
10.12493903   3.234550982    1
6.642287351   3.319983761    1
```

Listing 11.6: Small Contrived Dataset For Testing CART.

We can plot this dataset using separate colors for each class. You can see that it would not be difficult to manually pick a value of `X1` (x-axis on the plot) to split this dataset.

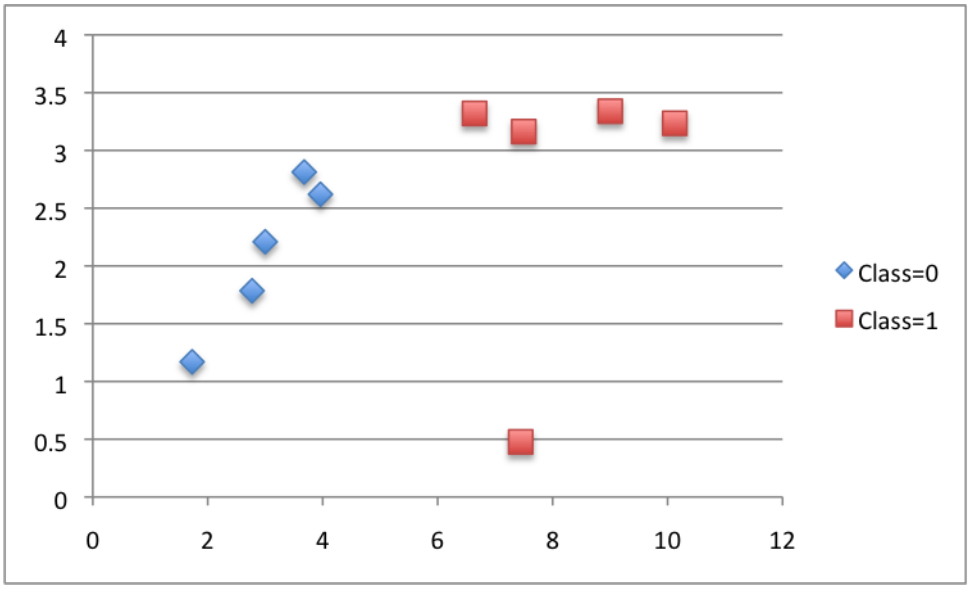

Figure 11.1: Plot of Small Contrived Dataset for Testing CART.

The example below puts all of this together.

```
# Example of getting the best split

# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
      right.append(row)
  return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):
  gini = 0.0
  for class_value in class_values:
```

```
    for group in groups:
      size = len(group)
      if size == 0:
        continue
      proportion = [row[-1] for row in group].count(class_value) / float(size)
      gini += (proportion * (1.0 - proportion))
  return gini

# Select the best split point for a dataset
def get_split(dataset):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  for index in range(len(dataset[0])-1):
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      print('X%d < %.3f Gini=%.3f' % ((index+1), row[index], gini))
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Test getting the best split
dataset = [[2.771244718,1.784783929,0],
  [1.728571309,1.169761413,0],
  [3.678319846,2.81281357,0],
  [3.961043357,2.61995032,0],
  [2.999208922,2.209014212,0],
  [7.497545867,3.162953546,1],
  [9.00220326,3.339047188,1],
  [7.444542326,0.476683375,1],
  [10.12493903,3.234550982,1],
  [6.642287351,3.319983761,1]]
split = get_split(dataset)
print('Split: [X%d < %.3f]' % ((split['index']+1), split['value']))
```

Listing 11.7: Example of Calculating The Best Split on a Contrived Dataset.

The `get_split()` function was modified to print out each split point and it's Gini index as it was evaluated. Running the example prints all of the Gini scores and then prints the score of best split in the dataset of $X1 < 6.642$ with a Gini Index of 0.0 or a perfect split.

```
X1 < 2.771 Gini=0.494
X1 < 1.729 Gini=0.500
X1 < 3.678 Gini=0.408
X1 < 3.961 Gini=0.278
X1 < 2.999 Gini=0.469
X1 < 7.498 Gini=0.408
X1 < 9.002 Gini=0.469
X1 < 7.445 Gini=0.278
X1 < 10.125 Gini=0.494
X1 < 6.642 Gini=0.000
X2 < 1.785 Gini=1.000
X2 < 1.170 Gini=0.494
X2 < 2.813 Gini=0.640
X2 < 2.620 Gini=0.819
X2 < 2.209 Gini=0.934
```

```
X2 < 3.163 Gini=0.278
X2 < 3.339 Gini=0.494
X2 < 0.477 Gini=0.500
X2 < 3.235 Gini=0.408
X2 < 3.320 Gini=0.469
Split: [X1 < 6.642]
```

Listing 11.8: Example Output of Fining the Best Split.

Now that we know how to find the best split points in a dataset or list of rows, let's see how we can use it to build out a decision tree.

11.2.3 Build a Tree

Creating the root node of the tree is easy. We call the above `get_split()` function using the entire dataset. Adding more nodes to our tree is more interesting. Building a tree may be divided into 3 main parts:

1. Terminal Nodes.
2. Recursive Splitting.
3. Building a Tree.

Terminal Nodes

We need to decide when to stop growing a tree. We can do that using the depth and the number of rows that the node is responsible for in the training dataset.

- **Maximum Tree Depth**. This is the maximum number of nodes from the root node of the tree. Once a maximum depth of the tree is met, we must stop adding new nodes. Deeper trees are more complex and are more likely to overfit the training data.
- **Minimum Node Records**. This is the minimum number of training patterns that a given node is responsible for. Once at or below this minimum, we must stop splitting and adding new nodes. Nodes that account for too few training patterns are expected to be too specific and are likely to overfit the training data.

These two approaches will be user-specified arguments to our tree building procedure. There is one more condition; it is possible to choose a split in which all rows belong to one group. In this case, we will be unable to continue splitting and adding child nodes as we will have no records to split on one side or another.

Now we have some ideas of when to stop growing the tree. When we do stop growing at a given point, that node is called a terminal node and is used to make a final prediction. This is done by taking the group of rows assigned to that node and selecting the most common class value in the group. This will be used to make predictions. Below is a function named `to_terminal()` that will select a class value for a group of rows. It returns the most common output value in a list of rows.

```
# Create a terminal node value
def to_terminal(group):
  outcomes = [row[-1] for row in group]
  return max(set(outcomes), key=outcomes.count)
```

Listing 11.9: Function To Create a Terminal Node.

Recursive Splitting

We know how and when to create terminal nodes; now we can build our tree. Building a decision tree involves calling the above developed `get_split()` function over and over again on the groups created for each node. New nodes added to an existing node are called child nodes. A node may have zero children (a terminal node), one child (one side makes a prediction directly) or two child nodes. We will refer to the child nodes as left and right in the dictionary representation of a given node.

Once a node is created, we can create child nodes recursively on each group of data from the split by calling the same function again. Below is a function that implements this recursive procedure. It takes a node as an argument as well as the maximum depth, minimum number of patterns in a node and the current depth of a node. You can imagine how this might be first called passing in the root node and the depth of 1. This function is best explained in steps:

1. Firstly, the two groups of data split by the node are extracted for use and deleted from the node. As we work on these groups the node no longer requires access to these data.
2. Next, we check if either left or right group of rows is empty and if so we create a terminal node using what records we do have.
3. We then check if we have reached our maximum depth and if so we create a terminal node.
4. We then process the left child, creating a terminal node if the group of rows is too small, otherwise creating and adding the left node in a depth first fashion until the bottom of the tree is reached on this branch.
5. The right side is then processed in the same manner, as we rise back up the constructed tree to the root.

```
# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
  left, right = node['groups']
  del(node['groups'])
  # check for a no split
  if not left or not right:
    node['left'] = node['right'] = to_terminal(left + right)
    return
  # check for max depth
  if depth >= max_depth:
    node['left'], node['right'] = to_terminal(left), to_terminal(right)
    return
  # process left child
  if len(left) <= min_size:
```

```
    node['left'] = to_terminal(left)
  else:
    node['left'] = get_split(left)
    split(node['left'], max_depth, min_size, depth+1)
  # process right child
  if len(right) <= min_size:
    node['right'] = to_terminal(right)
  else:
    node['right'] = get_split(right)
    split(node['right'], max_depth, min_size, depth+1)
```

Listing 11.10: Function To Create Split Points Recursively.

Building a Tree

We can now put all of the pieces together. Building the tree involves creating the root node and calling the `split()` function that then calls itself recursively to build out the whole tree. Below is the small `build_tree()` function that implements this procedure.

```
# Build a decision tree
def build_tree(train, max_depth, min_size):
  root = get_split(train)
  split(root, max_depth, min_size, 1)
  return root
```

Listing 11.11: Function To Create a Decision Tree.

We can test out this whole procedure using the small dataset we contrived above. Below is the complete example. Also included is a small `print_tree()` function that recursively prints out nodes of the decision tree with one line per node. Although not as striking as a real decision tree diagram, it gives an idea of the tree structure and decisions made throughout.

```
# Example of building a tree

# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
      right.append(row)
  return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):
  gini = 0.0
  for class_value in class_values:
    for group in groups:
      size = len(group)
      if size == 0:
        continue
      proportion = [row[-1] for row in group].count(class_value) / float(size)
      gini += (proportion * (1.0 - proportion))
  return gini
```

```
# Select the best split point for a dataset
def get_split(dataset):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  for index in range(len(dataset[0])-1):
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node value
def to_terminal(group):
  outcomes = [row[-1] for row in group]
  return max(set(outcomes), key=outcomes.count)

# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
  left, right = node['groups']
  del(node['groups'])
  # check for a no split
  if not left or not right:
    node['left'] = node['right'] = to_terminal(left + right)
    return
  # check for max depth
  if depth >= max_depth:
    node['left'], node['right'] = to_terminal(left), to_terminal(right)
    return
  # process left child
  if len(left) <= min_size:
    node['left'] = to_terminal(left)
  else:
    node['left'] = get_split(left)
    split(node['left'], max_depth, min_size, depth+1)
  # process right child
  if len(right) <= min_size:
    node['right'] = to_terminal(right)
  else:
    node['right'] = get_split(right)
    split(node['right'], max_depth, min_size, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size):
  root = get_split(train)
  split(root, max_depth, min_size, 1)
  return root

# Print a decision tree
def print_tree(node, depth=0):
  if isinstance(node, dict):
    print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value'])))
    print_tree(node['left'], depth+1)
    print_tree(node['right'], depth+1)
  else:
```

```
    print('%s[%s]' % ((depth*' ', node)))

dataset = [[2.771244718,1.784783929,0],
  [1.728571309,1.169761413,0],
  [3.678319846,2.81281357,0],
  [3.961043357,2.61995032,0],
  [2.999208922,2.209014212,0],
  [7.497545867,3.162953546,1],
  [9.00220326,3.339047188,1],
  [7.444542326,0.476683375,1],
  [10.12493903,3.234550982,1],
  [6.642287351,3.319983761,1]]
tree = build_tree(dataset, 1, 1)
print_tree(tree)
```

Listing 11.12: Example of Creating a Decision Tree From the Contrived Dataset.

We can vary the maximum depth argument as we run this example and see the effect on the printed tree. With a maximum depth of 1 (the second parameter in the call to the `build_tree()` function), we can see that the tree uses the perfect split we discovered in the previous section. This is a tree with one node, also called a decision stump.

```
[X1 < 6.642]
 [0]
 [1]
```

Listing 11.13: Example of a Decision Tree With Depth=1 (Decision Stump).

Increasing the maximum depth to 2, we are forcing the tree to make splits even when none are required. The `X1` attribute is then used again by both the left and right children of the root node to split up the already perfect mix of classes.

```
[X1 < 6.642]
 [X1 < 2.771]
  [0]
  [0]
 [X1 < 7.498]
  [1]
  [1]
```

Listing 11.14: Example of a Decision Tree With Depth=2.

Finally, and perversely, we can force one more level of splits with a maximum depth of 3.

```
[X1 < 6.642]
 [X1 < 2.771]
  [0]
  [X1 < 2.771]
   [0]
   [0]
 [X1 < 7.498]
  [X1 < 7.445]
   [1]
   [1]
  [X1 < 7.498]
   [1]
   [1]
```

Listing 11.15: Example of a Decision Tree With Depth=3.

These tests show that there is great opportunity to refine the implementation to avoid unnecessary splits. This is left as an extension. Now that we can create a decision tree, let's see how we can use it to make predictions on new data.

11.2.4 Make a Prediction

Making predictions with a decision tree involves navigating the tree with the specifically provided row of data. Again, we can implement this using a recursive function, where the same prediction routine is called again with the left or the right child nodes, depending on how the split affects the provided data. We must check if a child node is either a terminal value to be returned as the prediction, or if it is a dictionary node containing another level of the tree to be considered.

Below is the `predict()` function that implements this procedure. You can see how the index and value in a given node is used to evaluate whether the row of provided data falls on the left or the right of the split.

```
# Make a prediction with a decision tree
def predict(node, row):
  if row[node['index']] < node['value']:
    if isinstance(node['left'], dict):
      return predict(node['left'], row)
    else:
      return node['left']
  else:
    if isinstance(node['right'], dict):
      return predict(node['right'], row)
    else:
      return node['right']
```

Listing 11.16: Function To Make a Prediction With a Decision Tree.

We can use our contrived dataset to test this function. Below is an example that uses a hard-coded decision tree with a single node that best splits the data (a decision stump). The example makes a prediction for each row in the dataset.

```
# Example of making predictions

# Make a prediction with a decision tree
def predict(node, row):
  if row[node['index']] < node['value']:
    if isinstance(node['left'], dict):
      return predict(node['left'], row)
    else:
      return node['left']
  else:
    if isinstance(node['right'], dict):
      return predict(node['right'], row)
    else:
      return node['right']

# contrived dataset
dataset = [[2.771244718,1.784783929,0],
```

```
  [1.728571309,1.169761413,0],
  [3.678319846,2.81281357,0],
  [3.961043357,2.61995032,0],
  [2.999208922,2.209014212,0],
  [7.497545867,3.162953546,1],
  [9.00220326,3.339047188,1],
  [7.444542326,0.476683375,1],
  [10.12493903,3.234550982,1],
  [6.642287351,3.319983761,1]]
#  predict with a stump
stump = {'index': 0, 'right': 1, 'value': 6.642287351, 'left': 0}
for row in dataset:
  prediction = predict(stump, row)
  print('Expected=%d, Got=%d' % (row[-1], prediction))
```

Listing 11.17: Example of Making Predictions on the Contrived Dataset.

Running the example prints the correct prediction for each row, as expected.

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```

Listing 11.18: Example Output of Making a Prediction with a Decision Tree.

We now know how to create a decision tree and use it to make predictions. Now, let's apply it to a real dataset.

11.2.5 Banknote Case Study

This section applies the CART algorithm to the Bank Note dataset. The first step is to load the dataset and convert the loaded data to numbers that we can use to calculate split points. For this we will use the helper function `load_csv()` to load the file and `str_column_to_float()` to convert string numbers to floats.

We will evaluate the algorithm using k-fold cross-validation with 5 folds. This means that $\frac{1372}{5} = 274.4$ or just over 270 records will be used in each fold. We will use the helper functions `evaluate_algorithm()` to evaluate the algorithm with cross-validation and `accuracy_metric()` to calculate the accuracy of predictions.

A new function named `decision_tree()` was developed to manage the application of the CART algorithm, first creating the tree from the training dataset, then using the tree to make predictions on a test dataset. The complete example is listed below.

```
# Example of CART on the Banknote dataset
from random import seed
from random import randrange
from csv import reader
```

```
# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores
```

```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
      right.append(row)
  return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):
  gini = 0.0
  for class_value in class_values:
    for group in groups:
      size = len(group)
      if size == 0:
        continue
      proportion = [row[-1] for row in group].count(class_value) / float(size)
      gini += (proportion * (1.0 - proportion))
  return gini

# Select the best split point for a dataset
def get_split(dataset):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  for index in range(len(dataset[0])-1):
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node value
def to_terminal(group):
  outcomes = [row[-1] for row in group]
  return max(set(outcomes), key=outcomes.count)

# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
  left, right = node['groups']
  del(node['groups'])
  # check for a no split
  if not left or not right:
    node['left'] = node['right'] = to_terminal(left + right)
    return
  # check for max depth
  if depth >= max_depth:
    node['left'], node['right'] = to_terminal(left), to_terminal(right)
    return
  # process left child
  if len(left) <= min_size:
    node['left'] = to_terminal(left)
  else:
    node['left'] = get_split(left)
```

```
    split(node['left'], max_depth, min_size, depth+1)
  # process right child
  if len(right) <= min_size:
    node['right'] = to_terminal(right)
  else:
    node['right'] = get_split(right)
    split(node['right'], max_depth, min_size, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size):
  root = get_split(train)
  split(root, max_depth, min_size, 1)
  return root

# Make a prediction with a decision tree
def predict(node, row):
  if row[node['index']] < node['value']:
    if isinstance(node['left'], dict):
      return predict(node['left'], row)
    else:
      return node['left']
  else:
    if isinstance(node['right'], dict):
      return predict(node['right'], row)
    else:
      return node['right']

# Classification and Regression Tree Algorithm
def decision_tree(train, test, max_depth, min_size):
  tree = build_tree(train, max_depth, min_size)
  predictions = list()
  for row in test:
    prediction = predict(tree, row)
    predictions.append(prediction)
  return(predictions)

# Test CART on Bank Note dataset
seed(1)
# load and prepare data
filename = 'data_banknote_authentication.csv'
dataset = load_csv(filename)
# convert string attributes to integers
for i in range(len(dataset[0])):
  str_column_to_float(dataset, i)
# evaluate algorithm
n_folds = 5
max_depth = 5
min_size = 10
scores = evaluate_algorithm(dataset, decision_tree, n_folds, max_depth, min_size)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 11.19: Example of CART on the Banknote Dataset.

The example uses the max tree depth of 5 layers and the minimum number of rows per node to 10. These parameters to CART were chosen with a little experimentation, but by no means

are they optimal. Running the example prints the average classification accuracy on each fold as well as the average performance across all folds.

You can see that CART and the chosen configuration achieved a mean classification accuracy of about 83% which is dramatically better than the baseline performance of 50% accuracy.

```
Scores: [83.57664233576642, 82.84671532846716, 86.86131386861314, 79.92700729927007,
    82.11678832116789]
Mean Accuracy: 83.066%
```

Listing 11.20: Example Output for CART on the Banknote Dataset.

11.3 Extensions

This section lists extensions to this tutorial that you may wish to explore.

- **Algorithm Tuning**. The application of CART to the Bank Note dataset was not tuned. Experiment with different parameter values and see if you can achieve better performance.
- **Cross Entropy**. Another cost function for evaluating splits is cross entropy (logloss). You could implement and experiment with this alternative cost function.
- **Tree Pruning**. An important technique for reducing overfitting of the training dataset is to prune the trees. Investigate and implement tree pruning methods.
- **Categorical Dataset**. The example was designed for input data with numerical or ordinal input attributes, experiment with categorical input data and splits that may use equality instead of ranking.
- **Regression**. Adapt the tree for regression using a different cost function and method for creating terminal nodes.
- **More Datasets**. Apply the algorithm to more datasets on the UCI Machine Learning Repository.

11.4 Review

In this tutorial, you discovered how to implement the decision tree algorithm from scratch with Python. Specifically, you learned:

- How to select and evaluate split points in a training dataset.
- How to recursively build a decision tree from multiple splits.
- How to apply the CART algorithm to a real world classification predictive modeling problem.

11.4.1 Further Reading

- Section 8.1. The Basics of Decision Trees, page 303, *An Introduction to Statistical Learning*, 2014.
  http://amzn.to/2eeTyQX
- Section 18.3. Learning Decision Trees, page 697, *Artificial Intelligence: A Modern Approach*, 2010.
  http://amzn.to/2e3lFqP
- Section 8.1. Basic Regression Trees, page 175 and Section 14.1 Basic Regression, page 370, *Applied Predictive Modeling*, 2013
  http://amzn.to/2e3lNXF
- Section 4.3, Divide-and-conquer: Constructing Decision Trees, page 97 and Section 6.1, Decision Trees, page 189, *Data Mining: Practical Machine Learning Tools and Techniques*, second edition, 2005.
  http://amzn.to/2fj3SYY

11.4.2 Next

In the next tutorial, you will discover how to implement and apply the Naive Bayes algorithm for classification.

Chapter 12

Naive Bayes

We can use probability to make predictions. Perhaps the most widely used example is called the Naive Bayes algorithm. Not only is it straightforward to understand, but it also achieves surprisingly good results. In this tutorial you will discover how to implement the Naive Bayes algorithm from scratch in Python. After completing this tutorial you will know:

- How to calculate the probabilities required by the Naive Bayes algorithm.
- How to implement the Naive Bayes algorithm from scratch.
- How to apply Naive Bayes to a real-world predictive modeling problem.

Let's get started.

12.1 Descriptions

This section provides a brief overview of the Naive Bayes algorithm and the Iris flowers dataset that we will use in this tutorial.

12.1.1 Naive Bayes

Bayes' Theorem provides a way that we can calculate the probability of a piece of data belonging to a given class, given our prior knowledge. Bayes' Theorem is stated as:

$$P(class|data) = \frac{P(data|class) \times P(class)}{P(data)} \tag{12.1}$$

Where `P(class|data)` is the probability of class given the provided data. Naive Bayes is a classification algorithm for binary (two-class) and multiclass classification problems. It is called Naive Bayes or idiot Bayes because the calculations of the probabilities for each class are simplified to make their calculations tractable.

Rather than attempting to calculate the probabilities of each attribute value, they are assumed to be conditionally independent given the class value. This is a very strong assumption that is most unlikely in real data, i.e. that the attributes do not interact. Nevertheless, the approach performs surprisingly well on data where this assumption does not hold.

12.1.2 Iris Flower Species Dataset

In this tutorial we will use the Iris Flower Species Dataset. This dataset involves the prediction of iris flower species. The baseline performance on the problem is approximately 26%. You can learn more about it in Appendix A, Section A.7. Download the dataset and save it into your current working directory with the filename `iris.csv`.

12.2 Tutorial

This tutorial is broken down into 6 parts:

1. Separate By Class.
2. Summarize Dataset.
3. Summarize Data By Class.
4. Gaussian Probability Density Function.
5. Class Probabilities.
6. Iris Flower Species Case Study.

These steps will provide the foundation that you need to implement Naive Bayes from scratch and apply it to your own predictive modeling problems.

12.2.1 Separate By Class

We will need to calculate the probability of data by the class they belong to. This means that we will first need to separate our training data by class. A relatively straightforward operation. We can create a dictionary object where each key is the class value and then add a list of all the records as the value in the dictionary. Below is a function named `separate_by_class()` that implements this approach. It assumes that the last column in each row is the class value.

```
# Split the dataset by class values, returns a dictionary
def separate_by_class(dataset):
  separated = dict()
  for i in range(len(dataset)):
    vector = dataset[i]
    class_value = vector[-1]
    if (class_value not in separated):
      separated[class_value] = list()
    separated[class_value].append(vector)
  return separated
```

Listing 12.1: Function To Separate Rows By Class Value.

We can contrive a small dataset to test out this function.

```
X1           X2            Y
3.393533211  2.331273381   0
3.110073483  1.781539638   0
1.343808831  3.368360954   0
```

```
3.582294042  4.67917911    0
2.280362439  2.866990263   0
7.423436942  4.696522875   1
5.745051997  3.533989803   1
9.172168622  2.511101045   1
7.792783481  3.424088941   1
7.939820817  0.791637231   1
```

Listing 12.2: Small Contrived Dataset For Testing Naive Bayes.

We can plot this dataset and use separate colors for each class.

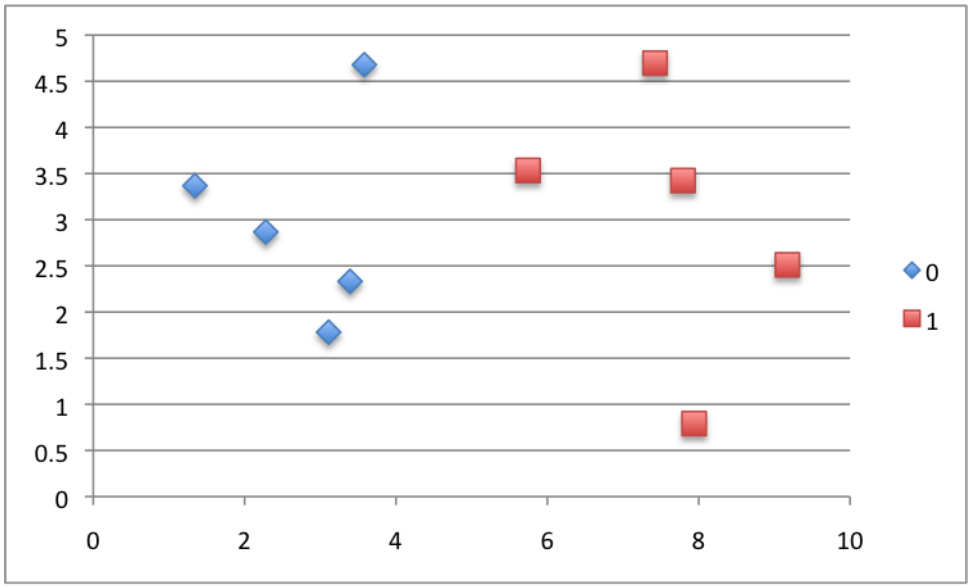

Figure 12.1: Plot of Small Contrived Dataset for Testing the Naive Bayes algorithm.

Putting this all together, we can test our `separate_by_class()` function on the contrived dataset.

```
# Example of separating data by class value

# Split the dataset by class values, returns a dictionary
def separate_by_class(dataset):
  separated = dict()
  for i in range(len(dataset)):
    vector = dataset[i]
    class_value = vector[-1]
    if (class_value not in separated):
      separated[class_value] = list()
    separated[class_value].append(vector)
  return separated

# Test separating data by class
dataset = [[3.393533211,2.331273381,0],
  [3.110073483,1.781539638,0],
  [1.343808831,3.368360954,0],
  [3.582294042,4.67917911,0],
  [2.280362439,2.866990263,0],
```

```
  [7.423436942,4.696522875,1],
  [5.745051997,3.533989803,1],
  [9.172168622,2.511101045,1],
  [7.792783481,3.424088941,1],
  [7.939820817,0.791637231,1]]
separated = separate_by_class(dataset)
for label in separated:
  print(label)
  for row in separated[label]:
    print(row)
```

Listing 12.3: Example of Separating Rows By Class Value on a Contrived Dataset.

Running the example sorts observations in the dataset by their class value, then prints the class value followed by all identified records.

```
0
[3.393533211, 2.331273381, 0]
[3.110073483, 1.781539638, 0]
[1.343808831, 3.368360954, 0]
[3.582294042, 4.67917911, 0]
[2.280362439, 2.866990263, 0]
1
[7.423436942, 4.696522875, 1]
[5.745051997, 3.533989803, 1]
[9.172168622, 2.511101045, 1]
[7.792783481, 3.424088941, 1]
[7.939820817, 0.791637231, 1]
```

Listing 12.4: Sample Output For Separating Rows by Class Value.

Next we can start to develop the functions needed to collect statistics.

12.2.2 Summarize Dataset

We need two statistics from a given set of data. We'll see how these statistics are used in the calculation of probabilities in a few steps. The two statistics we require from a given dataset are the mean and the standard deviation (average deviation from the mean). The mean is the average value and can be calculated as:

$$mean = \frac{\sum_{i=1}^{n} x_i}{count(x)} \tag{12.2}$$

Where `x` is the list of values or a column we are looking at and `i` is the index of a specific value. Below is a small function named `mean()` that calculates the mean of a list of numbers.

```
# Calculate the mean of a list of numbers
def mean(numbers):
  return sum(numbers)/float(len(numbers))
```

Listing 12.5: Function To Calculate the Mean of a List of Numbers.

The sample standard deviation is calculated as the mean difference from the mean value. This can be calculated as:

$$\text{standard deviation} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - mean(x))^2}{count(x) - 1}} \tag{12.3}$$

You can see that we square the difference between the mean and a given value, calculate the average squared difference from the mean, then take the square root to return the units back to their original value. Below is a small function named `standard_deviation()` that calculates the standard deviation of a list of numbers. You will notice that it calculates the mean. It might be more efficient to calculate the mean of a list of numbers once and pass it to the `standard_deviation()` function as a parameter. You can explore this optimization if you're interested later.

```
from math import sqrt

# Calculate the standard deviation of a list of numbers
def stdev(numbers):
  avg = mean(numbers)
  variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
  return sqrt(variance)
```

Listing 12.6: Function To Calculate the Standard Deviation of a List of Numbers.

We require the mean and standard deviation statistics to be calculated for each input attribute or each column of our data. We can do that by gathering all of the values for each column into a list and calculating the mean and standard deviation on that list. Once calculated, we can gather the statistics together into a list or tuple of statistics. Then, repeat this operation for each column in the dataset and return a list of tuples of statistics.

Below is a function named `summarize_dataset()` that implements this approach. It uses some Python tricks to cut down on the number of lines required.

```
# Calculate the mean, stdev and count for each column in a dataset
def summarize_dataset(dataset):
  summaries = [(mean(column), stdev(column), len(column)) for column in zip(*dataset)]
  del(summaries[-1])
  return summaries
```

Listing 12.7: Function To Summarize Each Column in a Dataset.

The first trick is the use of the `zip()` function that will aggregate elements from each provided argument. We pass in the dataset to the `zip()` function with the `*` operator that separates the dataset (that is a list of lists) into separate lists for each row. The `zip()` function then iterates over each element of each row and returns a column from the dataset as a list of numbers. A clever little trick.

We then calculate the mean, standard deviation and count of rows in each column. A tuple is created from these 3 numbers and a list of these tuples is stored. We then remove the statistics for the class variable as we will not need these statistics. Let's test all of these functions on our contrived dataset from above. Below is the complete example.

```
# Example of summarizing a dataset
from math import sqrt
```

```
# Calculate the mean of a list of numbers
def mean(numbers):
  return sum(numbers)/float(len(numbers))

# Calculate the standard deviation of a list of numbers
def stdev(numbers):
  avg = mean(numbers)
  variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
  return sqrt(variance)

# Calculate the mean, stdev and count for each column in a dataset
def summarize_dataset(dataset):
  summaries = [(mean(column), stdev(column), len(column)) for column in zip(*dataset)]
  del(summaries[-1])
  return summaries

# Test summarizing a dataset
dataset = [[3.393533211,2.331273381,0],
  [3.110073483,1.781539638,0],
  [1.343808831,3.368360954,0],
  [3.582294042,4.67917911,0],
  [2.280362439,2.866990263,0],
  [7.423436942,4.696522875,1],
  [5.745051997,3.533989803,1],
  [9.172168622,2.511101045,1],
  [7.792783481,3.424088941,1],
  [7.939820817,0.791637231,1]]
summary = summarize_dataset(dataset)
print(summary)
```

Listing 12.8: Example of Summarization By Column on a Contrived Dataset.

Running the example prints out the list of tuples of statistics on each of the two input variables. Interpreting the results, we can see that the mean value of `X1` is 5.178333386499999 and the standard deviation of `X1` is 2.7665845055177263.

```
[(5.178333386499999, 2.7665845055177263, 10), (2.9984683241, 1.218556343617447, 10)]
```

Listing 12.9: Sample Output Summarizing a Dataset by Column.

Now we are ready to use these functions on each group of rows in our dataset.

12.2.3 Summarize Data By Class

We require statistics from our training dataset organized by class. Above, we have developed the `separate_by_class()` function to separate a dataset into rows by class. And we have developed `summarize_dataset()` function to calculate summary statistics for each column.

We can put all of this together and summarize the columns in the dataset organized by class values. Below is a function named `summarize_by_class()` that implements this operation. The dataset is first split by class, then statistics are calculated on each subset. The results in the form of a list of tuples of statistics are then stored in a dictionary by their class value.

```
# Split dataset by class then calculate statistics for each row
def summarize_by_class(dataset):
  separated = selarate_by_class(dataset)
```

```
  summaries = dict()
  for class_value, rows in separated.iteritems():
    summaries[class_value] = summarize_dataset(rows)
  return summaries
```

Listing 12.10: Function To Summarize Each Column in a Dataset by Class.

Again, let's test out all of these behaviors on our contrived dataset.

```
# Example of summarizing data by class value
from math import sqrt

# Split the dataset by class values, returns a dictionary
def separate_by_class(dataset):
  separated = dict()
  for i in range(len(dataset)):
    vector = dataset[i]
    class_value = vector[-1]
    if (class_value not in separated):
      separated[class_value] = list()
    separated[class_value].append(vector)
  return separated

# Calculate the mean of a list of numbers
def mean(numbers):
  return sum(numbers)/float(len(numbers))

# Calculate the standard deviation of a list of numbers
def stdev(numbers):
  avg = mean(numbers)
  variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
  return sqrt(variance)

# Calculate the mean, stdev and count for each column in a dataset
def summarize_dataset(dataset):
  summaries = [(mean(column), stdev(column), len(column)) for column in zip(*dataset)]
  del(summaries[-1])
  return summaries

# Split dataset by class then calculate statistics for each row
def summarize_by_class(dataset):
  separated = separate_by_class(dataset)
  summaries = dict()
  for class_value, rows in separated.iteritems():
    summaries[class_value] = summarize_dataset(rows)
  return summaries

# Test summarizing by class
dataset = [[3.393533211,2.331273381,0],
  [3.110073483,1.781539638,0],
  [1.343808831,3.368360954,0],
  [3.582294042,4.67917911,0],
  [2.280362439,2.866990263,0],
  [7.423436942,4.696522875,1],
  [5.745051997,3.533989803,1],
  [9.172168622,2.511101045,1],
  [7.792783481,3.424088941,1],
```

```
  [7.939820817,0.791637231,1]]
summary = summarize_by_class(dataset)
for label in summary:
  print(label)
  for row in summary[label]:
    print(row)
```

Listing 12.11: Example of Summarization By Column on a Contrived Dataset By Class.

Running this example calculates the statistics for each input variable and prints them organized by class value. Interpreting the results, we can see that the `X1` values for rows for class 0 have a mean value of 2.7420144012.

```
0
(2.7420144012, 0.9265683289298018, 5)
(3.0054686692, 1.1073295894898725, 5)
1
(7.6146523718, 1.2344321550313704, 5)
(2.9914679790000003, 1.4541931384601618, 5)
```

Listing 12.12: Sample Output Summarizing a Dataset by Class.

There is one more piece we need before we start calculating probabilities.

12.2.4 Gaussian Probability Density Function

Calculating the probability or likelihood of observing a given real-value like `X1` is difficult. One way we can do this is to assume that `X1` values are drawn from a distribution, such as a bell curve or Gaussian distribution.

A Gaussian distribution can be summarized using only two numbers: the mean and the standard deviation. Therefore, with a little math, we can estimate the probability of a given value. This piece of math is called a Gaussian Probability Distribution Function (or Gaussian PDF) and can be calculated as:

$$probability(x) = \frac{1}{\sqrt{2 \times PI} \times \text{standard\_deviation}} \times e^{-(\frac{(x-mean(x))^2}{2 \times \text{standard\_deviation}^2})} \quad (12.4)$$

Below is a function that implements this. I tried to split it up to make it more readable.

```
# Calculate the Gaussian probability distribution function for x
def calculate_probability(x, mean, stdev):
  exponent = exp(-((x-mean)**2 / (2 * stdev**2 )))
  return (1 / (sqrt(2 * pi) * stdev)) * exponent
```

Listing 12.13: Function To Calculate the Gaussian PDF.

Let's test it out to see how it works. Below are some worked examples.

```
# Example of Gaussian PDF
from math import sqrt
from math import pi
from math import exp

# Calculate the Gaussian probability distribution function for x
def calculate_probability(x, mean, stdev):
  exponent = exp(-((x-mean)**2 / (2 * stdev**2 )))
```

```
  return (1 / (sqrt(2 * pi) * stdev)) * exponent

# Test Gaussian PDF
print(calculate_probability(1.0, 1.0, 1.0))
print(calculate_probability(2.0, 1.0, 1.0))
print(calculate_probability(0.0, 1.0, 1.0))
```

Listing 12.14: Example of Calculating a Gaussian PDF for ad hoc values.

Running it prints the probability of some input values. You can see that when the value is 1 and the mean and standard deviation is 1 our input is the most likely (top of the bell curve) and has the probability of 0.39. We can see that when we keep the statistics the same and change the x value to 1 standard deviation either side of the mean value (2 and 0 or the same distance either side of the bell curve) the probabilities of those input values are the same at 0.24.

```
0.398942280401
0.241970724519
0.241970724519
```

Listing 12.15: Sample Output Calculating Gaussian PDF values.

Now that we have all the pieces in place, let's see how we can calculate the probabilities we need for a Naive Bayes Theorem.

12.2.5 Class Probabilities

Now it is time to use the statistics calculated from our training data to calculate probabilities for new data. Probabilities are calculated separately for each class. This means that we first calculate the probability that a new piece of data belongs to the first class, then calculate probabilities that it belongs to the second class, and so on for all the classes. The probability that a piece of data belongs to a class is calculated as follows:

$$P(class|data) = P(X|class) \times P(class) \tag{12.5}$$

You may note that this is different from the Bayes Theorem described above. The division have been removed to simplify the calculation. This means that the result is no longer strictly a probability of the data belonging to a class. The value is still maximized, meaning that the calculation for the class that results in the largest value is taken as the prediction. This is a common implementation simplification as we are often more interested in the class prediction rather than the probability.

The input variables are treated separately, giving the technique it's name *naive*. For the above example where we have 2 input variables, the calculation of the probability that a row belongs to the first class 0 can be calculated as:

$$P(class = 0|X1, X2) = P(X1|class = 0) \times P(X2|class = 0) \times P(class = 0) \tag{12.6}$$

Now you can see why we need to separate the data by class value. The Gaussian Probability Density function in the previous step is how we calculate the probability of a real value like `X1` and the statistics we prepared are used in this calculation. Below is a function named `calculate_class_probabilities()` that ties all of this together. It takes a set of prepared summaries and a new row as input arguments.

First the total number of training records is calculated from the counts stored in the summary statistics. This is used in the calculation of the probability of a given class or `P(class)` as the ratio of rows with a given class of all rows in the training data.

Next, probabilities are calculated for each input value in the row using the Gaussian probability density function and the statistics for that column and of that class. Probabilities are multiplied together as they accumulated. This process is repeated for each class in the dataset. Finally a dictionary of probabilities is returned with one entry for each class.

```
# Calculate the probabilities of predicting each class for a given row
def calculate_class_probabilities(summaries, row):
  total_rows = sum([summaries[label][0][2] for label in summaries])
  probabilities = dict()
  for class_value, class_summaries in summaries.iteritems():
    probabilities[class_value] = summaries[class_value][0][2]/float(total_rows)
    for i in range(len(class_summaries)):
      mean, stdev, count = class_summaries[i]
      probabilities[class_value] *= calculate_probability(row[i], mean, stdev)
  return probabilities
```

Listing 12.16: Function To Calculate the Probabilities for Each Class.

Let's tie this together with an example on the contrived dataset. The example below first calculates the summary statistics by class for the training dataset, then uses these statistics to calculate the probability of the first record belonging to each class.

```
# Example of calculating class probabilities
from math import sqrt
from math import pi
from math import exp

# Split the dataset by class values, returns a dictionary
def separate_by_class(dataset):
  separated = dict()
  for i in range(len(dataset)):
    vector = dataset[i]
    class_value = vector[-1]
    if (class_value not in separated):
      separated[class_value] = list()
    separated[class_value].append(vector)
  return separated

# Calculate the mean of a list of numbers
def mean(numbers):
  return sum(numbers)/float(len(numbers))

# Calculate the standard deviation of a list of numbers
def stdev(numbers):
  avg = mean(numbers)
  variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
  return sqrt(variance)

# Calculate the mean, stdev and count for each column in a dataset
def summarize_dataset(dataset):
  summaries = [(mean(column), stdev(column), len(column)) for column in zip(*dataset)]
  del(summaries[-1])
  return summaries
```

```
# Split dataset by class then calculate statistics for each row
def summarize_by_class(dataset):
  separated = separate_by_class(dataset)
  summaries = dict()
  for class_value, rows in separated.iteritems():
    summaries[class_value] = summarize_dataset(rows)
  return summaries

# Calculate the Gaussian probability distribution function for x
def calculate_probability(x, mean, stdev):
  exponent = exp(-((x-mean)**2 / (2 * stdev**2 )))
  return (1 / (sqrt(2 * pi) * stdev)) * exponent

# Calculate the probabilities of predicting each class for a given row
def calculate_class_probabilities(summaries, row):
  total_rows = sum([summaries[label][0][2] for label in summaries])
  probabilities = dict()
  for class_value, class_summaries in summaries.iteritems():
    probabilities[class_value] = summaries[class_value][0][2]/float(total_rows)
    for i in range(len(class_summaries)):
      mean, stdev, count = class_summaries[i]
      probabilities[class_value] *= calculate_probability(row[i], mean, stdev)
  return probabilities

# Test calculating class probabilities
dataset = [[3.393533211,2.331273381,0],
  [3.110073483,1.781539638,0],
  [1.343808831,3.368360954,0],
  [3.582294042,4.67917911,0],
  [2.280362439,2.866990263,0],
  [7.423436942,4.696522875,1],
  [5.745051997,3.533989803,1],
  [9.172168622,2.511101045,1],
  [7.792783481,3.424088941,1],
  [7.939820817,0.791637231,1]]
summaries = summarize_by_class(dataset)
probabilities = calculate_class_probabilities(summaries, dataset[0])
print(probabilities)
```

Listing 12.17: Example of Calculating Probabilities For Each Class.

Running the example prints the probabilities calculated for each class. We can see that the probability of the first row belonging to the 0 class (0.0503) is higher than the probability of it belonging to the 1 class (0.0001). We would therefore correctly conclude that it belongs to the 0 class.

```
{0: 0.05032427673372075, 1: 0.00011557718379945765}
```

Listing 12.18: Sample Output Calculating the Probabilities for Each Class.

Now that we have seen how to implement the Naive Bayes algorithm, let's apply it to the Iris flowers dataset.

12.2.6 Iris Flower Species Case Study

This section applies the Naive Bayes algorithm to the Iris flowers dataset. The first step is to load the dataset and convert the loaded data to numbers that we can use with the mean and standard deviation calculations. For this we will use the helper function `load_csv()` to load the file, `str_column_to_float()` to convert string numbers to floats and `str_column_to_int()` to convert the class column to integer values.

We will evaluate the algorithm using k-fold cross-validation with 5 folds. This means that $\frac{150}{5} = 30$ records will be in each fold. We will use the helper functions `evaluate_algorithm()` to evaluate the algorithm with cross-validation and `accuracy_metric()` to calculate the accuracy of predictions. A new function named `predict()` was developed to manage the calculation of the probabilities of a new row belonging to each class and selecting the class with the largest probability value.

Another new function named `naive_bayes()` was developed to manage the application of the Naive Bayes algorithm, first learning the statistics from a training dataset and using them to make predictions for a test dataset. The complete example is listed below.

```
# Naive Bayes On The Iris Dataset
from csv import reader
from random import randrange
from math import sqrt
from math import exp
from math import pi

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
```

```
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# Split the dataset by class values, returns a dictionary
def separate_by_class(dataset):
  separated = dict()
  for i in range(len(dataset)):
    vector = dataset[i]
    class_value = vector[-1]
    if (class_value not in separated):
      separated[class_value] = list()
    separated[class_value].append(vector)
  return separated

# Calculate the mean of a list of numbers
def mean(numbers):
  return sum(numbers)/float(len(numbers))

# Calculate the standard deviation of a list of numbers
def stdev(numbers):
  avg = mean(numbers)
  variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
```

```
  return sqrt(variance)

# Calculate the mean, stdev and count for each column in a dataset
def summarize_dataset(dataset):
  summaries = [(mean(column), stdev(column), len(column)) for column in zip(*dataset)]
  del(summaries[-1])
  return summaries

# Split dataset by class then calculate statistics for each row
def summarize_by_class(dataset):
  separated = separate_by_class(dataset)
  summaries = dict()
  for class_value, rows in separated.iteritems():
    summaries[class_value] = summarize_dataset(rows)
  return summaries

# Calculate the Gaussian probability distribution function for x
def calculate_probability(x, mean, stdev):
  exponent = exp(-((x-mean)**2 / (2 * stdev**2 )))
  return (1 / (sqrt(2 * pi) * stdev)) * exponent

# Calculate the probabilities of predicting each class for a given row
def calculate_class_probabilities(summaries, row):
  total_rows = sum([summaries[label][0][2] for label in summaries])
  probabilities = dict()
  for class_value, class_summaries in summaries.iteritems():
    probabilities[class_value] = summaries[class_value][0][2]/float(total_rows)
    for i in range(len(class_summaries)):
      mean, stdev, count = class_summaries[i]
      probabilities[class_value] *= calculate_probability(row[i], mean, stdev)
  return probabilities

# Predict the class for a given row
def predict(summaries, row):
  probabilities = calculate_class_probabilities(summaries, row)
  best_label, best_prob = None, -1
  for class_value, probability in probabilities.iteritems():
    if best_label is None or probability > best_prob:
      best_prob = probability
      best_label = class_value
  return best_label

# Naive Bayes Algorithm
def naive_bayes(train, test):
  summarize = summarize_by_class(train)
  predictions = list()
  for row in test:
    output = predict(summarize, row)
    predictions.append(output)
  return(predictions)

# Test Naive Bayes on Iris Dataset
filename = 'iris.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
  str_column_to_float(dataset, i)
```

```
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# evaluate algorithm
n_folds = 5
scores = evaluate_algorithm(dataset, naive_bayes, n_folds)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 12.19: Example of Naive Bayes on the Iris Dataset.

Running the example prints the mean classification accuracy scores on each cross-validation fold as well as the mean accuracy score. We can see that the mean accuracy of 95.333% is dramatically better than the baseline accuracy of 26%.

```
Scores: [96.66666666666667, 90.0, 100.0, 93.33333333333333, 96.66666666666667]
Mean Accuracy: 95.333%
```

Listing 12.20: Sample Output of Naive Bayes on the Iris Dataset.

12.3 Extensions

This section lists extensions to the tutorial that you may wish to explore.

- **Categorical Input Variables**. Only real-valued inputs were supported in the tutorial. Update it to support categorical input values by calculating the probability as their frequency ratio.
- **Log Probabilities**. Probabilities become very small numbers in Naive Bayes because they are multiplied together. This can cause problems on datasets with more input features. Converting all probabilities to log values (e.g. $p = log(1 + p)$) before they are multiplied together can remedy this problem.

12.4 Review

In this tutorial you discovered how to implement the Naive Bayes algorithm from scratch in Python. Specifically, you learned:

- How to calculate the probabilities required by the Naive interpretation of Bayes Theorem.
- How to use probabilities to make predictions on new data.
- How to apply Naive Bayes to a real-world predictive modeling problem.

12.4.1 Further Reading

- Section 13.6 Naive Bayes, page 353, *Applied Predictive Modeling*, 2013
  http://amzn.to/2e3lNXF
- Section 4.2, Statistical modeling, page 88, *Data Mining: Practical Machine Learning Tools and Techniques*, second edition, 2005.
  http://amzn.to/2fj3SYY

12.4.2 Next

In the next tutorial, you will discover how to implement and apply the k-Nearest Neighbors algorithm for classification.

Chapter 13

k-Nearest Neighbors

A simple but powerful approach for making predictions is to use the most similar historical examples to the new data. This is the principle behind the k-Nearest Neighbors algorithm. In this tutorial you will discover how to implement the k-Nearest Neighbors algorithm from scratch with Python. After completing this tutorial you will know:

- How to calculate the similarity between two pieces of data.
- How to make a prediction using the most similar historical records.
- How to use k-Nearest Neighbors for both classification and regression problems.

Let's get started.

13.1 Description

This section will provide a brief background on the k-Nearest Neighbors algorithm that we will implement in this tutorial and the Abalone dataset to which we will apply it.

13.1.1 k-Nearest Neighbors

The k-Nearest Neighbors algorithm or KNN for short is a very simple technique. The entire training dataset is stored. When a prediction is required, the k-most similar records to a new record from the training dataset are then located. From these neighbors, a summarized prediction is made. Similarity between records can be measured many different ways. A problem or data-specific method can be used. Generally, with tabular data, a good starting point is the Euclidean distance.

Once the neighbors are discovered, the summary prediction can be made by returning the most common outcome or taking the average. As such, KNN can be used for classification or regression problems. There is no model to speak of other than holding the entire training dataset. Because no work is done until a prediction is required, KNN is often referred to as a lazy learning method.

13.1.2 Abalone Dataset

In this tutorial we will use the Abalone Dataset. This dataset involves the prediction of the age of abalone. The baseline performance on the problem is approximately 16% or an RMSE of approximately 3.2 rings. You can learn more about it in Appendix A, Section A.8. Download the dataset and save it into your current working directory with the filename `abalone.csv`.

13.2 Tutorial

This tutorial is broken down into 5 parts:

1. Euclidean Distance.
2. Get Neighbors.
3. Make Predictions.
4. Abalone Case Study as Classification.
5. Abalone Case Study as Regression.

These steps will teach you the fundamentals of implementing and applying the k-Nearest Neighbors algorithm for classification and regression predictive modeling problems.

13.2.1 Euclidean Distance

The first step needed is to calculate the distance between two rows in a dataset. Rows of data are mostly made up of numbers and an easy way to calculate the distance between two rows or vectors of numbers is to draw a straight line. This makes sense in 2D or 3D and scales nicely to higher dimensions. We can calculate the straight line distance between two vectors using the Euclidean distance measure. It is calculated as the square root of the sum of the squared differences between the two vectors.

$$distance = \sqrt{\sum_{i=1}^{n}(x1_i - x2_i)^2} \tag{13.1}$$

Where `x1` is the first row of data, `x2` is the second row of data and `i` is the index to a specific column as we sum across all columns. With Euclidean distance, the smaller the value, the more similar two records will be. A value of 0 means that there is no difference between two records. Below is a function named `euclidean_distance()` that implements this in Python.

```
# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)
```

Listing 13.1: Function To Calculate the Euclidean Distance Between Rows.

You can see that the function assumes that the last column in each row is an output value which is ignored from the distance calculation. We can test this distance function with a small contrived classification dataset. We will use this dataset a few times as we construct the elements needed for the KNN algorithm.

```
X1          X2             Y
2.7810836   2.550537003    0
1.465489372 2.362125076    0
3.396561688 4.400293529    0
1.38807019  1.850220317    0
3.06407232  3.005305973    0
7.627531214 2.759262235    1
5.332441248 2.088626775    1
6.922596716 1.77106367     1
8.675418651 -0.242068655   1
7.673756466 3.508563011    1
```

Listing 13.2: Small Contrived Dataset for Testing Logistic Regression.

Below is a plot of the dataset using different colors to show the different classes for each point.

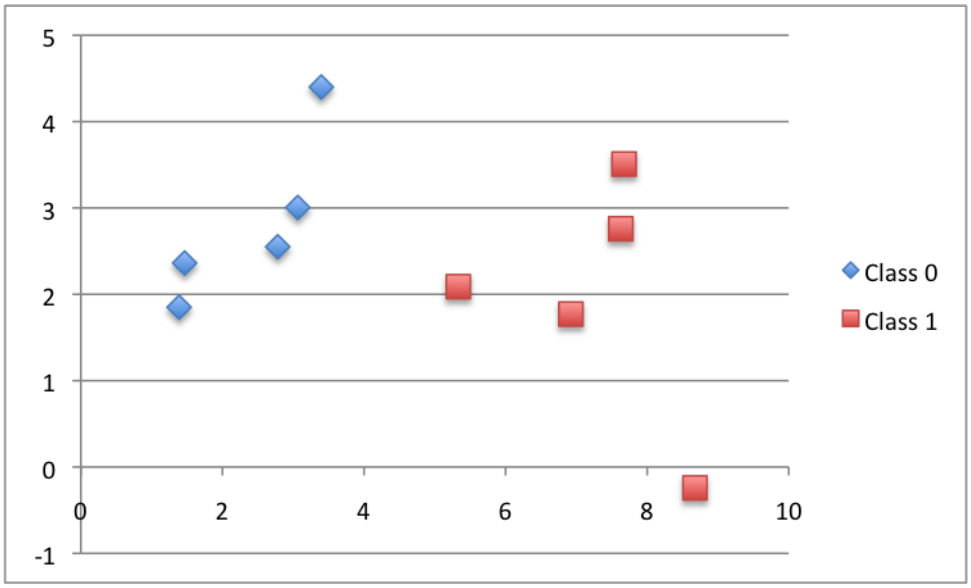

Figure 13.1: Plot of the Small Contrived Dataset for Testing the KNN algorithm.

Putting this all together, we can write a small example to test our distance function by printing the distance between the first row and all other rows. We would expect the distance between the first row and itself to be 0, a good thing to look out for. The full example is listed below.

```
# Example of calculating Euclidean distance
from math import sqrt

# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
```

```
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Test distance function
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
row0 = dataset[0]
for row in dataset:
  distance = euclidean_distance(row0, row)
  print(distance)
```

Listing 13.3: Example of Calculating Euclidean Distance on the Contrived Dataset.

Running this example prints the distances between the first row and every row in the dataset, including itself.

```
0.0
1.32901739153
1.94946466557
1.55914393855
0.535628072194
4.85094018699
2.59283375995
4.21422704263
6.52240998823
4.98558538245
```

Listing 13.4: Sample Output of Calculating Euclidean Distance.

Now it is time to use the distance calculation to locate neighbors within a dataset.

13.2.2 Get Neighbors

Neighbors for a new piece of data in the dataset are the k closest instances, as defined by our distance measure. To locate the neighbors for a new piece of data within a dataset we must first calculate the distance between each record in the dataset to the new piece of data. We can do this using our distance function above. Once distances are calculated, we must sort all of the records in the training dataset by their distance to the new data. We can then select the top k to return as the most similar neighbors.

We can do this by keeping track of the distance for each record in the dataset as a tuple, sort the list of tuples by the distance (in descending order) and then retrieve the neighbors. Below is a function named `get_neighbors()` that implements this.

```
# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
  distances = list()
  for train_row in train:
```

```
    dist = euclidean_distance(test_row, train_row)
    distances.append((train_row, dist))
  distances.sort(key=lambda tup: tup[1])
  neighbors = list()
  for i in range(num_neighbors):
    neighbors.append(distances[i][0])
  return neighbors
```

Listing 13.5: Function To Locate Neighbors for a New Data Row.

You can see that the `euclidean_distance()` function developed in the previous step is used to calculate the distance between each `train_row` and the new `test_row`. The list of `train_row` and distance tuples is sorted where a custom key is used ensuring that the second item in the tuple (`tup[1]`) is used in the sorting operation.

Finally, a list of the `num_neighbors` most similar neighbors to `test_row` is returned. We can test this function with the small contrived dataset prepared in the previous section. The complete example is listed below.

```
# Example of getting neighbours for an instance
from math import sqrt

# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
  distances = list()
  for train_row in train:
    dist = euclidean_distance(test_row, train_row)
    distances.append((train_row, dist))
  distances.sort(key=lambda tup: tup[1])
  neighbors = list()
  for i in range(num_neighbors):
    neighbors.append(distances[i][0])
  return neighbors

# Test distance function
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
neighbors = get_neighbors(dataset, dataset[0], 3)
for neighbor in neighbors:
  print(neighbor)
```

Listing 13.6: Example of Getting Most Similar Neighbors on the Contrived Dataset.

Running this example prints the 3 most similar records in the dataset to the first record, in order of similarity. As expected, the first record is the most similar to itself and is at the top of the list.

```
[2.7810836, 2.550537003, 0]
[3.06407232, 3.005305973, 0]
[1.465489372, 2.362125076, 0]
```

Listing 13.7: Sample Output of Calculating Most Similar Neighbors.

Now that we know how to get neighbors from the dataset, we can use them to make predictions.

13.2.3 Make Predictions

The most similar neighbors collected from the training dataset can be used to make predictions. In the case of classification, we can return the most represented class among the neighbors. We can achieve this by performing the `max()` function on the list of output values from the neighbors. Given a list of class values observed in the neighbors, the `max()` function takes a set of unique class values and calls the count on the list of class values for each class value in the set. Below is the function named `predict_classification()` that implements this.

```
# Make a classification prediction with neighbors
def predict_classification(train, test_row, num_neighbors):
  neighbors = get_neighbors(train, test_row, num_neighbors)
  output_values = [row[-1] for row in neighbors]
  prediction = max(set(output_values), key=output_values.count)
  return prediction
```

Listing 13.8: Function To Make Classification Predictions for a New Data Row.

We can test this function on the above contrived dataset. Below is a complete example.

```
# Example of making predictions
from math import sqrt

# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
  distances = list()
  for train_row in train:
    dist = euclidean_distance(test_row, train_row)
    distances.append((train_row, dist))
  distances.sort(key=lambda tup: tup[1])
  neighbors = list()
  for i in range(num_neighbors):
    neighbors.append(distances[i][0])
  return neighbors

# Make a classification prediction with neighbors
```

```
def predict_classification(train, test_row, num_neighbors):
  neighbors = get_neighbors(train, test_row, num_neighbors)
  output_values = [row[-1] for row in neighbors]
  prediction = max(set(output_values), key=output_values.count)
  return prediction

# Test distance function
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
prediction = predict_classification(dataset, dataset[0], 3)
print('Expected %d, Got %d.' % (dataset[0][-1], prediction))
```

Listing 13.9: Example of Making Predictions on the Contrived Dataset.

Running this example prints the expected classification of 0 and the actual classification predicted from the 3 most similar neighbors in the dataset.

```
Expected 0, Got 0.
```

Listing 13.10: Sample Output of Making Predictions.

We can imagine how the `predict_classification()` function can be changed to calculate the mean value of the outcome values. Below is a quick example named `predict_regression()` that we will use later.

```
# Make a prediction with neighbors
def predict_regression(train, test_row, num_neighbors):
  neighbors = get_neighbors(train, test_row, num_neighbors)
  output_values = [row[-1] for row in neighbors]
  prediction = sum(output_values) / float(len(output_values))
  return prediction
```

Listing 13.11: Function To Make Regression Predictions for a New Data Row.

We now have all of the pieces to make predictions with KNN. Let's apply it to the Abalone dataset.

13.2.4 Abalone Case Study as Classification

In this section we will apply the k-Nearest Neighbors algorithm to the Abalone dataset. The first step is to load the dataset and covert the loaded data to numbers that we can use with the Euclidean distance calculation. For this we will use the helper function `load_csv()` to load the file, `str_column_to_float()` to convert string numbers to floats and `str_column_to_int()` to convert the sex column (column 0) to integer values.

We will evaluate the algorithm using k-fold cross-validation with 5 folds. This means that $\frac{4177}{5} = 835.4$ or just over 830 records will be in each fold. We will use the helper functions

`evaluate_algorithm()` to evaluate the algorithm with cross-validation and `accuracy_metric()` to calculate the accuracy of predictions. The complete example is listed below.

```
# k-nearest neighbors on the Abalone Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Find the min and max values for each column
def dataset_minmax(dataset):
  minmax = list()
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    value_min = min(col_values)
    value_max = max(col_values)
    minmax.append([value_min, value_max])
  return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
```

```
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
  distances = list()
  for train_row in train:
    dist = euclidean_distance(test_row, train_row)
    distances.append((train_row, dist))
  distances.sort(key=lambda tup: tup[1])
  neighbors = list()
  for i in range(num_neighbors):
    neighbors.append(distances[i][0])
  return neighbors

# Make a prediction with neighbors
```

```
def predict_classification(train, test_row, num_neighbors):
  neighbors = get_neighbors(train, test_row, num_neighbors)
  output_values = [row[-1] for row in neighbors]
  prediction = max(set(output_values), key=output_values.count)
  return prediction

# kNN Algorithm
def k_nearest_neighbors(train, test, num_neighbors):
  predictions = list()
  for row in test:
    output = predict_classification(train, row, num_neighbors)
    predictions.append(output)
  return(predictions)

# Test the kNN on the Abalone dataset
seed(1)
# load and prepare data
filename = 'abalone.csv'
dataset = load_csv(filename)
for i in range(1, len(dataset[0])):
  str_column_to_float(dataset, i)
# convert first column to integers
str_column_to_int(dataset, 0)
# evaluate algorithm
n_folds = 5
num_neighbors = 5
scores = evaluate_algorithm(dataset, k_nearest_neighbors, n_folds, num_neighbors)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 13.12: Classification KNN on the Abalone Dataset.

A value of k=5 neighbors was used to make predictions. You may experiment with larger k values to increase accuracy. Running the example prints the accuracy scores achieved on each fold and the mean of those scores. We can see that the mean accuracy of 23% is better than the baseline of 16%, but is quite poor in general. This is because of the large number of classes making accuracy a poor judge of skill on this problem. This fact, combined with the fact that many of the classes have few or one example also makes the problem challenging.

```
Scores: [24.191616766467067, 24.431137724550897, 23.47305389221557, 22.035928143712574,
    23.11377245508982]
Mean Accuracy: 23.449%
```

Listing 13.13: Sample Output of Classification KNN on the Abalone Dataset.

We can also model the dataset as a regression predictive modeling problem. This is because the class values have a natural ordinal relationship as they are years.

13.2.5 Abalone Case Study as Regression

Regression may be a more useful way to model this problem given the large number of classes and sparseness of some class values. We can easily change our above example to regression by changing KNN to predict the mean of the neighbors and using Root Mean Squared Error to evaluate predictions. Below is a complete working example with these changes.

```
# k-Nearest Neighbors on the Abalone Dataset for Regression
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Find the min and max values for each column
def dataset_minmax(dataset):
  minmax = list()
  for i in range(len(dataset[0])):
    col_values = [row[i] for row in dataset]
    value_min = min(col_values)
    value_max = max(col_values)
    minmax.append([value_min, value_max])
  return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
  for row in dataset:
    for i in range(len(row)):
      row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
```

```
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate root mean squared error
def rmse_metric(actual, predicted):
  sum_error = 0.0
  for i in range(len(actual)):
    prediction_error = predicted[i] - actual[i]
    sum_error += (prediction_error ** 2)
  mean_error = sum_error / float(len(actual))
  return sqrt(mean_error)

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    rmse = rmse_metric(actual, predicted)
    scores.append(rmse)
  return scores

# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
  distances = list()
  for train_row in train:
    dist = euclidean_distance(test_row, train_row)
    distances.append((train_row, dist))
  distances.sort(key=lambda tup: tup[1])
  neighbors = list()
  for i in range(num_neighbors):
    neighbors.append(distances[i][0])
  return neighbors

# Make a prediction with neighbors
def predict_regression(train, test_row, num_neighbors):
```

```
  neighbors = get_neighbors(train, test_row, num_neighbors)
  output_values = [row[-1] for row in neighbors]
  prediction = sum(output_values) / float(len(output_values))
  return prediction

# kNN Algorithm
def k_nearest_neighbors(train, test, num_neighbors):
  predictions = list()
  for row in test:
    output = predict_regression(train, row, num_neighbors)
    predictions.append(output)
  return(predictions)

# Test the kNN on the Abalone dataset
seed(1)
# load and prepare data
filename = 'abalone.csv'
dataset = load_csv(filename)
for i in range(1, len(dataset[0])):
  str_column_to_float(dataset, i)
# convert first column to integers
str_column_to_int(dataset, 0)
# evaluate algorithm
n_folds = 5
num_neighbors = 5
scores = evaluate_algorithm(dataset, k_nearest_neighbors, n_folds, num_neighbors)
print('Scores: %s' % scores)
print('Mean RMSE: %.3f' % (sum(scores)/float(len(scores))))
```

Listing 13.14: Regression KNN on the Abalone Dataset.

Running this example prints the RMSE on each fold and the mean RMSE across all folds. We can see that the RMSE of 2.242 rings is better than the baseline of 3.222 rings. We also have a model that is perhaps more useful in the domain with an performance that is easier to understand.

```
Scores: [2.1235153320071554, 2.258503558410589, 2.2739767060988636, 2.3582090027389,
    2.196633243122396]
Mean RMSE: 2.242
```

Listing 13.15: Sample Output of Regression KNN on the Abalone Dataset.

13.3 Extensions

This section lists extensions to the tutorial you may wish to consider to investigate.

- **Tune KNN**. Try larger and larger k values to see if you can improve the performance of the algorithm on the Abalone dataset.
- **Regression for Classification**. Combine the approach used to make predictions for regression problems (take the mean) with the classification approach to making predictions (return an integer) and see if you can improve results.

- **More Distance Measures**. Implement other distance measures that you can use to find similar historical data, such as Hamming distance, Manhattan distance and Minkowski distance.
- **Data Preparation**. Distance measures are strongly affected by the scale of the input data. Experiment with normalization and standardization data preparation methods in order to improve results.
- **More Problems**. As always, experiment with the technique on more and different classification and regression problems.

13.4 Review

In this tutorial you discovered how to implement the k-Nearest Neighbors algorithm from scratch with Python. Specifically, you learned:

- How to implement k-Nearest Neighbors from scratch in Python.
- How to apply k-Nearest Neighbors to classification problems.
- How to apply k-Nearest Neighbors to regression problems.

13.4.1 Further Reading

- Section 3.5 Comparison of Linear Regression with K-Nearest Neighbors, page 104, *An Introduction to Statistical Learning*, 2014.
  http://amzn.to/2eeTyQX
- Section 18.8. Nonparametric Models, page 737, *Artificial Intelligence: A Modern Approach*, 2010.
  http://amzn.to/2e3lFqP
- Section 7.4 K-Nearest Neighbors, page 159, and Section 13.5 K-Nearest Neighbors, page 350 *Applied Predictive Modeling*, 2013
  http://amzn.to/2e3lNXF
- Section 4.7, Instance-based learning, page 128, *Data Mining: Practical Machine Learning Tools and Techniques*, second edition, 2005.
  http://amzn.to/2fj3SYY

13.4.2 Next

In the next tutorial, you will discover how to implement and apply the Learning Vector Optimization algorithm for classification.

Chapter 14

Learning Vector Quantization

A limitation of k-Nearest Neighbors is that you must keep a large database of training examples in order to make predictions. The Learning Vector Quantization algorithm addresses this by learning a much smaller subset of patterns that best represent the training data. In this tutorial, you will discover how to implement the Learning Vector Quantization algorithm from scratch with Python. After completing this tutorial, you will know:

- How to learn a set of codebook vectors from a training data set.
- How to make predictions using learned codebook vectors.
- How to apply Learning Vector Quantization to a real predictive modeling problem.

Let's get started.

14.1 Description

This section provides a brief introduction to the Learning Vector Quantization algorithm and the Ionosphere classification problem that we will use in this tutorial

14.1.1 Learning Vector Quantization

The Learning Vector Quantization (LVQ) algorithm is a lot like k-Nearest Neighbors. Predictions are made by finding the best match among a library of patterns. The difference is that the library of patterns is learned from training data, rather than using the training patterns themselves.

The library of patterns is called codebook vectors and each pattern is called a codebook. The codebook vectors are initialized to randomly selected values from the training dataset. Then, over a number of epochs, they are adapted to best summarize the training data using a learning algorithm. The learning algorithm shows one training record at a time, finds the best matching unit among the codebook vectors and moves it closer to the training record if they have the same class, or further away if they have different classes.

Once prepared, the codebook vectors are used to make predictions using the k-Nearest Neighbors algorithm where k=1. The algorithm was developed for classification predictive modeling problems, but can be adapted for use with regression problems.

14.1.2 Ionosphere Dataset

In this tutorial we will use the Ionosphere Dataset. This dataset involves the prediction of structure in the atmosphere. The baseline performance on the problem is approximately 64%. You can learn more about it in Appendix A, Section A.9. Download the dataset and save it into your current working directory with the filename `ionosphere.csv`.

14.2 Tutorial

This tutorial is broken down into 4 parts:

1. Euclidean Distance.
2. Best Matching Unit.
3. Training Codebook Vectors.
4. Ionosphere Case Study.

These steps will lay the foundation for implementing and applying the LVQ algorithm to your own predictive modeling problems.

14.2.1 Euclidean Distance

The first step needed is to calculate the distance between two rows in a dataset. We will use Euclidean distance, first introduced in Chapter 13. The calculation of Euclidean distance can be summarized as:

$$distance = \sqrt{\sum_{i=1}^{n}(x1_i - x2_i)^2} \tag{14.1}$$

Where `x1` is the first row of data, `x2` is the second row of data and `i` is the index for a specific column as we sum across all columns. With Euclidean distance, the smaller the value, the more similar two records will be. A value of 0 means that there is no difference between two records. Below is a function named `euclidean_distance()` that implements this in Python.

```
# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)
```

Listing 14.1: Function To Calculate the Euclidean Distance Between Rows.

You can see that the function assumes that the last column in each row is an output value which is ignored from the distance calculation. We can test this distance function with a small contrived classification dataset. We will use this dataset a few times as we construct the elements needed for the LVQ algorithm.

```
X1          X2             Y
2.7810836   2.550537003    0
1.465489372 2.362125076    0
3.396561688 4.400293529    0
1.38807019  1.850220317    0
3.06407232  3.005305973    0
7.627531214 2.759262235    1
5.332441248 2.088626775    1
6.922596716 1.77106367     1
8.675418651 -0.242068655   1
7.673756466 3.508563011    1
```

Listing 14.2: Small Contrived Dataset for Testing Logistic Regression.

Below is a plot of the dataset using different colors to show the different classes for each point.

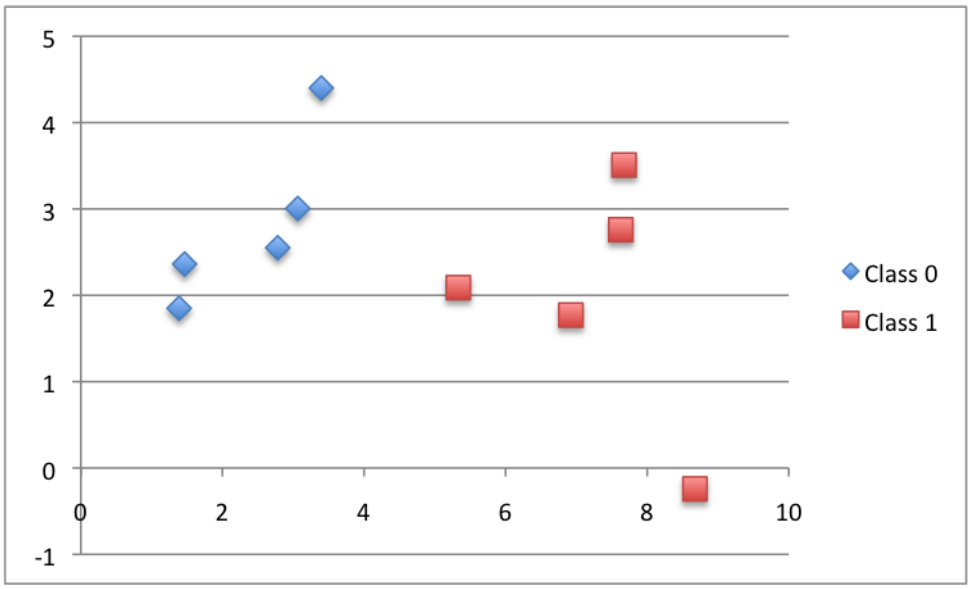

Figure 14.1: Plot of the Small Contrived Dataset for Testing the KNN algorithm.

Putting this all together, we can write a small example to test our distance function by printing the distance between the first row and all other rows. We would expect the distance between the first row and itself to be 0, a good thing to look out for. The full example is listed below.

```
# Example of calculating Euclidean distance
from math import sqrt

# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Test distance function
```

```
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
row0 = dataset[0]
for row in dataset:
  distance = euclidean_distance(row0, row)
  print(distance)
```

Listing 14.3: Example of Calculating Euclidean Distance on the Contrived Dataset.

Running this example prints the distances between the first row and every row in the dataset, including itself.

```
0.0
1.32901739153
1.94946466557
1.55914393855
0.535628072194
4.85094018699
2.59283375995
4.21422704263
6.52240998823
4.98558538245
```

Listing 14.4: Sample Output of Calculating Euclidean Distance.

Now it is time to use the distance calculation to locate the best matching unit within a dataset.

14.2.2 Best Matching Unit

The Best Matching Unit or BMU is the codebook vector that is most similar to a new piece of data. To locate the BMU for a new piece of data within a dataset we must first calculate the distance between each codebook to the new piece of data. We can do this using our distance function above. Once distances are calculated, we must sort all of the codebooks by their distance to the new data. We can then return the first or most similar codebook vector.

We can do this by keeping track of the distance for each record in the dataset as a tuple, sort the list of tuples by the distance (in descending order) and then retrieve the BMU. Below is a function named `get_best_matching_unit()` that implements this.

```
# Locate the best matching unit
def get_best_matching_unit(codebooks, test_row):
  distances = list()
  for codebook in codebooks:
    dist = euclidean_distance(codebook, test_row)
    distances.append((codebook, dist))
  distances.sort(key=lambda tup: tup[1])
  return distances[0][0]
```

Listing 14.5: Function To Calculate the Best Matching Unit for a new Row.

You can see that the `euclidean_distance()` function developed in the previous step is used to calculate the distance between each codebook and the new `test_row`. The list of codebook and distance tuples is sorted where a custom key is used ensuring that the second item in the tuple (`tup[1]`) is used in the sorting operation. Finally, the top or most similar codebook vector is returned as the BMU. We can test this function with the small contrived dataset prepared in the previous section. The complete example is listed below.

```
# Example of getting the BMU
from math import sqrt

# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Locate the best matching unit
def get_best_matching_unit(codebooks, test_row):
  distances = list()
  for codebook in codebooks:
    dist = euclidean_distance(codebook, test_row)
    distances.append((codebook, dist))
  distances.sort(key=lambda tup: tup[1])
  return distances[0][0]

# Test best matching unit function
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
test_row = dataset[0]
bmu = get_best_matching_unit(dataset, test_row)
print(bmu)
```

Listing 14.6: Example of Calculating the BMU on the Contrived Dataset.

Running this example prints the BMU in the dataset to the first record. As expected, the first record is the most similar to itself and is at the top of the list.

```
[2.7810836, 2.550537003, 0]
```

Listing 14.7: Sample Output of Calculating the BMU.

Making predictions with a set of codebook vectors is the same thing. We use the 1-nearest neighbor algorithm. That is, for each new pattern we wish to make a prediction for, we locate the most similar codebook vector in the set and return its associated class value. Now that we

know how to get the best matching unit from a set of codebook vectors, we need to learn how to train them.

14.2.3 Training Codebook Vectors

The first step in training a set of codebook vectors is to initialize the set. We can initialize it with patterns constructed from random features in the training dataset. Below is a function named `random_codebook()` that implements this. Random input and output features are selected from the training data.

```
# Create a random codebook vector
def random_codebook(train):
  n_records = len(train)
  n_features = len(train[0])
  codebook = [train[randrange(n_records)][i] for i in range(n_features)]
  return codebook
```

Listing 14.8: Function To Create a Random Codebook Vector.

After the codebook vectors are initialized to a random set, they must be adapted to best summarize the training data. This is done iteratively.

- **Epochs**: At the top level, the process is repeated for a fixed number of epochs or exposures of the training data.
- **Training Dataset**: Within an epoch, each training pattern is used one at a time to update the set of codebook vectors.
- **Pattern Features**: For a given training pattern, each feature of a best matching codebook vector is updated to move it closer or further away.

The best matching unit is found for each training pattern and only this best matching unit is updated. The difference between the training pattern and the BMU is calculated as the error. The class values (assumed to be the last value in the list) are compared. If they match, the error is added to the BMU to bring it closer to the training pattern, otherwise it is subtracted to push it further away.

The amount that the BMU is adjusted is controlled by a learning rate. This is a weighting on the amount of change made to all BMUs. For example, a learning rate of 0.3 means that BMUs are only moved by 30% of the error or difference between training patterns and BMUs.

Further, the learning rate is adjusted so that it has maximum effect in the first epoch and less effect as training continues until it has a minimal effect in the final epoch. This is called a linear decay learning rate schedule and can also be used in artificial neural networks. We can summarize this decay in learning rate by epoch number as follows:

$$rate = \text{learning\_rate} \times (1.0 - \frac{epoch}{\text{total\_epochs}}) \tag{14.2}$$

We can test this equation by assuming a learning rate of 0.3 and 10 epochs. The learning rate of each epoch would be as follows:

```
Epoch Effective Learning Rate
0   0.3
1   0.27
2   0.24
3   0.21
4   0.18
5   0.15
6   0.12
7   0.09
8   0.06
9   0.03
```

Listing 14.9: Test of Linear Decay of Learning Rate.

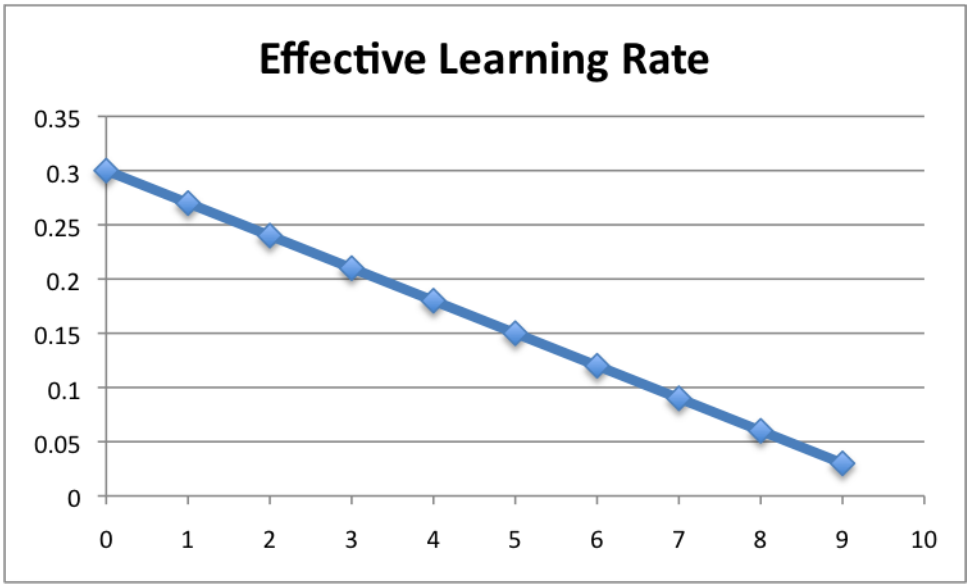

Figure 14.2: Plot of the LVQ Learning Rate.

We can put all of this together. Below is a function named `train_codebooks()` that implements the procedure for training a set of codebook vectors given a training dataset. The function takes 3 additional arguments to the training dataset, the number of codebook vectors to create and train, the initial learning rate and the number of epochs for which to train the codebook vectors.

You can also see that the function keeps track of the sum squared error each epoch and prints a message showing the epoch number, effective learning rate and sum squared error score. This is helpful when debugging the training function or the specific configuration for a given prediction problem. You can see the use of the `random_codebook()` to initialize the codebook vectors and the `get_best_matching_unit()` function to find the BMU for each training pattern within an epoch.

```
# Train a set of codebook vectors
def train_codebooks(train, n_codebooks, lrate, epochs):
  codebooks = [random_codebook(train) for i in range(n_codebooks)]
  for epoch in range(epochs):
    rate = lrate * (1.0-(epoch/float(epochs)))
```

```
    sum_error = 0.0
    for row in train:
      bmu = get_best_matching_unit(codebooks, row)
      for i in range(len(row)-1):
        error = row[i] - bmu[i]
        sum_error += error**2
        if bmu[-1] == row[-1]:
          bmu[i] += rate * error
        else:
          bmu[i] -= rate * error
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, rate, sum_error))
  return codebooks
```

Listing 14.10: Function To Train a Population of Codebook Vectors.

We can put this together with the examples above and learn a set of codebook vectors for our contrived dataset. Below is the complete example.

```
# Example of training a set of codebook vectors
from math import sqrt
from random import randrange
from random import seed

# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Locate the best matching unit
def get_best_matching_unit(codebooks, test_row):
  distances = list()
  for codebook in codebooks:
    dist = euclidean_distance(codebook, test_row)
    distances.append((codebook, dist))
  distances.sort(key=lambda tup: tup[1])
  return distances[0][0]

# Create a random codebook vector
def random_codebook(train):
  n_records = len(train)
  n_features = len(train[0])
  codebook = [train[randrange(n_records)][i] for i in range(n_features)]
  return codebook

# Train a set of codebook vectors
def train_codebooks(train, n_codebooks, lrate, epochs):
  codebooks = [random_codebook(train) for i in range(n_codebooks)]
  for epoch in range(epochs):
    rate = lrate * (1.0-(epoch/float(epochs)))
    sum_error = 0.0
    for row in train:
      bmu = get_best_matching_unit(codebooks, row)
      for i in range(len(row)-1):
        error = row[i] - bmu[i]
        sum_error += error**2
```

```
        if bmu[-1] == row[-1]:
          bmu[i] += rate * error
        else:
          bmu[i] -= rate * error
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, rate, sum_error))
  return codebooks

# Test the training function
seed(1)
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
learn_rate = 0.3
n_epochs = 10
n_codebooks = 2
codebooks = train_codebooks(dataset, n_codebooks, learn_rate, n_epochs)
print('Codebooks: %s' % codebooks)
```

Listing 14.11: Example of Training Codebook Vectors on the Contrived Dataset.

Running this example trains a set of 2 codebook vectors for 10 epochs with an initial learning rate of 0.3. The details are printed each epoch and the set of 2 codebook vectors learned from the training data is displayed. We can see that the changes to learning rate meet our expectations explored above for each epoch. We can also see that the sum squared error each epoch does continue to drop at the end of training and that there may be an opportunity to tune the example further to achieve less error.

```
>epoch=0, lrate=0.300, error=139.812
>epoch=1, lrate=0.270, error=47.368
>epoch=2, lrate=0.240, error=27.535
>epoch=3, lrate=0.210, error=26.241
>epoch=4, lrate=0.180, error=25.509
>epoch=5, lrate=0.150, error=24.778
>epoch=6, lrate=0.120, error=24.053
>epoch=7, lrate=0.090, error=23.344
>epoch=8, lrate=0.060, error=22.653
>epoch=9, lrate=0.030, error=21.981
Codebooks: [[7.3188612290158614, 1.9696349335193466, 1], [2.4304257696446854,
    2.8396012380964555, 0]]
```

Listing 14.12: Example Output of Training Codebook Vectors.

Now that we know how to train a set of codebook vectors, let's see how we can use this algorithm on a real dataset.

14.2.4 Ionosphere Case Study

In this section, we will apply the Learning Vector Quantization algorithm to the Ionosphere dataset. The first step is to load the dataset and convert the loaded data to numbers that we can use with the Euclidean distance calculation. For this we will use the helper function `load_csv()` to load the file, `str_column_to_float()` to convert string numbers to floats and `str_column_to_int()` to convert the class column to integer values.

We will evaluate the algorithm using k-fold cross-validation with 5 folds. This means that $\frac{351}{5} = 70.2$ or just over 70 records will be in each fold. We will use the helper functions `evaluate_algorithm()` to evaluate the algorithm with cross-validation and `accuracy_metric()` to calculate the accuracy of predictions. The complete example is listed below.

```
# LVQ for the Ionosphere Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
```

```
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Locate the best matching unit
def get_best_matching_unit(codebooks, test_row):
  distances = list()
  for codebook in codebooks:
    dist = euclidean_distance(codebook, test_row)
    distances.append((codebook, dist))
  distances.sort(key=lambda tup: tup[1])
  return distances[0][0]

# Make a prediction with codebook vectors
def predict(codebooks, test_row):
  bmu = get_best_matching_unit(codebooks, test_row)
  return bmu[-1]

# Create a random codebook vector
def random_codebook(train):
  n_records = len(train)
  n_features = len(train[0])
  codebook = [train[randrange(n_records)][i] for i in range(n_features)]
```

```
  return codebook

# Train a set of codebook vectors
def train_codebooks(train, n_codebooks, lrate, epochs):
  codebooks = [random_codebook(train) for i in range(n_codebooks)]
  for epoch in range(epochs):
    rate = lrate * (1.0-(epoch/float(epochs)))
    for row in train:
      bmu = get_best_matching_unit(codebooks, row)
      for i in range(len(row)-1):
        error = row[i] - bmu[i]
        if bmu[-1] == row[-1]:
          bmu[i] += rate * error
        else:
          bmu[i] -= rate * error
  return codebooks

# LVQ Algorithm
def learning_vector_quantization(train, test, n_codebooks, lrate, epochs):
  codebooks = train_codebooks(train, n_codebooks, lrate, epochs)
  predictions = list()
  for row in test:
    output = predict(codebooks, row)
    predictions.append(output)
  return(predictions)

# Test LVQ on Ionosphere dataset
seed(1)
# load and prepare data
filename = 'ionosphere.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
  str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# evaluate algorithm
n_folds = 5
learn_rate = 0.3
n_epochs = 50
n_codebooks = 20
scores = evaluate_algorithm(dataset, learning_vector_quantization, n_folds, n_codebooks,
    learn_rate, n_epochs)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 14.13: Example of LVQ on the Ionosphere Dataset.

Running this example prints the classification accuracy on each fold and the mean classification accuracy across all folds. We can see that the accuracy of 87.143% is better than the baseline of 64.286%. We can also see that our library of 20 codebook vectors is far fewer than holding the entire training dataset.

```
Scores: [90.0, 88.57142857142857, 84.28571428571429, 87.14285714285714, 85.71428571428571]
Mean Accuracy: 87.143%
```

Listing 14.14: Example Output of LVQ on the Ionosphere Dataset.

14.3 Extensions

This section lists extensions to the tutorial that you may wish to explore.

- **Tune Parameters**. The parameters in the above example were not tuned; try different values to improve the classification accuracy.
- **Different Distance Measures**. Experiment with different distance measures such as Manhattan distance and Minkowski distance.
- **Multiple-Pass LVQ**. The codebook vectors may be updated by multiple training runs. Experiment by training with large learning rates followed by a large number of epochs with smaller learning rates to fine tune the codebooks.
- **Update More BMUs**. Experiment with selecting more than one BMU when training and pushing and pulling them away from the training data.
- **More Problems**. Apply LVQ to more classification problems on the UCI Machine Learning Repository.

14.4 Review

In this tutorial, you discovered how to implement the learning vector quantization algorithm from scratch in Python. Specifically, you learned:

- How to calculate the distance between patterns and locate the best matching unit.
- How to train a set of codebook vectors to best summarize the training dataset.
- How to apply the learning vector quantization algorithm to a real predictive modeling problem.

14.4.1 Further Reading

- Chapter 6. Learning Vector Quantization, page 245, *Self-Organizing Maps*, 2000
  http://amzn.to/2f77mhU

14.4.2 Next

In the next tutorial, you will discover how to implement and apply the Back-Propagation algorithm for classification.

Chapter 15

Back-Propagation

The back-propagation algorithm is the classical feedforward artificial neural network. It is the technique still used to train large deep learning networks. In this tutorial, you will discover how to implement the back-propagation algorithm from scratch with Python. After completing this tutorial, you will know:

- How to forward-propagate an input to calculate an output.
- How to back-propagate error and train a network.
- How to apply the back-propagation algorithm to a real-world predictive modeling problem.

Let's get started.

15.1 Description

This section provides a brief introduction to the Back-propagation Algorithm and the Wheat Seeds dataset that we will be using in this tutorial.

15.1.1 Back-propagation Algorithm

The Back-propagation algorithm is a supervised learning method for multilayer feedforward networks from the field of Artificial Neural Networks. Feedforward neural networks are inspired by the information processing of one or more neural cells, called a neuron. A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. The axon carries the signal out to synapses, which are the connections of a cell's axon to other cell's dendrites.

The principle of the back-propagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method where the error between the system's output and a known expected output is presented to the system and used to modify its internal state.

Technically, the back-propagation algorithm is a method for training the weights in a multilayer feedforward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one hidden layer, and one output layer. Back-propagation can be

used for both classification and regression problems, but we will focus on classification in this tutorial.

In classification problems, best results are achieved when the network has one neuron in the output layer for each class value. For example, a 2-class or binary classification problem with the class values of A and B. These expected outputs would have to be transformed into binary vectors with one column for each class value. Such as [1, 0] and [0, 1] for A and B respectively. This is called a one hot encoding.

15.1.2 Wheat Seeds Dataset

In this tutorial we will use the Wheat Seeds Dataset. This dataset involves the prediction of the species of wheat seeds. The baseline performance on the problem is approximately 28%. You can learn more about it in Appendix A, Section A.10. Download the dataset and save it into your current working directory with the filename `seeds_dataset.csv`. The dataset is in tab-separated format, so you must convert it to CSV using a text editor or a spreadsheet program.

15.2 Tutorial

This tutorial is broken down into 6 parts:

1. Initialize Network.
2. Forward-Propagate.
3. Back-propagate Error.
4. Train Network.
5. Predict.
6. Wheat Seeds Case Study.

These steps will provide the foundation that you need to implement the back-propagation algorithm from scratch and apply it to your own predictive modeling problems.

15.2.1 Initialize Network

Let's start with something easy: the creation of a new network ready for training. Each neuron has a set of weights that need to be maintained. One weight for each input connection and an additional weight for the bias. We will need to store additional properties for a neuron during training, therefore we will use a dictionary to represent each neuron and store properties by names such as `weights` for the weights.

A network is organized into layers. The input layer is really just a row from our training dataset. The first real layer is the hidden layer. This is followed by the output layer that has one neuron for each class value. We will organize layers as arrays of dictionaries and treat the whole network as an array of layers. It is good practice to initialize the network weights to small random numbers. In this case, will we use random numbers in the range of 0 to 1.

Below is a function named `initialize_network()` that creates a new neural network ready for training. It accepts three parameters: the number of inputs, the number of neurons to have in the hidden layer and the number of outputs. You can see that for the hidden layer we create `n_hidden` neurons and each neuron in the hidden layer has `n_inputs + 1` weights, one for each input column in a dataset and an additional one for the bias.

You can also see that the output layer that connects to the hidden layer has `n_outputs` neurons, each with `n_hidden + 1` weights. This means that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

```
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
  network = list()
  hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in
      range(n_hidden)]
  network.append(hidden_layer)
  output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in
      range(n_outputs)]
  network.append(output_layer)
  return network
```

Listing 15.1: Function To Initialize a Multilayer Perceptron Network.

Let's test out this function. Below is a complete example that creates a small network.

```
# Example of initializing a network
from random import seed
from random import random

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
  network = list()
  hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in
      range(n_hidden)]
  network.append(hidden_layer)
  output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in
      range(n_outputs)]
  network.append(output_layer)
  return network

# Test initializing a network
seed(1)
network = initialize_network(2, 1, 2)
for layer in network:
  print(layer)
```

Listing 15.2: Example of Initializing a Multilayer Perceptron Network.

Running the example, you can see that the code prints out each layer one by one. You can see the hidden layer has one neuron with 2 input weights plus the bias. The output layer has 2 neurons, each with 1 weight plus the bias.

```
[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}]
[{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights': [0.4494910647887381,
    0.651592972722763]}]
```

Listing 15.3: Sample Output from Initializing a Network.

Now that we know how to create and initialized a network, let's see how we can use it to calculate an output.

15.2.2 Forward-Propagate

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values. We call this forward-propagation. It is the technique we will need to generate predictions during training that will need to be corrected, and it is the method we will need after the network is trained to make predictions on new data. We can break forward-propagation down into three parts:

1. Neuron Activation.
2. Neuron Transfer.
3. Forward-Propagation.

Neuron Activation

The first step is to calculate the activation of one neuron given an input. The input could be a row from our training dataset, as in the case of the hidden layer. It may also be the outputs from each neuron in the hidden layer, in the case of the output layer. Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression.

$$activation = bias + \sum_{i=1}^{n} weight_i \times input_i \tag{15.1}$$

Where `weight` is a network weight, `input` is an input value, `i` is the index of a weight or an input and `bias` is a special weight that has no input to multiply with (or you can think of the input as always being 1.0). Below is an implementation of this in a function named `activate()`. You can see that the function assumes that the bias is the last weight in the list of weights. This helps here and later to make the code easier to read.

```
# Calculate neuron activation for an input
def activate(weights, inputs):
  activation = weights[-1]
  for i in range(len(weights)-1):
    activation += weights[i] * inputs[i]
  return activation
```

Listing 15.4: Function To Activate a Neuron.

Now, let's see how to use the neuron activation.

Neuron Transfer

Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is. Different transfer functions can be used. It is traditional to use the sigmoid activation function, but you can also use the tanh (hyperbolic tangent) function to transfer outputs. More recently, the rectifier transfer function has been popular with large deep learning networks.

The sigmoid activation function looks like an S shape: it's also called the logistic function. It can take any input value and produce a number between 0 and 1 on an S-curve. It is also a function of which we can easily calculate the derivative (slope) that we will need later when back-propagating error. We can transfer an activation function using the sigmoid function as follows:

$$output = \frac{1}{1 + e^{-activation}} \tag{15.2}$$

Where `e` is the base of the natural logarithms (Euler's number). Below is a function named `transfer()` that implements the sigmoid equation.

```
# Transfer neuron activation
def transfer(activation):
  return 1.0 / (1.0 + exp(-activation))
```

Listing 15.5: Function To Transfer a Neuron's Activation.

Now that we have the pieces, let's see how they are used.

Forward-Propagation

Forward-propagating an input is straightforward. We work through each layer of our network calculating the outputs for each neuron. All of the outputs from one layer become inputs to the neurons on the next layer. Below is a function named `forward_propagate()` that implements the forward-propagation for a row of data from our dataset with our neural network.

You can see that a neuron's output value is stored in the neuron with the name `output`. You can also see that we collect the outputs for a layer in an array named `new_inputs` that becomes the array `inputs` and is used as inputs for the following layer. The function returns the outputs from the last layer also called the output layer.

```
# Forward-propagate input to a network output
def forward_propagate(network, row):
  inputs = row
  for layer in network:
    new_inputs = []
    for neuron in layer:
      activation = activate(neuron['weights'], inputs)
      neuron['output'] = transfer(activation)
      new_inputs.append(neuron['output'])
    inputs = new_inputs
  return inputs
```

Listing 15.6: Function To Forward-Propagate Input Through a Network.

Let's put all of these pieces together and test out the forward-propagation of our network. We define our network inline with one hidden neuron that expects 2 input values and an output layer with two neurons.

```
# Example of forward propagating input
from math import exp

# Calculate neuron activation for an input
def activate(weights, inputs):
  activation = weights[-1]
```

```
  for i in range(len(weights)-1):
    activation += weights[i] * inputs[i]
  return activation

# Transfer neuron activation
def transfer(activation):
  return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
  inputs = row
  for layer in network:
    new_inputs = []
    for neuron in layer:
      activation = activate(neuron['weights'], inputs)
      neuron['output'] = transfer(activation)
      new_inputs.append(neuron['output'])
    inputs = new_inputs
  return inputs

# test forward propagation
network = [[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}],
    [{'weights': [0.2550690257394217, 0.49543508709194095]}, {'weights':
        [0.4494910647887381, 0.651592972722763]}]]
row = [1, 0, None]
output = forward_propagate(network, row)
print(output)
```

Listing 15.7: Example of Forward-Propagating an Input Through a Network.

Running the example propagates the input pattern [1, 0] and produces an output value that is printed. Because the output layer has two neurons, we get a list of two numbers as output. The actual output values are just nonsense for now, but next, we will start to learn how to make the weights in the neurons more useful.

```
[0.6629970129852887, 0.7253160725279748]
```

Listing 15.8: Sample Output from Forward-Propagate Input Through a Network.

15.2.3 Back-propagate Error

The back-propagation algorithm is named for the way in which weights are trained. Error is calculated between the expected outputs and the outputs forward-propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go. The math for back-propagating error is rooted in calculus, but we will remain high level in this section and focus on what is calculated and how rather than why the calculations take this particular form.

This part is broken down into two sections.

1. Transfer Derivative.
2. Error Back-propagation.

Transfer Derivative

Given an output value from a neuron, we need to calculate it's slope. We are using the sigmoid transfer function, the derivative of which can be calculated as follows:

$$derivative = output \times (1.0 - output) \tag{15.3}$$

Below is a function named `transfer_derivative()` that implements this equation.

```
# Calculate the derivative of an neuron output
def transfer_derivative(output):
  return output * (1.0 - output)
```

Listing 15.9: Function To Calculate the Derivative of a Neuron's Output.

Now, let's see how this can be used.

Error Back-propagation

The first step is to calculate the error for each output neuron; this will give us our error signal (input) to propagate backwards through the network. The error for a given neuron can be calculated as follows:

$$error = (expected - output) \times \text{transfer\_derivative}(output) \tag{15.4}$$

Where `expected` is the expected `output` value for the neuron, output is the output value for the neuron and `transfer_derivative()` calculates the slope of the neuron's output value, as shown above. This error calculation is used for neurons in the output layer. The expected value is the class value itself. In the hidden layer, things are a little more complicated.

The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer. Think of the error traveling back along the weights of the output layer to the neurons in the hidden layer. The back-propagated error signal is accumulated and then used to determine the error for the neuron in the hidden layer, as follows:

$$error = (weight_k \times error_j) \times \text{transfer\_derivative}(output) \tag{15.5}$$

Where `error_j` is the error signal from the `j`th neuron in the output layer, `weight_k` is the weight that connects the kth neuron to the current neuron and output is the output for the current neuron. Below is a function named `backward_propagate_error()` that implements this procedure.

You can see that the error signal calculated for each neuron is stored with the name `delta`. You can see that the layers of the network are iterated in reverse order, starting at the output and working backwards. This ensures that the neurons in the output layer have delta values calculated first that neurons in the hidden layer can use in the subsequent iteration. I chose the name delta to reflect the change the error implies on the neuron (e.g. the weight delta).

You can see that the error signal for neurons in the hidden layer is accumulated from neurons in the output layer where the hidden neuron number `j` is also the index of the neuron's weight in the output layer `neuron['weights'][j]`.

```
# Back-propagate error and store in neurons
def backward_propagate_error(network, expected):
  for i in reversed(range(len(network))):
```

```
    layer = network[i]
    errors = list()
    if i != len(network)-1:
      for j in range(len(layer)):
        error = 0.0
        for neuron in network[i + 1]:
          error += (neuron['weights'][j] * neuron['delta'])
        errors.append(error)
    else:
      for j in range(len(layer)):
        neuron = layer[j]
        errors.append(expected[j] - neuron['output'])
    for j in range(len(layer)):
      neuron = layer[j]
      neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

Listing 15.10: Function To Back-propagate Error Through a Network.

Let's put all of the pieces together and see how it works. We define a fixed neural network with output values and back-propagate an expected output pattern. The complete example is listed below.

```
# Example of backpropagating error

# Calculate the derivative of an neuron output
def transfer_derivative(output):
  return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
  for i in reversed(range(len(network))):
    layer = network[i]
    errors = list()
    if i != len(network)-1:
      for j in range(len(layer)):
        error = 0.0
        for neuron in network[i + 1]:
          error += (neuron['weights'][j] * neuron['delta'])
        errors.append(error)
    else:
      for j in range(len(layer)):
        neuron = layer[j]
        errors.append(expected[j] - neuron['output'])
    for j in range(len(layer)):
      neuron = layer[j]
      neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])

# test backpropagation of error
network = [[{'output': 0.7105668883115941, 'weights': [0.13436424411240122,
    0.8474337369372327, 0.763774618976614]}],
    [{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.49543508709194095]},
        {'output': 0.6573693455986976, 'weights': [0.4494910647887381, 0.651592972722763]}]]
expected = [0, 1]
backward_propagate_error(network, expected)
for layer in network:
  print(layer)
```

Listing 15.11: Example of Back-propagating Error Through a Network.

Running the example prints the network after the back-propagation of error is complete. You can see that error values are calculated and stored in the neurons for the output layer and the hidden layer.

```
[{'output': 0.7105668883115941, 'weights': [0.13436424411240122, 0.8474337369372327,
    0.763774618976614], 'delta': -0.0005348048046610517}]
[{'output': 0.6213859615555266, 'weights': [0.2550690257394217, 0.4954350870919409 5],
    'delta': -0.14619064683582808}, {'output': 0.6573693455986976, 'weights':
    [0.4494910647887381, 0.651592972722763], 'delta': 0.0771723774346327}]
```

Listing 15.12: Sample Output from Back-propagate Error Through a Network.

Now let's use the back-propagation of error to train the network.

15.2.4 Train Network

The network is trained using stochastic gradient descent. Gradient descent was introduced and described in Section 8.1.2. The procedure involves multiple iterations of exposing a training dataset to the network and for each row of data forward-propagating the inputs, back-propagating the error and updating the network weights. This part is broken down into two sections:

1. Update Weights.
2. Train Network.

Update Weights

Once errors are calculated for each neuron in the network via the back-propagation method above, they can be used to update weights. Network weights are updated as follows:

$$weight = weight + \text{learning\_rate} \times error \times input \tag{15.6}$$

Where `weight` is a given weight, `learning_rate` is a parameter that you must specify, `error` is the error calculated by the back-propagation procedure for the neuron and `input` is the input value that caused the error. The same procedure can be used for updating the bias weight, except there is no input term, or input is the fixed value of 1.0.

Learning rate controls how much to change the weight to correct for the error. For example, a value of 0.1 will update the weight 10% of the amount that it possibly could be updated. Small learning rates are preferred that cause slower learning over a large number of training iterations. This increases the likelihood of the network finding a good set of weights across all layers rather than the fastest set of weights that minimize error (called premature convergence).

Below is a function named `update_weights()` that updates the weights for a network given an input row of data, a learning rate and assume that a forward and backward propagation have already been performed. Remember that the input for the output layer is a collection of outputs from the hidden layer.

```
# Update network weights with error
def update_weights(network, row, l_rate):
  for i in range(len(network)):
    inputs = row[:-1]
    if i != 0:
      inputs = [neuron['output'] for neuron in network[i - 1]]
    for neuron in network[i]:
      for j in range(len(inputs)):
        neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
      neuron['weights'][-1] += l_rate * neuron['delta']
```

Listing 15.13: Function To Update Weights in a Network.

Now that we know how to update network weights, let's see how we can do it repeatedly.

Train Network

As mentioned, the network is updated using stochastic gradient descent. This involves first looping for a fixed number of epochs and within each epoch updating the network for each row in the training dataset.

Because updates are made for each training pattern, this type of learning is called online learning. If errors were accumulated across an epoch before updating the weights, this is called batch learning or batch gradient descent. Below is a function that implements the training of an already initialized neural network with a given training dataset, learning rate, fixed number of epochs and an expected number of output values.

The expected number of output values is used to transform class values in the training data into a one hot encoding. That is a binary vector with one column for each class value to match the output of the network. This is required to calculate the error for the output layer. You can also see that the sum squared error between the expected output and the network output is accumulated each epoch and printed. This is helpful to create a trace of how much the network is learning and improving each epoch.

```
# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
  for epoch in range(n_epoch):
    sum_error = 0
    for row in train:
      outputs = forward_propagate(network, row)
      expected = [0 for i in range(n_outputs)]
      expected[row[-1]] = 1
      sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
      backward_propagate_error(network, expected)
      update_weights(network, row, l_rate)
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
```

Listing 15.14: Function To Train a Neural Network on a Dataset.

We now have all of the pieces to train the network. We can put together an example that includes everything we've seen so far including network initialization and train a network on a small dataset. Below is a small contrived dataset that we can use to test out training our neural network.

```
X1          X2          Y
```

```
2.7810836    2.550537003    0
1.465489372  2.362125076    0
3.396561688  4.400293529    0
1.38807019   1.850220317    0
3.06407232   3.005305973    0
7.627531214  2.759262235    1
5.332441248  2.088626775    1
6.922596716  1.77106367     1
8.675418651  -0.242068655   1
7.673756466  3.508563011    1
```

Listing 15.15: Small Contrived Dataset for Testing Logistic Regression.

Below is a plot of the dataset using different colors to show the different classes for each point.

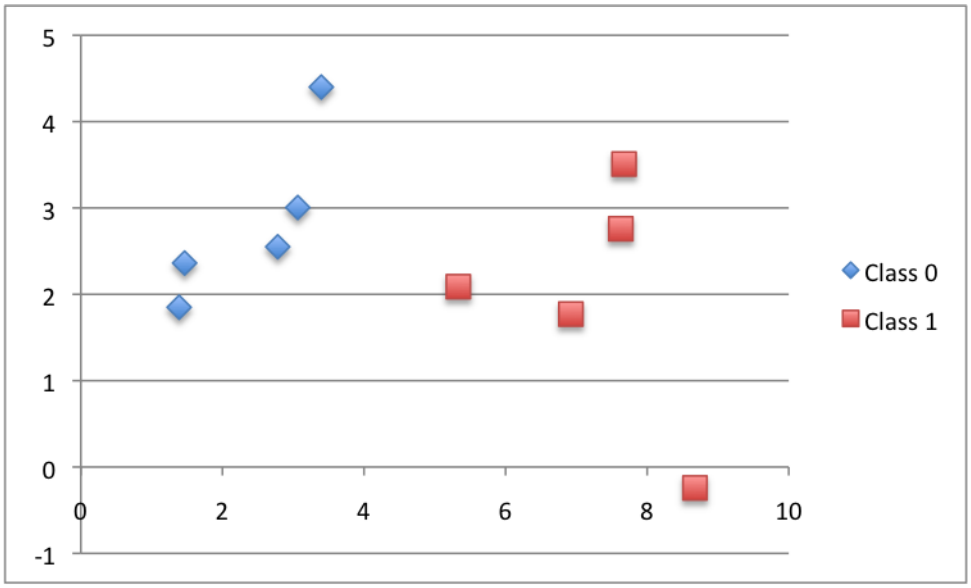

Figure 15.1: Plot of the Small Contrived Dataset for Testing the Back-Propagation algorithm.

Below is the complete example. We will use 2 neurons in the hidden layer. It is a binary classification problem (2 classes) so there will be two neurons in the output layer. The network will be trained for 20 epochs with a learning rate of 0.5, which is high because we are training for so few iterations.

```
# Example of training a network by backpropagation
from math import exp
from random import seed
from random import random

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
  network = list()
  hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in
      range(n_hidden)]
  network.append(hidden_layer)
  output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in
      range(n_outputs)]
```

```
  network.append(output_layer)
  return network

# Calculate neuron activation for an input
def activate(weights, inputs):
  activation = weights[-1]
  for i in range(len(weights)-1):
    activation += weights[i] * inputs[i]
  return activation

# Transfer neuron activation
def transfer(activation):
  return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
  inputs = row
  for layer in network:
    new_inputs = []
    for neuron in layer:
      activation = activate(neuron['weights'], inputs)
      neuron['output'] = transfer(activation)
      new_inputs.append(neuron['output'])
    inputs = new_inputs
  return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
  return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
  for i in reversed(range(len(network))):
    layer = network[i]
    errors = list()
    if i != len(network)-1:
      for j in range(len(layer)):
        error = 0.0
        for neuron in network[i + 1]:
          error += (neuron['weights'][j] * neuron['delta'])
        errors.append(error)
    else:
      for j in range(len(layer)):
        neuron = layer[j]
        errors.append(expected[j] - neuron['output'])
    for j in range(len(layer)):
      neuron = layer[j]
      neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])

# Update network weights with error
def update_weights(network, row, l_rate):
  for i in range(len(network)):
    inputs = row[:-1]
    if i != 0:
      inputs = [neuron['output'] for neuron in network[i - 1]]
    for neuron in network[i]:
```

```
      for j in range(len(inputs)):
        neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
      neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
  for epoch in range(n_epoch):
    sum_error = 0
    for row in train:
      outputs = forward_propagate(network, row)
      expected = [0 for i in range(n_outputs)]
      expected[row[-1]] = 1
      sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
      backward_propagate_error(network, expected)
      update_weights(network, row, l_rate)
    print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))

# Test training backprop algorithm
seed(1)
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
n_inputs = len(dataset[0]) - 1
n_outputs = len(set([row[-1] for row in dataset]))
network = initialize_network(n_inputs, 2, n_outputs)
train_network(network, dataset, 0.5, 20, n_outputs)
for layer in network:
  print(layer)
```

Listing 15.16: Example of Training a Network on the Contrived Dataset.

Running the example first prints the sum squared error each training epoch. We can see a trend of this error decreasing with each epoch. Once trained, the network is printed, showing the learned weights. Also still in the network are output and delta values that can be ignored. We could update our training function to delete these data if we wanted.

```
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.221
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
```

```
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'output': 0.029980305604426185, 'weights': [-1.4688375095432327, 1.850887325439514,
    1.0858178629550297], 'delta': -0.0059546604162323625}, {'output': 0.9456229000211323,
    'weights': [0.37711098142462157, -0.0625909894552989, 0.2765123702642716], 'delta':
    0.0026279652850863837}]
[{'output': 0.23648794202357587, 'weights': [2.515394649397849, -0.3391927502445985,
    -0.9671565426390275], 'delta': -0.04270059278364587}, {'output': 0.7790535202438367,
    'weights': [-2.5584149848484263, 1.0036422106209202, 0.42383086467582715], 'delta':
    0.03803132596437354}]
```

Listing 15.17: Example Output from Training a Network on the Contrived Dataset.

Once a network is trained, we need to use it to make predictions.

15.2.5 Predict

Making predictions with a trained neural network is easy enough. We have already seen how to forward-propagate an input pattern to get an output. This is all we need to do to make a prediction. We can use the output values themselves directly as the probability of a pattern belonging to each output class.

It may be more useful to turn this output back into a crisp class prediction. We can do this by selecting the class value with the larger probability. This is also called the `arg max` function. Below is a function named `predict()` that implements this procedure. It returns the index in the network output that has the largest probability. It assumes that class values have been converted to integers starting at 0.

```
# Make a prediction with a network
def predict(network, row):
  outputs = forward_propagate(network, row)
  return outputs.index(max(outputs))
```

Listing 15.18: Function To Make a Prediction With a Network.

We can put this together with our code above for forward-propagating input and with our small contrived dataset to test making predictions with an already-trained network. The example hardcodes a network trained from the previous step. The complete example is listed below.

```
# Example of making predictions
from math import exp

# Calculate neuron activation for an input
def activate(weights, inputs):
  activation = weights[-1]
  for i in range(len(weights)-1):
    activation += weights[i] * inputs[i]
  return activation
```

```
# Transfer neuron activation
def transfer(activation):
  return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
  inputs = row
  for layer in network:
    new_inputs = []
    for neuron in layer:
      activation = activate(neuron['weights'], inputs)
      neuron['output'] = transfer(activation)
      new_inputs.append(neuron['output'])
    inputs = new_inputs
  return inputs

# Make a prediction with a network
def predict(network, row):
  outputs = forward_propagate(network, row)
  return outputs.index(max(outputs))

# Test making predictions with the network
dataset = [[2.7810836,2.550537003,0],
  [1.465489372,2.362125076,0],
  [3.396561688,4.400293529,0],
  [1.38807019,1.850220317,0],
  [3.06407232,3.005305973,0],
  [7.627531214,2.759262235,1],
  [5.332441248,2.088626775,1],
  [6.922596716,1.77106367,1],
  [8.675418651,-0.242068655,1],
  [7.673756466,3.508563011,1]]
network = [[{'weights': [-1.482313569067226, 1.8308790073202204, 1.078381922048799]},
    {'weights': [0.23244990332399884, 0.3621998343835864, 0.40289821191094327]}],
  [{'weights': [2.5001872433501404, 0.7887233511355132, -1.1026649757805829]}, {'weights':
      [-2.429350576245497, 0.8357651039198697, 1.0699217181280656]}]]
for row in dataset:
  prediction = predict(network, row)
  print('Expected=%d, Got=%d' % (row[-1], prediction))
```

Listing 15.19: Example of Making a Prediction on the Contrived Dataset.

Running the example prints the expected output for each record in the training dataset, followed by the crisp prediction made by the network. It shows that the network achieves 100% accuracy on this small dataset.

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```

Listing 15.20: Example Output from Making Predictions on the Contrived Dataset.

Now we are ready to apply our back-propagation algorithm to a real world dataset.

15.2.6 Wheat Seeds Case Study

This section applies the Back-propagation algorithm to the wheat seeds dataset. The first step is to load the dataset and convert the loaded data to numbers that we can use in our neural network. For this we will use the helper function `load_csv()` to load the file, `str_column_to_float()` to convert string numbers to floats and `str_column_to_int()` to convert the class column to integer values.

Input values vary in scale and need to be normalized to the range of 0 and 1. It is generally good practice to normalize input values to the range of the chosen transfer function, in this case, the sigmoid function that outputs values between 0 and 1. The `dataset_minmax()` and `normalize_dataset()` helper functions were used to normalize the input values.

We will evaluate the algorithm using k-fold cross-validation with 5 folds. This means that $\frac{201}{5} = 40.2$ or 40 records will be in each fold. We will use the helper functions `evaluate_algorithm()` to evaluate the algorithm with cross-validation and `accuracy_metric()` to calculate the accuracy of predictions.

A new function named `back_propagation()` was developed to manage the application of the Back-propagation algorithm, first initializing a network, training it on the training dataset and then using the trained network to make predictions on a test dataset. The complete example is listed below.

```
# Backprop on the Seeds Dataset
from random import seed
from random import randrange
from random import random
from csv import reader
from math import exp

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
```

```
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Find the min and max values for each column
def dataset_minmax(dataset):
  minmax = list()
  stats = [[min(column), max(column)] for column in zip(*dataset)]
  return stats

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
  for row in dataset:
    for i in range(len(row)-1):
      row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
```

```
    scores.append(accuracy)
  return scores

# Calculate neuron activation for an input
def activate(weights, inputs):
  activation = weights[-1]
  for i in range(len(weights)-1):
    activation += weights[i] * inputs[i]
  return activation

# Transfer neuron activation
def transfer(activation):
  return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
  inputs = row
  for layer in network:
    new_inputs = []
    for neuron in layer:
      activation = activate(neuron['weights'], inputs)
      neuron['output'] = transfer(activation)
      new_inputs.append(neuron['output'])
    inputs = new_inputs
  return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
  return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
  for i in reversed(range(len(network))):
    layer = network[i]
    errors = list()
    if i != len(network)-1:
      for j in range(len(layer)):
        error = 0.0
        for neuron in network[i + 1]:
          error += (neuron['weights'][j] * neuron['delta'])
        errors.append(error)
    else:
      for j in range(len(layer)):
        neuron = layer[j]
        errors.append(expected[j] - neuron['output'])
    for j in range(len(layer)):
      neuron = layer[j]
      neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])

# Update network weights with error
def update_weights(network, row, l_rate):
  for i in range(len(network)):
    inputs = row[:-1]
    if i != 0:
      inputs = [neuron['output'] for neuron in network[i - 1]]
    for neuron in network[i]:
```

```
      for j in range(len(inputs)):
        neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
      neuron['weights'][-1] += l_rate * neuron['delta']

# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
  for epoch in range(n_epoch):
    for row in train:
      outputs = forward_propagate(network, row)
      expected = [0 for i in range(n_outputs)]
      expected[row[-1]] = 1
      backward_propagate_error(network, expected)
      update_weights(network, row, l_rate)

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
  network = list()
  hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in
      range(n_hidden)]
  network.append(hidden_layer)
  output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in
      range(n_outputs)]
  network.append(output_layer)
  return network

# Make a prediction with a network
def predict(network, row):
  outputs = forward_propagate(network, row)
  return outputs.index(max(outputs))

# Backpropagation Algorithm With Stochastic Gradient Descent
def back_propagation(train, test, l_rate, n_epoch, n_hidden):
  n_inputs = len(train[0]) - 1
  n_outputs = len(set([row[-1] for row in train]))
  network = initialize_network(n_inputs, n_hidden, n_outputs)
  train_network(network, train, l_rate, n_epoch, n_outputs)
  predictions = list()
  for row in test:
    prediction = predict(network, row)
    predictions.append(prediction)
  return(predictions)

# Test Backprop on Seeds dataset
seed(1)
# load and prepare data
filename = 'seeds_dataset.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
  str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# normalize input variables
minmax = dataset_minmax(dataset)
normalize_dataset(dataset, minmax)
# evaluate algorithm
n_folds = 5
```

```
l_rate = 0.3
n_epoch = 500
n_hidden = 5
scores = evaluate_algorithm(dataset, back_propagation, n_folds, l_rate, n_epoch, n_hidden)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 15.21: Back-propagation Algorithm on the Wheat Seeds Dataset.

A network with 5 neurons in the hidden layer and 3 neurons in the output layer was constructed. The network was trained for 500 epochs with a learning rate of 0.3. These parameters were found with a little trial and error, but you may be able to do much better. Running the example prints the average classification accuracy on each fold as well as the average performance across all folds.

You can see that back-propagation and the chosen configuration achieved a mean classification accuracy of 95.238% which is dramatically better than the baseline of 28% accuracy.

```
Scores: [95.23809523809523, 97.61904761904762, 95.23809523809523, 92.85714285714286,
    95.23809523809523]
Mean Accuracy: 95.238%
```

Listing 15.22: Example Output from the Back-propagation Algorithm on the Wheat Seeds Dataset.

15.3 Extensions

This section lists extensions to the tutorial that you may wish to explore.

- **Tune Algorithm Parameters**. Try larger or smaller networks trained for longer or shorter. See if you can get better performance on the seeds dataset.
- **Additional Methods**. Experiment with different weight initialization techniques (such as small random numbers) and different transfer functions (such as tanh).
- **More Layers**. Add support for more hidden layers, trained in just the same way as the one hidden layer used in this tutorial.
- **Regression**. Change the network so that there is only one neuron in the output layer and that a real value is predicted. Pick a regression dataset to practice on. A linear transfer function could be used for neurons in the output layer, or the output values of the chosen dataset could be scaled to values between 0 and 1.
- **Batch Gradient Descent**. Change the training procedure from online to batch gradient descent and update the weights only at the end of each epoch.

15.4 Review

In this tutorial, you discovered how to implement the Back-propagation algorithm from scratch. Specifically, you learned:

- How to forward-propagate an input to calculate a network output.
- How to back-propagate error and update network weights.
- How to apply the back-propagation algorithm to a real world dataset.

15.4.1 Further Reading

- Section 18.7. Artificial Neural Networks, page 717, *Artificial Intelligence: A Modern Approach*, 2010.
  http://amzn.to/2e3lFqP
- Section 7.1 Neural Networks, page 141 and Section 13.2 Neural Networks, page 333, *Applied Predictive Modeling*, 2013
  http://amzn.to/2e3lNXF
- Chapter 5. Back-Propagation, page 39, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999
  http://amzn.to/2fmFWUC

15.4.2 Next

This ends Part 3 on nonlinear algorithms. Next, in Part 4 you will discover ensemble algorithms. In the next tutorial, you will discover how to implement and apply the Bootstrap Aggregation algorithm for classification.

Part IV

Ensemble Algorithms

Chapter 16

Bootstrap Aggregation

Decision trees are a simple and powerful predictive modeling technique, but they suffer from high-variance. This means that trees can get very different results given different training data. A technique to make decision trees more robust and to achieve better performance is called bootstrap aggregation or bagging for short. In this tutorial, you will discover how to implement the bagging procedure with decision trees from scratch with Python. After completing this tutorial, you will know:

- How to create a bootstrap sample of your dataset.
- How to make predictions with bootstrapped models.
- How to apply bagging to your own predictive modeling problems.

Let's get started.

16.1 Descriptions

This section provides a brief description to Bootstrap Aggregation and the Sonar dataset that will be used in this tutorial.

16.1.1 Bootstrap Aggregation Algorithm

A bootstrap is a sample of a dataset with replacement. This means that a new dataset is created from a random sample of an existing dataset where a given row may be selected and added more than once to the sample. It is a useful approach to use when estimating values such as the mean for a broader dataset, when you only have a limited dataset available. By creating samples of your dataset and estimating the mean from those samples, you can take the average of those estimates and get a better idea of the true mean of the underlying problem.

This same approach can be used with machine learning algorithms that have a high variance, such as decision trees (CART) introduced in Chapter 11. A separate model is trained on each bootstrap sample of data and the average output of those models used to make predictions. This technique is called bootstrap aggregation or bagging for short. Variance means that an algorithm's performance is sensitive to the training data, with high variance suggesting that the more the training data is changed, the more the performance of the algorithm will vary.

The performance of high variance machine learning algorithms like unpruned decision trees can be improved by training many trees and taking the average of their predictions. Results are often better than a single decision tree. Another benefit of bagging in addition to improved performance is that the bagged decision trees cannot overfit the problem. Trees can continue to be added until a maximum in performance is achieved.

16.1.2 Sonar Dataset

In this tutorial we will use the Sonar Dataset. This dataset involves the discrimination between mines and rocks. The baseline performance on the problem is approximately 53%. You can learn more about it in Appendix A, Section A.5. Download the dataset and save it into your current working directory with the filename `sonar.all-data.csv`.

16.2 Tutorial

This tutorial is broken down into 2 parts:

- Bootstrap Resample.
- Sonar Case Study.

These steps provide the foundation that you need to implement and apply bootstrap aggregation with decision trees to your own predictive modeling problems.

16.2.1 Bootstrap Resample

Let's start off by getting a strong idea of how the bootstrap method works. We can create a new sample of a dataset by randomly selecting rows from the dataset and adding them to a new list. We can repeat this for a fixed number of rows or until the size of the new dataset matches a ratio of the size of the original dataset.

We can allow sampling with replacement by not removing the row that was selected so that it is available for future selections. Below is a function named `subsample()` that implements this procedure. The `randrange()` function from the random module is used to select a random row index to add to the sample each iteration of the loop. The default size of the sample is the size of the original dataset.

```
# Create a random subsample from the dataset with replacement
def subsample(dataset, ratio=1.0):
  sample = list()
  n_sample = round(len(dataset) * ratio)
  while len(sample) < n_sample:
    index = randrange(len(dataset))
    sample.append(dataset[index])
  return sample
```

Listing 16.1: Function To Make a Subsample of a Dataset.

We can use this function to estimate the mean of a contrived dataset. First, we can create a dataset with 20 rows and a single column of random numbers between 0 and 9 and calculate the

mean value. We can then make bootstrap samples of the original dataset, calculate the mean, and repeat this process until we have a list of means. Taking the average of these sample means should give us a robust estimate of the mean of the entire dataset.

The complete example is listed below. Each bootstrap sample is created as a 10% sample of the original 20 observation dataset (or 2 observations). We then experiment by creating 1, 10, 100 bootstrap samples of the original dataset, calculate their mean value, then average all of those estimated mean values.

```
# Example of subsampling a dataste
from random import seed
from random import random
from random import randrange

# Create a random subsample from the dataset with replacement
def subsample(dataset, ratio=1.0):
  sample = list()
  n_sample = round(len(dataset) * ratio)
  while len(sample) < n_sample:
    index = randrange(len(dataset))
    sample.append(dataset[index])
  return sample

# Calculate the mean of a list of numbers
def mean(numbers):
  return sum(numbers) / float(len(numbers))

# Test subsampling a dataset
seed(1)
# True mean
dataset = [[randrange(10)] for i in range(20)]
print('True Mean: %.3f' % mean([row[0] for row in dataset]))
# Estimated means
ratio = 0.10
for size in [1, 10, 100]:
  sample_means = list()
  for i in range(size):
    sample = subsample(dataset, ratio)
    sample_mean = mean([row[0] for row in sample])
    sample_means.append(sample_mean)
  print('Samples=%d, Estimated Mean: %.3f' % (size, mean(sample_means)))
```

Listing 16.2: Example of Subsampling a Dataset.

Running the example prints the original mean value we aim to estimate. We can then see the estimated mean from the various different numbers of bootstrap samples. We can see that with 100 samples we achieve a good estimate of the mean.

```
True Mean: 4.450
Samples=1, Estimated Mean: 4.500
Samples=10, Estimated Mean: 3.300
Samples=100, Estimated Mean: 4.480
```

Listing 16.3: Example Output from Subsampling a Dataset.

Instead of calculating the mean value, we can create a model from each subsample. Next, let's see how we can combine the predictions from multiple bootstrap models.

16.2.2 Sonar Case Study

In this section, we will apply the Random Forest algorithm to the Sonar dataset. The example assumes that a CSV copy of the dataset is in the current working directory with the file name `sonar.all-data.csv`.

The dataset is first loaded, the string values converted to numeric and the output column is converted from strings to the integer values of 0 to 1. This is achieved with helper functions `load_csv()`, `str_column_to_float()` and `str_column_to_int()` to load and prepare the dataset.

We will use k-fold cross-validation to estimate the performance of the learned model on unseen data. This means that we will construct and evaluate k models and estimate the performance as the mean model error. Classification accuracy will be used to evaluate each model. These behaviors are provided in the `cross_validation_split()`, `accuracy_metric()` and `evaluate_algorithm()` helper functions.

We will also use an implementation of the Classification and Regression Trees (CART) algorithm adapted for bagging with the helper functions from Chapter 11 including `test_split()` to split a dataset into groups, `gini_index()` to evaluate a split point, `get_split()` to find an optimal split point, `to_terminal()`, `split()` and `build_tree()` used to create a single decision tree, `predict()` to make a prediction with a decision tree and the `subsample()` function described in the previous step to make a subsample of the training dataset.

A new function named `bagging_predict()` is developed that is responsible for making a prediction with each decision tree and combining the predictions into a single return value. This is achieved by selecting the most common prediction from the list of predictions made by the bagged trees.

Finally, a new function named `bagging()` is developed that is responsible for creating the samples of the training dataset, training a decision tree on each, then making predictions on the test dataset using the list of bagged trees. The complete example is listed below.

```
# Bagging Algorithm on the Sonar dataset
from random import seed
from random import randrange
from csv import reader

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
```

```
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
```

```
      right.append(row)
  return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):
  gini = 0.0
  for class_value in class_values:
    for group in groups:
      size = len(group)
      if size == 0:
        continue
      proportion = [row[-1] for row in group].count(class_value) / float(size)
      gini += (proportion * (1.0 - proportion))
  return gini

# Select the best split point for a dataset
def get_split(dataset):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  for index in range(len(dataset[0])-1):
    for row in dataset:
    # for i in range(len(dataset)):
    #   row = dataset[randrange(len(dataset))]
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node value
def to_terminal(group):
  outcomes = [row[-1] for row in group]
  return max(set(outcomes), key=outcomes.count)

# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
  left, right = node['groups']
  del(node['groups'])
  # check for a no split
  if not left or not right:
    node['left'] = node['right'] = to_terminal(left + right)
    return
  # check for max depth
  if depth >= max_depth:
    node['left'], node['right'] = to_terminal(left), to_terminal(right)
    return
  # process left child
  if len(left) <= min_size:
    node['left'] = to_terminal(left)
  else:
    node['left'] = get_split(left)
    split(node['left'], max_depth, min_size, depth+1)
  # process right child
  if len(right) <= min_size:
    node['right'] = to_terminal(right)
  else:
```

```
    node['right'] = get_split(right)
    split(node['right'], max_depth, min_size, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size):
  root = get_split(train)
  split(root, max_depth, min_size, 1)
  return root

# Make a prediction with a decision tree
def predict(node, row):
  if row[node['index']] < node['value']:
    if isinstance(node['left'], dict):
      return predict(node['left'], row)
    else:
      return node['left']
  else:
    if isinstance(node['right'], dict):
      return predict(node['right'], row)
    else:
      return node['right']

# Create a random subsample from the dataset with replacement
def subsample(dataset, ratio):
  sample = list()
  n_sample = round(len(dataset) * ratio)
  while len(sample) < n_sample:
    index = randrange(len(dataset))
    sample.append(dataset[index])
  return sample

# Make a prediction with a list of bagged trees
def bagging_predict(trees, row):
  predictions = [predict(tree, row) for tree in trees]
  return max(set(predictions), key=predictions.count)

# Bootstrap Aggregation Algorithm
def bagging(train, test, max_depth, min_size, sample_size, n_trees):
  trees = list()
  for i in range(n_trees):
    sample = subsample(train, sample_size)
    tree = build_tree(sample, max_depth, min_size)
    trees.append(tree)
  predictions = [bagging_predict(trees, row) for row in test]
  return(predictions)

# Test bagging on the sonar dataset
seed(1)
# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)
# convert string attributes to integers
for i in range(len(dataset[0])-1):
  str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
```

```
# evaluate algorithm
n_folds = 5
max_depth = 6
min_size = 2
sample_size = 0.50
for n_trees in [1, 5, 10, 50]:
  scores = evaluate_algorithm(dataset, bagging, n_folds, max_depth, min_size, sample_size,
      n_trees)
  print('Trees: %d' % n_trees)
  print('Scores: %s' % scores)
  print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 16.4: Example of Bagging on the Sonar Dataset.

A k value of 5 was used for cross-validation, giving each fold $\frac{208}{5} = 41.6$ or just over 40 records to be evaluated upon each iteration.

Deep trees were constructed with a max depth of 6 and a minimum number of training rows at each node of 2. Samples of the training dataset were created with 50% the size of the original dataset. This was to force some variety in the dataset subsamples used to train each tree. The default for bagging is to have the size of sample datasets match the size of the original training dataset.

A series of 4 different numbers of trees were evaluated to show the behavior of the algorithm. The accuracy from each fold and the mean accuracy for each configuration are printed. We can see a trend of some minor lift in performance as the number of trees is increased.

```
Trees: 1
Scores: [60.97560975609756, 65.85365853658537, 58.536585365853654, 63.41463414634146,
    65.85365853658537]
Mean Accuracy: 62.927%

Trees: 5
Scores: [65.85365853658537, 56.09756097560976, 68.29268292682927, 68.29268292682927,
    53.65853658536586]
Mean Accuracy: 62.439%

Trees: 10
Scores: [60.97560975609756, 75.60975609756098, 75.60975609756098, 73.17073170731707,
    60.97560975609756]
Mean Accuracy: 69.268%

Trees: 50
Scores: [75.60975609756098, 68.29268292682927, 68.29268292682927, 65.85365853658537,
    73.17073170731707]
Mean Accuracy: 70.244%
```

Listing 16.5: Example Output of Bagging on the Sonar Dataset.

A difficulty of this method is that even though deep trees are constructed, the bagged trees that are created are very similar. In turn, the predictions made by these trees are also similar, and the high variance we desire among the trees trained on different samples of the training dataset is diminished.

This is because of the greedy algorithm used in the construction of the trees selecting the same or similar split points. The tutorial tried to re-inject this variance by constraining the sample size used to train each tree. A more robust technique is to constrain the features that

may be evaluated when creating each split point. This is the method used in the Random Forest algorithm.

16.3 Extensions

- **Tune the Example**. Explore different configurations for the number of trees and even individual tree configurations to see if you can further improve results.
- **Bag Another Algorithm**. Other algorithms can be used with bagging. For example, a k-nearest neighbor algorithm with a low value of k will have a high variance and is a good candidate for bagging.
- **Regression Problems**. Bagging can be used with regression trees. Instead of predicting the most common class value from the set of predictions, you can return the average of the predictions from the bagged trees. Experiment on regression problems.

16.4 Review

In this tutorial, you discovered how to implement bootstrap aggregation from scratch with Python. Specifically, you learned:

- How to create a subsample and estimate bootstrap quantities.
- How to create an ensemble of decision trees and use them to make predictions.
- How to apply bagging to a real world predictive modeling problem.

16.4.1 Further Reading

- Section 8.2. Bagging, Random Forests, Boosting, page 316, *An Introduction to Statistical Learning*, 2014.
  http://amzn.to/2eeTyQX
- Section 8.4 Bagged Trees, page 192 and Section 14.3, Bagged Trees, page 385, *Applied Predictive Modeling*, 2013
  http://amzn.to/2e3lNXF
- Section 7.5, Combining multiple models, page 315, *Data Mining: Practical Machine Learning Tools and Techniques*, second edition, 2005.
  http://amzn.to/2fj3SYY

16.4.2 Next

In the next tutorial, you will discover how to implement and apply the Random Forest algorithm for classification.

Chapter 17

Random Forest

Decision trees can suffer from high variance which makes their results fragile to the specific training data used. Building multiple models from samples of your training data, called bagging, can reduce this variance, but the trees are highly correlated.

Random Forest is an extension of bagging that in addition to building trees based on multiple samples of your training data, it also constrains the features that can be used to build the trees, forcing trees to be different. This, in turn, can give a lift in performance. In this tutorial, you will discover how to implement the Random Forest algorithm from scratch in Python. After completing this tutorial, you will know:

- The difference between bagged decision trees and the random forest algorithm.
- How to construct bagged decision trees with more variance.
- How to apply the random forest algorithm to a predictive modeling problem.

Let's get started.

17.1 Description

This section provides a brief introduction to the Random Forest algorithm and the Sonar dataset used in this tutorial.

17.1.1 Random Forest Algorithm

Decision trees involve the greedy selection of the best split point from the dataset at each step. This algorithm makes decision trees susceptible to high variance if they are not pruned. This high variance can be harnessed and reduced by creating multiple trees with different samples of the training dataset (different views of the problem) and combining their predictions. This approach is called bootstrap aggregation or bagging for short.

A limitation of bagging is that the same greedy algorithm is used to create each tree, meaning that it is likely that the same or very similar split points will be chosen in each tree making the different trees very similar (trees will be correlated). This, in turn, makes their predictions similar, mitigating the variance originally sought.

We can force the decision trees to be different by limiting the features (columns) that the greedy algorithm can evaluate at each split point when creating the tree. This is called the Random Forest algorithm. Like bagging, multiple samples of the training dataset are taken and a different tree trained on each. The difference is that at each point a split is made in the data and added to the tree, only a fixed subset of attributes can be considered. For classification problems, the type of problems we will look at in this tutorial, the number of attributes to be considered for the split is limited to the square root of the number of input features.

$$\text{num\_features\_for\_split} = \sqrt{\text{total\_input\_features}} \tag{17.1}$$

The result of this one small change are trees that are more different from each other (uncorrelated) resulting in predictions that are more diverse and a combined prediction that often has better performance than a single tree or bagging alone.

17.1.2 Sonar Dataset

In this tutorial we will use the Sonar Dataset. This dataset involves the discrimination between mines and rocks. The baseline performance on the problem is approximately 53%. You can learn more about it in Appendix A, Section A.5. Download the dataset and save it into your current working directory with the filename `sonar.all-data.csv`.

17.2 Tutorial

This tutorial is broken down into 2 steps.

1. Calculating Splits.
2. Sonar Case Study.

These steps provide the foundation that you need to implement and apply the Random Forest algorithm to your own predictive modeling problems.

17.2.1 Calculating Splits

In a decision tree, split points are chosen by finding the attribute and the value of that attribute that results in the lowest cost. For classification problems, this cost function is often the Gini index that calculates the purity of the groups of data created by the split point. A Gini index of 0 is perfect purity where class values are perfectly separated into two groups, in the case of a two-class classification problem.

Finding the best split point in a decision tree involves evaluating the cost of each value in the training dataset for each input variable. For bagging and random forest, this procedure is executed upon a sample of the training dataset, made with replacement. Sampling with replacement means that the same row may be chosen and added to the sample more than once.

We can update this procedure for Random Forest. Instead of enumerating all values for input attributes in search of the split with the lowest cost, we can create a sample of the input attributes to consider. This sample of input attributes can be chosen randomly and without

replacement, meaning that each input attribute needs to only be considered once when looking for the split point with the lowest cost.

Below is a function name `get_split()` that implements this procedure. It takes a dataset and a fixed number of input features to evaluate as input arguments, where the dataset may be a sample of the actual training dataset. The helper function `test_split()` is used to split the dataset by a candidate split point and `gini_index()` is used to evaluate the cost of a given split by the groups of rows created.

We can see that a list of features is created by randomly selecting feature indices and adding them to a list (called `features`), this list of features is then enumerated and specific values in the training dataset evaluated as split points.

```
# Select the best split point for a dataset
def get_split(dataset, n_features):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  features = list()
  while len(features) < n_features:
    index = randrange(len(dataset[0])-1)
    if index not in features:
      features.append(index)
  for index in features:
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}
```

Listing 17.1: Function To Make Split Points For Random Forest Decision Trees.

Now that we know how a decision tree algorithm can be modified for use with the Random Forest algorithm, we can piece this together with an implementation of bagging and apply it to a real-world dataset.

17.2.2 Sonar Case Study

In this section, we will apply the Random Forest algorithm to the Sonar dataset. The example assumes that a CSV copy of the dataset is in the current working directory with the file name `sonar.all-data.csv`.

The dataset is first loaded, the string values converted to numeric and the output column is converted from strings to the integer values of 0 and 1. This is achieved with helper functions `load_csv()`, `str_column_to_float()` and `str_column_to_int()` to load and prepare the dataset.

We will use k-fold cross-validation to estimate the performance of the learned model on unseen data. This means that we will construct and evaluate k models and estimate the performance as the mean model error. Classification accuracy will be used to evaluate each model. These behaviors are provided in the `cross_validation_split()`, `accuracy_metric()` and `evaluate_algorithm()` helper functions.

We will also use an implementation of the Classification and Regression Trees (CART) algorithm adapted for bagging and the helper functions from Chapter 11 including `test_split()` to split a dataset into groups, `gini_index()` to evaluate a split point, our modified `get_split()`

function discussed in the previous step, `to_terminal()`, `split()` and `build_tree()` used to create a single decision tree, `predict()` to make a prediction with a decision tree, `subsample()` to make a subsample of the training dataset and `bagging_predict()` to make a prediction with a list of decision trees.

A new function name `random_forest()` is developed that first creates a list of decision trees from subsamples of the training dataset and then uses them to make predictions. As we stated above, the key difference between Random Forest and bagged decision trees is the one small change to the way that trees are created here in the `get_split()` function. The complete example is listed below.

```
# Random Forest Algorithm on Sonar Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split
```

```
# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
  left, right = list(), list()
  for row in dataset:
    if row[index] < value:
      left.append(row)
    else:
      right.append(row)
  return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):
  gini = 0.0
  for class_value in class_values:
    for group in groups:
      size = len(group)
      if size == 0:
        continue
      proportion = [row[-1] for row in group].count(class_value) / float(size)
      gini += (proportion * (1.0 - proportion))
  return gini

# Select the best split point for a dataset
def get_split(dataset, n_features):
  class_values = list(set(row[-1] for row in dataset))
  b_index, b_value, b_score, b_groups = 999, 999, 999, None
  features = list()
  while len(features) < n_features:
```

```
    index = randrange(len(dataset[0])-1)
    if index not in features:
      features.append(index)
  for index in features:
    for row in dataset:
      groups = test_split(index, row[index], dataset)
      gini = gini_index(groups, class_values)
      if gini < b_score:
        b_index, b_value, b_score, b_groups = index, row[index], gini, groups
  return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node value
def to_terminal(group):
  outcomes = [row[-1] for row in group]
  return max(set(outcomes), key=outcomes.count)

# Create child splits for a node or make terminal
def split(node, max_depth, min_size, n_features, depth):
  left, right = node['groups']
  del(node['groups'])
  # check for a no split
  if not left or not right:
    node['left'] = node['right'] = to_terminal(left + right)
    return
  # check for max depth
  if depth >= max_depth:
    node['left'], node['right'] = to_terminal(left), to_terminal(right)
    return
  # process left child
  if len(left) <= min_size:
    node['left'] = to_terminal(left)
  else:
    node['left'] = get_split(left, n_features)
    split(node['left'], max_depth, min_size, n_features, depth+1)
  # process right child
  if len(right) <= min_size:
    node['right'] = to_terminal(right)
  else:
    node['right'] = get_split(right, n_features)
    split(node['right'], max_depth, min_size, n_features, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size, n_features):
  root = get_split(train, n_features)
  split(root, max_depth, min_size, n_features, 1)
  return root

# Make a prediction with a decision tree
def predict(node, row):
  if row[node['index']] < node['value']:
    if isinstance(node['left'], dict):
      return predict(node['left'], row)
    else:
      return node['left']
  else:
    if isinstance(node['right'], dict):
```

```
      return predict(node['right'], row)
    else:
      return node['right']

# Create a random subsample from the dataset with replacement
def subsample(dataset, ratio):
  sample = list()
  n_sample = round(len(dataset) * ratio)
  while len(sample) < n_sample:
    index = randrange(len(dataset))
    sample.append(dataset[index])
  return sample

# Make a prediction with a list of bagged trees
def bagging_predict(trees, row):
  predictions = [predict(tree, row) for tree in trees]
  return max(set(predictions), key=predictions.count)

# Random Forest Algorithm
def random_forest(train, test, max_depth, min_size, sample_size, n_trees, n_features):
  trees = list()
  for i in range(n_trees):
    sample = subsample(train, sample_size)
    tree = build_tree(sample, max_depth, min_size, n_features)
    trees.append(tree)
  predictions = [bagging_predict(trees, row) for row in test]
  return(predictions)

# Test the random forest algorithm on sonar dataset
seed(1)
# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)
# convert string attributes to integers
for i in range(0, len(dataset[0])-1):
  str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# evaluate algorithm
n_folds = 5
max_depth = 10
min_size = 1
sample_size = 1.0
n_features = int(sqrt(len(dataset[0])-1))
for n_trees in [1, 5, 10]:
  scores = evaluate_algorithm(dataset, random_forest, n_folds, max_depth, min_size,
      sample_size, n_trees, n_features)
  print('Trees: %d' % n_trees)
  print('Scores: %s' % scores)
  print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 17.2: Example of Random Forest on the Sonar Dataset.

A k value of 5 was used for cross-validation, giving each fold $\frac{208}{5} = 41.6$ or just over 40 records to be evaluated upon each iteration. Deep trees were constructed with a max depth of 10 and a minimum number of training rows at each node of 1. Samples of the training dataset

were created with the same size as the original dataset, which is a default expectation for the Random Forest algorithm.

The number of features considered at each split point was set to $\sqrt{\text{num\_features}}$ or $\sqrt{60} = 7.74$ rounded to 7 features. A suite of 3 different numbers of trees were evaluated for comparison, showing the increasing skill as more trees are added. Running the example prints the scores for each fold and mean score for each configuration.

```
Trees: 1
Scores: [51.21951219512195, 78.04878048780488, 58.536585365853654, 65.85365853658537,
    53.65853658536586]
Mean Accuracy: 61.463%

Trees: 5
Scores: [63.41463414634146, 60.97560975609756, 56.09756097560976, 60.97560975609756,
    56.09756097560976]
Mean Accuracy: 59.512%

Trees: 10
Scores: [63.41463414634146, 53.65853658536586, 70.73170731707317, 56.09756097560976,
    68.29268292682927]
Mean Accuracy: 62.439%
```

Listing 17.3: Example Output of Random Forest on the Sonar Dataset.

17.3 Extensions

This section lists extensions to this tutorial that you may be interested in exploring.

- **Algorithm Tuning**. The configuration used in the tutorial was found with a little trial and error but was not optimized. Experiment with larger numbers of trees, different numbers of features and even different tree configurations to improve performance.
- **More Problems**. Apply the technique to other classification problems and even adapt it for regression with a new cost function and a new method for combining the predictions from trees.

17.4 Review

In this tutorial, you discovered how to implement the Random Forest algorithm from scratch. Specifically, you learned:

- The difference between Random Forest and Bagged Decision Trees.
- How to update the creation of decision trees to accommodate the Random Forest procedure.
- How to apply the Random Forest algorithm to a real-world predictive modeling problem.

17.4.1 Further Reading

- Section 8.2. Bagging, Random Forests, Boosting, page 316, *An Introduction to Statistical Learning*, 2014.
  http://amzn.to/2eeTyQX
- Section 8.5 Random Forests, page 198 and Section 14.4, Random Forests, page 386, *Applied Predictive Modeling*, 2013
  http://amzn.to/2e3lNXF

17.4.2 Next

In the next tutorial, you will discover how to implement and apply the Stacking algorithm for classification.

Chapter 18

Stacked Generalization

Ensemble methods are an excellent way to improve predictive performance on your machine learning problems. Stacked Generalization or stacking is an ensemble technique that uses a new model to learn how to best combine the predictions from two or more models trained on your dataset. In this tutorial, you will discover how to implement stacking from scratch in Python. After completing this tutorial, you will know:

- How to learn to combine the predictions from multiple models on a dataset.
- How to apply stacked generalization to a real-world predictive modeling problem.

Let's get started.

18.1 Description

This section provides a brief overview of the Stacked Generalization algorithm and the Sonar dataset used in this tutorial.

18.1.1 Stacked Generalization Algorithm

Stacked Generalization is an ensemble algorithm where a new model is trained to combine the predictions from two or more models already trained or your dataset. The predictions from the existing models or submodels are combined using a new model, and as such stacking is often referred to as blending, as the predictions from submodels are blended together.

It is typical to use a simple linear method to combine the predictions for submodels. Examples include: simple averaging called voting, a weighted sum using linear regression. and logistic regression. Models that have their predictions combined must have skill on the problem, but do not need to be the best possible models. This means that you do not need to tune the submodels intently, as long as the model shows some advantage over a baseline prediction.

It is important that submodels produce different predictions, so-called uncorrelated predictions. Stacking works best when the predictions that are combined are all skillful, but skillful in different ways. This may be achieved by using algorithms that use very different internal representations (trees compared to instances) and/or models trained on different representations or projections of the training data.

In this tutorial, we will look at taking two very different and untuned submodels and combining their predictions with a simple logistic regression algorithm.

18.1.2 Sonar Dataset

In this tutorial we will use the Sonar Dataset. This dataset involves the discrimination between mines and rocks. The baseline performance on the problem is approximately 53%. You can learn more about it in Appendix A, Section A.5. Download the dataset and save it into your current working directory with the filename `sonar.all-data.csv`.

18.2 Tutorial

This tutorial is broken down into 3 steps:

1. Submodels and Aggregator.
2. Combining Predictions.
3. Sonar Case Study.

These steps provide the foundation that you need to understand and implement stacking on your own predictive modeling problems.

18.2.1 Submodels and Aggregator

We are going to use two models as submodels for stacking and a linear model as the aggregator model.

This part is divided into 3 sections:

1. Submodel #1: k-Nearest Neighbors.
2. Submodel #2: Perceptron.
3. Aggregator Model: Logistic Regression.

Each model will be described in terms of the functions used to train the model and a function used to make predictions.

Submodel #1: k-Nearest Neighbors

The k-Nearest Neighbors algorithm or KNN uses the entire training dataset as the model. Therefore training the model involves retaining the training dataset. Below is a function named `knn_model()` that does just this.

```
# Prepare the KNN model
def knn_model(train):
  return train
```

Listing 18.1: Function To Train a KNN Model.

Making predictions involves finding the k most similar records in the training dataset and selecting the most common class values. The Euclidean distance function is used to calculate the similarity between new rows of data and rows in the training dataset.

Below are these helper functions that involve making predictions for a KNN model. The function `euclidean_distance()` calculates the distance between two rows of data, `get_neighbors()` locates all neighbors for in the training dataset for a new row of data and `knn_predict()` makes a prediction from the neighbors for a new row of data.

```
# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Locate neighbors for a new row
def get_neighbors(train, test_row, num_neighbors):
  distances = list()
  for train_row in train:
    dist = euclidean_distance(test_row, train_row)
    distances.append((train_row, dist))
  distances.sort(key=lambda tup: tup[1])
  neighbors = list()
  for i in range(num_neighbors):
    neighbors.append(distances[i][0])
  return neighbors

# Make a prediction with KNN
def knn_predict(model, test_row, num_neighbors=2):
  neighbors = get_neighbors(model, test_row, num_neighbors)
  output_values = [row[-1] for row in neighbors]
  prediction = max(set(output_values), key=output_values.count)
  return prediction
```

Listing 18.2: Functions To Make Predictions with a KNN Model.

You can see that the number of neighbors (k) is set to 2 as a default parameter on the `knn_predict()` function. This number was chosen with a little trial and error and was not tuned. Now that we have the building blocks for a KNN model, let's look at the Perceptron algorithm. For more on KNN see Chapter 13.

Submodel #2: Perceptron

The model for the Perceptron algorithm is a set of weights learned from the training data. In order to train the weights, many predictions need to be made on the training data in order to calculate error values. Therefore, both model training and prediction require a function for prediction.

Below are the helper functions for implementing the Perceptron algorithm. The `perceptron_model()` function trains the Perceptron model on the training dataset and `perceptron_predict()` is used to make a prediction for a row of data.

```
# Make a prediction with weights
def perceptron_predict(model, row):
  activation = model[0]
```

```
  for i in range(len(row)-1):
    activation += model[i + 1] * row[i]
  return 1.0 if activation >= 0.0 else 0.0

# Estimate Perceptron weights using stochastic gradient descent
def perceptron_model(train, l_rate=0.01, n_epoch=5000):
  weights = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    for row in train:
      prediction = perceptron_predict(weights, row)
      error = row[-1] - prediction
      weights[0] = weights[0] + l_rate * error
      for i in range(len(row)-1):
        weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
  return weights
```

Listing 18.3: Functions To Train and Make Predictions with a Perceptron Model.

The `perceptron_model()` function specifies both a learning rate and number of training epochs as default parameters. Again, these parameters were chosen with a little bit of trial and error, but were not tuned on the dataset. We now have implementations for both submodels; let's look at implementing the aggregator model. For more on the Perceptron algorithm see Chapter 10.

Aggregator Model: Logistic Regression

Like the Perceptron algorithm, Logistic Regression uses a set of weights, called coefficients, as the representation of the model. And like the Perceptron algorithm, the coefficients are learned by iteratively making predictions on the training data and updating them.

Below are the helper functions for implementing the logistic regression algorithm. The `logistic_regression_model()` function is used to train the coefficients on the training dataset and `logistic_regression_predict()` is used to make a prediction for a row of data.

```
# Make a prediction with coefficients
def logistic_regression_predict(model, row):
  yhat = model[0]
  for i in range(len(row)-1):
    yhat += model[i + 1] * row[i]
  return 1.0 / (1.0 + exp(-yhat))

# Estimate logistic regression coefficients using stochastic gradient descent
def logistic_regression_model(train, l_rate=0.01, n_epoch=5000):
  coef = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    for row in train:
      yhat = logistic_regression_predict(coef, row)
      error = row[-1] - yhat
      coef[0] = coef[0] + l_rate * error * yhat * (1.0 - yhat)
      for i in range(len(row)-1):
        coef[i + 1] = coef[i + 1] + l_rate * error * yhat * (1.0 - yhat) * row[i]
  return coef
```

Listing 18.4: Functions To Train and Make Predictions with a Logistic Regression Model.

The `logistic_regression_model()` defines a learning rate and number of epochs as default parameters, and as with the other algorithms, these parameters were found with a little trial and error and were not optimized. Now that we have implementations of submodels and the aggregator model, let's see how we can combine the predictions from multiple models. For more on Logistic Regression see Chapter 9.

18.2.2 Combining Predictions

For a machine learning algorithm, learning how to combine predictions is much the same as learning from a training dataset. A new training dataset can be constructed from the predictions of the submodels, as follows:

- Each row represents one row in the training dataset.
- The first column contains predictions for each row in the training dataset made by the first submodel, such as k-Nearest Neighbors.
- The second column contains predictions for each row in the training dataset made by the second submodel, such as the Perceptron algorithm.
- The third column contains the expected output value for the row in the training dataset.

Below is a contrived example of what a constructed stacking dataset may look like:

```
KNN, Per, Y
0,   0    0
1,   0    1
0,   1    0
1,   1    1
0,   1    0
```

Listing 18.5: Small Contrived Stacked Dataset.

A machine learning algorithm, such as logistic regression can then be trained on this new dataset. In essence, this new meta-algorithm learns how to best combine the prediction from multiple submodels. Below is a function named `to_stacked_row()` that implements this procedure for creating new rows for this stacked dataset.

The function takes a list of models as input; these are used to make predictions. The function also takes a list of functions as input, one function used to make a prediction for each model. Finally, a single row from the training dataset is included. A new row is constructed one column at a time. Predictions are calculated using each model and the row of training data. The expected output value from the training dataset row is then added as the last column to the row.

```
# Make predictions with submodels and construct a new stacked row
def to_stacked_row(models, predict_list, row):
  stacked_row = list()
  for i in range(len(models)):
    prediction = predict_list[i](models[i], row)
    stacked_row.append(prediction)
  stacked_row.append(row[-1])
  return stacked_row
```

Listing 18.6: Function To Make Predictions with Submodels and Build a Stacked Row.

On some predictive modeling problems, it is possible to get an even larger boost by training the aggregated model on both the training row and the predictions made by submodels. This improvement gives the aggregator model the context of all the data in the training row to help determine how and when to best combine the predictions of the submodels.

We can update our `to_stacked_row()` function to include this by aggregating the training row (minus the final column) and the stacked row as created above. Below is an updated version of the `to_stacked_row()` function that implements this improvement.

```
# Make predictions with sub-models and construct a new stacked row
def to_stacked_row(models, predict_list, row):
  stacked_row = list()
  for i in range(len(models)):
    prediction = predict_list[i](models[i], row)
    stacked_row.append(prediction)
  stacked_row.append(row[-1])
  return row[0:len(row)-1] + stacked_row
```

Listing 18.7: Function To Make Predictions with Submodels and Build a Stacked Row Including Input Data.

It is a good idea to try both approaches on your problem to see which works best. Now that we have all of the pieces for stacked generalization, we can apply the algorithm to a real-world problem.

18.2.3 Sonar Case Study

In this section, we will apply the Stacking algorithm to the Sonar dataset. The example assumes that a CSV copy of the dataset is in the current working directory with the filename `sonar.all-data.csv`.

The dataset is first loaded, the string values converted to numeric and the output column is converted from strings to the integer values of 0 to 1. This is achieved with helper functions `load_csv()`, `str_column_to_float()` and `str_column_to_int()` to load and prepare the dataset.

We will use k-fold cross-validation to estimate the performance of the learned model on unseen data. This means that we will construct and evaluate k models and estimate the performance as the mean model error. Classification accuracy will be used to evaluate the model. These behaviors are provided in the `cross_validation_split()`, `accuracy_metric()` and `evaluate_algorithm()` helper functions.

We will use the k-Nearest Neighbors, Perceptron and Logistic Regression algorithms implemented above. We will also use our technique for creating the new stacked dataset defined in the previous step. A new function name `stacking()` is developed. This function does 4 things:

1. It first trains a list of models (KNN and Perceptron).
2. It then uses the models to make predictions and create a new stacked dataset.
3. It then trains an aggregator model (logistic regression) on the stacked dataset.
4. It then uses the submodels and aggregator model to make predictions on the test dataset.

The complete example is listed below.

```
# Stacking on the sonar dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt
from math import exp

# Load a CSV file
def load_csv(filename):
  dataset = list()
  with open(filename, 'r') as file:
    csv_reader = reader(file)
    for row in csv_reader:
      if not row:
        continue
      dataset.append(row)
  return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
  for row in dataset:
    row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
  class_values = [row[column] for row in dataset]
  unique = set(class_values)
  lookup = dict()
  for i, value in enumerate(unique):
    lookup[value] = i
  for row in dataset:
    row[column] = lookup[row[column]]
  return lookup

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / n_folds)
  for i in range(n_folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
    dataset_split.append(fold)
  return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
  correct = 0
  for i in range(len(actual)):
    if actual[i] == predicted[i]:
      correct += 1
  return correct / float(len(actual)) * 100.0
```

```
# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
  folds = cross_validation_split(dataset, n_folds)
  scores = list()
  for fold in folds:
    train_set = list(folds)
    train_set.remove(fold)
    train_set = sum(train_set, [])
    test_set = list()
    for row in fold:
      row_copy = list(row)
      test_set.append(row_copy)
      row_copy[-1] = None
    predicted = algorithm(train_set, test_set, *args)
    actual = [row[-1] for row in fold]
    accuracy = accuracy_metric(actual, predicted)
    scores.append(accuracy)
  return scores

# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i])**2
  return sqrt(distance)

# Locate neighbors for a new row
def get_neighbors(train, test_row, num_neighbors):
  distances = list()
  for train_row in train:
    dist = euclidean_distance(test_row, train_row)
    distances.append((train_row, dist))
  distances.sort(key=lambda tup: tup[1])
  neighbors = list()
  for i in range(num_neighbors):
    neighbors.append(distances[i][0])
  return neighbors

# Make a prediction with kNN
def knn_predict(model, test_row, num_neighbors=2):
  neighbors = get_neighbors(model, test_row, num_neighbors)
  output_values = [row[-1] for row in neighbors]
  prediction = max(set(output_values), key=output_values.count)
  return prediction

# Prepare the kNN model
def knn_model(train):
  return train

# Make a prediction with weights
def perceptron_predict(model, row):
  activation = model[0]
  for i in range(len(row)-1):
    activation += model[i + 1] * row[i]
  return 1.0 if activation >= 0.0 else 0.0
```

```
# Estimate Perceptron weights using stochastic gradient descent
def perceptron_model(train, l_rate=0.01, n_epoch=5000):
  weights = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    for row in train:
      prediction = perceptron_predict(weights, row)
      error = row[-1] - prediction
      weights[0] = weights[0] + l_rate * error
      for i in range(len(row)-1):
        weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
  return weights

# Make a prediction with coefficients
def logistic_regression_predict(model, row):
  yhat = model[0]
  for i in range(len(row)-1):
    yhat += model[i + 1] * row[i]
  return 1.0 / (1.0 + exp(-yhat))

# Estimate logistic regression coefficients using stochastic gradient descent
def logistic_regression_model(train, l_rate=0.01, n_epoch=5000):
  coef = [0.0 for i in range(len(train[0]))]
  for epoch in range(n_epoch):
    for row in train:
      yhat = logistic_regression_predict(coef, row)
      error = row[-1] - yhat
      coef[0] = coef[0] + l_rate * error * yhat * (1.0 - yhat)
      for i in range(len(row)-1):
        coef[i + 1] = coef[i + 1] + l_rate * error * yhat * (1.0 - yhat) * row[i]
  return coef

# Make predictions with sub-models and construct a new stacked row
def to_stacked_row(models, predict_list, row):
  stacked_row = list()
  for i in range(len(models)):
    prediction = predict_list[i](models[i], row)
    stacked_row.append(prediction)
  stacked_row.append(row[-1])
  return row[0:len(row)-1] + stacked_row

# Stacked Generalization Algorithm
def stacking(train, test):
  model_list = [knn_model, perceptron_model]
  predict_list = [knn_predict, perceptron_predict]
  models = list()
  for i in range(len(model_list)):
    model = model_list[i](train)
    models.append(model)
  stacked_dataset = list()
  for row in train:
    stacked_row = to_stacked_row(models, predict_list, row)
    stacked_dataset.append(stacked_row)
  stacked_model = logistic_regression_model(stacked_dataset)
  predictions = list()
  for row in test:
    stacked_row = to_stacked_row(models, predict_list, row)
```

```
    stacked_dataset.append(stacked_row)
    prediction = logistic_regression_predict(stacked_model, stacked_row)
    prediction = round(prediction)
    predictions.append(prediction)
  return predictions

# Test stacking on the sonar dataset
seed(1)
# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)
# convert string attributes to integers
for i in range(len(dataset[0])-1):
  str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
n_folds = 3
scores = evaluate_algorithm(dataset, stacking, n_folds)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Listing 18.8: Example of Stacking on the Sonar Dataset.

A k value of 3 was used for cross-validation, giving each fold $\frac{208}{3} = 69.3$ or just under 70 records to be evaluated upon each iteration. I evaluated each submodel separately on the training dataset and achieved 74.879% for KNN and 72.464% for the Perceptron algorithm; both algorithms are better than the baseline accuracy of 53

Training logistic regression on the stacked dataset without the training rows for context achieves an accuracy of 75.845%. Including the training rows in the stacked dataset gives a further bump to 76.329%. Running the example prints the scores and mean of the scores for the final configuration.

```
Scores: [78.26086956521739, 78.26086956521739, 72.46376811594203]
Mean Accuracy: 76.329%
```

Listing 18.9: Example Output of Stacking on the Sonar Dataset.

18.3 Extensions

This section lists extensions to this tutorial that you may be interested in exploring.

- **Tune Algorithms**. The algorithms used for the submodels and the aggregate model in this tutorial were not tuned. Explore alternate configurations and see if you can further lift performance.
- **Prediction Correlations**. Stacking works better if the predictions of submodels are weakly correlated. Implement calculations to estimate the correlation between the predictions of submodels.
- **Different Submodels**. Implement more and different submodels to be combined using the stacking procedure.

- **Different Aggregating Model**. Experiment with simpler models (like averaging and voting) and more complex aggregation models to see if you can boost performance.
- **More Datasets**. Apply stacking to more datasets on the UCI Machine Learning Repository.

18.4 Review

In this tutorial, you discovered how to implement the stacking algorithm from scratch in Python. Specifically, you learned:

- How to combine the predictions from multiple models.
- How to apply stacking to a real-world predictive modeling problem.

18.4.1 Further Reading

- Section 7.5, Combining multiple models, page 315, *Data Mining: Practical Machine Learning Tools and Techniques*, second edition, 2005.
  http://amzn.to/2fj3SYY

18.4.2 Next

This concludes Part 4 on ensemble algorithms. Next in Part 5 you we will conclude the book.

Part V

Conclusions

Chapter 19

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come.

- You know how to load and prepare machine learning data ready for modeling.
- You know how to evaluate predictions made by machine learning models and estimate the skill of models.
- You know how to estimate a baseline of performance on a problem and compare multiple models on a problem.
- You know how to implement and apply the Linear regression, Logistic Regression and Perceptron linear machine learning algorithms
- You know how to implement and apply the CART, Naive Bayes, KNN, LVQ and Back-propagation nonlinear machine learning algorithms.
- You know how to implement and apply the Bagging, Random Forest and Stacking ensemble machine learning algorithms.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable skill of being able to implement and work through machine learning problems using Python. This is a platform that is used by a majority of working data scientist professionals. The sky is the limit.

I want to take a moment and sincerely thank you for letting me help you start your machine learning algorithm journey with Python. I hope you keep learning and have fun as you continue to master machine learning.

Chapter 20

Getting More Help

This is just the beginning of your journey with machine learning algorithms. As you start to work on new algorithms or expand your existing knowledge of algorithms you may need help. This chapter points out some of the best sources of help on machine learning algorithms you can find.

20.1 Machine Learning Books

This book contains everything that you need to get started with machine learning algorithms, but if you are like me, then you love books. There are many machine learning books available, but below is a small selection that I recommend as the next step.

- **An Introduction to Statistical Learning**. Excellent coverage of machine learning algorithms from a statistical perspective. Recommended as the next step.
  http://amzn.to/1pgirl0
- **Applied Predictive Modeling**. An excellent introduction to predictive modeling with coverage of a large number of algorithms. This book is better for breadth rather than depth on any one algorithm.
  http://amzn.to/1n5MSsq
- **Artificial Intelligence: A Modern Approach**. An excellent book on artificial intelligence in general, but the chapters on machine learning give a superb computer science perspective of the algorithms covered in this book.
  http://amzn.to/1TGk1rr

20.2 Forums and Q&A Websites

Question and answer sites are perhaps the best way to get answers to your specific technical questions about machine learning. You can search them for similar questions, browse through topics to learn about solutions to common problems and ask your own technical questions. The best Q&A sites I would recommend for your machine learning algorithm questions are:

- Cross Validated: http://stats.stackexchange.com/

- Stack Overflow: http://stackoverflow.com/questions/tagged/machine-learning
- Data Science: http://datascience.stackexchange.com/
- Reddit Machine Learning: http://www.reddit.com/r/machinelearning
- Quora Machine Learning: https://www.quora.com/topic/Machine-Learning/top_stories

Make heavy use of the search feature on these sites. Also note the list of *Related* questions in the right-hand navigation bar when viewing a specific question on Cross Validated. These are often relevant and useful.

20.3 Contact the Author

If you ever have any questions about machine learning algorithms or this book, please contact me directly. I will do my best to help.

Jason Brownlee
Jason@MachineLearningMastery.com

Part VI

Appendix

Appendix A

Standard Datasets

The key to getting good at applied machine learning is practicing on lots of different datasets. This is because each problem is different, requiring subtly different data preparation and modeling methods. In this appendix, you will discover a suite of standard machine learning datasets that you can use for practice. These are the datasets used throughout the book.

A.1 Overview

A.1.1 A structured Approach

Each dataset is summarized in a consistent way. This makes them easy to compare and navigate for you to practice a specific data preparation technique or modeling method. The aspects that you need to know about each dataset are:

- **Name**: How to refer to the dataset.
- **Problem Type**: Whether the problem is regression or classification.
- **Inputs and Outputs**: The numbers and known names of input and output features.
- **Performance**: Baseline performance for comparison using the Zero Rule algorithm, as well as best known performance (if known).
- **Sample**: A snapshot of the first 5 rows of raw data.
- **Links**: Where you can download the dataset and learn more.

A.1.2 Standard Datasets

Below is a list of the datasets we'll cover. Each dataset is small enough to fit into memory and review in a spreadsheet. All datasets are comprised of tabular data and no (explicitly) missing values.

1. Swedish Auto Insurance Dataset.
2. Wine Quality Dataset.

3. Pima Indians Diabetes Dataset.
4. Sonar Dataset.
5. Banknote Dataset.
6. Iris Flowers Dataset.
7. Abalone Dataset.
8. Ionosphere Dataset.
9. Wheat Seeds Dataset.

A.2 Swedish Auto Insurance Dataset

The Swedish Auto Insurance Dataset involves predicting the total payment for all claims in thousands of Swedish Kronor, given the total number of claims. It is a regression problem. It is comprised of 63 observations with 1 input variable and 1 output variable. The variable names are as follows:

1. Number of claims.
2. Total payment for all claims in thousands of Swedish Kronor.

The baseline performance of predicting the mean value is an RMSE of approximately 72.251 thousand Kronor. A sample of the first 5 rows is listed below.

```
108,392.5
19,46.2
13,15.7
124,422.2
40,119.4
```

Listing A.1: Sample of the Swedish Insurance Dataset.

Below is a scatter plot of the entire dataset.

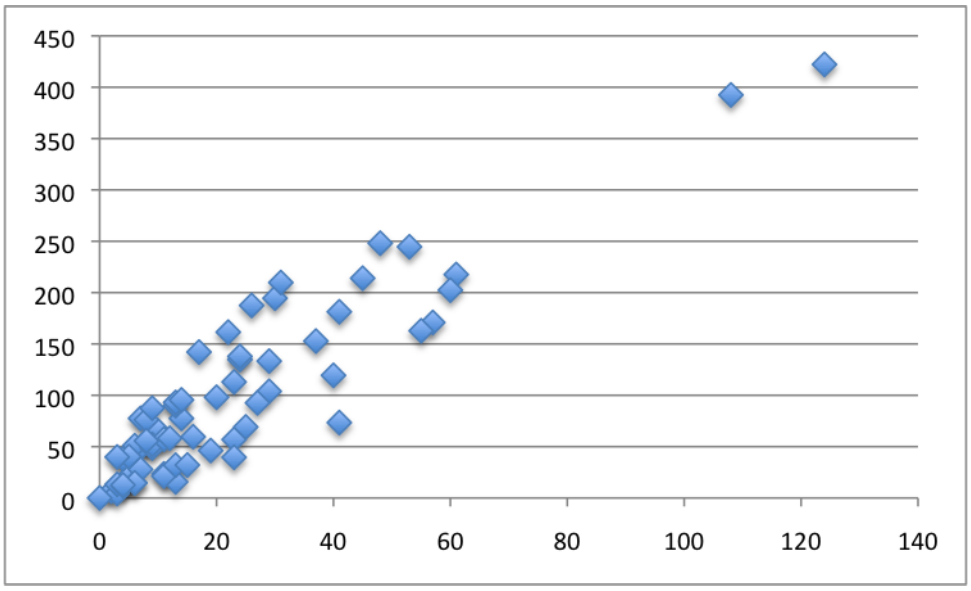

Figure A.1: Plot of the Swedish Insurance Dataset.

- Download: https://goo.gl/qPSNqY
- More Information: https://goo.gl/oMdwwR

A.3 Wine Quality Dataset

The Wine Quality Dataset involves predicting the quality of white wines on a scale given chemical measures of each wine. It is a multiclass classification problem, but could also be framed as a regression problem. The number of observations for each class is not balanced. There are 4,898 observations with 11 input variables and 1 output variable. The variable names are as follows:

1. Fixed acidity.
2. Volatile acidity.
3. Citric acid.
4. Residual sugar.
5. Chlorides.
6. Free sulfur dioxide.
7. Total sulfur dioxide.
8. Density.
9. pH.
10. Sulphates.

11. Alcohol.
12. Quality (score between 0 and 10).

The baseline performance of predicting the mean value is an RMSE of approximately 0.148 quality points. A sample of the first 5 rows is listed below.

```
7,0.27,0.36,20.7,0.045,45,170,1.001,3,0.45,8.8,6
6.3,0.3,0.34,1.6,0.049,14,132,0.994,3.3,0.49,9.5,6
8.1,0.28,0.4,6.9,0.05,30,97,0.9951,3.26,0.44,10.1,6
7.2,0.23,0.32,8.5,0.058,47,186,0.9956,3.19,0.4,9.9,6
7.2,0.23,0.32,8.5,0.058,47,186,0.9956,3.19,0.4,9.9,6
```

Listing A.2: Sample of the Wine Quality Dataset.

- Download: https://goo.gl/8jsvFK
- More Information: https://goo.gl/py6RCA

A.4 Pima Indians Diabetes Dataset

The Pima Indians Diabetes Dataset involves predicting the onset of diabetes within 5 years in Pima Indians given medical details. It is a binary (2-class) classification problem. The number of observations for each class is not balanced. There are 768 observations with 8 input variables and 1 output variable. Missing values are believed to be encoded with 0 values. The variable names are as follows:

1. Number of times pregnant.
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test.
3. Diastolic blood pressure (mm Hg).
4. Triceps skinfold thickness (mm).
5. 2-Hour serum insulin (mu U/ml).
6. Body mass index (weight in kg/(height in m)$\hat{2}$).
7. Diabetes pedigree function.
8. Age (years).
9. Class variable (0 or 1).

The baseline performance of predicting the most prevalent class is a classification accuracy of approximately 65%. Top results achieve a classification accuracy of approximately 77%. A sample of the first 5 rows is listed below.

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
```

Listing A.3: Sample of the Pima Indians Diabetes Dataset.

- Download: https://goo.gl/4EIcpc
- More Information: https://goo.gl/9sBPzQ
- Top Results: https://goo.gl/lyfjYr

A.5 Sonar Dataset

The Sonar Dataset involves the prediction of whether or not an object is a mine or a rock given the strength of sonar returns at different angles. It is a binary (2-class) classification problem. The number of observations for each class is not balanced. There are 208 observations with 60 input variables and 1 output variable. The variable names are as follows:

1. Sonar returns at different angles
2. ...
3. Class (`M` for mine and `R` for rock)

The baseline performance of predicting the most prevalent class is a classification accuracy of approximately 53%. Top results achieve a classification accuracy of approximately 88%. A sample of the first 5 rows is not provided as it is too large to fit on the page.

- Download: https://goo.gl/FXFWwF
- More Information: https://goo.gl/JMF2Iv
- Top Results: https://goo.gl/3wNAjv

A.6 Banknote Dataset

The Banknote Dataset involves predicting whether a given banknote is authentic given a number of measures taken from a photograph. It is a binary (2-class) classification problem. The number of observations for each class is not balanced. There are 1,372 observations with 4 input variables and 1 output variable. The variable names are as follows:

1. Variance of Wavelet Transformed image (continuous).
2. Skewness of Wavelet Transformed image (continuous).
3. Kurtosis of Wavelet Transformed image (continuous).

4. Entropy of image (continuous).
5. Class (0 for authentic, 1 for inauthentic).

The baseline performance of predicting the most prevalent class is a classification accuracy of approximately 50%. A sample of the first 5 rows is listed below.

```
3.6216,8.6661,-2.8073,-0.44699,0
4.5459,8.1674,-2.4586,-1.4621,0
3.866,-2.6383,1.9242,0.10645,0
3.4566,9.5228,-4.0112,-3.5944,0
0.32924,-4.4552,4.5718,-0.9888,0
4.3684,9.6718,-3.9606,-3.1625,0
```

Listing A.4: Sample of the Banknote Dataset.

- Download: https://goo.gl/rLPrHO
- More Information: https://goo.gl/CUWnce

A.7 Iris Flower Dataset

The Iris Flower Dataset involves predicting the flower species given measurements of iris flowers. It is a multiclass classification problem. The number of observations for each class is balanced. There are 150 observations with 4 input variables and 1 output variable. The variable names are as follows:

1. Sepal length in cm.
2. Sepal width in cm.
3. Petal length in cm.
4. Petal width in cm.
5. Class (`Iris Setosa`, `Iris Versicolour`, `Iris Virginica`).

The baseline performance of predicting the most prevalent class is a classification accuracy of approximately 26%. A sample of the first 5 rows is listed below.

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
```

Listing A.5: Sample of the Iris Flowers Dataset.

- Download: https://goo.gl/uqoqh7
- More Information: https://goo.gl/eA2Tff

A.8 Abalone Dataset

The Abalone Dataset involves predicting the age of abalone given objective measures of individuals. It is a multiclass classification problem, but can also be framed as a regression. The number of observations for each class is not balanced. There are 4,177 observations with 8 input variables and 1 output variable. The variable names are as follows:

1. Sex (M, F, I).
2. Length.
3. Diameter.
4. Height.
5. Whole weight.
6. Shucked weight.
7. Viscera weight.
8. Shell weight.
9. Rings.

The baseline performance of predicting the most prevalent class is a classification accuracy of approximately 16%. The baseline performance of predicting the mean value is an RMSE of approximately 3.2 rings. A sample of the first 5 rows is listed below.

```
M,0.455,0.365,0.095,0.514,0.2245,0.101,0.15,15
M,0.35,0.265,0.09,0.2255,0.0995,0.0485,0.07,7
F,0.53,0.42,0.135,0.677,0.2565,0.1415,0.21,9
M,0.44,0.365,0.125,0.516,0.2155,0.114,0.155,10
I,0.33,0.255,0.08,0.205,0.0895,0.0395,0.055,7
```

Listing A.6: Sample of the Abalone Dataset.

- Download: https://goo.gl/BDYgh5
- More Information: https://goo.gl/ieNz0R

A.9 Ionosphere Dataset

The Ionosphere Dataset requires the prediction of structure in the atmosphere given radar returns targeting free electrons in the ionosphere. It is a binary (2-class) classification problem. The number of observations for each class is not balanced. There are 351 observations with 34 input variables and 1 output variable. The variable names are as follows:

1. 17 pairs of radar return data.
2. ...

3. Class (`g` for good and `b` for bad).

The baseline performance of predicting the most prevalent class is a classification accuracy of approximately 64%. Top results achieve a classification accuracy of approximately 94%. A sample of the first 5 rows is not provided as it is too large to fit on the page.

- Download: https://goo.gl/fHX5Cx
- More Information: https://goo.gl/JJRyoX
- Top Results: https://goo.gl/U70K44

A.10 Wheat Seeds Dataset

The Wheat Seeds Dataset involves the prediction of species given measurements of seeds from different varieties of wheat. It is a binary (2-class) classification problem. The number of observations for each class is balanced. There are 210 observations with 7 input variables and 1 output variable. The variable names are as follows:

1. Area.
2. Perimeter.
3. Compactness
4. Length of kernel.
5. Width of kernel.
6. Asymmetry coefficient.
7. Length of kernel groove.
8. Class (1, 2, 3).

The baseline performance of predicting the most prevalent class is a classification accuracy of approximately 28%. A sample of the first 5 rows is listed below.

```
15.26,14.84,0.871,5.763,3.312,2.221,5.22,1
14.88,14.57,0.8811,5.554,3.333,1.018,4.956,1
14.29,14.09,0.905,5.291,3.337,2.699,4.825,1
13.84,13.94,0.8955,5.324,3.379,2.259,4.805,1
16.14,14.99,0.9034,5.658,3.562,1.355,5.175,1
```

Listing A.7: Sample of the Wheat Seeds Dataset.

- Download: https://goo.gl/na700k
- More Information: https://goo.gl/To7jT9

Appendix B

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.

B.1 Assignment

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

B.1.1 Strings

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

Listing B.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 B.2: Output of example working with strings.

B.1.2 Numbers

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

Listing B.3: Example of working with numbers.

Running the example prints:

```
123.1
10
```

Listing B.4: Output of example working with numbers.

B.1.3 Boolean

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

Listing B.5: Example of working with booleans.

Running the example prints:

```
(True, False)
```

Listing B.6: Output of example working with booleans.

B.1.4 Multiple Assignment

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

Listing B.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 B.8: Output of example working with multiple assignment.

B.1.5 No Value

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

Listing B.9: Example of working with no value.

Running the example prints:

```
None
```

Listing B.10: Output of example working with no value.

B.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.

B.2.1 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 B.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:

```
That is fast
```

Listing B.12: Output of example working with an If-Then-Else conditional.

B.2.2 For-Loop

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

Listing B.13: Example of working with a For-Loop.

Running the example prints:

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

Listing B.14: Output of example working with a For-Loop.

B.2.3 While-Loop

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

Listing B.15: Example of working with a While-Loop.

Running the example prints:

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

Listing B.16: Output of example working with a While-Loop.

B.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.

B.3.1 Tuple

Tuples are read-only collections of items.

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

Listing B.17: Example of working with a Tuple.

Running the example prints:

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

Listing B.18: Output of example working with a Tuple.

B.3.2 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 B.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 B.20: Output of example working with a List.

B.3.3 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 B.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 B.22: Output of example working with a Dictionary.

B.3.4 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 B.23: Example of working with a custom function.

Running the example prints:

```
4
```

Listing B.24: Output of example working with a custom function.