Master Machine Learning Algorithms

Discover How They Work and Implement Them From Scratch

Jason Brownlee

MACHINE
LEARNING
MASTERY

Jason Brownlee

Master Machine Learning Algorithms

Discover How They Work and Implement Them From Scratch

Master Machine Learning Algorithms

Edition, v1.1
http://MachineLearningMastery.com

Contents

Preface

Machine learning algorithms dominate applied machine learning. Because algorithms are such a big part of machine learning you must spend time to get familiar with them and really understand how they work. I wrote this book to help you start this journey.

You can describe machine learning algorithms using statistics, probability and linear algebra. The mathematical descriptions are very precise and often unambiguous. But this is not the only way to describe machine learning algorithms. Writing this book, I set out to describe machine learning algorithms for developers (like myself). As developers, we think in repeatable procedures. The best way to describe a machine learning algorithm for us is:

1. In terms of the representation used by the algorithm (the actual numbers stored in a file).
2. In terms of the abstract repeatable procedures used by the algorithm to learn a model from data and later to make predictions with the model.
3. With clear worked examples showing exactly how real numbers plug into the equations and what numbers to expect as output.

This book cuts through the mathematical talk around machine learning algorithms and shows you exactly how they work so that you can implement them yourself in a spreadsheet, in code with your favorite programming language or however you like. Once you possess this intimate knowledge, it will always be with you. You can implement the algorithms again and again. More importantly, you can translate the behavior of an algorithm back to the underlying procedure and really know what is going on and how to get the most from it.

This book is your tour of machine learning algorithms and I'm excited and honored to be your tour guide. Let's dive in.

Jason Brownlee
Melbourne, Australia
2016

Part I

Introduction

Chapter 1

Welcome

Welcome to *Master Machine Learning Algorithms*. This book will teach you 10 powerful machine learning algorithms from scratch.

Developers learn best with a mixture of algorithm descriptions and practical examples. This book was carefully designed to teach developers about machine learning algorithms. The structure includes both procedural descriptions of machine learning algorithms and step-by-step tutorials that show you exactly how to plug-in numbers into the various equations and exactly what numbers to expect on the other side. This book was written to pull back the curtain on machine learning algorithms for you so that nothing is hidden. After reading through the algorithm descriptions and tutorials in this book you will be able to:

1. Understand and explain how the top machine learning algorithms work.
2. Implement algorithm prototypes in your language or tool of choice.

This book is your guided tour to the internals of machine learning algorithms.

1.1 Audience

This book was written for developers. It does not assume a background in statistics, probability or linear algebra. If you know a little statistics and probability it can help as we will be talking about concepts such as means, standard deviations and Gaussian distributions. Don't worry if you are rusty or unsure, you will have the equations and worked examples to be able to fit it all together.

This book also does not assume a background in machine learning. It helps if you know the broad strokes, but the goal of this book is to teach you machine learning algorithms from scratch. Specifically, we are concerned with the type of machine learning where we build models in order to make predictions on new data called `predictive modeling`. Don't worry if this is new to you, we will get into the details of the types of machine learning algorithms soon.

Finally, this book does not assume that you know how to code or code well. You can follow along all of the examples in a spreadsheet. In fact you are strongly encouraged to follow along in a spreadsheet. If you're a programmer, you can also port the examples to your favorite programming language as part of the learning process.

1.2 Algorithm Descriptions

The description and presentation of algorithms in this book was carefully designed. Each algorithm is described in terms of three key properties:

1. The representation used by the algorithm in terms of the actual numbers and structure that could be stored in a file.
2. The procedure used by the algorithm to learn from training data.
3. The procedure used by the algorithm to make predictions given a learned model.

There will be very little mathematics used in this book. Those equations that are included were included because they are the very best way to get an idea across. Whenever possible, each equation will also be described textually and a worked example will be provided to show you exactly how to use it.

Finally, and most importantly, every algorithm described in this book will include a step-by-step tutorial. This is so that you can see exactly how the learning and prediction procedures work with real numbers. Each tutorial is provided in sufficient detail to allow you to follow along in a spreadsheet or in a programming language of your choice. This includes the raw input data and the output of each equation including all of the gory precision. Nothing is hidden or held back. You will see it all.

1.3 Book Structure

This book is broken into four parts:

1. Background on machine learning algorithms.
2. Linear machine learning algorithms.
3. Nonlinear machine learning algorithms.
4. Ensemble machine learning algorithms.

Let's take a closer look at each of the five parts:

1.3.1 Algorithms Background

This part will give you a foundation in machine learning algorithms. It will teach you how all machine learning algorithms are connected and attempt to solve the same underlying problem. This will give you the context to be able to understand any machine learning algorithm. You will discover:

- Terminology used in machine learning when describing data.
- The framework for understanding the problem solved by all machine learning algorithms.
- Important differences between parametric and nonparametric algorithms.

- Contrast between supervised, unsupervised and semi-supervised machine learning problems.
- Error introduced by bias and variance the trade-off between these concerns.
- Battle in applied machine learning to overcome the problem of overfitting data.

1.3.2 Linear Algorithms

This part will ease you into machine learning algorithms by starting with simpler linear algorithms. These may be simple algorithms but they are also the important foundation for understanding the more powerful techniques. You will discover the following linear algorithms:

- Gradient descent optimization procedure that may be used in the heart of many machine learning algorithms.
- Linear regression for predicting real values with two tutorials to make sure it really sinks in.
- Logistic regression for classification on problems with two categories.
- Linear discriminant analysis for classification on problems with more than two categories.

1.3.3 Nonlinear Algorithms

This part will introduce more powerful nonlinear machine learning algorithms that build upon the linear algorithms. These are techniques that make fewer assumptions about your problem and are able to learn a large variety of problem types. But this power needs to be used carefully because they can learn too well and overfit your training data. You will discover the following nonlinear algorithms:

- Classification and regression trees the staple decision tree algorithm.
- Naive Bayes using probability for classification with two tutorials showing you useful ways this technique can be used.
- K-Nearest Neighbors that do not require any model at all other than your dataset.
- Learning Vector Quantization which extends K-Nearest Neighbors by learning to compress your training dataset down in size.
- Support vector machines which are perhaps one of the most popular and powerful out of the box algorithms.

1.3.4 Ensemble Algorithms

A powerful and more advanced type of machine learning algorithm are ensemble algorithms. These are techniques that combine the predictions from multiple models in order to provide more accurate predictions. In this part you will be introduced to two of the most used ensemble methods:

- Bagging and Random Forests which are among the most powerful algorithms available.
- Boosting ensemble and the AdaBoost algorithm that successively corrects the predictions of weaker models.

1.4 What This Book is Not

- This is not a machine learning textbook. We will not be going into the theory behind why things work or the derivations of equations. This book is about teaching how machine learning algorithms work, not why they work.
- This is not a machine learning programming book. We will not be designing machine learning algorithms for production or operational use. All examples in this book are for demonstration purposes only.

1.5 How To Best Use this Book

This book is intended to be read linearly from one end to the other. Reading this book is not enough. To make the concepts stick and actually learn machine learning algorithms you need to work through the tutorials. You will get the most out of this book if you open a spreadsheet along side the book and work through each tutorial.

Working through the tutorials will give context to the representation, learning and prediction procedures described for each algorithm. From there, you can translate the ideas to your own programs and to your usage of these algorithms in practice.

I recommend completing one chapter per day, ideally in the evening at the computer so you can immediately try out what you have learned. I have intentionally repeated key equations and descriptions to allow you to pick up where you left off from day to day.

1.6 Summary

It is time to finally understand machine learning. This book is your ticket to machine learning algorithms. Next up you will build a foundation to understand the underlying problem that all machine learning algorithms are trying to solve.

Part II

Background

Chapter 2

How To Talk About Data in Machine Learning

Data plays a big part in machine learning. It is important to understand and use the right terminology when talking about data. In this chapter you will discover exactly how to describe and talk about data in machine learning. After reading this chapter you will know:

- Standard data terminology used in general when talking about spreadsheets of data.
- Data terminology used in statistics and the statistical view of machine learning.
- Data terminology used in the computer science perspective of machine learning.

This will greatly help you with understanding machine learning algorithms in general. Let's get started.

2.1 Data As you Know It

How do you think about data? Think of a spreadsheet. You have columns, rows, and cells.

	A	B	C	D
1		Column 1	Column 2	Column 3
2	Row 1	2.2	2.3	1
3	Row 2	2.3	2.6	0
4	Row 3	2.1	2	1

Figure 2.1: Data Terminology in Data in Machine Learning.

- **Column**: A column describes data of a single type. For example, you could have a column of weights or heights or prices. All the data in one column will have the same scale and have meaning relative to each other.
- **Row**: A row describes a single entity or observation and the columns describe properties about that entity or observation. The more rows you have, the more examples from the problem domain that you have.

- **Cell**: A cell is a single value in a row and column. It may be a real value (1.5) an integer (2) or a category (*red*).

This is how you probably think about data, columns, rows and cells. Generally, we can call this type of data: tabular data. This form of data is easy to work with in machine learning. There are different flavors of machine learning that give different perspectives on the field. For example there is a the statistical perspective and the computer science perspective. Next we will look at the different terms used to refer to data as you know it.

2.2 Statistical Learning Perspective

The statistical perspective frames data in the context of a hypothetical function (f) that the machine learning algorithm is trying to learn. That is, given some input variables (*input*), what is the predicted output variable (*output*).

$$Output = f(Input) \tag{2.1}$$

Those columns that are the inputs are referred to as input variables. Whereas the column of data that you may not always have and that you would like to predict for new input data in the future is called the output variable. It is also called the response variable.

$$OutputVariable = f(InputVariables) \tag{2.2}$$

	A	B	C
1	X1	X2	Y
2	2.2	2.3	1
3	2.3	2.6	0
4	2.1	2	1
5			

Figure 2.2: Statistical Learning Perspective of Data in Machine Learning.

Typically, you have more than one input variable. In this case the group of input variables are referred to as the input vector.

$$OutputVariable = f(InputVector) \tag{2.3}$$

If you have done a little statistics in your past you may know of another more traditional terminology. For example, a statistics text may talk about the input variables as independent variables and the output variable as the dependent variable. This is because in the phrasing of the prediction problem the output is dependent or a function of the input or independent variables.

$$DependentVariable = f(IndependentVariables) \tag{2.4}$$

The data is described using a short hand in equations and descriptions of machine learning algorithms. The standard shorthand used in the statistical perspective is to refer to the input variables as capital x (X) and the output variables as capital y (Y).

$$Y = f(X) \tag{2.5}$$

When you have multiple input variables they may be dereferenced with an integer to indicate their ordering in the input vector, for example $X1$, $X2$ and $X3$ for data in the first three columns.

2.3 Computer Science Perspective

There is a lot of overlap in the computer science terminology for data with the statistical perspective. We will look at the key differences. A row often describes an entity (like a person) or an observation about an entity. As such, the columns for a row are often referred to as attributes of the observation. When modeling a problem and making predictions, we may refer to input attributes and output attributes.

$$OutputAttribute = Program(InputAttributes) \tag{2.6}$$

	A	B	C	D
1		Attribute 1	Attribute 2	Output Attribute
2	Instance 1	2.2	2.3	1
3	Instance 2	2.3	2.6	0
4	Instance 3	2.1	2	1
5				

Figure 2.3: Computer Science Perspective of Data in Machine Learning.

Another name for columns is features, used for the same reason as attribute, where a feature describes some property of the observation. This is more common when working with data where features must be extracted from the raw data in order to construct an observation. Examples of this include analog data like images, audio and video.

$$Output = Program(InputFeatures) \tag{2.7}$$

Another computer science phrasing is that for a row of data or an observation as an instance. This is used because a row may be considered a single example or single instance of data observed or generated by the problem domain.

$$Prediction = Program(Instance) \tag{2.8}$$

2.4 Models and Algorithms

There is one final note of clarification that is important and that is between algorithms and models. This can be confusing as both algorithm and model can be used interchangeably. A

perspective that I like is to think of the model as the specific representation learned from data and the algorithm as the process for learning it.

$$Model = Algorithm(Data) \tag{2.9}$$

For example, a decision tree or a set of coefficients are a model and the C5.0 and Least Squares Linear Regression are algorithms to learn those respective models.

2.5 Summary

In this chapter you discovered the key terminology used to describe data in machine learning.

- You started with the standard understanding of tabular data as seen in a spreadsheet as columns, rows and cells.
- You learned the statistical terms of input and output variables that may be denoted as X and sY respectively.
- You learned the computer science terms of attribute, feature and instance.
- Finally you learned that talk of models and algorithms can be separated into learned representation and process for learning.

You now know how to talk about data in machine learning. In the next chapter you will discover the paradigm that underlies all machine learning algorithms.

Chapter 3

Algorithms Learn a Mapping From Input to Output

How do machine learning algorithms work? There is a common principle that underlies all supervised machine learning algorithms for predictive modeling. In this chapter you will discover how machine learning algorithms actually work by understanding the common principle that underlies all algorithms. After reading this chapter you will know:

- The mapping problem that all supervised machine learning algorithms aim to solve.
- That the subfield of machine learning focused on making predictions is called predictive modeling.
- That different machine learning algorithms represent different strategies for learning the mapping function.

Let's get started.

3.1 Learning a Function

Machine learning algorithms are described as learning a target function (f) that best maps input variables (X) to an output variable (Y).

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

This is a general learning task where we would like to make predictions in the future (Y) given new examples of input variables (X). We don't know what the function (f) looks like or it's form. If we did, we would use it directly and we would not need to learn it from data using machine learning algorithms. It is harder than you think. There is also error (e) that is independent of the input data (X).

$$Y = f(X) + e \tag{3.2}$$

This error might be error such as not having enough attributes to sufficiently characterize the best mapping from X to Y. This error is called irreducible error because no matter how good we get at estimating the target function (f), we cannot reduce this error. This is to say, that the problem of learning a function from data is a difficult problem and this is the reason why the field of machine learning and machine learning algorithms exist.

3.2 Learning a Function To Make Predictions

The most common type of machine learning is to learn the mapping $Y = f(X)$ to make predictions of Y for new X. This is called predictive modeling or predictive analytics and our goal is to make the most accurate predictions possible.

As such, we are not really interested in the shape and form of the function (f) that we are learning, only that it makes accurate predictions. We could learn the mapping of $Y = f(X)$ to learn more about the relationship in the data and this is called statistical inference. If this were the goal, we would use simpler methods and value understanding the learned model and form of (f) above making accurate predictions.

When we learn a function (f) we are estimating its form from the data that we have available. As such, this estimate will have error. It will not be a perfect estimate for the underlying hypothetical best mapping from Y given X. Much time in applied machine learning is spent attempting to improve the estimate of the underlying function and in term improve the performance of the predictions made by the model.

3.3 Techniques For Learning a Function

Machine learning algorithms are techniques for estimating the target function (f) to predict the output variable (Y) given input variables (X). Different representations make different assumptions about the form of the function being learned, such as whether it is linear or nonlinear.

Different machine learning algorithms make different assumptions about the shape and structure of the function and how best to optimize a representation to approximate it. This is why it is so important to try a suite of different algorithms on a machine learning problem, because we cannot know before hand which approach will be best at estimating the structure of the underlying function we are trying to approximate.

3.4 Summary

In this chapter you discovered the underlying principle that explains the objective of all machine learning algorithms for predictive modeling.

- You learned that machine learning algorithms work to estimate the mapping function (f) of output variables (Y) given input variables (X), or $Y = f(X)$.
- You also learned that different machine learning algorithms make different assumptions about the form of the underlying function.
- That when we don't know much about the form of the target function we must try a suite of different algorithms to see what works best.

You now know the principle that underlies all machine learning algorithms. In the next chapter you will discover the two main classes of machine learning algorithms: parametric and nonparametric algorithms.

Chapter 4

Parametric and Nonparametric Machine Learning Algorithms

What is a parametric machine learning algorithm and how is it different from a nonparametric machine learning algorithm? In this chapter you will discover the difference between parametric and nonparametric machine learning algorithms. After reading this chapter you will know:

- That parametric machine learning algorithms simply the mapping to a know functional form.
- That nonparametric algorithms can learn any mapping from inputs to outputs.
- That all algorithms can be organized into parametric or nonparametric groups.

Let's get started.

4.1 Parametric Machine Learning Algorithms

Assumptions can greatly simplify the learning process, but can also limit what can be learned. Algorithms that simplify the function to a known form are called parametric machine learning algorithms.

> A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a parametric model. No matter how much data you throw at a parametric model, it won't change its mind about how many parameters it needs.

– Artificial Intelligence: A Modern Approach, page 737

The algorithms involve two steps:

1. Select a form for the function.
2. Learn the coefficients for the function from the training data.

An easy to understand functional form for the mapping function is a line, as is used in linear regression:

$$B0 + B1 \times X1 + B2 \times X2 = 0 \tag{4.1}$$

Where $B0$, $B1$ and $B2$ are the coefficients of the line that control the intercept and slope, and $X1$ and $X2$ are two input variables. Assuming the functional form of a line greatly simplifies the learning process. Now, all we need to do is estimate the coefficients of the line equation and we have a predictive model for the problem.

Often the assumed functional form is a linear combination of the input variables and as such parametric machine learning algorithms are often also called *linear machine learning algorithms*. The problem is, the actual unknown underlying function may not be a linear function like a line. It could be almost a line and require some minor transformation of the input data to work right. Or it could be nothing like a line in which case the assumption is wrong and the approach will produce poor results.

Some more examples of parametric machine learning algorithms include:

- Logistic Regression
- Linear Discriminant Analysis
- Perceptron

Benefits of Parametric Machine Learning Algorithms:

- **Simpler**: These methods are easier to understand and interpret results.
- **Speed**: Parametric models are very fast to learn from data.
- **Less Data**: They do not require as much training data and can work well even if the fit to the data is not perfect.

Limitations of Parametric Machine Learning Algorithms:

- **Constrained**: By choosing a functional form these methods are highly constrained to the specified form.
- **Limited Complexity**: The methods are more suited to simpler problems.
- **Poor Fit**: In practice the methods are unlikely to match the underlying mapping function.

4.2 Nonparametric Machine Learning Algorithms

Algorithms that do not make strong assumptions about the form of the mapping function are called nonparametric machine learning algorithms. By not making assumptions, they are free to learn any functional form from the training data.

> Nonparametric methods are good when you have a lot of data and no prior knowledge, and when you don't want to worry too much about choosing just the right features.

– Artificial Intelligence: A Modern Approach, page 757

Nonparametric methods seek to best fit the training data in constructing the mapping function, whilst maintaining some ability to generalize to unseen data. As such, they are able to fit a large number of functional forms. An easy to understand nonparametric model is the k-nearest neighbors algorithm that makes predictions based on the k most similar training patterns for a new data instance. The method does not assume anything about the form of the mapping function other than patterns that are close are likely have a similar output variable. Some more examples of popular nonparametric machine learning algorithms are:

- Decision Trees like CART and C4.5
- Naive Bayes
- Support Vector Machines
- Neural Networks

Benefits of Nonparametric Machine Learning Algorithms:

- **Flexibility**: Capable of fitting a large number of functional forms.
- **Power**: No assumptions (or weak assumptions) about the underlying function.
- **Performance**: Can result in higher performance models for prediction.

Limitations of Nonparametric Machine Learning Algorithms:

- **More data**: Require a lot more training data to estimate the mapping function.
- **Slower**: A lot slower to train as they often have far more parameters to train.
- **Overfitting**: More of a risk to overfit the training data and it is harder to explain why specific predictions are made.

4.3 Summary

In this chapter you have discovered the difference between parametric and nonparametric machine learning algorithms.

- You learned that parametric methods make large assumptions about the mapping of the input variables to the output variable and in turn are faster to train, require less data but may not be as powerful.
- You also learned that nonparametric methods make few or no assumptions about the target function and in turn require a lot more data, are slower to train and have a higher model complexity but can result in more powerful models.

You now know the difference between parametric and nonparametric machine learning algorithms. In the next chapter you will discover another way to group machine learning algorithms by the way they learn: supervised and unsupervised learning.

Chapter 5

Supervised, Unsupervised and Semi-Supervised Learning

What is supervised machine learning and how does it relate to unsupervised machine learning? In this chapter you will discover supervised learning, unsupervised learning and semis-supervised learning. After reading this chapter you will know:

- About the classification and regression supervised learning problems.
- About the clustering and association unsupervised learning problems.
- Example algorithms used for supervised and unsupervised problems.

A problem that sits in between supervised and unsupervised learning called semi-supervised learning. Let's get started.

5.1 Supervised Machine Learning

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

$$Y = f(X) \tag{5.1}$$

The goal is to approximate the mapping function so well that when you have new input data (X) that you can predict the output variables (Y) for that data. It is called supervised learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. We know the correct answers, the algorithm iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the algorithm achieves an acceptable level of performance. Supervised learning problems can be further grouped into regression and classification problems.

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

- Some common types of problems built on top of classification and regression include recommendation and time series prediction respectively.

Some popular examples of supervised machine learning algorithms are:

- Linear regression for regression problems.
- Random forest for classification and regression problems.
- Support vector machines for classification problems.

5.2 Unsupervised Machine Learning

Unsupervised learning is where you you only have input data (X) and no corresponding output variables. The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn more about the data.

These are called unsupervised learning because unlike supervised learning above there is no correct answers and there is no teacher. Algorithms are left to their own devises to discover and present the interesting structure in the data. Unsupervised learning problems can be further grouped into clustering and association problems.

- **Clustering**: A clustering problem is where you want to discover the inherent groupings in the data, such as grouping customers by purchasing behavior.
- **Association**: An association rule learning problem is where you want to discover rules that describe large portions of your data, such as people that buy A also tend to buy B.

Some popular examples of unsupervised learning algorithms are:

- k-means for clustering problems.
- Apriori algorithm for association rule learning problems.

5.3 Semi-Supervised Machine Learning

Problems where you have a large amount of input data (X) and only some of the data is labeled (Y) are called semi-supervised learning problems. These problems sit in between both supervised and unsupervised learning. A good example is a photo archive where only some of the images are labeled, (e.g. dog, cat, person) and the majority are unlabeled. Many real world machine learning problems fall into this area. This is because it can be expensive or time consuming to label data as it may require access to domain experts. Whereas unlabeled data is cheap and easy to collect and store.

You can use unsupervised learning techniques to discover and learn the structure in the input variables. You can also use supervised learning techniques to make best guess predictions for the unlabeled data, feed that data back into the supervised learning algorithm as training data and use the model to make predictions on new unseen data.

5.4 Summary

In this chapter you learned the difference between supervised, unsupervised and semi-supervised learning. You now know that:

- Supervised: All data is labeled and the algorithms learn to predict the output from the input data.
- Unsupervised: All data is unlabeled and the algorithms learn to inherent structure from the input data.
- Semi-supervised: Some data is labeled but most of it is unlabeled and a mixture of supervised and unsupervised techniques can be used.

You now know that you can group machine learning algorithms as supervised, unsupervised and semi-supervised learning. In the next chapter you will discover the two biggest sources of error when learning from data, namely bias and variance and the tension between these two concerns.

Chapter 6

The Bias-Variance Trade-Off

Supervised machine learning algorithms can best be understood through the lens of the bias-variance trade-off. In this chapter you will discover the Bias-Variance Trade-Off and how to use it to better understand machine learning algorithms and get better performance on your data. After reading this chapter you will know.

- That all learning error can be broken down into bias or variance error.
- That bias refers to the simplifying assumptions made by the algorithm to make the problem easier to solve.
- That variance refers to the sensitivity of a model to changes to the training data.
- That all of applied machine learning for predictive model is best understood through the framework of bias and variance.

Let's get started.

6.1 Overview of Bias and Variance

In supervised machine learning an algorithm learns a model from training data. The goal of any supervised machine learning algorithm is to best estimate the mapping function (f) for the output variable (Y) given the input data (X). The mapping function is often called the target function because it is the function that a given supervised machine learning algorithm aims to approximate. The prediction error for any machine learning algorithm can be broken down into three parts:

- Bias Error
- Variance Error
- Irreducible Error

The irreducible error cannot be reduced regardless of what algorithm is used. It is the error introduced from the chosen framing of the problem and may be caused by factors like unknown variables that influence the mapping of the input variables to the output variable. In this chapter we will focus on the two parts we can influence with our machine learning algorithms. The bias error and the variance error.

6.2 Bias Error

Bias are the simplifying assumptions made by a model to make the target function easier to learn. Generally parametric algorithms have a high bias making them fast to learn and easier to understand but generally less flexible. In turn they are have lower predictive performance on complex problems that fail to meet the simplifying assumptions of the algorithms bias.

- **Low Bias**: Suggests more assumptions about the form of the target function.
- **High-Bias**: Suggests less assumptions about the form of the target function.

Examples of low-bias machine learning algorithms include: Decision Trees, k-Nearest Neighbors and Support Vector Machines. Examples of high-bias machine learning algorithms include: Linear Regression, Linear Discriminant Analysis and Logistic Regression.

6.3 Variance Error

Variance is the amount that the estimate of the target function will change if different training data was used. The target function is estimated from the training data by a machine learning algorithm, so we should expect the algorithm to have some variance. Ideally, it should not change too much from one training dataset to the next, meaning that the algorithm is good at picking out the hidden underlying mapping between the inputs and the output variables. Machine learning algorithms that have a high variance are strongly influenced by the specifics of the training data. This means that the specifics of the training have influences the number and types of parameters used to characterize the mapping function.

- **Low Variance**: Suggests small changes to the estimate of the target function with changes to the training dataset.
- **High Variance**: Suggests large changes to the estimate of the target function with changes to the training dataset.

Generally nonparametric machine learning algorithms that have a lot of flexibility have a high bias. For example decision trees have a high bias, that is even higher if the trees are not pruned before use. Examples of low-variance machine learning algorithms include: Linear Regression, Linear Discriminant Analysis and Logistic Regression. Examples of high-variance machine learning algorithms include: Decision Trees, k-Nearest Neighbors and Support Vector Machines.

6.4 Bias-Variance Trade-Off

The goal of any supervised machine learning algorithm is to achieve low bias and low variance. In turn the algorithm should achieve good prediction performance. You can see a general trend in the examples above:

- Parametric or linear machine learning algorithms often have a high bias but a low variance.

- Nonparametric or nonlinear machine learning algorithms often have a low bias but a high variance.

The parameterization of machine learning algorithms is often a battle to balance out bias and variance. Below are two examples of configuring the bias-variance trade-off for specific algorithms:

- The k-nearest neighbors algorithm has low bias and high variance, but the trade-off can be changed by increasing the value of k which increases the number of neighbors that contribute t the prediction and in turn increases the bias of the model.
- The support vector machine algorithm has low bias and high variance, but the trade-off can be changed by increasing the C parameter that influences the number of violations of the margin allowed in the training data which increases the bias but decreases the variance.

There is no escaping the relationship between bias and variance in machine learning.

- Increasing the bias will decrease the variance.
- Increasing the variance will decrease the bias.

There is a trade-off at play between these two concerns and the algorithms you choose and the way you choose to configure them are finding different balances in this trade-off for your problem. In reality we cannot calculate the real bias and variance error terms because we do not know the actual underlying target function. Nevertheless, as a framework, bias and variance provide the tools to understand the behavior of machine learning algorithms in the pursuit of predictive performance.

6.5 Summary

In this chapter you discovered bias, variance and the bias-variance trade-off for machine learning algorithms. You now know that:

- Bias is the simplifying assumptions made by the model to make the target function easier to approximate.
- Variance is the amount that the estimate of the target function will change given different training data.
- Trade-off is tension between the error introduced by the bias and the variance.

You know know about bias and variance, the two sources of error when learning form data. In the next chapter you will discover the practical implications of bias and variance when applying machine learning to problems, namely overfitting and underfitting.

Chapter 7

Overfitting and Underfitting

The cause of poor performance in machine learning is either overfitting or underfitting the data. In this chapter you will discover the concept of generalization in machine learning and the problems of overfitting and underfitting that go along with it. After reading this chapter you will know:

- That overfitting refers to learning the training data too well at the expense of not generalizing well to new data.
- That underfitting refers to failing to learn the problem from the training data sufficiently.
- That overfitting is the most common problem in practice and can be addressed by using resampling methods and a held-back verification dataset.

Let's get started.

7.1 Generalization in Machine Learning

In machine learning we describe the learning of the target function from training data as inductive learning. Induction refers to learning general concepts from specific examples which is exactly the problem that supervised machine learning problems aim to solve. This is different from deduction that is the other way around and seeks to learn specific concepts from general rules.

Generalization refers to how well the concepts learned by a machine learning model apply to specific examples not seen by the model when it was learning. The goal of a good machine learning model is to generalize well from the training data to any data from the problem domain. This allows us to make predictions in the future on data the model has never seen. There is a terminology used in machine learning when we talk about how well a machine learning model learns and generalizes to new data, namely overfitting and underfitting. Overfitting and underfitting are the two biggest causes for poor performance of machine learning algorithms.

7.2 Statistical Fit

In statistics a fit refers to how well you approximate a target function. This is good terminology to use in machine learning, because supervised machine learning algorithms seek to approximate the unknown underlying mapping function for the output variables given the input variables.

Statistics often describe the goodness of fit which refers to measures used to estimate how well the approximation of the function matches the target function. Some of these methods are useful in machine learning (e.g. calculating the residual errors), but some of these techniques assume we know the form of the target function we are approximating, which is not the case in machine learning. If we knew the form of the target function, we would use it directly to make predictions, rather than trying to learn an approximation from samples of noisy training data.

7.3 Overfitting in Machine Learning

Overfitting refers to a model that models the training data too well. Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance on the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize.

Overfitting is more likely with nonparametric and nonlinear models that have more flexibility when learning a target function. As such, many nonparametric machine learning algorithms also include parameters or techniques to limit and constrain how much detail the model learns. For example, decision trees are a nonparametric machine learning algorithm that is very flexible and is subject to overfitting training data. This problem can be addressed by pruning a tree after it has learned in order to remove some of the detail it has picked up.

7.4 Underfitting in Machine Learning

Underfitting refers to a model that can neither model the training data not generalize to new data. An underfit machine learning model is not a suitable model and will be obvious as it will have poor performance on the training data. Underfitting is often not discussed as it is easy to detect given a good performance metric. The remedy is to move on and try alternate machine learning algorithms. Nevertheless, it does provide good contrast to the problem of concept of overfitting.

7.5 A Good Fit in Machine Learning

Ideally, you want to select a model at the sweet spot between underfitting and overfitting. This is the goal, but is very difficult to do in practice.

To understand this goal, we can look at the performance of a machine learning algorithm over time as it is learning a training data. We can plot both the skill on the training data an the skill on a test dataset we have held back from the training process. Over time, as the algorithm learns, the error for the model on the training data goes down and so does the error on the test dataset. If we train for too long, the performance on the training dataset may continue to decrease because the model is overfitting and learning the irrelevant detail and noise in the training dataset. At the same time the error for the test set starts to rise again as the model's ability to generalize decreases.

The sweet spot is the point just before the error on the test dataset starts to increase where the model has good skill on both the training dataset and the unseen test dataset. You can perform this experiment with your favorite machine learning algorithms. This is often not

useful technique in practice, because by choosing the stopping point for training using the skill on the test dataset it means that the testset is no longer *unseen* or a standalone objective measure. Some knowledge (a lot of useful knowledge) about that data has leaked into the training procedure. There are two additional techniques you can use to help find the sweet spot in practice: resampling methods and a validation dataset.

7.6 How To Limit Overfitting

Both overfitting and underfitting can lead to poor model performance. But by far the most common problem in applied machine learning is overfitting. Overfitting is such a problem because the evaluation of machine learning algorithms on training data is different from the evaluation we actually care the most about, namely how well the algorithm performs on unseen data. There are two important techniques that you can use when evaluating machine learning algorithms to limit overfitting:

1. Use a resampling technique to estimate model accuracy.
2. Hold back a validation dataset.

The most popular resampling technique is k-fold cross validation. It allows you to train and test your model k-times on different subsets of training data and build up an estimate of the performance of a machine learning model on unseen data.

A validation dataset is simply a subset of your training data that you hold back from your machine learning algorithms until the very end of your project. After you have selected and tuned your machine learning algorithms on your training dataset you can evaluate the learned models on the validation dataset to get a final objective idea of how the models might perform on unseen data. Using cross validation is a gold standard in applied machine learning for estimating model accuracy on unseen data. If you have the data, using a validation dataset is also an excellent practice.

7.7 Summary

In this chapter you discovered that machine learning is solving problems by the method of induction. You learned that generalization is a description of how well the concepts learned by a model apply to new data. Finally you learned about the terminology of generalization in machine learning of overfitting and underfitting:

- Overfitting: Good performance on the training data, poor generalization to other data.
- Underfitting: Poor performance on the training data and poor generalization to other data.

You now know about the risks of overfitting and underfitting data. This chapter draws your background on machine learning algorithms to an end. In the next part you will start learning about machine learning algorithms, starting with linear algorithms.

Part III

Linear Algorithms

Chapter 8

Crash-Course in Spreadsheet Math

The tutorials in this book were designed for you to complete using a spreadsheet program. This chapter gives you a quick crash course in some mathematical functions you should know about in order to complete the tutorials in this book. After completing this chapter you will know:

- How to perform basic arithmetic operations in a spreadsheet.
- How to use statistical functions to summarize data.
- How to create random numbers to use as test data.

It does not matter which spreadsheet program you use to complete the tutorials. All functions used are generic across spreadsheet programs. Some recommended programs that you can use include:

- Microsoft Office with Excel.
- LibreOffice with Calc.
- Numbers on the Mac.
- Google Sheets in Google Drive.

If you are already proficient with using a spreadsheet program, you can skip this chapter. Alternatively, you do not need to use a spreadsheet and could implement the tutorials directly in your programming language of choice. Let's get started.

8.1 Arithmetic

Let's start with some basic spreadsheet navigation and arithmetic.

- A cell can evaluate an expression using the equals (`=`) and then the expression. For example the expression `=1+1` will evaluate as 2.
- You can add the values from multiple cells using the `SUM()` function. For example the expression `=SUM(A7:C7)` will evaluate the sum or the values in the range from cell A7 to cell C7. Often summing over a range, say 1 to n using the iterator variable i is written mathematically as $\sum_{i=1}^{n}$.

- You can count cells in a range using the `COUNT()` function. For example the expression `=COUNT(A7:C7)` will evaluate to 3 because there are 3 cells in the range.

Let's try working with exponents.

- You can raise a number to a power using the `^` operator. For example the expression `=2^2` will square the number 2 and evaluate as 4. This is often written as 2^2.
- You can calculate the logarithm of a number for a base using the `LOG()` function, defaulting to base 10. Remember that the log is the inverse operation of raising a number to a power. For example the expression `=LOG(2,2)` will calculate the logarithm of 4 using base 2 and will evaluate as 2.
- You can calculate the square root of a number using the `SQRT()` function. For example, the expression `=SQRT(4)` evaluates as 2. This is often written as $\sqrt{4}$.

Let's try working with the mathematical constant Euler's number (e).

- We can raise a number to e using the function `EXP()`. For example the expression `=EXP(2)` will evaluate as 7.389056099. This can also be written as e^2.
- We can calculate the natural logarithm of a number using the function `LN()`. Remember that the natural logarithm is the inverse operation of raising e to a power. For example the expression `=LN(7.389056099)` will evaluate as 2.

Some other useful stuff:

- You can calculate the mathematical constant PI using the `PI()` function. For example, the expression `=PI()` evaluates as 3.141592654. PI is usually written as π.

8.2 Statistical Summaries

Many machine learning algorithms need to use statistical summaries of input data. Let's take a look at how we can summarize data.

- You can calculate the mean or average of a list of numbers using the `AVERAGE()` function. Remember that the average is the middle or central tendency of a list of numbers. For example, the expression `=AVERAGE(1,2,3)` will evaluate as 2. Often the mean is referred to as μ (mu).
- You can calculate the mode of a list of numbers using the `MODE()` function. Remember that the mode of a list of numbers is the most common value in the list. For example the expression `=MODE(2,2,3)` will evaluate as 2.
- You can calculate the standard deviation of a list of numbers using the `STDEV()` function. Remember that the standard deviation is the average spread of the points from the mean value. For example the expression `=STDEV(1,2,3)` evaluates as 1. Often the standard deviation is referred to as σ (sigma).

- You can calculate the correlation between two lists of numbers using the `PEARSON()` function. Remember that a correlation of 1 and -1 indicate a perfect positive and negative correlation respectively. For example the expression `=PEARSON({2,3,4},{4,5,6})` evaluates as 1 (perfectly positively correlated).

All of these examples used in-line lists of numbers but can just as easily use ranges of cells as inputs.

8.3 Random Numbers

You need sample data when implementing machine learning algorithms in a spreadsheet. The best source of controlled sample data is to use random numbers.

- You can calculate a uniformly random number in the range between 0 and 1 using the `RAND()` function.
- You can calculate a Gaussian random number using the `NORMINV()` function. Remember that Gaussian refers to distribution that has a bell shape. Many linear machine learning algorithms assume a Gaussian distribution. For example, the expression `=NORMINV(RAND(), 10, 1)` will generate Gaussian random numbers with a mean of 10 and a standard deviation of 1.

8.4 Flow Control

You can do basic flow control in your spreadsheet.

- You can conditionally evaluate a cell using the `IF()` function. It takes three arguments, the first is the condition to evaluate, the second is the expression to use if the condition evaluates true, and the final argument is the expression to use if the condition evaluates false. For example the expression `=IF(1>2,"YES","NO")` evaluates as `NO`.

8.5 More Help

You do not need to be an expert in the functions presented in this chapter, but you should be comfortable with using them. As we go through the tutorials in the book, I will remind you about which functions to use. If you are unsure, come back to this chapter and use it as a reference.

Spreadsheets have excellent help. If you want to know more about the functions used in this crash course or other functions please refer to the built in help for the functions in your spreadsheet program. The help is excellent and you can learn more by using the functions in small test spreadsheets and running test data through them.

8.6 Summary

You now know enough of the mathematical functions in a spreadsheet in order to complete all of the tutorials in this book. You learned:

- How to perform basic arithmetic in a spreadsheet such as counts, sums, logarithms and exponents.
- How to use statistical functions to calculate summaries of data such as mean, mode and standard deviation.
- How to generate uniform and Gaussian random numbers to use as test data.

You know how to drive a spreadsheet. More than that, you have the basic tools that you can use to implement and play with any machine learning algorithm in a spreadsheet. In the next chapter you will discover the most common optimization algorithm in machine learning called gradient descent.

Chapter 9

Gradient Descent For Machine Learning

Optimization is a big part of machine learning. Almost every machine learning algorithm has an optimization algorithm at it's core. In this chapter you will discover a simple optimization algorithm that you can use with any machine learning algorithm. It is easy to understand and easy to implement. After reading this chapter you will know:

- About the gradient descent optimization algorithm.
- How gradient descent can be used in algorithms like linear regression.
- How gradient descent can scale to very large datasets.
- Tips for getting the most from gradient descent in practice.

Let's get started.

9.1 Gradient Descent

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function ($cost$). Gradient descent is best used when the parameters cannot be calculated analytically (e.g. using linear algebra) and must be searched for by an optimization algorithm.

9.1.1 Intuition for Gradient Descent

Think of a large bowl like what you would eat serial out of or store fruit in. This bowl is a plot of the cost function (f). A random position on the surface of the bowl is the cost of the current values of the coefficients (cost). The bottom of the bowl is the cost of the best set of coefficients, the minimum of the function.

The goal is to continue to try different values for the coefficients, evaluate their cost and select new coefficients that have a slightly better (lower) cost. Repeating this process enough times will lead to the bottom of the bowl and you will know the values of the coefficients that result in the minimum cost.

9.1.2 Gradient Descent Procedure

The procedure starts off with initial values for the coefficient or coefficients for the function. These could be 0.0 or a small random value.

$$\text{coefficient} = 0.0 \tag{9.1}$$

The cost of the coefficients is evaluated by plugging them into the function and calculating the cost.

$$\begin{aligned} cost &= f(\text{coefficient}) \\ cost &= evaluate(f(\text{coefficient})) \end{aligned} \tag{9.2}$$

The derivative of the cost is calculated. The derivative is a concept from calculus and refers to the slope of the function at a given point. We need to know the slope so that we know the direction (sign) to move the coefficient values in order to get a lower cost on the next iteration.

$$delta = derivative(cost) \tag{9.3}$$

Now that we know from the derivative which direction is downhill, we can now update the coefficient values. A learning rate parameter ($alpha$) must be specified that controls how much the coefficients can change on each update.

$$\text{coefficient} = \text{coefficient} - (alpha \times delta) \tag{9.4}$$

This process is repeated until the cost of the coefficients (cost) is 0.0 or no further improvements in cost can be achieved. You can see how simple gradient descent is. It does require you to know the gradient of your cost function or the function you are optimizing, but besides that, it's very straightforward. Next we will see how we can use this in machine learning algorithms.

9.2 Batch Gradient Descent

The goal of all supervised machine learning algorithms is to best estimate a target function (f) that maps input data (X) onto output variables (Y). This describes all classification and regression problems. Some machine learning algorithms have coefficients that characterize the algorithms estimate for the target function (f). Different algorithms have different representations and different coefficients, but many of them require a process of optimization to find the set of coefficients that result in the best estimate of the target function. Common examples of algorithms with coefficients that can be optimized using gradient descent are Linear Regression and Logistic Regression.

The evaluation of how close a fit a machine learning model estimates the target function can be calculated a number of different ways, often specific to the machine learning algorithm. The cost function involves evaluating the coefficients in the machine learning model by calculating a prediction for each training instance in the dataset and comparing the predictions to the actual output values then calculating a sum or average error (such as the Sum of Squared Residuals or SSR in the case of linear regression).

From the cost function a derivative can be calculated for each coefficient so that it can be updated using exactly the update equation described above. The cost is calculated for a machine learning algorithm over the entire training dataset for each iteration of the gradient

descent algorithm. One iteration of the algorithm is called one batch and this form of gradient descent is referred to as batch gradient descent. Batch gradient descent is the most common form of gradient descent described in machine learning.

9.3 Stochastic Gradient Descent

Gradient descent can be slow to run on very large datasets. Because one iteration of the gradient descent algorithm requires a prediction for each instance in the training dataset, it can take a long time when you have many millions of instances. In situations when you have large amounts of data, you can use a variation of gradient descent called stochastic gradient descent. In this variation, the gradient descent procedure described above is run but the update to the coefficients is performed for each training instance, rather than at the end of the batch of instances.

The first step of the procedure requires that the order of the training dataset is randomized. This is to mix up the order that updates are made to the coefficients. Because the coefficients are updated after every training instance, the updates will be noisy, jumping all over the place, and so will the corresponding cost function. By mixing up the order for the updates to the coefficients, it harnesses this random walk and avoids getting stuck.

The update procedure for the coefficients is the same as that above, except the cost is not summed or averaged over all training patterns, but instead calculated for one training pattern. The learning can be much faster with stochastic gradient descent for very large training datasets and often you only need a small number of passes through the dataset to reach a good or good enough set of coefficients, e.g. 1-to-10 passes through the dataset.

9.4 Tips for Gradient Descent

This section lists some tips and tricks for getting the most out of the gradient descent algorithm for machine learning.

- **Plot Cost versus Time**: Collect and plot the cost values calculated by the algorithm each iteration. The expectation for a well performing gradient descent run is a decrease in cost each iteration. If it does not decrease, try reducing your learning rate.
- **Learning Rate**: The learning rate value is a small real value such as 0.1, 0.001 or 0.0001. Try different values for your problem and see which works best.
- **Rescale Inputs**: The algorithm will reach the minimum cost faster if the shape of the cost function is not skewed and distorted. You can achieve this by rescaling all of the input variables (X) to the same range, such as between 0 and 1.
- **Few Passes**: Stochastic gradient descent often does not need more than 1-to-10 passes through the training dataset to converge on good or good enough coefficients.
- **Plot Mean Cost**: The updates for each training dataset instance can result in a noisy plot of cost over time when using stochastic gradient descent. Taking the average over 10, 100, or 1000 updates can give you a better idea of the learning trend for the algorithm.

9.5 Summary

In this chapter you discovered gradient descent for machine learning. You learned that:

- Optimization is a big part of machine learning.
- Gradient descent is a simple optimization procedure that you can use with many machine learning algorithms.
- Batch gradient descent refers to calculating the derivative from all training data before calculating an update.
- Stochastic gradient descent refers to calculating the derivative from each training data instance and calculating the update immediately.

You now know about the gradient descent optimization algorithm, a foundation for many machine learning algorithms. In the next chapter you will discover the linear regression algorithm for making predictions for real-valued data.

Chapter 10

Linear Regression

Linear regression is perhaps one of the most well known and well understood algorithms in statistics and machine learning. In this chapter you will discover the linear regression algorithm, how it works and how you can best use it in on your machine learning projects. In this chapter you will learn:

- Why linear regression belongs to both statistics and machine learning.
- The many names by which linear regression is known.
- The representation and learning algorithms used to create a linear regression model.
- How to best prepare your data when modeling using linear regression.

Let's get started.

10.1 Isn't Linear Regression from Statistics?

Before we dive into the details of linear regression, you may be asking yourself why we are looking at this algorithm. Isn't it a technique from statistics?

Machine learning, more specifically the field of predictive modeling is primarily concerned with minimizing the error of a model or making the most accurate predictions possible, at the expense of explainability. In applied machine learning we will borrow, reuse and steal algorithms from many different fields, including statistics and use them towards these ends.

As such, linear regression was developed in the field of statistics and is studied as a model for understanding the relationship between input and output numerical variables, but has been borrowed by machine learning. It is both a statistical algorithm and a machine learning algorithm. Next, let's review some of the common names used to refer to a linear regression model.

10.2 Many Names of Linear Regression

When you start looking into linear regression, things can get very confusing. The reason is because linear regression has been around for so long (more than 200 years). It has been studied from every possible angle and often each angle has a new and different name.

Linear regression is a linear model, e.g. a model that assumes a linear relationship between the input variables (x) and the single output variable (y). More specifically, that y can be calculated from a linear combination of the input variables (x). When there is a single input variable (x), the method is referred to as simple linear regression. When there are multiple input variables, literature from statistics often refers to the method as multiple linear regression.

Different techniques can be used to prepare or train the linear regression equation from data, the most common of which is called Ordinary Least Squares. It is common to therefore refer to a model prepared this way as Ordinary Least Squares Linear Regression or just Least Squares Regression. Now that we know some names used to describe linear regression, let's take a closer look at the representation used.

10.3 Linear Regression Model Representation

Linear regression is an attractive model because the representation is so simple. The representation is a linear equation that combines a specific set of input values (x) the solution to which is the predicted output for that set of input values (y). As such, both the input values (x) and the output value are numeric.

The linear equation assigns one scale factor to each input value or column, called a coefficient that is commonly represented by the Greek letter Beta (β). One additional coefficient is also added, giving the line an additional degree of freedom (e.g. moving up and down on a two-dimensional plot) and is often called the intercept or the bias coefficient. For example, in a simple regression problem (a single x and a single y), the form of the model would be:

$$y = B0 + B1 \times x \tag{10.1}$$

In higher dimensions when we have more than one input (x), the line is called a plane or a hyper-plane. The representation therefore is the form of the equation and the specific values used for the coefficients (e.g. $B0$ and $B1$ in the above example). It is common to talk about the complexity of a regression model like linear regression. This refers to the number of coefficients used in the model.

When a coefficient becomes zero, it effectively removes the influence of the input variable on the model and therefore from the prediction made from the model ($0 \times x = 0$). This becomes relevant if you look at regularization methods that change the learning algorithm to reduce the complexity of regression models by putting pressure on the absolute size of the coefficients, driving some to zero. Now that we understand the representation used for a linear regression model, let's review some ways that we can learn this representation from data.

10.4 Linear Regression Learning the Model

Learning a linear regression model means estimating the values of the coefficients used in the representation with the data that we have available. In this section we will take a brief look at four techniques to prepare a linear regression model. This is not enough information to implement them from scratch, but enough to get a flavor of the computation and trade-offs involved.

There are many more techniques because the model is so well studied. Take note of Ordinary Least Squares because it is the most common method used in general. Also take note of Gradient Descent as it is the most common technique taught from a machine learning perspective.

10.4.1 Simple Linear Regression

With simple linear regression when we have a single input, we can use statistics to estimate the coefficients. This requires that you calculate statistical properties from the data such as means, standard deviations, correlations and covariance. All of the data must be available to traverse and calculate statistics. This is fun as an exercise in a spreadsheet, but not really useful in practice.

10.4.2 Ordinary Least Squares

When we have more than one input we can use Ordinary Least Squares to estimate the values of the coefficients. The Ordinary Least Squares procedure seeks to minimize the sum of the squared residuals. This means that given a regression line through the data we calculate the distance from each data point to the regression line, square it, and sum all of the squared errors together. This is the quantity that ordinary least squares seeks to minimize.

This approach treats the data as a matrix and uses linear algebra operations to estimate the optimal values for the coefficients. It means that all of the data must be available and you must have enough memory to fit the data and perform matrix operations. It is unusual to implement the Ordinary Least Squares procedure yourself unless as an exercise in linear algebra. It is more likely that you will call a procedure in a linear algebra library. This procedure is very fast to calculate.

10.5 Gradient Descent

When there are one or more inputs you can use a process of optimizing the values of the coefficients by iteratively minimizing the error of the model on your training data. This operation is called Gradient Descent and works by starting with zero values for each coefficient. The sum of the squared errors are calculated for each pair of input and output values. A learning rate is used as a scale factor and the coefficients are updated in the direction towards minimizing the error. The process is repeated until a minimum sum squared error is achieved or no further improvement is possible.

When using this method, you must select a learning rate (*alpha*) parameter that determines the size of the improvement step to take on each iteration of the procedure. Gradient descent is often taught using a linear regression model because it is relatively straightforward to understand. In practice, it is useful when you have a very large dataset either in the number of rows or the number of columns that may not fit into memory.

10.5.1 Regularized Linear Regression

There are extensions of the training of the linear model called regularization methods. These seek to both minimize the sum of the squared error of the model on the training data (using Ordinary Least Squares) but also to reduce the complexity of the model (like the number or

absolute size of the sum of all coefficients in the model). Two popular examples of regularization procedures for linear regression are:

- Lasso Regression: where Ordinary Least Squares is modified to also minimize the absolute sum of the coefficients (called $L1$ regularization).
- Ridge Regression: where Ordinary Least Squares is modified to also minimize the squared absolute sum of the coefficients (called $L2$ regularization).

These methods are effective to use when there is collinearity in your input values and ordinary least squares would overfit the training data. Now that you know some techniques to learn the coefficients in a linear regression model, let's look at how we can use a model to make predictions on new data.

10.6 Making Predictions with Linear Regression

Given the representation is a linear equation, making predictions is as simple as solving the equation for a specific set of inputs. Let's make this concrete with an example. Imagine we are predicting weight (y) from height (x). Our linear regression model representation for this problem would be:

$$\begin{aligned} y &= B0 + B1 \times X1 \\ weight &= B0 + B1 \times height \end{aligned} \tag{10.2}$$

Where $B0$ is the bias coefficient and $B1$ is the coefficient for the height column. We use a learning technique to find a good set of coefficient values. Once found, we can plug in different height values to predict the weight. For example, let's use $B0 = 0.1$ and $B1 = 0.5$. Let's plug them in and calculate the weight (in kilograms) for a person with the height of 182 centimeters.

$$\begin{aligned} weight &= 0.1 + 0.05 \times 182 \\ weight &= 91.1 \end{aligned} \tag{10.3}$$

You can see that the above equation could be plotted as a line in two-dimensions. The $B0$ is our starting point regardless of what height we have. We can run through a bunch of heights from 100 to 250 centimeters and plug them to the equation and get weight values, creating our line.

Now that we know how to make predictions given a learned linear regression model, let's look at some rules of thumb for preparing our data to make the most of this type of model.

10.7 Preparing Data For Linear Regression

Linear regression is been studied at great length, and there is a lot of literature on how your data must be structured to make best use of the model. As such, there is a lot of sophistication when talking about these requirements and expectations which can be intimidating. In practice, you can uses these rules more as rules of thumb when using Ordinary Least Squares Regression, the most common implementation of linear regression. Try different preparations of your data using these heuristics and see what works best for your problem.

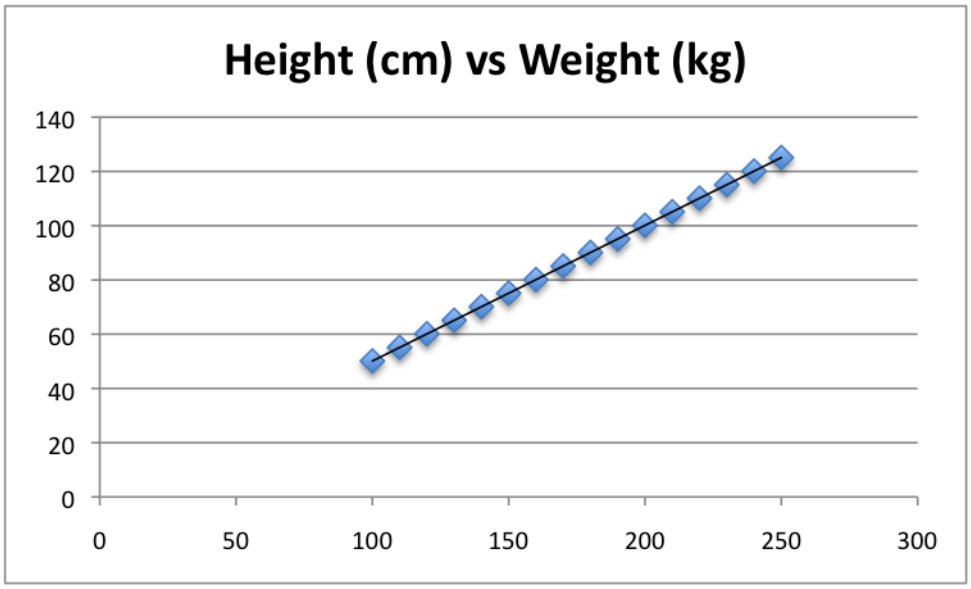

Figure 10.1: Sample Height vs Weight Linear Regression.

- **Linear Assumption**. Linear regression assumes that the relationship between your input and output is linear. It does not support anything else. This may be obvious, but it is good to remember when you have a lot of attributes. You may need to transform data to make the relationship linear (e.g. log transform for an exponential relationship).
- **Remove Noise**. Linear regression assumes that your input and output variables are not noisy. Consider using data cleaning operations that let you better expose and clarify the signal in your data. This is most important for the output variable and you want to remove outliers in the output variable (y) if possible.
- **Remove Collinearity**. Linear regression will over-fit your data when you have highly correlated input variables. Consider calculating pairwise correlations for your input data and removing the most correlated.
- **Gaussian Distributions**. Linear regression will make more reliable predictions if your input and output variables have a Gaussian distribution. You may get some benefit using transforms (e.g. log or BoxCox) on you variables to make their distribution more Gaussian looking.
- **Rescale Inputs**: Linear regression will often make more reliable predictions if you rescale input variables using standardization or normalization.

10.8 Summary

In this chapter you discovered the linear regression algorithm for machine learning. You covered a lot of ground including:

- The common names used when describing linear regression models.
- The representation used by the model.

- Learning algorithms used to estimate the coefficients in the model.
- Rules of thumb to consider when preparing data for use with linear regression.

You now know about the linear regression algorithm for making real-valued predictions. In the next chapter you will discover how to implement the simple linear regression algorithm from scratch.

Chapter 11

Simple Linear Regression Tutorial

Linear regression is a very simple method but has proven to be very useful for a large number of situations. In this chapter you will discover exactly how linear regression works step-by-step. After reading this chapter you will know:

- How to calculate a simple linear regression step-by-step.
- How to make predictions on new data using your model.
- A shortcut that greatly simplifies the calculation.

Let's get started.

11.1 Tutorial Data Set

The data set we are using is completely made up. Below is the raw data.

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

Listing 11.1: Tutorial Data Set.

The attribute x is the input variable and y is the output variable that we are trying to predict. If we got more data, we would only have x values and we would be interested in predicting y values. Below is a simple scatter plot of x versus y.

We can see the relationship between x and y looks kind-of linear. As in, we could probably draw a line somewhere diagonally from the bottom left of the plot to the top right to generally describe the relationship between the data. This is a good indication that using linear regression might be appropriate for this little dataset.

11.2 Simple Linear Regression

When we have a single input attribute (x) and we want to use linear regression, this is called simple linear regression. If we had multiple input attributes (e.g. $X1$, $X2$, $X3$, etc.) This would

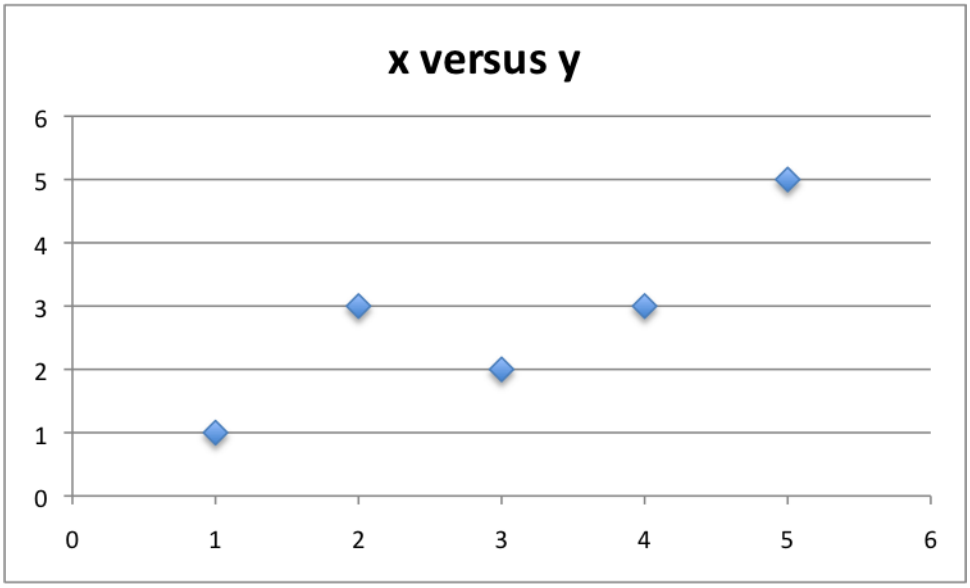

Figure 11.1: Simple Linear Regression Dataset.

be called multiple linear regression. The procedure for linear regression is different and simpler than that for multiple linear regression, so it is a good place to start. In this section we are going to create a simple linear regression model from our training data, then make predictions for our training data to get an idea of how well the model learned the relationship in the data. With simple linear regression we want to model our data as follows:

$$y = B0 + B1 \times x \tag{11.1}$$

This is a line where y is the output variable we want to predict, x is the input variable we know and $B0$ and $B1$ are coefficients that we need to estimate that move the line around. Technically, $B0$ is called the intercept because it determines where the line intercepts the y-axis. In machine learning we can call this the bias, because it is added to offset all predictions that we make. The $B1$ term is called the slope because it defines the slope of the line or how x translates into a y value before we add our bias.

The goal is to find the best estimates for the coefficients to minimize the errors in predicting y from x. Simple regression is great, because rather than having to search for values by trial and error or calculate them analytically using more advanced linear algebra, we can estimate them directly from our data. We can start off by estimating the value for $B1$ as:

$$B1 = \frac{\sum_{i=1}^{n}(x_i - mean(x)) \times (y_i - mean(y))}{\sum_{i=1}^{n}(x_i - mean(x))^2} \tag{11.2}$$

Where $mean()$ is the average value for the variable in our dataset. The x_i and y_i refer to the fact that we need to repeat these calculations across all values in our dataset and i refers to the i'th value of x or y. We can calculate $B0$ using $B1$ and some statistics from our dataset, as follows:

$$B0 = mean(y) - B1 \times mean(x) \tag{11.3}$$

Not that bad right? We can calculate these right in our spreadsheet.

11.2.1 Estimating The Slope (B1)

Let's start with the top part of the equation, the numerator. First we need to calculate the mean value of x and y. The mean is calculated as:

$$\frac{1}{n} \times \sum_{i=1}^{n} x_i \tag{11.4}$$

Where n is the number of values (5 in this case). You can use the `AVERAGE()` function in your spreadsheet. Let's calculate the mean value of our x and y variables:

$$\begin{aligned} mean(x) &= 3 \\ mean(y) &= 2.8 \end{aligned} \tag{11.5}$$

Now we need to calculate the error of each variable from the mean. Let's do this with x first:

```
x   mean(x)    x - mean(x)
1   3          -2
2              -1
4              1
3              0
5              2
```

Listing 11.2: Residual of each x value from the mean.

Now let's do that for the y variable.

```
y   mean(y)    y - mean(y)
1   2.8        -1.8
3              0.2
3              0.2
2              -0.8
5              2.2
```

Listing 11.3: Residual of each y value from the mean.

We now have the parts for calculating the numerator. All we need to do is multiple the error for each x with the error for each y and calculate the sum of these multiplications.

```
x - mean(x)    y - mean(y)    Multiplication
-2             -1.8           3.6
-1             0.2            -0.2
1              0.2            0.2
0              -0.8           0
2              2.2            4.4
```

Listing 11.4: Multiplication of the x and y residuals from their means.

Summing the final column we have calculated our numerator as 8. Now we need to calculate the bottom part of the equation for calculating $B1$, or the denominator. This is calculated as the sum of the squared differences of each x value from the mean. We have already calculated the difference of each x value from the mean, all we need to do is square each value and calculate the sum.

```
x - mean(x)    squared
-2             4
-1             1
```

```
1               1
0               0
2               4
```

Listing 11.5: Squared residual of each x value from the mean.

Calculating the sum of these squared values gives us up denominator of 10. Now we can calculate the value of our slope.

$$\begin{aligned} B1 &= \frac{8}{10} \\ B1 &= 0.8 \end{aligned} \tag{11.6}$$

11.2.2 Estimating The Intercept (B0)

This is much easier as we already know the values of all of the terms involved.

$$\begin{aligned} B0 &= mean(y) - B1 \times mean(x) \\ B0 &= 2.8 - 0.8 \times 3 \\ B0 &= 0.4 \end{aligned} \tag{11.7}$$

11.3 Making Predictions

We now have the coefficients for our simple linear regression equation.

$$\begin{aligned} y &= B0 + B1 \times x \\ y &= 0.4 + 0.8 \times x \end{aligned} \tag{11.8}$$

Let's try out the model by making predictions for our training data.

```
x   Predicted Y
1   1.2
2   2
4   3.6
3   2.8
5   4.4
```

Listing 11.6: Predicted y value for each x input value.

We can plot these predictions as a line with our data. This gives us a visual idea of how well the line models our data.

11.4 Estimating Error

We can calculate an error score for our predictions called the Root Mean Squared Error or RMSE.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(p_i - y_i)^2}{n}} \tag{11.9}$$

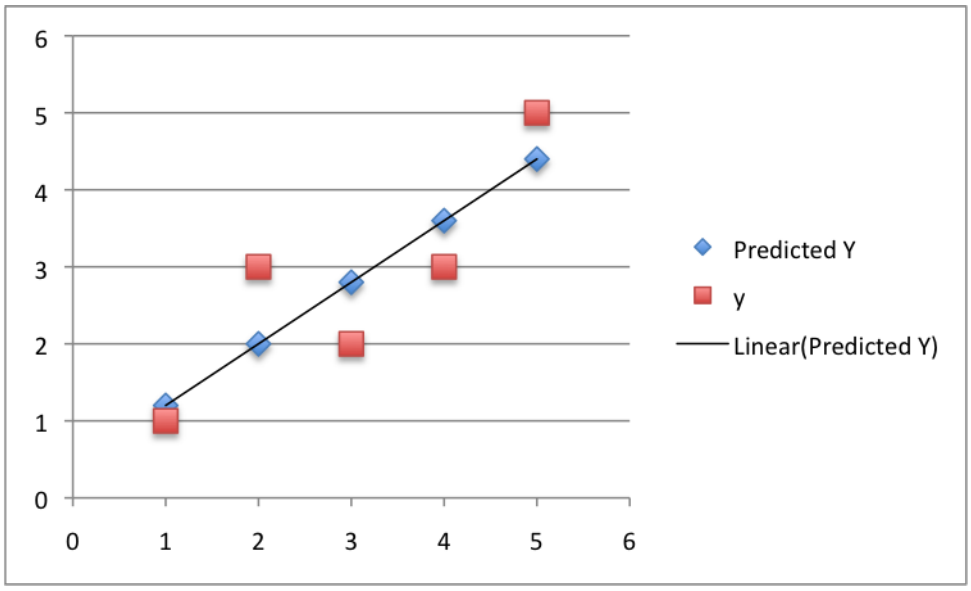

Figure 11.2: Simple Linear Regression Predictions.

Where you can use `SQRT()` function in your spreadsheet to calculate the square root, p is the predicted value and y is the actual value, i is the index for a specific instance, because we must calculate the error across all predicted values. First we must calculate the difference between each model prediction and the actual y values.

```
Predicted  y   Predicted - y
1.2        1   0.2
2          3   -1
3.6        3   0.6
2.8        2   0.8
4.4        5   -0.6
```

Listing 11.7: Error for predicted values.

We can easily calculate the square of each of these error values ($error \times error$ or $error^2$).

```
Predicted - y     squared error
0.2               0.04
-1                1
0.6               0.36
0.8               0.64
-0.6              0.36
```

Listing 11.8: Squared error for predicted values.

The sum of these errors is 2.4 units, dividing by 5 and taking the square root gives us:

$$RMSE = 0.692820323 \tag{11.10}$$

Or, each prediction is on average wrong by about 0.692 units.

11.5 Shortcut

Before we wrap up I want to show you a quick shortcut for calculating the coefficients. Simple linear regression is the simplest form of regression and the most studied. There is a shortcut that you can use to quickly estimate the values for $B0$ and $B1$. Really it is a shortcut for calculating $B1$. The calculation of $B1$ can be re-written as:

$$B1 = corr(x, y) \times \frac{stdev(y)}{stdev(x)} \tag{11.11}$$

Where $corr(x)$ is the correlation between x and y an $stdev()$ is the calculation of the standard deviation for a variable. Correlation (also known as Pearson's correlation coefficient) is a measure of how related two variables are in the range of -1 to 1. A value of 1 indicates that the two variables are perfectly positively correlated, they both move in the same direction and a value of -1 indicates that they are perfectly negatively correlated, when one moves the other moves in the other direction.

Standard deviation is a measure of how much on average the data is spread out from the mean. You can use the function `PEARSON()` in your spreadsheet to calculate the correlation of x and y as 0.852 (highly correlated) and the function `STDEV()` to calculate the standard deviation of x as 1.5811 and y as 1.4832. Plugging these values in we have:

$$\begin{aligned} B1 &= 0.852802865 \times \frac{1.483239697}{1.58113883} \\ B1 &= 0.8 \end{aligned} \tag{11.12}$$

11.6 Summary

In this chapter you discovered how to implement simple linear regression step-by-step in a spreadsheet. You learned:

- How to estimate the coefficients for a simple linear regression model from your training data.
- How to make predictions using your learned model.

You now know how to implement the simple linear regression algorithm from scratch. In the next section, you will discover how you can implement linear regression from scratch using stochastic gradient descent.

Chapter 12

Linear Regression Tutorial Using Gradient Descent

Stochastic Gradient Descent is an important and widely used algorithm in machine learning. In this chapter you will discover how to use Stochastic Gradient Descent to learn the coefficients for a simple linear regression model by minimizing the error on a training dataset. After reading this chapter you will know:

- How stochastic gradient descent can be used to search for the coefficients of a regression model.
- How repeated iterations of gradient descent can create an accurate regression model.

Let's get started.

12.1 Tutorial Data Set

The dataset is the same as that used in the previous chapter on Simple Linear Regression. It is listed again for completeness.

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

Listing 12.1: Tutorial Data Set.

12.2 Stochastic Gradient Descent

Gradient Descent is the process of minimizing a function by following the gradients of the cost function. This involves knowing the form of the cost as well as the derivative so that from a given point you know the gradient and can move in that direction, e.g. downhill towards the minimum value. In machine learning we can use a technique that evaluates and update 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 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, called weights (w) in machine learning language are updated using the equation:

$$w = w - alpha \times delta \tag{12.1}$$

Where w is the coefficient or weight being optimized, *alpha* is a learning rate that you must configure (e.g. 0.1) and gradient is the error for the model on the training data attributed to the weight.

12.3 Simple Linear Regression with Stochastic Gradient Descent

The coefficients used in simple linear regression can be found using stochastic gradient descent. Stochastic gradient descent is not used to calculate the coefficients for linear regression in practice unless the dataset prevents traditional Ordinary Least Squares being used (e.g. a very large dataset). Nevertheless, linear regression does provide a useful exercise for practicing stochastic gradient descent which is an important algorithm used for minimizing cost functions by machine learning algorithms. As stated in the previous chapter, our linear regression model is defined as follows:

$$y = B0 + B1 \times x \tag{12.2}$$

12.3.1 Gradient Descent Iteration #1

Let's start with values of 0.0 for both coefficients.

$$\begin{aligned} B0 &= 0.0 \\ B1 &= 0.0 \\ y &= 0.0 + 0.0 \times x \end{aligned} \tag{12.3}$$

We can calculate the error for a prediction as follows:

$$error = p(i) - y(i) \tag{12.4}$$

Where $p(i)$ is the prediction for the i'th instance in our dataset and $y(i)$ is the i'th output variable for the instance in the dataset. We can now calculate he predicted value for y using our starting point coefficients for the first training instance: $x = 1, y = 1$.

$$\begin{aligned} p(i) &= 0.0 + 0.0 \times 1 \\ p(i) &= 0 \end{aligned} \tag{12.5}$$

Using the predicted output, we can calculate our error:

$$\begin{aligned} error &= (0 - 1) \\ error &= -1 \end{aligned} \tag{12.6}$$

We can now use this error in our equation for gradient descent to update the weights. We will start with updating the intercept first, because it is easier. We can say that $B0$ is accountable for all of the error. This is to say that updating the weight will use just the error as the gradient. We can calculate the update for the $B0$ coefficient as follows:

$$B0(t+1) = B0(t) - alpha \times error \tag{12.7}$$

Where $B0(t+1)$ is the updated version of the coefficient we will use on the next training instance, $B0(t)$ is the current value for $B0$, $alpha$ is our learning rate and error is the error we calculate for the training instance. Let's use a small learning rate of 0.01 and plug the values into the equation to work out what the new and slightly optimized value of $B0$ will be:

$$\begin{aligned} B0(t+1) &= 0.0 - 0.01 \times -1.0 \\ B0(t+1) &= 0.01 \end{aligned} \tag{12.8}$$

Now, let's look at updating the value for $B1$. We use the same equation with one small change. The error is filtered by the input that caused it. We can update $B1$ using the equation:

$$B1(t+1) = B1(t) - alpha \times error \times x \tag{12.9}$$

Where $B1(t+1)$ is the update coefficient, $B1(t)$ is the current version of the coefficient, $alpha$ is the same learning rate described above, error is the same error calculated above and x is the input value. We can plug in our numbers into the equation and calculate the updated value for $B1$:

$$\begin{aligned} B1(t+1) &= 0.0 - 0.01 \times -1 \times 1 \\ B1(t+1) &= 0.01 \end{aligned} \tag{12.10}$$

We have just finished the first iteration of gradient descent and we have updated our weights to be $B0 = 0.01$ and $B1 = 0.01$. This process must be repeated for the remaining 4 instances from our dataset. One pass through the training dataset is called an epoch.

12.3.2 Gradient Descent Iteration #20

Let's jump ahead. You can repeat this process another 19 times. This is 4 complete epochs of the training data being exposed to the model and updating the coefficients. Here is a list of all of the values for the coefficients over the 20 iterations that you should see:

```
B0            B1
0.01          0.01
0.0397        0.0694
0.066527      0.176708
0.08056049    0.21880847
0.118814462   0.410078328
0.123525534   0.4147894
0.14399449    0.455727313
0.154325453   0.497051164
```

```
0.157870663    0.507686795
0.180907617    0.622871563
0.182869825    0.624833772
0.198544452    0.656183024
0.200311686    0.663251962
0.19841101     0.657549935
0.213549404    0.733241901
0.21408149     0.733773988
0.227265196    0.760141398
0.224586888    0.749428167
0.219858174    0.735242025
0.230897491    0.79043861
```

Listing 12.2: Simple linear regression coefficients after 20 iterations.

I think that 20 iterations or 4 epochs is a nice round number and a good place to stop. You could keep going if you wanted. Your values should match closely, but may have minor differences due to different spreadsheet programs and different precisions. You can plug each pair of coefficients back into the simple linear regression equation. This is useful because we can calculate a prediction for each training instance and in turn calculate the error.

Below is a plot of the error for each set of coefficients as the learning process unfolded. This is a useful graph as it shows us that error was decreasing with each iteration and starting to bounce around a bit towards the end.

Figure 12.1: Simple Linear Regression Performance Versus Iteration.

You can see that our final coefficients have the values $B0 = 0.230897491$ and $B1 = 0.79043861$. Let's plug them into our simple linear Regression model and make a prediction for each point in our training dataset.

```
x   Prediction
1   1.021336101
2   1.811774711
4   3.392651932
3   2.602213322
```

```
5   4.183090542
```

Listing 12.3: Simple linear regression predictions for the training dataset.

We can plot our dataset again with these predictions overlaid (x vs y and x vs *prediction*). Drawing a line through the 5 predictions gives us an idea of how well the model fits the training data.

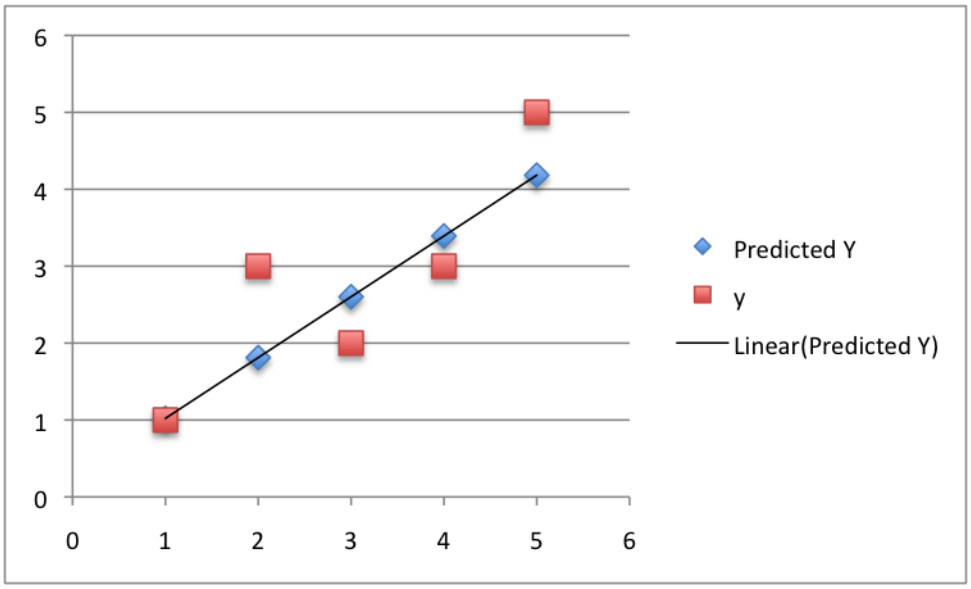

Figure 12.2: Simple Linear Regression Predictions.

We can calculate the RMSE for these predictions as we did in the previous chapter. The result comes out to be $RMSE = 0.720626401$.

12.4 Summary

In this chapter you discovered the simple linear regression model and how to train it using stochastic gradient descent. You learned:

- How to work through the application of the update rule for gradient descent.
- How to make predictions using a learned linear regression model.

You now know how to implement linear regression using stochastic gradient descent. In the next chapter you will discover the logistic regression algorithm for binary classification.

Chapter 13

Logistic Regression

Logistic regression is another technique borrowed by machine learning from the field of statistics. It is the go-to method for binary classification problems (problems with two class values). In this chapter you will discover the logistic regression algorithm for machine learning. After reading this chapter you will know:

- The many names and terms used when describing logistic regression (like log odds and logit).
- The representation used for a logistic regression model.
- Techniques used to learn the coefficients of a logistic regression model from data.
- How to actually make predictions using a learned logistic regression model.
- Where to go for more information if you want to dig a little deeper.

Let's get started.

13.1 Logistic Function

Logistic regression is named for the function used at the core of the method, the logistic function. The logistic function, also called the sigmoid function was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It's an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.

$$\frac{1}{1 + e^{-value}} \tag{13.1}$$

Where e is the base of the natural logarithms (Euler's number or the `EXP()` function in your spreadsheet) and $value$ is the actual numerical value that you want to transform. Below is a plot of the numbers between -5 and 5 transformed into the range 0 and 1 using the logistic function.

Now that we know what the logistic function is, let's see how it is used in logistic regression.

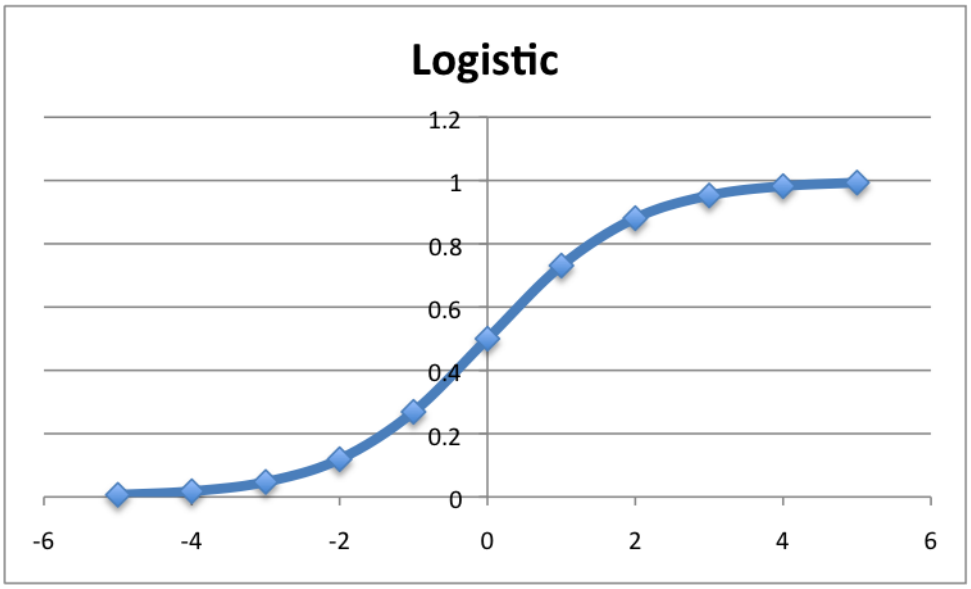

Figure 13.1: Logistic Function.

13.2 Representation Used for Logistic Regression

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 values (0 or 1) rather than a numeric value.

Below is an example logistic regression equation:

$$y = \frac{e^{B0+B1\times x}}{1 + e^{B0+B1\times x}} \tag{13.2}$$

Where y is the predicted output, $B0$ is the bias or intercept term and $B1$ is the coefficient for the single input value (x). 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 are the coefficients in the equation (the beta value or B's).

13.3 Logistic Regression Predicts Probabilities

Logistic regression models the probability of the default class (e.g. the first class). For example, if we are modeling people's sex as male or female from their height, then the first class could be male and the logistic regression model could be written as the probability of male given a person's height, or more formally:

$$P(sex = male|height) \tag{13.3}$$

Written another way, we are modeling the probability that an input (X) belongs to the default class ($Y = 1$), we can write this formally as:

$$P(X) = P(Y = 1|X) \tag{13.4}$$

We're predicting probabilities? I thought logistic regression was a classification algorithm? Note that the probability prediction must be transformed into a binary values (0 or 1) in order to actually make a crisp prediction. More on this later when we talk about making predictions.

Logistic regression is a linear method, but the predictions are transformed using the logistic function. The impact of this is that we can no longer understand the predictions as a linear combination of the inputs as we can with linear regression, for example, continuing on from above, the model can be stated as:

$$p(X) = \frac{e^{B0+B1\times X}}{1 + e^{B0+B1\times X}} \tag{13.5}$$

I don't want to dive into the math too much, but we can turn around the above equation as follows (remember we can remove the e from one side by adding a $ln()$ to the other):

$$ln(\frac{p(X)}{1 - p(X)}) = B0 + B1 \times X \tag{13.6}$$

This is useful because we can see that the calculation of the output on the right is linear again (just like linear regression), and the input on the left is a natural logarithm of the probability of the default class. This ratio on the left is called the odds of the default class (it's historical that we use odds, for example, odds are used in horse racing rather than probabilities). Odds are calculated as a ratio of the probability of the event divided by the probability of not the event, e.g. $\frac{0.8}{1-0.8}$ which has the odds of 4. So we could instead write:

$$ln(odds) = B0 + B1 \times X \tag{13.7}$$

Because the odds are log transformed, we call this left hand side the log-odds or the probit. It is possible to use other types of functions for the transform (which is out of scope), but as such it is common to refer to the transform that relates the linear regression equation to the probabilities as the link function, e.g. the logit link function. We can move the exponent back to the right and write it as:

$$odds = e^{B0+B1\times X} \tag{13.8}$$

All of this helps us understand that indeed the model is still a linear combination of the inputs, but that this linear combination relates to the log-odds of the default class.

13.4 Learning the Logistic Regression Model

The coefficients of the logistic regression algorithm must be estimated from your training data. This is done using maximum-likelihood estimation. Maximum-likelihood estimation is a common learning algorithm used by a variety of machine learning algorithms, although it does make assumptions about the distribution of your data (more on this when we talk about preparing your data).

The best coefficients would result in a model that would predict a value very close to 1 (e.g. male) for the default class and a value very close to 0 (e.g. female) for the other class. The intuition for maximum-likelihood for logistic regression is that a search procedure seeks values for the coefficients that minimize the error in the probabilities predicted by the model to those in the data (e.g. probability of 1 if the data is the primary class).

We are not going to go into the math of maximum likelihood. It is enough to say that a minimization algorithm is used to optimize the best values for the coefficients for your training data. This is often implemented in practice using efficient numerical optimization algorithm (like the Quasi-newton method). When you are learning logistic, you can implement it yourself from scratch using the much simpler gradient descent algorithm.

13.5 Making Predictions with Logistic Regression

Making predictions with a logistic regression model is as simple as plugging in numbers into the logistic regression equation and calculating a result. Let's make this concrete with a specific example. Let's say we have a model that can predict whether a person is male or female based on their height (completely fictitious). Given a height of 150 cm is the person male or female.

We have learned the coefficients of $B0 = -100$ and $B1 = 0.6$. Using the equation above we can calculate the probability of male given a height of 150 cm or more formally $P(male|height = 150)$. We will use `EXP()` for e, because that is what you can use if you type this example into your spreadsheet:

$$\begin{aligned} y &= \frac{e^{B0+B1\times X}}{1+e^{B0+B1\times X}} \\ y &= \frac{EXP(-100+0.6\times 150)}{1+EXP(-100+0.6\times X)} \\ y &= 0.0000453978687 \end{aligned} \tag{13.9}$$

Or a probability of near zero that the person is a male. In practice we can use the probabilities directly. Because this is classification and we want a crisp answer, we can snap the probabilities to a binary class value, for example:

$$\begin{aligned} prediction &= 0 \text{ IF } p(male) < 0.5 \\ prediction &= 1 \text{ IF } p(male) \geq 0.5 \end{aligned} \tag{13.10}$$

Now that we know how to make predictions using logistic regression, let's look at how we can prepare our data to get the most from the technique.

13.6 Prepare Data for Logistic Regression

The assumptions made by logistic regression about the distribution and relationships in your data are much the same as the assumptions made in linear regression. Much study has gone into defining these assumptions and precise probabilistic and statistical language is used. My advice is to use these as guidelines or rules of thumb and experiment with different data preparation schemes. Ultimately in predictive modeling machine learning projects you are laser focused on making accurate predictions rather than interpreting the results. As such, you can break some assumptions as long as the model is robust and performs well.

- **Binary Output Variable**: This might be obvious as we have already mentioned it, but logistic regression is intended for binary (two-class) classification problems. It will predict the probability of an instance belonging to the default class, which can be snapped into a 0 or 1 classification.

- **Remove Noise**: Logistic regression assumes no error in the output variable (y), consider removing outliers and possibly misclassified instances from your training data.
- **Gaussian Distribution**: Logistic regression is a linear algorithm (with a nonlinear transform on output). It does assume a linear relationship between the input variables with the output. Data transforms of your input variables that better expose this linear relationship can result in a more accurate model. For example, you can use log, root, Box-Cox and other univariate transforms to better expose this relationship.
- **Remove Correlated Inputs**: Like linear regression, the model can overfit if you have multiple highly-correlated inputs. Consider calculating the pairwise correlations between all inputs and removing highly correlated inputs.
- **Fail to Converge**: It is possible for the expected likelihood estimation process that learns the coefficients to fail to converge. This can happen if there are many highly correlated inputs in your data or the data is very sparse (e.g. lots of zeros in your input data).

13.7 Summary

In this chapter you discovered the logistic regression algorithm for machine learning and predictive modeling. You covered a lot of ground and learned:

- What the logistic function is and how it is used in logistic regression.
- That the key representation in logistic regression are the coefficients, just like linear regression.
- That the coefficients in logistic regression are estimated using a process called maximum-likelihood estimation.
- That making predictions using logistic regression is so easy that you can do it in a spreadsheet.
- That the data preparation for logistic regression is much like linear regression.

You now know about the logistic regression algorithm for binary classification. In the next chapter you will discover how to implement logistic regression from scratch using stochastic gradient descent.

Chapter 14

Logistic Regression Tutorial

Logistic regression is one of the most popular machine learning algorithms for binary classification. This is because it is a simple algorithm that performs very well on a wide range of problems. In this chapter you are going to discover the logistic regression algorithm for binary classification, step-by-step. After reading this chapter you will know:

- How to calculate the logistic function.
- How to learn the coefficients for a logistic regression model using stochastic gradient descent.
- How to make predictions using a logistic regression model.

Let's get started.

14.1 Tutorial Dataset

In this tutorial we will use a contrived dataset. This dataset has two input variables ($X1$ and $X2$) and one output variable (Y). The input variables are real-valued random numbers drawn from a Gaussian distribution. The output variable has two values, making the problem a binary classification problem. The raw data is listed below.

```
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 14.1: Tutorial Data Set.

Below is a plot of the dataset. You can see that it is completely contrived and that we can easily draw a line to separate the classes. This is exactly what we are going to do with the logistic regression model.

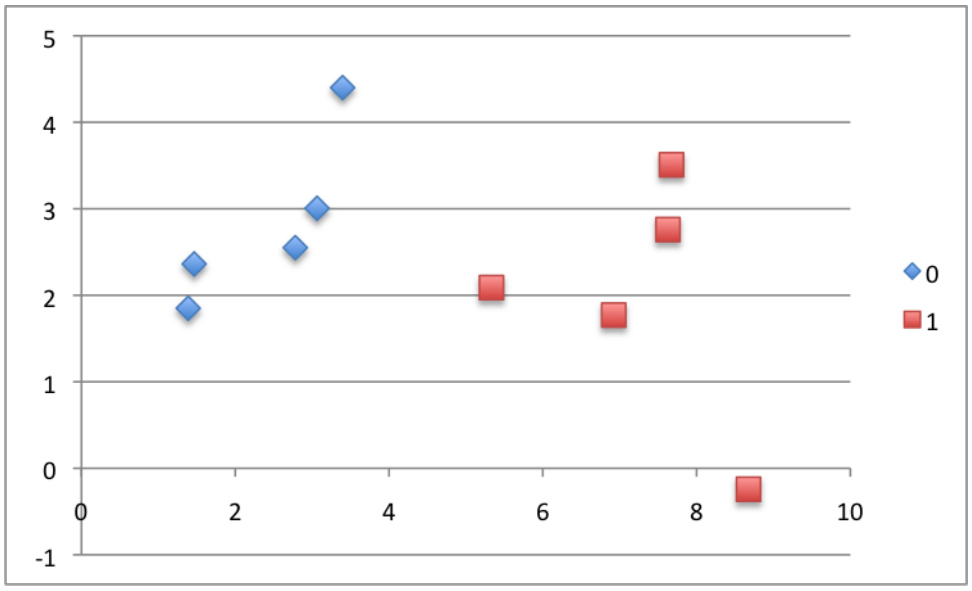

Figure 14.1: Logistic Regression Dataset.

14.2 Logistic Regression Model

The logistic regression model takes real-valued inputs and makes a prediction as to the probability of the input belonging to the default class (class 0). If the probability is greater than 0.5 we can take the output as a prediction for the default class (class 0), otherwise the prediction is for the other class (class 1). For this dataset, the logistic regression has three coefficients just like linear regression, for example:

$$output = B0 + B1 \times X1 + B2 \times X2 \tag{14.1}$$

The job of the learning algorithm will be to discover the best values for the coefficients ($B0$, $B1$ and $B2$) based on the training data. Unlike linear regression, the output is transformed into a probability using the logistic function:

$$p(class = 0) = \frac{1}{1 + e^{-output}} \tag{14.2}$$

In your spreadsheet this would be written as:

$$p(class = 0) = \frac{1}{1 + EXP(-output)} \tag{14.3}$$

14.3 Logistic Regression by Stochastic Gradient Descent

We can estimate the values of the coefficients using stochastic gradient descent. We can apply stochastic gradient descent to the problem of finding the coefficients for the logistic regression model.

14.3.1 Calculate Prediction

Let's start off by assigning 0.0 to each coefficient and calculating the probability of the first training instance that belongs to class 0.

$$\begin{aligned} B0 &= 0.0 \\ B1 &= 0.0 \\ B2 &= 0.0 \end{aligned} \tag{14.4}$$

The first training instance is: $X1 = 2.7810836$, $X2 = 2.550537003$, $Y = 0$. Using the above equation we can plug in all of these numbers and calculate a prediction:

$$\begin{aligned} prediction &= \frac{1}{1 + e^{-(B0 + B1 \times X1 + B2 \times X2)}} \\ prediction &= \frac{1}{1 + e^{-(0.0 + 0.0 \times 2.7810836 + 0.0 \times 2.550537003)}} \\ prediction &= 0.5 \end{aligned} \tag{14.5}$$

14.3.2 Calculate New Coefficients

We can calculate the new coefficient values using a simple update equation.

$$b = b + alpha \times (y - prediction) \times prediction \times (1 - prediction) \times x \tag{14.6}$$

Where b is the coefficient we are updating and prediction is the output of making a prediction using the model. *Alpha* is a parameter that you must specify at the beginning of the training run. This is the learning rate and controls how much the coefficients (and therefore the model) changes or learns each time it is updated. Larger learning rates are used in online learning (when we update the model for each training instance). Good values might be in the range 0.1 to 0.3. Let's use a value of 0.3.

You will notice that the last term in the equation is x, this is the input value for the coefficient. You will notice that the $B0$ does not have an input. This coefficient is often called the bias or the intercept and we can assume it always has an input value of 1.0. This assumption can help when implementing the algorithm using vectors or arrays. Let's update the coefficients using the prediction (0.5) and coefficient values (0.0) from the previous section.

$$\begin{aligned} B0 &= 0.0 + 0.3 \times (0 - 0.5) \times 0.5 \times (1 - 0.5) \times 1.0 \\ B1 &= B1 + 0.3 \times (0 - 0.5) \times 0.5 \times (1 - 0.5) \times 2.7810836 \\ B2 &= B2 + 0.3 \times (0 - 0.5) \times 0.5 \times (1 - 0.5) \times 2.550537003 \end{aligned} \tag{14.7}$$

or

$$\begin{aligned} B0 &= -0.0375 \\ B1 &= -0.104290635 \\ B2 &= -0.095645138 \end{aligned} \tag{14.8}$$

14.3.3 Repeat the Process

We can repeat this process and update the model for each training instance in the dataset. A single iteration through the training dataset is called an epoch. It is common to repeat the stochastic gradient descent procedure for a fixed number of epochs. At the end of epoch you can calculate error values for the model. Because this is a classification problem, it would be nice to get an idea of how accurate the model is at each iteration. The graph below show a plot of accuracy of the model over 10 epochs.

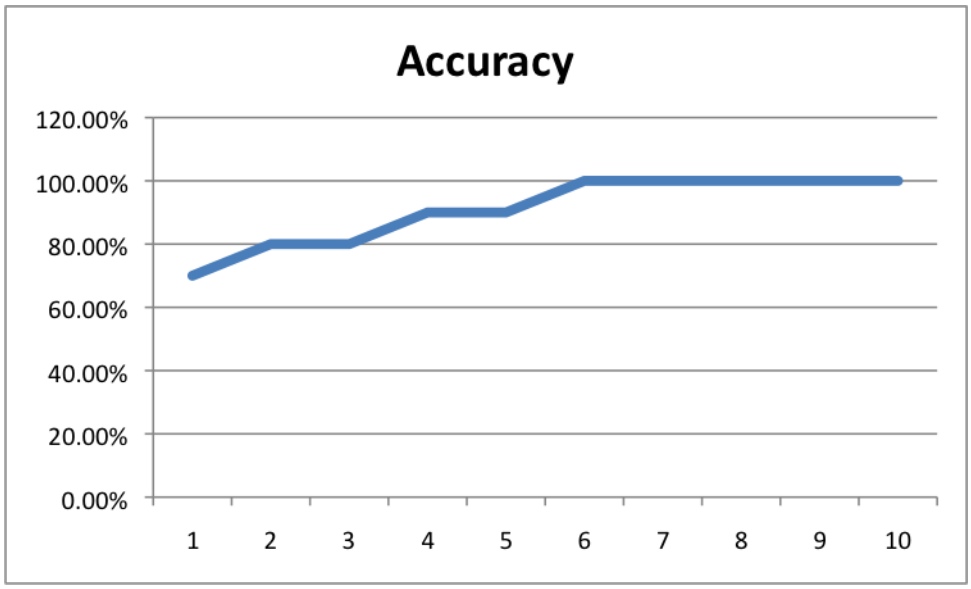

Figure 14.2: Logistic Regression with Gradient Descent Accuracy versus Iteration.

You can see that the model very quickly achieves 100% accuracy on the training dataset. The coefficients calculated after 10 epochs of stochastic gradient descent are:

$$\begin{aligned} B0 &= -0.406605464 \\ B1 &= 0.852573316 \\ B2 &= -1.104746259 \end{aligned} \tag{14.9}$$

14.3.4 Make Predictions

Now that we have trained the model, we can use it to make predictions. We can make predictions on the training dataset, but this could just as easily be new data. Using the coefficients above learned after 10 epochs, we can calculate output values for each training instance:

```
X1            X2            Prediction
2.7810836     2.550537003   0.298756986
1.465489372   2.362125076   0.145951056
3.396561688   4.400293529   0.085333265
1.38807019    1.850220317   0.219737314
3.06407232    3.005305973   0.247059
7.627531214   2.759262235   0.954702135
5.332441248   2.088626775   0.862034191
6.922596716   1.77106367    0.971772905
```

```
8.675418651    -0.242068655   0.999295452
7.673756466    3.508563011    0.905489323
```

Listing 14.2: Raw Logistic Regression Predictions.

These are the probabilities of each instance belonging to $Y = 0$. We can convert these into crisp class values using:

$$prediction = \text{IF } (output < 0.5) \text{ Then 0 Else 1} \tag{14.10}$$

With this simple procedure we can convert all of the outputs to class values:

```
Prediction     Crisp
0.298756986    0
0.145951056    0
0.085333265    0
0.219737314    0
0.247059       0
0.954702135    1
0.862034191    1
0.971772905    1
0.999295452    1
0.905489323    1
```

Listing 14.3: Crisp Logistic Regression Predictions.

Finally, we can calculate the accuracy for the model on the training dataset:

$$\begin{aligned} accuracy &= \frac{CorrectPredictions}{TotalPredictions} \times 100 \\ accuracy &= \frac{10}{10} \times 100 \\ accuracy &= 100\% \end{aligned} \tag{14.11}$$

14.4 Summary

In this chapter you discovered how you can implement logistic regression from scratch, step-by-step. You learned:

- How to calculate the logistic function.
- How to learn the coefficients for a logistic regression model using stochastic gradient descent.
- How to make predictions using a logistic regression model.

You now know how to implement logistic regression from scratch using stochastic gradient descent. In the next chapter you will discover the linear discriminant analysis algorithm for classification.

Chapter 15

Linear Discriminant Analysis

Logistic regression is a classification algorithm traditionally limited to only two-class classification problems. If you have more than two classes then the Linear Discriminant Analysis is the preferred linear classification technique. In this chapter you will discover the Linear Discriminant Analysis (LDA) algorithm for classification predictive modeling problems. After reading this chapter you will know.

- The limitations of logistic regression and the need for linear discriminant analysis.
- The representation of the model that is learned from data and can be saved to file.
- How the model is estimated from your data.
- How to make predictions from a learned LDA model.
- How to prepare your data to get the most from the LDA model.

Let's get started.

15.1 Limitations of Logistic Regression

Logistic regression is a simple and powerful linear classification algorithm. It also has limitations that suggest at the need for alternate linear classification algorithms.

- **Two-Class Problems**. Logistic regression is intended for two-class or binary classification problems. It can be extended for multiclass classification, but is rarely used for this purpose.
- **Unstable With Well Separated Classes**. Logistic regression can become unstable when the classes are well separated.
- **Unstable With Few Examples**. Logistic regression can become unstable when there are few examples from which to estimate the parameters.

Linear discriminant analysis does address each of these points and is the go-to linear method for multiclass classification problems. Even with binary-classification problems, it is a good idea to try both logistic regression and linear discriminant analysis.

15.2 Representation of LDA Models

The representation of LDA is pretty straight forward. It consists of statistical properties of your data, calculated for each class. For a single input variable (x) this is the mean and the variance of the variable for each class.

For multiple variables, this is same properties calculated over the multivariate Gaussian, namely the means and the covariance matrix (this is a multi-dimensional generalization of variance). These statistical properties are estimated from your data and plug into the LDA equation to make predictions. These are the model values that you would save to file for your model. Let's look at how these parameters are estimated.

15.3 Learning LDA Models

LDA makes some simplifying assumptions about your data

- That your data is Gaussian, that each variable is is shaped like a bell curve when plotted.
- That each attribute has the same variance, that values of each variable vary around the mean by the same amount on average.

With these assumptions, the LDA model estimates the mean and variance from your data for each class. It is easy to think about this in the univariate (single input variable) case with two classes. The mean ($mean$) value of each input (x) for each class (k) can be estimated in the normal way by dividing the sum of values by the total number of values.

$$mean_k = \frac{1}{n_k} \times \sum_{i=1}^{n} x_i \tag{15.1}$$

Where $mean_k$ is the mean value of x for the class k, n_k is the number of instances with class k. The variance is calculated across all classes as the average squared difference of each value from the mean.

$$sigma^2 = \frac{1}{n - K} \times \sum_{i=1}^{n} (x_i - mean_k)^2 \tag{15.2}$$

Where $sigma^2$ is the variance across all inputs (x), n is the number of instances, K is the number of classes and $mean_k$ is the mean of x for the class to which x_i belongs. Put another way, we calculate the squared difference of each value from the mean within the class groups but average these differences across all class groups. Remember that that when we talk about variance ($variance^2$, $sigma^2$ or σ^2) that the units are squared units, not that we need to square the variance value.

15.4 Making Predictions with LDA

LDA makes predictions by estimating the probability that a new set of inputs belongs to each class. The class that gets the highest probability is the output class and a prediction is made. The model uses Bayes Theorem to estimate the probabilities. Briefly Bayes Theorem can be

used to estimate the probability of the output class (k) given the input (x) using the probability of each class and the probability of the data belonging to each class:

$$P(Y = k|X = x) = \frac{P(k) \times P(x|k)}{\sum_{l=1}^{K}(P(l) \times P(x|l)} \tag{15.3}$$

Where:

- $P(Y = k|X = x)$ is the probability of the class $Y = k$ given the input data x.
- $P(k)$ is the base probability of a given class k we are considering ($Y = k$), e.g. the ratio of instances with this class in the training dataset.
- $P(x|k)$ is the estimated probability of x belonging to the class k.
- The denominator normalizes across for each class l, e.g. the probability of the class $P(l)$ and the probability of the input given the class $P(x|l)$.

A Gaussian distribution function is can be used to estimate $P(x|k)$. Plugging the Gaussian into the above equation and simplifying we end up with the equation below. It is no longer a probability as we discard some terms. Instead it is called a discriminate function for class k. It is calculated for each class k and the class that has the largest discriminant value will make the output classification ($Y = k$):

$$D_k(x) = x \times \frac{mean_k}{sigma^2} - \frac{{mean_k}^2}{2 \times sigma^2} + ln(P(k)) \tag{15.4}$$

$D_k(x)$ is the discriminate function for class k given input x, the $mean_k$, $sigma^2$ and $P(k)$ are all estimated from your data. The $ln()$ function is the natural logarithm.

15.5 Preparing Data For LDA

This section lists some suggestions you may consider when preparing your data for use with LDA.

- **Classification Problems**. This might go without saying, but LDA is intended for classification problems where the output variable is categorical. LDA supports both binary and multiclass classification.
- **Gaussian Distribution**. The standard implementation of the model assumes a Gaussian distribution of the input variables. Consider reviewing the univariate distributions of each attribute and using transforms to make them more Gaussian-looking (e.g. log and root for exponential distributions and Box-Cox for skewed distributions).
- **Remove Outliers**. Consider removing outliers from your data. These can skew the basic statistics used to separate classes in LDA such the mean and the standard deviation.
- **Same Variance**. LDA assumes that each input variable has the same variance. It almost always a good idea to standardize your data before using LDA so that it has a mean of 0 and a standard deviation of 1.

15.6 Extensions to LDA

Linear Discriminant Analysis is a simple and effective method for classification. Because it is simple and so well understood, there are many extensions and variations to the method. Some popular extensions include:

- Quadratic Discriminant Analysis: Each class uses its own estimate of variance (or covariance when there are multiple input variables).
- Flexible Discriminant Analysis: Where nonlinear combination of inputs is used such as splines.
- Regularized Discriminant Analysis: Introduces regularization into the estimate of the variance (or covariance), moderating the influence of different variables on LDA.

The original development was called the Linear Discriminant or Fisher's Discriminant Analysis. The multiclass version was referred to Multiple Discriminant Analysis. These are all simply referred to as Linear Discriminant Analysis now.

15.7 Summary

In this chapter you discovered Linear Discriminant Analysis for classification predictive modeling problems. You learned:

- The model representation for LDA and what is actually distinct about a learned model.
- How the parameters of the LDA model can be estimated from training data.
- How the model can be used to make predictions on new data.
- How to prepare your data to get the most from the method.

You now know about the linear discriminant analysis algorithm for classification. In the next chapter you will discover how to implement the LDA algorithm from scratch for classification.

Chapter 16

Linear Discriminant Analysis Tutorial

Linear Discriminant Analysis is a linear method for classification predictive modeling problems. In this chapter you will discover how Linear Discriminant Analysis (LDA) works by implementing the algorithm step-by-step from scratch for a simple dataset. After reading this chapter you will know:

- The assumptions made by LDA about your training data.
- How to calculate statistics required by the LDA model from your data.
- How to make predictions using the learned LDA model.

Let's get started.

16.1 Tutorial Overview

We are going to step through how to calculate an LDA model for simple dataset with one input and one output variable. This is the simplest case for LDA. This tutorial will to cover:

1. **Dataset**: Introduce the dataset that we are going to model. We will use the same dataset as the training and the test dataset.
2. **Learning the Model**: How to learn the LDA model from the dataset including all of the statistics needed to make predictions.
3. **Making Predictions**: How to use the learned model to make predictions for each instance in the training dataset.

16.2 Tutorial Dataset

LDA makes some assumptions about your data:

- The input variables have a Gaussian (bell curve) distribution.
- The variance (average squared difference from the mean) calculated for each input variables by class grouping is the same.

- That the mix of classes in your training set is representative of the problem.

Below is a contrived simple two-dimensional dataset containing the input variable X and the output class variable Y. All values for X were drawn from a Gaussian distribution and the class variable Y has the value 0 or 1. The instances in the two classes were separated to make the prediction problem simpler. All instances in class 0 were drawn from a Gaussian distribution with a mean of 5 and a standard deviation of 1. All instances in class 1 were drawn from a Gaussian distribution with a mean of 20 and a standard deviation of 1.

The classes do not interact and should be separable with a linear model like LDA. It is also handy to know the actual statistical properties of the data because we can generate more test instances later to see how well LDA has learned the model. Below is the complete dataset.

```
X             Y
4.667797637   0
5.509198779   0
4.702791608   0
5.956706641   0
5.738622413   0
5.027283325   0
4.805434058   0
4.425689143   0
5.009368635   0
5.116718815   0
6.370917709   0
2.895041947   0
4.666842365   0
5.602154638   0
4.902797978   0
5.032652964   0
4.083972925   0
4.875524106   0
4.732801047   0
5.385993407   0
20.74393514   1
21.41752855   1
20.57924186   1
20.7386947    1
19.44605384   1
18.36360265   1
19.90363232   1
19.10870851   1
18.18787593   1
19.71767611   1
19.09629027   1
20.52741312   1
20.63205608   1
19.86218119   1
21.34670569   1
20.333906     1
21.02714855   1
18.27536089   1
21.77371156   1
20.65953546   1
```

Listing 16.1: LDA Tutorial Data Set.

Below is a plot of the dataset, showing the separation of the two classes.

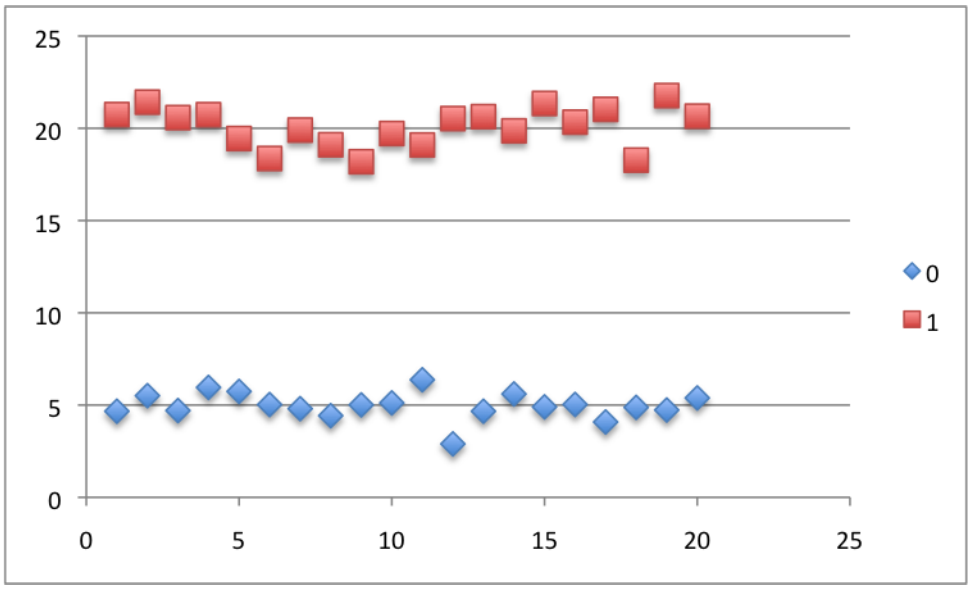

Figure 16.1: LDA Tutorial dataset.

16.3 Learning The Model

The LDA model requires the estimation of statistics from the training data:

1. Mean of each input value for each class
2. Probability of an instance belong to each class.
3. Covariance for the input data for each class.

16.3.1 Calculate the Class Means

The mean can be calculated for each class using:

$$mean(x) = \frac{1}{n} \times \sum_{i=1}^{n} x_i \tag{16.1}$$

Where x are the input values for a class and n is the total number of input values. You can use the `AVERAGE()` function in you spreadsheet. The mean value of x for each class is:

- $Y = 0$: 4.975415507
- $Y = 1$ 1: 20.06447458

16.3.2 Calculate the Class Probabilities

Next we need to estimate the probability that a given instance will belong to $Y = 0$ or $Y = 1$. This can be calculated as:

$$\begin{aligned} P(y = 0) &= \frac{count(y = 0)}{count(y = 0) + count(y = 1)} \\ P(y = 1) &= \frac{count(y = 1)}{count(y = 0) + count(y = 1)} \end{aligned} \tag{16.2}$$

or

$$\begin{aligned} P(y = 0) &= \frac{20}{20 + 20} \\ P(y = 1) &= \frac{20}{20 + 20} \end{aligned} \tag{16.3}$$

This is 0.5 for each class. We knew this already because we created the dataset, but it is a good idea to work through each step of the model learning process.

16.3.3 Calculate the Variance

Now we need to calculate the variance for the input variable for each class. You can understand the variance as the difference of each instance from the mean. The difference is squared so the variance is often written to include these units. It does not mean you need to square the variance value when using it. We can calculate the variance for our dataset in two steps:

1. Calculate the squared difference for each input variable from the group mean.
2. Calculate the mean of the squared difference.

We can divide the dataset into two groups by the Y values 0 and 1. We can then calculate the difference for each input value X from the mean from that group. We can calculate the difference of each input value from the mean using:

$$SquaredDifference = (x - mean_k)^2 \tag{16.4}$$

Where $mean_k$ is the mean value of x for the class k to which x belongs. We can sum those values which gives us (rounded to 4 places):

- $Y = 0$: 10.15823013
- $Y = 1$: 21.49316708

Next we can calculate the variance as the average squared difference from the mean as:

$$\begin{aligned} variance &= \frac{1}{count(x) - count(classes)} \times \sum_{i=1}^{n} SquaredDifference(x_i) \\ variance &= \frac{1}{20 - 2} \times (10.15823013 + 21.49316708) \\ variance &= 0.832931506 \end{aligned} \tag{16.5}$$

Notice that this is slightly difference from using the mean above as we need to subtract two degrees of freedom for the two class values or two groups we are using. The explanation for exactly why this is the case is a beyond this tutorial.

16.4 Making Predictions

We are now ready to make predictions. Predictions are made by calculating the discriminant function for each class and predicting the class with the largest value. The discriminant function for a class given an input (x) is calculated using:

$$discriminant(x) = x \times \frac{mean}{variance} - \frac{mean^2}{2 \times variance} + ln(probability) \tag{16.6}$$

Where x is the input value, mean, variance and probability are calculated above for the class we are discriminating. After calculating the discriminant value for each class, the class with the largest discriminant value is taken as the prediction. Let's step through the calculation of the discriminate value of each class for the first instance. The first instance in the dataset is: $X = 4.667797637$ and $Y = 0$. The discriminant value for $Y = 0$ is calculated as follows:

$$\begin{aligned} discriminant(Y = 0|x) &= 4.667797637 \times \frac{4.975415507}{0.832931506} - \frac{4.975415507^2}{2 \times 0.832931506} + ln(0.5) \\ discriminant(Y = 0|x) &= 12.3293558 \end{aligned} \tag{16.7}$$

We can also calculate the discriminant value for $Y = 1$:

$$\begin{aligned} discriminant(Y = 1|x) &= 4.667797637 \times \frac{20.08706292}{0.832931506} - \frac{20.08706292^2}{2 \times 0.832931506} + ln(0.5) \\ discriminant(Y = 1|x) &= -130.3349038 \end{aligned} \tag{16.8}$$

We can see that the discriminant value for $Y = 0$ (12.3293558) is larger than the discriminate value for $Y = 1$ (-130.3349038), therefore the model predicts $Y = 0$. Which we know is correct. You can proceed to calculate the Y values for each instance in the dataset, as follows:

X	Disc. Y=0	Disc. Y=1	Prediction
4.667797637	12.3293558	-130.3349038	0
5.509198779	17.35536365	-110.0435863	0
4.702791608	12.53838805	-129.4909856	0
5.956706641	20.02849769	-99.25144007	0
5.738622413	18.72579795	-104.510782	0
5.027283325	14.47670003	-121.6655095	0
4.805434058	13.15151029	-127.0156496	0
4.425689143	10.88315002	-136.1736175	0
5.009368635	14.3696888	-122.0975421	0
5.116718815	15.0109321	-119.5086739	0
6.370917709	22.50273735	-89.26228283	0
2.895041947	1.740014309	-173.0868646	0
4.666842365	12.3236496	-130.3579413	0
5.602154638	17.91062422	-107.8018531	0
4.902797978	13.73310188	-124.6676111	0

```
5.032652964    14.50877492    -121.5360148   0
4.083972925    8.841949563    -144.4144814   0
4.875524106    13.57018471    -125.3253507   0
4.732801047    12.7176458     -128.7672748   0
5.385993407    16.61941128    -113.0148198   0
20.74393514    108.3582168    257.3589021    1
21.41752855    112.3818455    273.603351     1
20.57924186    107.3744415    253.3871419    1
20.7386947     108.3269137    257.2325231    1
19.44605384    100.60548      226.0590616    1
18.36360265    94.1395889     199.954556     1
19.90363232    103.3387697    237.0940718    1
19.10870851    98.59038855    217.9236065    1
18.18787593    93.08990662    195.7167121    1
19.71767611    102.2279828    232.6095325    1
19.09629027    98.5162097     217.6241269    1
20.52741312    107.0648488    252.1372346    1
20.63205608    107.6899208    254.6608151    1
19.86218119    103.0911664    236.0944321    1
21.34670569    111.9587938    271.8953795    1
20.333906      105.9089574    247.4705967    1
21.02714855    110.0499579    264.1889062    1
18.27536089    93.61248743    197.8265085    1
21.77371156    114.5094616    282.1930975    1
20.65953546    107.8540656    255.3235107    1
```

Listing 16.2: LDA Predictions for the dataset.

If you compare the predictions to the dataset, you can see that LDA has achieved an accuracy of 100% (no errors). This is not surprising given that the dataset was contrived so that the groups for $Y = 0$ and $Y = 1$ were clearly separable.

16.5 Summary

In this chapter you discovered how the LDA algorithm works step-by-step with a simple worked example. You learned:

- How to calculate the statistics from your dataset required by the LDA model.
- How to use the LDA model to calculate a discriminant value for each class and make a prediction.

You now know how to implement the linear discriminant analysis algorithm from scratch for classification. This concludes your introduction to linear machine learning algorithms. In the next part you will discover nonlinear machine learning algorithms, starting with decision trees.

Part IV

Nonlinear Algorithms

Chapter 17

Classification and Regression Trees

Decision Trees are an important type of algorithm for predictive modeling machine learning. The classical decision tree algorithms have been around for decades and modern variations like random forest are among the most powerful techniques available. In this chapter you will discover the humble decision tree algorithm known by it's more modern name CART which stands for Classification And Regression Trees. After reading this chapter, you will know:

- The many names used to describe the CART algorithm for machine learning.
- The representation used by learned CART models that is actually stored on disk.
- How a CART model can be learned from training data.
- How a learned CART model can be used to make predictions on unseen data.
- Additional resources that you can use to learn more about CART and related algorithms.

If you have taken an algorithms and data structures course, it might be hard to hold you back from implementing this simple and powerful algorithm. And from there, you're a small step away from your own implementation of Random Forests. Let's get started.

17.1 Decision Trees

Classification and Regression Trees or CART for short is a term introduced by Leo Breiman to refer to Decision Tree algorithms that can be used for classification or regression predictive modeling problems. Classically, this algorithm is referred to as *decision trees*, but on some platforms like R they are referred to by the more modern term CART. The CART algorithm provides a foundation for important algorithms like bagged decision trees, random forest and boosted decision trees.

17.2 CART Model Representation

The representation for the CART model is a binary tree. This is your binary tree from algorithms and data structures, nothing too fancy. Each node represents a single input variable (x) and a split point on that variable (assuming the variable is numeric). The leaf nodes of the tree

contain an output variable (y) which is used to make a prediction. Given a dataset with two inputs of height in centimeters and weight in kilograms the output of sex as male or female, below is a crude example of a binary decision tree (completely fictitious for demonstration purposes only).

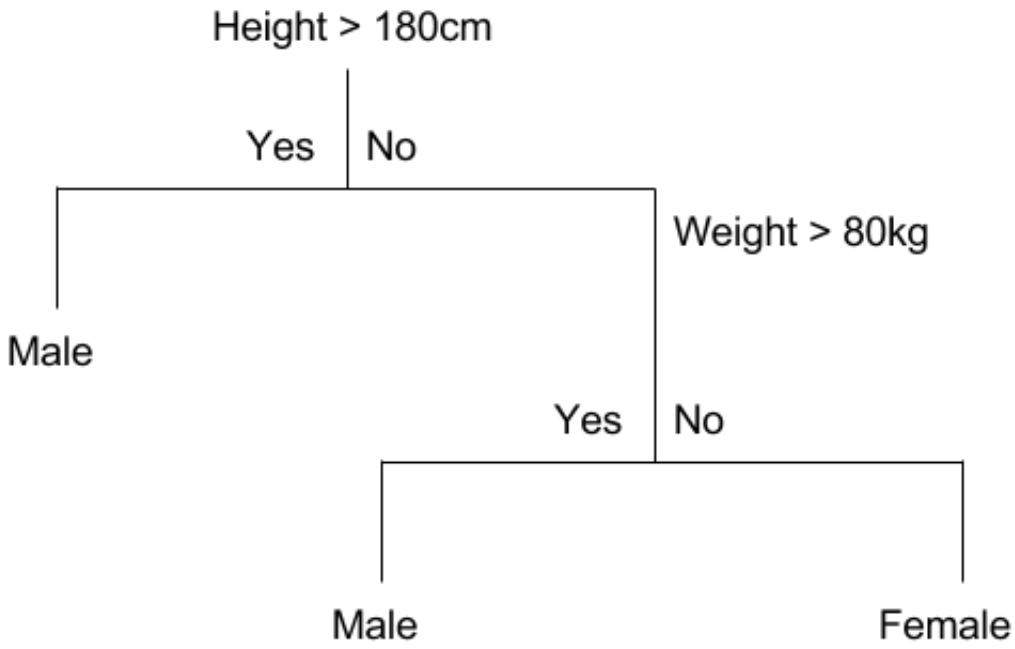

Figure 17.1: Example Decision Tree.

The tree can be stored to file as a graph or a set of rules. For example, below is the above decision tree as a set of rules.

```
If Height > 180 cm Then Male
If Height <= 180 cm AND Weight > 80 kg Then Male
If Height <= 180 cm AND Weight <= 80 kg Then Female
Make Predictions With CART Models
```

Listing 17.1: Example of a Rule Representation of a Decision Tree.

17.3 Making Predictions

With the binary tree representation of the CART model described above, making predictions is relatively straightforward. Given a new input, the tree is traversed by evaluating the specific input started at the root node of the tree. A learned binary tree is actually a partitioning of the input space. You can think of each input variable as a dimension on an p-dimensional space. The decision tree split this up into rectangles (when $p = 2$ input variables) or hyper-rectangles with more inputs. New data is filtered through the tree and lands in one of the rectangles and

the output value for that rectangle is the prediction made by the model. This gives you some feeling for the type of decisions that a CART model is capable of making, e.g. boxy decision boundaries. For example, given the input of $height = 160$ cm and $weight = 65$ kg, we would traverse the above tree as follows:

1. Height > 180 cm: No
2. Weight > 80 kg: No
3. Therefore: Female

17.4 Learn a CART Model From Data

Creating a binary decision tree is actually a process of dividing up the input space. A greedy approach is used to divide the space 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 (e.g. the very best split point is chosen each time). For regression predictive modeling problems the cost function that is minimized to choose spit points is the sum squared error across all training samples that fall within the rectangle:

$$\sum_{i=1}^{n}(y_i - prediction_i)^2 \tag{17.1}$$

Where y is the output for the training sample and prediction is the predicted output for the rectangle. For classification the Gini cost function is used which provides an indication of how *pure* the leaf nodes are (how mixed the training data assigned to each node is).

$$G = \sum_{k=1}^{n} p_k \times (1 - p_k) \tag{17.2}$$

Where G is the Gini cost over all classes, p_k are the number of training instances with class k in the rectangle of interest. A node that has all classes of the same type (perfect class purity) will have $G = 0$, where as a G that has a 50-50 split of classes for a binary classification problem (worst purity) will have a $G = 0.5$.

17.4.1 Stopping Criterion

The recursive binary splitting procedure described above needs to know when to stop splitting as it works its way down the tree with the training data. The most common stopping procedure is to use a minimum count on the number of training instances assigned to each leaf node. If the count is less than some minimum then the split is not accepted and the node is taken as a final leaf node. The count of training members is tuned to the dataset, e.g. 5 or 10. It defines how specific to the training data the tree will be. Too specific (e.g. a count of 1) and the tree will overfit the training data and likely have poor performance on the test set.

17.4.2 Pruning The Tree

The stopping criterion is important as it strongly influences the performance of your tree. You can use pruning after learning your tree to further lift performance. The complexity of a decision tree is defined as the number of splits in the tree. Simpler trees are preferred. They are easy to understand (you can print them out and show them to subject matter experts), and they are less likely to overfit your data.

The fastest and simplest pruning method is to work through each leaf node in the tree and evaluate the effect of removing it using a hold-out test set. Leaf nodes are removed only if it results in a drop in the overall cost function on the entire test set. You stop removing nodes when no further improvements can be made. More sophisticated pruning methods can be used such as cost complexity pruning (also called weakest link pruning) where a learning parameter (*alpha*) is used to weigh whether nodes can be removed based on the size of the sub-tree.

17.5 Preparing Data For CART

CART does not require any special data preparation other than a good representation of the problem.

17.6 Summary

In this chapter you have discovered the Classification And Regression Trees (CART) for machine learning. You learned:

- The classical name Decision Tree and the more Modern name CART for the algorithm.
- The representation used for CART is a binary tree.
- Predictions are made with CART by traversing the binary tree given a new input record.
- The tree is learned using a greedy algorithm on the training data to pick splits in the tree
- Stopping criteria define how much a tree learns and pruning can be used to improve generalization on a learned tree.

You now know about the Classification and Regression Trees machine learning algorithm for classification and regression. In the next chapter you will discover how you can implement CART from scratch.

Chapter 18

Classification and Regression Trees Tutorial

Decision trees are a flexible and very powerful machine learning method for regression and classification predictive modeling problems. In this chapter you will discover how to implement the CART machine learning algorithm from scratch step-by-step. After completing this chapter you will know:

- How to calculate the Gini index for a given split in a decision tree.
- How to evaluate different split points when constructing a decision tree.
- How to make predictions on new data with a learned decision tree.

Let's get started.

18.1 Tutorial Dataset

In this tutorial we will work through a simple binary (two-class) classification problem for CART. To keep things simple we will work with a two input variables ($X1$ and $X2$) and a single output variable (Y). This is not a real problem but a contrived problem to demonstrate how to implement the CART model and make predictions. The example was designed so that the algorithm will find at least two split points in order to best classify the training dataset. The raw data for this problem is as follows:

```
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 18.1: CART Tutorial Data Set.

When visualized as a two-dimensional scatter plot the dataset looks as follows:

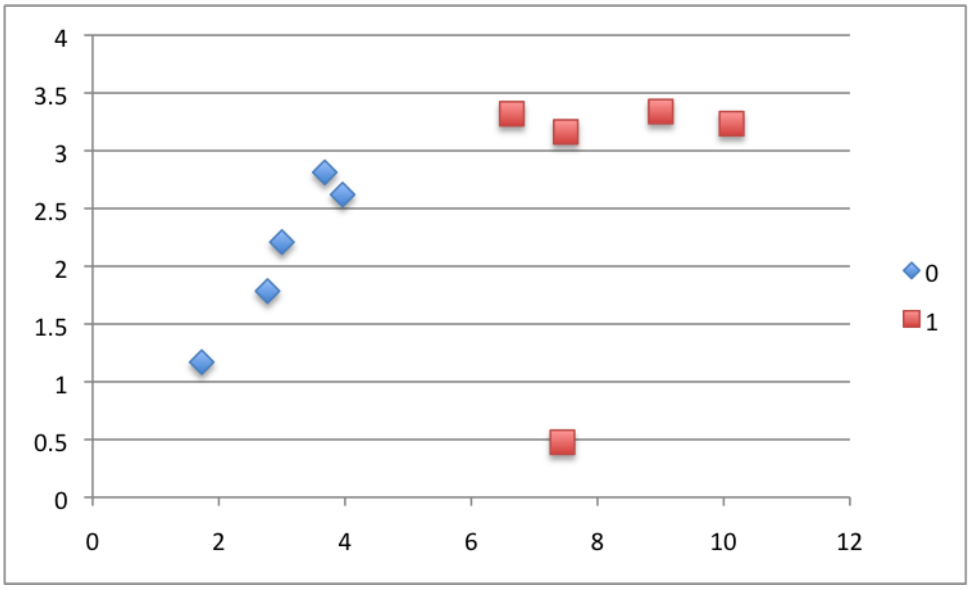

Figure 18.1: Classification And Regression Trees Tutorial Dataset.

18.2 Learning a CART Model

The CART model is learned by looking for split points in the data. A split point is a single value of a single attribute, e.g. the first value of the $X1$ attribute 2.771244718. Partitioning data at a split point involves separating all data at that node into two groups, left of the split point and right of the split point. If we are working on the first split point in the tree, then all of the dataset is affected. If we are working on say a split point one level deep, then only the data that has filtered down the tree from nodes above and is sitting at that node is affected by the split point.

We are not concerned with what the class value is of the chosen split point. We only care about the composition of the data assigned to the LEFT and to the RIGHT child nodes of the split point. A cost function is used to evaluate the mix of classes of training data assigned to each side of the split. In classification problems the Gini index cost function is used.

18.2.1 Gini Index Cost Function

We calculate the Gini index for a split point as follows:

$$Gini(split) = \sum_{k=1}^{n}(p_k \times (1 - p_k)) \tag{18.1}$$

For each class (k), for each group (left and right). Where p is the proportion of training instances with a given class in a given group. We will always have two groups, a left and right group because we are using a binary tree. And we know from our dataset that we only have two classes. Therefore we can calculate the Gini index of any split point in our dataset as the sum of:

$$\begin{aligned} Gini(split) = &(left(0) \times (1 - left(0))) + (right(0) \times (1 - right(0)) + \\ &(left(1) \times (1 - left(1))) + (right(1) \times (1 - right(1)) \end{aligned} \tag{18.2}$$

Where left(0) is the proportion of data instances in the left group with class 0, right(0) is the proportion of data instances in the right group with class 0, and so on. The proportion of data instances of a class is easy to calculate. If a LEFT group has 3 instances with class 0 and 4 instances with class 1, then the proportion of data instances with class 0 would be $\frac{3}{7}$ or 0.428571429. To get a feeling for Gini index scores, take a look at the table below. It provides 7 different scenarios for mixes of 0 and 1 classes in a single group.

```
Class 0    Class 1    Count   Class 0/Count     Class 1/Count     Gini
10         10         20      0.5               0.5               0.5
19         1          20      0.95              0.05              0.095
1          19         20      0.05              0.95              0.095
15         5          20      0.75              0.25              0.375
5          15         20      0.25              0.75              0.375
11         9          20      0.55              0.45              0.495
20         0          20      1                 0                 0
```

Listing 18.2: Sample Gini calculations.

It is easy to visualize these scenarios for one group, but the concepts translate to summing the proportions across the LEFT and RIGHT groups. You can see when the group has a 50-50 mix in the first row, that Gini is 0.5. This is the worst possible split. You can also see a case where the group only has data instances with class 0 on the last row and a Gini index of 0. This is an example of a perfect split. Our goal in selecting a split point is to evaluate the Gini index of all possible split points and greedily select the split point with the lowest cost. Let's make this more concrete by calculating the cost of selecting different data points as our split point.

18.2.2 First Candidate Split Point

The first step is to choose a split that will become the stump or root node of our decision tree. We will start with the first candidate split point which is the $X1$ attribute and the value of $X1$ in the first instance: $X1 = 2.771244718$.

- **IF** $X1 < 2.771244718$ **THEN** LEFT
- **IF** $X1 \geq 2.771244718$ **THEN** RIGHT

Let's apply this rule to each $X1$ value in our training dataset. Below is the answer we get for each numbered instance in the dataset:

```
X1             Y   Group
2.771244718    0   RIGHT
1.728571309    0   LEFT
3.678319846    0   RIGHT
3.961043357    0   RIGHT
2.999208922    0   RIGHT
7.497545867    1   RIGHT
9.00220326     1   RIGHT
7.444542326    1   RIGHT
10.12493903    1   RIGHT
6.642287351    1   RIGHT
```

Listing 18.3: Separation of Training Dataset by Candidate Split.

How good was this split? We can evaluate the mixture of the classes in each of the LEFT and RIGHT nodes as a single cost of choosing this split point for our root node. The LEFT group only has one member, where as the RIGHT group has 9 members. Starting with the LEFT group, we can calculate the proportion of training instances that have each class:

- $Y = 0$: $\frac{1}{1}$ or 1.0
- $Y = 1$: $\frac{0}{1}$ or 0.0

The RIGHT group is more interesting as it is comprised of a mix of classes (we are probably going to get a high Gini index).

- $Y = 0$: $\frac{7}{9}$ or 0.444444444
- $Y = 1$: $\frac{5}{9}$ or 0.555555556

We now have enough information to calculate the Gini index for this split:

$$\begin{aligned} Gini(X1 = 2.7712) = &(1.0 \times (1 - 1.0)) \ + (0.0 \times (1 - 0.0)) + \\ &(0.444444444 \times (1 - 0.444444444) + \\ &(0.555555556 \times (1 - 0.555555556)) \end{aligned} \tag{18.3}$$

or

$$Gini(X1 = 2.771244718) = 0.49382716 \tag{18.4}$$

18.2.3 Best Candidate Split Point

We can evaluate each candidate split point using the process above with the values from $X1$ and $X2$. If we look at the graph of the data, we can see that we can probably draw a vertical line to separate the classes. This would translate to a split point for $X1$ with a value around 0.5. A close fit would be the value for $X1$ in the last instance: $X1 = 6.642287351$.

- **IF** $X1 < 6.642287351$ **THEN** LEFT
- **IF** $X1 \geq 6.642287351$ **THEN** RIGHT

Let's apply this rule to all instances, below we get the assigned group for each numbered data instance:

```
X1            Y   Group
2.771244718   0   LEFT
1.728571309   0   LEFT
3.678319846   0   LEFT
3.961043357   0   LEFT
2.999208922   0   LEFT
7.497545867   1   RIGHT
9.00220326    1   RIGHT
7.444542326   1   RIGHT
10.12493903   1   RIGHT
6.642287351   1   RIGHT
```

Listing 18.4: Separation of Training Dataset by Bets Split.

There are 5 instances in each group, this looks like a good split. Starting with the LEFT group, we can calculate the proportion of training instances that have each class:

- $Y = 0$: $\frac{5}{5}$ or 1.0
- $Y = 1$: $\frac{0}{5}$ or 0.0

The RIGHT group has the opposite proportions.

- $Y = 0$: $\frac{0}{5}$ or 0.0
- $Y = 1$: $\frac{5}{5}$ or 1.0

We now have enough information to calculate the Gini index for this split:

$$\begin{aligned} Gini(X1 = 6.642287351) = &(1.0 \times (1 - 1.0)) + \\ &(0.0 \times (1 - 0.0)) + \\ &(1.0 \times (1 - 1.0)) + \\ &(0.0 \times (1 - 0.0)) \end{aligned} \tag{18.5}$$

or

$$Gini(X1 = 6.642287351) = 0.0 \tag{18.6}$$

This is a split that results in a pure Gini index because the classes are perfectly separated. The LEFT child node will classify instances as class 0 and the right as class 1. We can stop there. You can see how this process could be repeated for each child node to build up a more complicated tree for a more challenging dataset.

18.3 Making Predictions on Data

We can now use this decision tree to make some predictions for all of the training instances. But we have already done that when we calculated the Gini index above. Instead, let's classify some new data generated for each class using the same distribution. Here is the test dataset:

```
X1            X2            Y
2.343875381   2.051757824   0
3.536904049   3.032932531   0
2.801395588   2.786327755   0
3.656342926   2.581460765   0
2.853194386   1.052331062   0
8.907647835   3.730540859   1
9.752464513   3.740754624   1
8.016361622   3.013408249   1
6.58490395    2.436333477   1
7.142525173   3.650120799   1
```

Listing 18.5: Test Dataset.

Using the decision tree with a single split at, $X1 = 6.642287351$ we can classify the test instances as follows:

```
Y   Prediction
0   0
0   0
0   0
0   0
0   0
1   1
1   1
1   1
1   0
1   1
```

Listing 18.6: Predictions for Test Dataset.

Again, this is a perfect classification or 100% accurate.

18.4 Summary

In this chapter you discovered exactly how to construct a CART model and use it to make predictions. You learned how to:

- Calculate the Gini index for a candidate split in a decision tree.
- Evaluate any candidate split points using Gini index.
- How to create a decision tree from scratch to make predictions.
- How to make predictions on new data using a learned decision tree.

You now know how to implement CART from scratch. In the next chapter in you will discover the Naive Bayes machine learning algorithm for classification.

Chapter 19

Naive Bayes

Naive Bayes is a simple but surprisingly powerful algorithm for predictive modeling. In this chapter you will discover the Naive Bayes algorithm for classification. After reading this chapter, you will know:

- The representation used by naive Bayes that is actually stored when a model is written to a file.
- How a learned model can be used to make predictions.
- How you can learn a naive Bayes model from training data.
- How to best prepare your data for the naive Bayes algorithm.
- Where to go for more information on naive Bayes.

Let's get started.

19.1 Quick Introduction to Bayes' Theorem

In machine learning we are often interested in selecting the best hypothesis (h) given data (d). In a classification problem, our hypothesis (h) may be the class to assign for a new data instance (d). One of the easiest ways of selecting the most probable hypothesis given the data that we have that we can use as our prior knowledge about the problem. Bayes' Theorem provides a way that we can calculate the probability of a hypothesis given our prior knowledge. Bayes' Theorem is stated as:

$$P(h|d) = \frac{P(d|h) \times P(h)}{P(d)} \tag{19.1}$$

Where:

- $P(h|d)$ is the probability of hypothesis h given the data d. This is called the posterior probability.
- $P(d|h)$ is the probability of data d given that the hypothesis h was true.

- $P(h)$ is the probability of hypothesis h being true (regardless of the data). This is called the prior probability of h.
- $P(d)$ is the probability of the data (regardless of the hypothesis).

You can see that we are interested in calculating the posterior probability of $P(h|d)$ from the prior probability $p(h)$ with $P(D)$ and $P(d|h)$. After calculating the posterior probability for a number of different hypotheses, you can select the hypothesis with the highest probability. This is the maximum probable hypothesis and may formally be called the maximum a posteriori (MAP) hypothesis. This can be written as:

$$\begin{aligned} MAP(h) &= max(P(h|d)) \\ MAP(h) &= max(\frac{P(d|h) \times P(h)}{P(d)}) \\ MAP(h) &= max(P(d|h) \times P(h)) \end{aligned} \tag{19.2}$$

The $P(d)$ is a normalizing term which allows us to calculate the probability. We can drop it when we are interested in the most probable hypothesis as it is constant and only used to normalize. Back to classification, if we have an even number of instances in each class in our training data, then the probability of each class (e.g. $P(h)$) will be the same value for each class (e.g. 0.5 for a 50-50 split). Again, this would be a constant term in our equation and we could drop it so that we end up with:

$$MAP(h) = max(P(d|h)) \tag{19.3}$$

This is a useful exercise, because when reading up further on Naive Bayes you may see all of these forms of the theorem.

19.2 Naive Bayes Classifier

Naive Bayes is a classification algorithm for binary (two-class) and multiclass classification problems. The technique is easiest to understand when described using binary or categorical input values. It is called naive Bayes or idiot Bayes because the calculation of the probabilities for each hypothesis are simplified to make their calculation tractable. Rather than attempting to calculate the values of each attribute value $P(d1, d2, d3|h)$, they are assumed to be conditionally independent given the target value and calculated as $P(d1|h) \times P(d2|h)$ and so on. 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.

19.2.1 Representation Used By Naive Bayes Models

The representation for naive Bayes is probabilities. A list of probabilities is stored to file for a learned naive Bayes model. This includes:

- Class Probabilities: The probabilities of each class in the training dataset.
- Conditional Probabilities: The conditional probabilities of each input value given each class value.

19.2.2 Learn a Naive Bayes Model From Data

Learning a naive Bayes model from your training data is fast. Training is fast because only the probability of each class and the probability of each class given different input (x) values need to be calculated. No coefficients need to be fitted by optimization procedures.

Calculating Class Probabilities

The class probabilities are simply the frequency of instances that belong to each class divided by the total number of instances. For example in a binary classification the probability of an instance belonging to class 1 would be calculated as:

$$P(class = 1) = \frac{count(class = 1)}{count(class = 0) + count(class = 1)} \tag{19.4}$$

In the simplest case each class would have the probability of 0.5 or 50% for a binary classification problem with the same number of instances in each class.

Calculating Conditional Probabilities

The conditional probabilities are the frequency of each attribute value for a given class value divided by the frequency of instances with that class value. For example, if a `weather` attribute had the values `sunny` and `rainy` and the class attribute had the class values `go-out` and `stay-home`, then the conditional probabilities of each weather value for each class value could be calculated as:

$$\begin{aligned}
P(weather = sunny|class = \text{go-out}) &= \frac{count(weather = sunny \wedge \text{class=go-out})}{count(\text{class=go-out})} \\
P(weather = sunny|class = \text{stay-home}) &= \frac{count(weather = sunny \wedge class = \text{stay-home})}{count(class = \text{stay-home})} \\
P(weather = rainy|class = \text{go-out}) &= \frac{count(weather = rainy \wedge class = \text{go-out})}{count(class = \text{go-out})} \\
P(weather = rainy|class = \text{stay-home}) &= \frac{count(weather = rainy \wedge class = \text{stay-home})}{count(class = \text{stay-home})}
\end{aligned} \tag{19.5}$$

Where $\wedge$ means conjunction (and).

19.2.3 Make Predictions With a Naive Bayes Model

Given a naive Bayes model, you can make predictions for new data using Bayes theorem.

$$MAP(h) = max(P(d|h) \times P(h)) \tag{19.6}$$

Using our example above, if we had a new instance with the weather of `sunny`, we can calculate:

$$\begin{aligned}
\text{go-out} &= P(weather = sunny|class = \text{go-out}) \times P(class = \text{go-out}) \\
\text{stay-home} &= P(weather = sunny|class = \text{stay-home}) \times P(class = \text{stay-home})
\end{aligned} \tag{19.7}$$

We can choose the class that has the largest calculated value. We can turn these values into probabilities by normalizing them as follows:

$$\begin{aligned} P(\text{go-out}|weather = sunny) &= \frac{\text{go-out}}{\text{go-out} + \text{stay-home}} \\ P(\text{stay-home}|weather = sunny) &= \frac{\text{stay-home}}{\text{go-out} + \text{stay-home}} \end{aligned} \tag{19.8}$$

If we had more input variables we could extend the above example. For example, pretend we have a `car` attribute with the values `working` and `broken`. We can multiply this probability into the equation. For example below is the calculation for the `go-out` class label with the addition of the car input variable set to `working`:

$$\begin{aligned} \text{go-out} = &P(weather = sunny|class = \text{go-out})\times \\ &P(car = working|class = \text{go-out})\times \\ &P(class = \text{go-out}) \end{aligned} \tag{19.9}$$

19.3 Gaussian Naive Bayes

Naive Bayes can be extended to real-valued attributes, most commonly by assuming a Gaussian distribution. This extension of naive Bayes is called Gaussian Naive Bayes. Other functions can be used to estimate the distribution of the data, but the Gaussian (or Normal distribution) is the easiest to work with because you only need to estimate the mean and the standard deviation from your training data.

19.3.1 Representation for Gaussian Naive Bayes

Above, we calculated the probabilities for input values for each class using a frequency. With real-valued inputs, we can calculate the mean and standard deviation of input values (x) for each class to summarize the distribution. This means that in addition to the probabilities for each class, we must also store the mean and standard deviations for each input variable for each class.

19.3.2 Learn a Gaussian Naive Bayes Model From Data

This is as simple as calculating the mean and standard deviation values of each input variable (x) for each class value.

$$mean(x) = \frac{1}{n} \times \sum_{i=1}^{n} x_i \tag{19.10}$$

Where n is the number of instances and x are the values for an input variable in your training data. We can calculate the standard deviation using the following equation:

$$StandardDeviation(x) = \sqrt{\frac{1}{n} \times \sum_{i=1}^{n} (x_i - mean(x))^2} \tag{19.11}$$

This is the square root of the average squared difference of each value of x from the mean value of x, where n is the number of instances, x_i is a specific value of the x variable for the i'th instance and $mean(x)$ is described above.

19.3.3 Make Predictions With a Gaussian Naive Bayes Model

Probabilities of new x values are calculated using the Gaussian Probability Density Function (PDF). When making predictions these parameters can be plugged into the Gaussian PDF with a new input for the variable, and in return the Gaussian PDF will provide an estimate of the probability of that new input value for that class.

$$pdf(x, mean, sd) = \frac{1}{\sqrt{2 \times \pi} \times sd} \times e^{-(\frac{(x-mean)^2}{2 \times sd^2})} \tag{19.12}$$

Where $pdf(x)$ is the Gaussian PDF, $mean$ and sd are the mean and standard deviation calculated above, π is the numerical constant PI, e is the numerical constant Euler's number raised to power and x is the input value for the input variable. We can then plug in the probabilities into the equation above to make predictions with real-valued inputs. For example, adapting one of the above calculations with numerical values for weather and car:

$$\begin{aligned} \text{go-out} = & P(pdf(weather)|class = \text{go-out}) \times \\ & P(pdf(car)|class = \text{go-out}) \times \\ & P(class = \text{go-out}) \end{aligned} \tag{19.13}$$

19.4 Preparing Data For Naive Bayes

This section provides some tips for preparing your data for Naive Bayes.

- **Categorical Inputs**: Naive Bayes assumes label attributes such as binary, categorical or nominal.
- **Gaussian Inputs**: If the input variables are real-valued, a Gaussian distribution is assumed. In which case the algorithm will perform better if the univariate distributions of your data are Gaussian or near-Gaussian. This may require removing outliers (e.g. values that are more than 3 or 4 standard deviations from the mean).
- **Classification Problems**: Naive Bayes is a classification algorithm suitable for binary and multiclass classification.
- **Log Probabilities**: The calculation of the likelihood of different class values involves multiplying a lot of small numbers together. This can lead to an underflow of numerical precision. As such it is good practice to use a log transform of the probabilities to avoid this underflow.
- **Kernel Functions**: Rather than assuming a Gaussian distribution for numerical input values, more complex distributions can be used such as a variety of kernel density functions.
- **Update Probabilities**: When new data becomes available, you can simply update the probabilities of your model. This can be helpful if the data changes frequently.

19.5 Summary

In this chapter you discovered the Naive Bayes algorithm for classification. You learned about:

- The Bayes Theorem and how to calculate it in practice.
- Naive Bayes algorithm including representation, making predictions and learning the model.
- The adaptation of Naive Bayes for real-valued input data called Gaussian Naive Bayes.
- How to prepare data for Naive Bayes.

You now know about the Naive Bayes algorithm for classification. In the next chapter you will discover how to implement Naive Bayes from scratch for categorical variables.

Chapter 20

Naive Bayes Tutorial

Naive Bayes is a very simple classification algorithm that makes some strong assumptions about the independence of each input variable. Nevertheless, it has been shown to be effective in a large number of problem domains. In this chapter you will discover the Naive Bayes algorithm for categorical data. After reading this chapter you will know.

- How to work with categorical data for Naive Bayes.
- How to prepare the class and conditional probabilities for a Naive Bayes model.
- How to use a learned Naive Bayes model to make predictions.

Let's get started.

20.1 Tutorial Dataset

The dataset describes two categorical input variables and a class variable that has two outputs.

```
Weather    Car        Class
sunny      working    go-out
rainy      broken     go-out
sunny      working    go-out
sunny      working    go-out
sunny      working    go-out
rainy      broken     stay-home
rainy      broken     stay-home
sunny      working    stay-home
sunny      broken     stay-home
rainy      broken     stay-home
```

Listing 20.1: Naive Bayes Tutorial Data Set.

We can convert this into numbers. Each input has only two values and the output class variable has two values. We can convert each variable to binary as follows:

- Weather: sunny = 1, rainy = 0
- Car: working = 1, broken = 0
- Class: go-out = 1, stay-home = 0

Therefore, we can restate the dataset as:

```
Weather    Car    Class
1          1      1
0          0      1
1          1      1
1          1      1
1          1      1
0          0      0
0          0      0
1          1      0
1          0      0
0          0      0
```

Listing 20.2: Simplified Naive Bayes Tutorial Data Set.

This can make the data easier to work with in a spreadsheet or code if you are following along.

20.2 Learn a Naive Bayes Model

There are two types of quantities that need to be calculated from the dataset for the naive Bayes model:

- Class Probabilities.
- Conditional Probabilities.

Let's start with the class probabilities.

20.2.1 Calculate the Class Probabilities

The dataset is a two class problem and we already know the probability of each class because we contrived the dataset. Nevertheless, we can calculate the class probabilities for classes 0 and 1 as follows:

$$\begin{aligned} P(class = 1) &= \frac{count(class = 1)}{count(class = 0) + count(class = 1)} \\ P(class = 0) &= \frac{count(class = 0)}{count(class = 0) + count(class = 1)} \end{aligned} \tag{20.1}$$

or

$$\begin{aligned} P(class = 1) &= \frac{5}{5 + 5} \\ P(class = 0) &= \frac{5}{5 + 5} \end{aligned} \tag{20.2}$$

This works out to be a probability of 0.5 for any given data instance belonging to class 0 or class 1.

20.2.2 Calculate the Conditional Probabilities

The conditional probabilities are the probability of each input value given each class value. The conditional probabilities for the dataset can be calculated as follows:

Weather Input Variable

$$
\begin{aligned}
P(weather = sunny|class = \text{go-out}) &= \frac{count(weather = sunny \wedge class = \text{go-out})}{count(class = \text{go-out})} \\
P(weather = rainy|class = \text{go-out}) &= \frac{count(weather = rainy \wedge class = \text{go-out})}{count(class = \text{go-out})} \\
P(weather = sunny|class = \text{stay-home}) &= \frac{count(weather = sunny \wedge class = \text{stay-home})}{count(class = \text{stay-home})} \\
P(weather = rainy|class = \text{stay-home}) &= \frac{count(weather = rainy \wedge class = \text{stay-home})}{count(class = \text{stay-home})}
\end{aligned}
\tag{20.3}
$$

Remember that the $\wedge$ symbol is just a shorthand for conjunction (AND). Plugging in the numbers we get:

$$
\begin{aligned}
&P(weather = sunny|class = \text{go-out}) = 0.8 \\
&P(weather = rainy|class = \text{go-out}) = 0.2 \\
&P(weather = sunny|class = \text{stay-home}) = 0.4 \\
&P(weather = rainy|class = \text{stay-home}) = 0.6
\end{aligned}
\tag{20.4}
$$

Car Input Variable

$$
\begin{aligned}
P(car = working|class = \text{go-out}) &= \frac{count(car = working \wedge class = \text{go-out})}{count(class = \text{go-out})} \\
P(car = broken|class = \text{go-out}) &= \frac{count(car = broken \wedge class = \text{go-out})}{count(class = \text{go-out})} \\
P(car = working|class = \text{stay-home}) &= \frac{count(car = working \wedge class = \text{stay-home})}{count(class = \text{stay-home})} \\
P(car = broken|class = \text{stay-home}) &= \frac{count(car = broken \wedge class = \text{stay-home})}{count(class = \text{stay-home})}
\end{aligned}
\tag{20.5}
$$

Plugging in the numbers we get:

$$
\begin{aligned}
&P(car = working|class = \text{go-out}) = 0.8 \\
&P(car = broken|class = \text{go-out}) = 0.2 \\
&P(car = working|class = \text{stay-home}) = 0.2 \\
&P(car = broken|class = \text{stay-home}) = 0.8
\end{aligned}
\tag{20.6}
$$

We now have every thing we need to make predictions using the Naive Bayes model.

20.3 Make Predictions with Naive Bayes

We can make predictions using Bayes Theorem, defined and explained in the previous chapter.

$$P(h|d) = \frac{P(d|h) \times P(h)}{P(d)} \tag{20.7}$$

In fact, we don't need a probability to predict the most likely class for a new data instance. We only need the numerator and the class that gives the largest response, which will be the predicted output.

$$MAP(h) = max(P(d|h) \times P(h)) \tag{20.8}$$

Let's take the first record from our dataset and use our learned model to predict which class we think it belongs. First instance: $weather = sunny, car = working$.

We plug the probabilities for our model in for both classes and calculate the response. Starting with the response for the output `go-out`. We multiply the conditional probabilities together and multiply it by the probability of any instance belonging to the class.

$$\begin{aligned} \text{go-out} = & P(weather = sunny|class = \text{go-out}) \times \\ & P(car = working|class = \text{go-out}) \times \\ & P(class = \text{go-out}) \end{aligned} \tag{20.9}$$

or

$$\begin{aligned} \text{go-out} &= 0.8 \times 0.8 \times 0.5 \\ \text{go-out} &= 0.32 \end{aligned} \tag{20.10}$$

We can perform the same calculation for the stay-home case:

$$\begin{aligned} \text{stay-home} = & P(weather = sunny|class = \text{stay-home}) \times \\ & P(car = working|class = \text{stay-home}) \times \\ & P(class = \text{stay-home}) \end{aligned} \tag{20.11}$$

or

$$\begin{aligned} \text{stay-home} &= 0.4 \times 0.2 \times 0.5 \\ \text{stay-home} &= 0.04 \end{aligned} \tag{20.12}$$

We can see that 0.32 is greater than 0.04, therefore we predict `go-out` for this instance, which is correct. We can repeat this operation for the entire dataset, as follows:

Weather	Car	Class	go-out?	stay-home?	Prediction
sunny	working	go-out	0.32	0.04	go-out
rainy	broken	go-out	0.02	0.24	stay-home
sunny	working	go-out	0.32	0.04	go-out
sunny	working	go-out	0.32	0.04	go-out
sunny	working	go-out	0.32	0.04	go-out
rainy	broken	stay-home	0.02	0.24	stay-home
rainy	broken	stay-home	0.02	0.24	stay-home
sunny	working	stay-home	0.32	0.04	go-out
sunny	broken	stay-home	0.08	0.16	stay-home
rainy	broken	stay-home	0.02	0.24	stay-home

Listing 20.3: Naive Bayes Predictions for the Dataset.

If we tally up the predictions compared to the actual class values, we get an accuracy of 80%, which is excellent given that there are conflicting examples in the dataset.

20.4 Summary

In this chapter you discovered exactly how to implement Naive Bayes from scratch. You learned:

- How to work with categorical data with Naive Bayes.
- How to calculate class probabilities from training data.
- How to calculate conditional probabilities from training data.
- How to use a learned Naive Bayes model to make predictions on new data.

You now know how to implement Naive Bayes from scratch for categorical data. In the next chapter you will discover how to can implement Naive Bayes from scratch for real-valued data.

Chapter 21

Gaussian Naive Bayes Tutorial

Naive Bayes is a simple model that uses probabilities calculated from your training data to make predictions on new data. The basic Naive Bayes algorithm assumes categorical data. A simple extension for real-valued data is called Gaussian Naive Bayes. In this chapter you will discover how to implement Gaussian Naive Bayes from scratch. After reading this chapter you will know:

- The Gaussian Probability Density Function and how to calculate the probability of real values.
- How to learn the properties for a Gaussian Naive Bayes model from your training data.
- How to use a learned Gaussian Naive Bayes model to make predictions on new data.

Let's get started.

21.1 Tutorial Dataset

A simple dataset was contrived for our purposes. It is comprised of two input variables $X1$ and $X2$ and one output variable Y. The input variables are drawn from a Gaussian distribution, which is one assumption made by Gaussian Naive Bayes. The class variable has two values, 0 and 1, therefore the problem is a binary classification problem.

Data from class 0 was drawn randomly from a Gaussian distribution with a standard deviation of 1.0 for $X1$ and $X2$. Data from class 1 was drawn randomly from a Gaussian distribution with a mean of 7.5 for $X1$ and 2.5 for $X2$. This means that the classes are nicely separated if we plot the input data on a scatter plot. The raw dataset is listed below:

```
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 21.1: Gaussian Naive Bayes Tutorial Data Set.

You can clearly see the separation of the classes in the plot below. This will make the data relatively easy to work with as we implement and test a Gaussian Naive Bayes model.

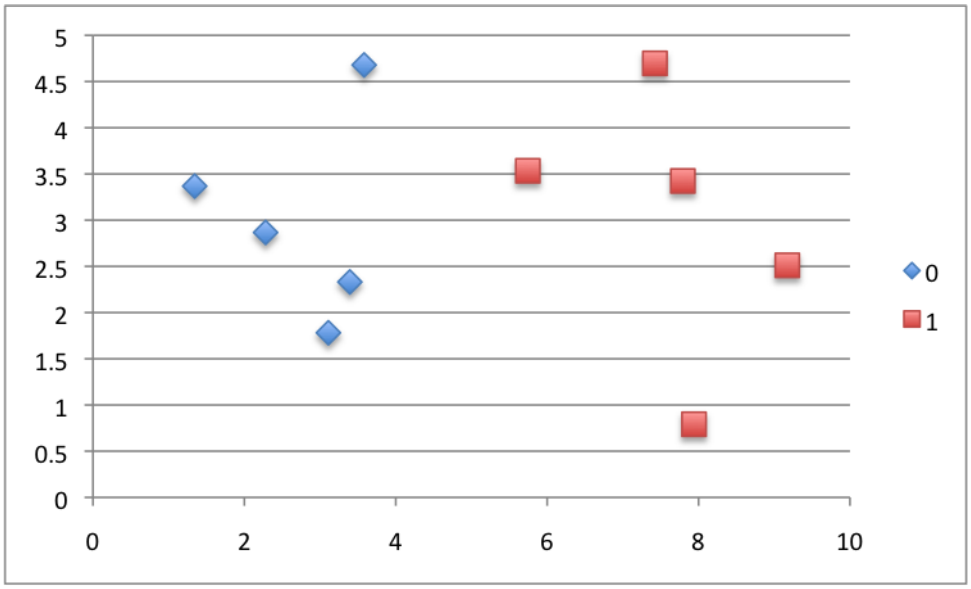

Figure 21.1: Gaussian Naive Bayes Tutorial Dataset.

21.2 Gaussian Probability Density Function

The Gaussian Probability Density Function (PDF) will calculate the probability of a value given the mean and standard deviation of the distribution from which it came. The Gaussian PDF is calculated as follows:

$$pdf(x, mean, sd) = \frac{1}{\sqrt{2 \times \pi} \times sd} \times e^{-(\frac{(x-mean)^2}{2 \times sd^2})} \tag{21.1}$$

Where $pdf(x)$ is the Gaussian PDF, $mean$ and sd are the mean and standard deviation calculated above, π is the numerical constant PI, e is Euler's number raised to a power and x is the input value for the input variable. Let's look at an example. Let's assume we have real values drawn from a population that has a mean of 0 and a standard deviation of 1. Using the Gaussian PDF we can estimate the likelihood of range of values.

```
X       PDF(x)
-5      1.48672E-06
-4      0.00013383
-3      0.004431848
-2      0.053990967
-1      0.241970725
0       0.39894228
1       0.241970725
2       0.053990967
```

```
3       0.004431848
4       0.00013383
5       1.48672E-06
```

Listing 21.2: Test of the Gaussian Probability Density Function.

You can see that the mean has the highest probability of nearly 0.4 (40%). You can also see values that are far away from the mean like -5 and +5 (5 standard deviations from the mean) have a very low probability. Below is a plot of the probabilities values.

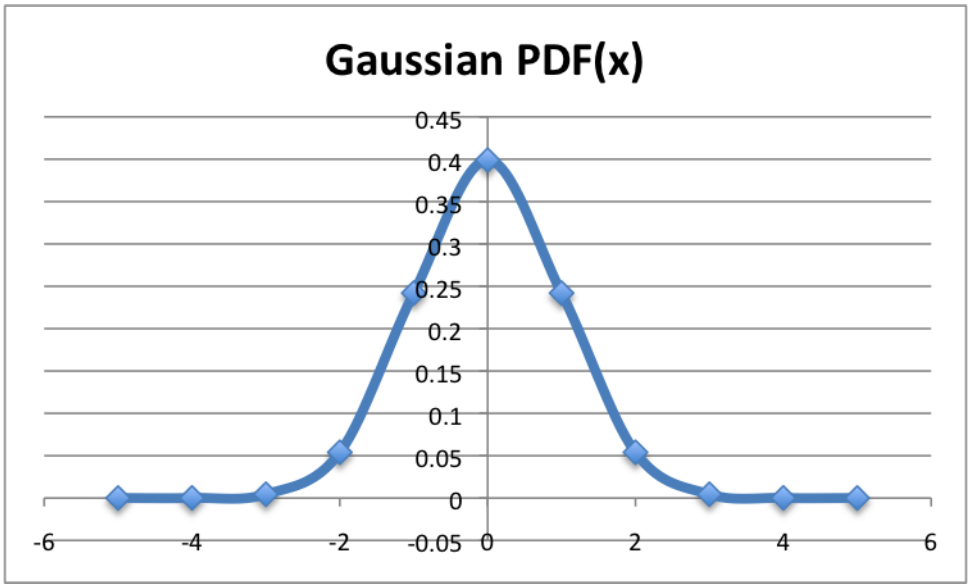

Figure 21.2: Gaussian Probability Density Function.

This function is really useful for Naive Bayes. We can assume that the input variables are each drawn from a Gaussian distribution. By calculating the mean and standard deviation of each input variable from the training data, we can use the Gaussian PDF to estimate the likelihood of each value for each class. We will see how this is used to calculate conditional probabilities in the next section.

21.3 Learn a Gaussian Naive Bayes Model

There are two types of probabilities that we need to summarize from our training data for the naive Bayes model:

- Class Probabilities.
- Conditional Probabilities.

21.3.1 Class Probabilities

The dataset is a two class problem and we already know the probability of each class because we contrived the dataset. Nevertheless, we can calculate the class probabilities for classes 0 and

1 as follows:

$$P(Y=1) = \frac{count(Y=1)}{count(Y=0)+count(Y=1)}$$
$$P(Y=0) = \frac{count(Y=0)}{count(Y=0)+count(Y=1)} \tag{21.2}$$

or

$$P(Y=1) = \frac{5}{5+5}$$
$$P(Y=0) = \frac{5}{5+5} \tag{21.3}$$

This works out to be a probability of 0.5 (50%) for any given data instance belonging to class 0 or class 1.

21.3.2 Conditional Probabilities

The conditional probabilities are the probabilities of each input value given each class value. The conditional probabilities that need to be collected from the training data are as follows:

$$P(X1|Y=0)$$
$$P(X1|Y=1)$$
$$P(X2|Y=0)$$
$$P(X2|Y=1) \tag{21.4}$$

The $X1$ and $X2$ input variables are real values. As such we will model them as having being drawn from a Gaussian distribution. This will allow us to estimate the probability of a given value using the Gaussian PDF described above. The Gaussian PDF requires two parameters in addition to the value for which the probability is being estimated: the mean and the standard deviation. Therefore we must estimate the mean and the standard deviation for each group of conditional probabilities that we require. We can estimate these directly from the dataset. The results are summarized below.

```
       P(X1|Y=0)      P(X1|Y=1)      P(X2|Y=0)      P(X2|Y=1)
Mean   2.742014401    7.614652372    3.005468669    2.991467979
Stdev  0.926568329    1.234432155    1.107329589    1.454193138
```

Listing 21.3: Summary of population statistics by class.

We now have enough information to make predictions for the training data or even a new dataset.

21.4 Make Prediction with Gaussian Naive Bayes

We can make predictions using Bayes Theorem, introduced and explained in a previous chapter. We don't need a probability to predict the most likely class for a new data instance. We only need the numerator and the class that gives the largest response is the predicted response.

$$MAP(h) = max(P(d|h) \times P(h)) \tag{21.5}$$

Let's take the first record from our dataset and use our learned model to predict which class we think it belongs. Instance: $X1 = 3.393533211$, $X2 = 2.331273381$, $Y = 0$. We can plug the probabilities for our model in for both classes and calculate the response. Starting with the response for the output class 0. We multiply the conditional probabilities together and multiply it by the probability of any instance belonging to the class.

$$\begin{aligned} \text{class } 0 &= P(pdf(X1)|class = 0) \times P(pdf(X2)|class = 0) \times P(class = 0) \\ \text{class } 0 &= 0.358838152 \times 0.272650889 \times 0.5 \\ \text{class } 0 &= 0.048918771 \end{aligned} \tag{21.6}$$

We can perform the same calculation for class 1:

$$\begin{aligned} \text{class } 1 &= P(pdf(X1)|class = 1) \times P(pdf(X2)|class = 1) \times P(class = 1) \\ \text{class } 1 &= 4.10796E-07 \times 0.173039018 \times 0.5 \\ \text{class } 1 &= 3.55418E-08 \end{aligned} \tag{21.7}$$

We can see that 0.048918771 is greater than 3.55418E-08, therefore we predict the class as 0 for this instance, which is correct. Repeating this process for all instances in the dataset we get the following outcomes for class 0 and class 1. By selecting the class with the highest output we make accurate predictions for all instances in the training dataset.

X1	X2	Output Y=0	Output Y=1	Prediction
3.393533211	2.331273381	0.048918771	3.55418E-08	0
3.110073483	1.781539638	0.02920928	1.82065E-09	0
1.343808831	3.368360954	0.030910813	3.7117E-15	0
3.582294042	4.67917911	0.010283134	9.08728E-09	0
2.280362439	2.866990263	0.069951664	1.67539E-11	0
7.423436942	4.696522875	1.10289E-06	0.001993521	1
5.745051997	3.533989803	0.001361494	0.002264317	1
9.172168622	2.511101045	1.30731E-09	0.00547065	1
7.792783481	3.424088941	1.22227E-06	0.0355024	1
7.939820817	0.791637231	3.53132E-08	0.000245214	1

Listing 21.4: Predictions using Gaussian Naive Bayes.

The prediction accuracy is 100%, as was expected given the clear separation of the classes.

21.5 Summary

In this chapter you discovered how to implement the Gaussian Naive Bayes classifier from scratch. You learned about:

- The Gaussian Probability Density Function for estimating the probability of any given real value.
- How to estimate the probabilities required by the Naive Bayes model from a training dataset.
- How to use the learned Naive Bayes model to make predictions.

You now know how to implement Gaussian Naive Bayes from scratch for real-valued data. In the next chapter you will discover the K-Nearest Neighbors algorithm for classification and regression.

Chapter 22

K-Nearest Neighbors

In this chapter you will discover the k-Nearest Neighbors (KNN) algorithm for classification and regression. After reading this chapter you will know.

- The model representation used by KNN.
- How a model is learned using KNN (hint, it's not).
- How to make predictions using KNN
- The many names for KNN including how different fields refer to it.
- How to prepare your data to get the most from KNN.
- Where to look to learn more about the KNN algorithm.

Let's get started.

22.1 KNN Model Representation

The model representation for KNN is the entire training dataset. It is as simple as that. KNN has no model other than storing the entire dataset, so there is no learning required. Efficient implementations can store the data using complex data structures like k-d trees to make look-up and matching of new patterns during prediction efficient. Because the entire training dataset is stored, you may want to think carefully about the consistency of your training data. It might be a good idea to curate it, update it often as new data becomes available and remove erroneous and outlier data.

22.2 Making Predictions with KNN

KNN makes predictions using the training dataset directly. Predictions are made for a new data point by searching through the entire training set for the K most similar instances (the neighbors) and summarizing the output variable for those K instances. For regression this might be the mean output variable, in classification this might be the mode (or most common) class value.

To determine which of the K instances in the training dataset are most similar to a new input a distance measure is used. For real-valued input variables, the most popular distance measure is Euclidean distance. Euclidean distance is calculated as the square root of the sum of the squared differences between a point a and point b across all input attributes i.

$$EuclideanDistance(a, b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2} \tag{22.1}$$

Other popular distance measures include:

- Hamming Distance: Calculate the distance between binary vectors.
- Manhattan Distance: Calculate the distance between real vectors using the sum of their absolute difference. Also called City Block Distance.
- Minkowski Distance: Generalization of Euclidean and Manhattan distance.

There are many other distance measures that can be used, such as Tanimoto, Jaccard, Mahalanobis and cosine distance. You can choose the best distance metric based on the properties of your data. If you are unsure, you can experiment with different distance metrics and different values of K together and see which mix results in the most accurate models. Euclidean is a good distance measure to use if the input variables are similar in type (e.g. all measured widths and heights). Manhattan distance is a good measure to use if the input variables are not similar in type (such as age, gender, height, etc.).

The value for K can be found by algorithm tuning. It is a good idea to try many different values for K (e.g. values from 1 to 21) and see what works best for your problem. The computational complexity of KNN increases with the size of the training dataset. For very large training sets, KNN can be made stochastic by taking a sample from the training dataset from which to calculate the K-most similar instances. KNN has been around for a long time and has been very well studied. As such, different disciplines have different names for it, for example:

- **Instance-Based Learning**: The raw training instances are used to make predictions. As such KNN is often referred to as instance-based learning or a case-based learning (where each training instance is a case from the problem domain).
- **Lazy Learning**: No learning of the model is required and all of the work happens at the time a prediction is requested. As such, KNN is often referred to as a lazy learning algorithm.
- **Nonparametric**: KNN makes no assumptions about the functional form of the problem being solved. As such KNN is referred to as a nonparametric machine learning algorithm.

KNN can be used for regression and classification problems.

22.2.1 KNN for Regression

When KNN is used for regression problems the prediction is based on the mean or the median of the K-most similar instances.

22.2.2 KNN for Classification

When KNN is used for classification, the output can be calculated as the class with the highest frequency from the K-most similar instances. Each instance in essence votes for their class and the class with the most votes is taken as the prediction. Class probabilities can be calculated as the normalized frequency of samples that belong to each class in the set of K most similar instances for a new data instance. For example, in a binary classification problem (class is 0 or 1):

$$p(class = 0) = \frac{count(class = 0)}{count(class = 0) + count(class = 1)} \tag{22.2}$$

If you are using K and you have an even number of classes (e.g. 2) it is a good idea to choose a K value with an odd number to avoid a tie. And the inverse, use an even number for K when you have an odd number of classes. Ties can be broken consistently by expanding K by 1 and looking at the class of the next most similar instance in the training dataset.

22.3 Curse of Dimensionality

KNN works well with a small number of input variables (p), but struggles when the number of inputs is very large. Each input variable can be considered a dimension of a p-dimensional input space. For example, if you had two input variables $X1$ and $X2$, the input space would be 2-dimensional. As the number of dimensions increases the volume of the input space increases at an exponential rate. In high dimensions, points that may be similar may have very large distances. All points will be far away from each other and our intuition for distances in simple 2 and 3-dimensional spaces breaks down. This might feel unintuitive at first, but this general problem is called the *Curse of Dimensionality*.

22.4 Preparing Data For KNN

- **Rescale Data**: KNN performs much better if all of the data has the same scale. Normalizing your data to the range between 0 and 1 is a good idea. It may also be a good idea to standardize your data if it has a Gaussian distribution.
- **Address Missing Data**: Missing data will mean that the distance between samples cannot be calculated. These samples could be excluded or the missing values could be imputed.
- **Lower Dimensionality**: KNN is suited for lower dimensional data. You can try it on high dimensional data (hundreds or thousands of input variables) but be aware that it may not perform as well as other techniques. KNN can benefit from feature selection that reduces the dimensionality of the input feature space.

22.5 Summary

In this chapter you discovered the KNN machine learning algorithm. You learned that:

- KNN stores the entire training dataset which it uses as its representation.
- KNN does not learn any model.
- KNN makes predictions just-in-time by calculating the similarity between an input sample and each training instance.
- There are many distance measures to choose from to match the structure of your input data.
- That it is a good idea to rescale your data, such as using normalization, when using KNN.

You now know about the K-Nearest Neighbors algorithm for classification and regression. In the next chapter you will discover how to implement KNN from scratch for classification.

Chapter 23

K-Nearest Neighbors Tutorial

The K-Nearest Neighbors (KNN) algorithm is very simple and very effective. In this chapter you will discover exactly how to implement it from scratch, step-by-step. After reading this chapter you will know:

- How to calculate the Euclidean distance between real valued vectors.
- How to use Euclidean distance and the training dataset to make predictions for new data.

Let's get started.

23.1 Tutorial Dataset

The problem is a binary (two-class) classification problem. This problem was contrived for this tutorial. The dataset contains two input variables ($X1$ and $X1$) and the class output variable with the values 0 and 1. The dataset contains 10 records, 5 that belong to each class.

```
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 23.1: KNN Tutorial Data Set.

You can see that the data for each class is quite separated. This is intentionally to make the problem easy to work with so that we can focus on the learning algorithm.

23.2 KNN and Euclidean Distance

KNN uses a distance measure to locate the K most similar instances from the training dataset when making a prediction. The distance measure selected must respect the structure of the

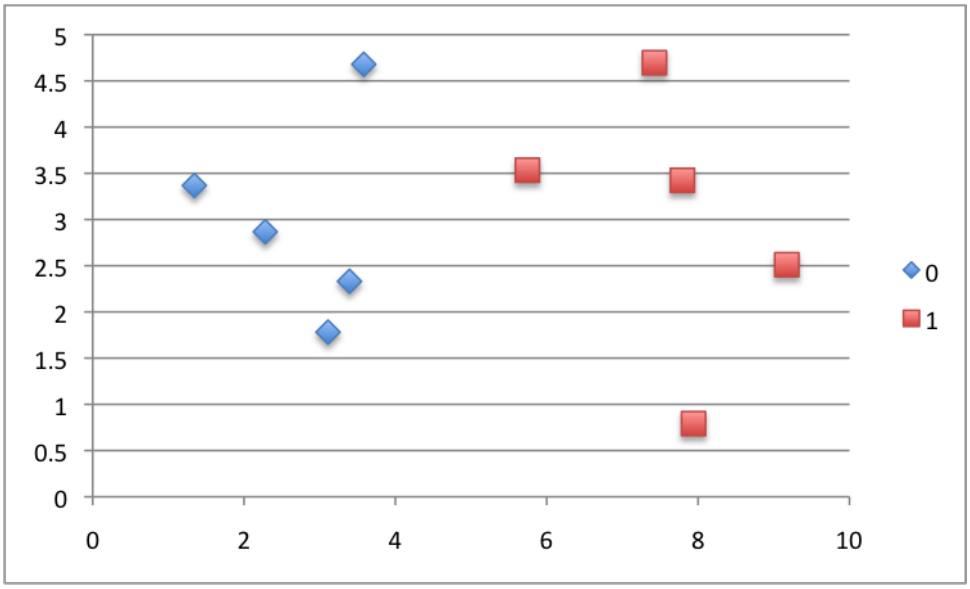

Figure 23.1: KNN Tutorial Dataset Scatter plot.

problem so that data instances that are close to each other according to the distance measure also belong to the same class. The most common distance measure for real values that have the same units or scale is the Euclidean distance. Euclidean distance is calculated as the square root of the sum of the squared differences between a point a and point b across all input attributes i.

$$EuclideanDistance(a, b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2} \tag{23.1}$$

To make this concrete, we will work through the calculation of the Euclidean distance for two instances from our dataset.

```
Instance    X1             X2
1           3.393533211    2.331273381
2           3.110073483    1.781539638
```

Listing 23.2: Two Instances For Calculating Euclidean Distance.

The first step is to calculate the squared difference for each attribute:

$$\begin{aligned} SquaredDifference1 &= (X1_1 - X1_2)^2 \\ SquaredDifference2 &= (X2_1 - X2_2)^2 \end{aligned} \tag{23.2}$$

or

$$\begin{aligned} SquaredDifference1 &= (3.393533211 - 3.110073483)^2 \\ SquaredDifference2 &= (2.331273381 - 1.781539638)^2 \end{aligned} \tag{23.3}$$

or

$$\begin{aligned} SquaredDifference1 &= 0.08034941698 \\ SquaredDifference2 &= 0.3022071889 \end{aligned} \tag{23.4}$$

We calculate the sum of these squared differences as:

$$\begin{aligned} SumSquaredDifference &= SquaredDifference1 + SquaredDifference2 \\ SumSquaredDifference &= 0.080349417 + 0.302207188 \\ SumSquaredDifference &= 0.382556606 \end{aligned} \tag{23.5}$$

Finally, we need to take the square root of the sum. This will convert the units of the difference between the data instances (real vectors) from squared units to their original units.

$$\begin{aligned} Distance &= \sqrt{SumSquaredDifference} \\ Distance &= 0.618511605 \end{aligned} \tag{23.6}$$

This final step can be skipped for performance reasons. You probably don't need the distance in the actual units, and the square root function is relatively expensive compared to other operations and will be performed many times per new data instance that is to be classified. Now that you know how the Euclidean distance measure is calculated, we can use it with the dataset to make predictions for new data.

23.3 Making Predictions with KNN

Given a new data instance for which we would like to make a prediction, the K instances with the smallest distance to the new data instance are chosen to contribute to that prediction. For classification, this involves allowing each of the K members to vote of which class the new data instance belongs. To make this concrete, we will work through making a prediction for a new data instance using the training dataset as the model. The new data instance is listed below. We are cheating because we contrived the dataset, we know what class the instance should be allocated. Instance: $X1 = 8.093607318$, $X2 = 3.365731514$, $Y = 1$.

The first step is to calculate the Euclidean distance between the new input instance and all instances in the training dataset. The table below lists the distance between each training instance and the new data.

No.	X1	X2	Y	(X1-X1)^2	(X2-X2)^2	Sum	Distance
1	3.393533211	2.331273381	0	22.09069661	1.070103629	23.16080024	4.812566908
2	3.110073483	1.781539638	0	24.83560948	2.5096639	27.34527338	5.229270827
3	1.343808831	3.368360954	0	45.55977962	6.91395E-06	45.55978653	6.749798999
4	3.582294042	4.679179110	0	20.35194747	1.725144587	22.07709206	4.698626614
5	2.280362439	2.866990263	0	33.79381602	0.248742835	34.04255886	5.834600146
6	7.423436942	4.696522875	1	0.449128333	1.771005647	2.220133979	1.490011402
7	5.745051997	3.533989803	1	5.515712096	0.028310852	5.544022948	2.354574897
8	9.172168622	2.511101045	1	1.163294486	0.730393239	1.893687725	1.376113268
9	7.792783481	3.424088941	1	0.090494981	0.003405589	0.09390057	0.306431999
10	7.939820817	0.791637231	1	0.023650288	6.625961377	6.649611665	2.578684096

Listing 23.3: Euclidean Distances for Training Data to New Instance.

We will set K to 3 and choose the 3 most similar neighbors to the data instance. A value of $K = 3$ is small and easy to use in this example, it is also an odd number, meaning that when the neighbors vote on the output class, that we cannot have a tie. The $K = 3$ most similar neighbors to the new data instance are:

```
No.     Distance      Y
9       0.306431999   1
8       1.376113268   1
6       1.490011402   1
```

Listing 23.4: K=3 Most Similar Neighbors To New Instance.

Making a prediction is as easy as selecting the majority class in the neighbors. Because we are using 0 and 1 for the class values, we can use the `MODE()` statistical function in a spreadsheet to return the most frequent value.

$$prediction = mode(class(i)) \tag{23.7}$$

In this case all 3 neighbors have a class of 1, therefore the prediction for this instance is 1, which is correct.

23.4 Summary

In this chapter you discovered how you can use K-Nearest Neighbors to make predictions on a binary classification problem. You learned about:

- The Euclidean distance measure and how to calculate it step-by-step.
- How to use the Euclidean distance to locate the nearest neighbors for a new data instance.
- How to make a prediction from the K-nearest neighbors.

You now know how to implement K-Nearest Neighbors from scratch for classification. In the next chapter you will discover an extension to KNN called Learning Vector Quantization for classification.

Chapter 24

Learning Vector Quantization

A downside of K-Nearest Neighbors is that you need to hang on to your entire training dataset. The Learning Vector Quantization algorithm (or LVQ for short) is an artificial neural network algorithm that allows you choose how many training instances to hang onto and learns exactly what those instances should look like. In this chapter you will discover the Learning Vector Quantization algorithm. After reading this chapter you will know:

- The representation used by the LVQ algorithm that you actually save to a file.
- The procedure that you can use to make predictions with a learned LVQ model.
- How to learn an LVQ model from training data.
- The data preparation to use to get the best performance from the LVQ algorithm.
- Where to look for more information on LVQ.

Let's get started.

24.1 LVQ Model Representation

The representation for LVQ is a collection of codebook vectors. LVQ was developed and is best understood as a classification algorithm. It supports both binary (two-class) and multiclass classification problems. A codebook vector is a list of numbers that have the same input and output attributes as your training data. For example, if your problem is a binary classification with classes 0 and 1, and the inputs width, length height, then a codebook vector would be comprised of all four attributes: width, length, height and class.

The model representation is a fixed pool of codebook vectors, learned from the training data. They look like training instances, but the values of each attribute have been adapted based on the learning procedure. In the language of neural networks, each codebook vector may be called a neuron, each attribute on a codebook vector is called a weight and the collection of codebook vectors is called a network.

24.2 Making Predictions with an LVQ Model

Predictions are made using the LVQ codebook vectors in the same way as K-Nearest Neighbors. Predictions are made for a new instance by searching through all codebook vectors for the K most similar instances and summarizing the output variable for those K instances. For classification this is the mode (or most common) class value. Typically predictions are made with $K = 1$, and the codebook vector that matches is called the Best Matching Unit (BMU).

To determine which of the K instances in the training dataset are most similar to a new input a distance measure is used. For real-valued input variables, the most popular distance measure is Euclidean distance. Euclidean distance is calculated as the square root of the sum of the squared differences between a point a and point b across all input attributes i.

$$EuclideanDistance(a, b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2} \tag{24.1}$$

24.3 Learning an LVQ Model From Data

The LVQ algorithm learns the codebook vectors from the training data. You must choose the number of codebook vectors to use, such as 20 or 40. You can find the best number of codebook vectors to use by testing different configurations on your training dataset. The learning algorithm starts with a pool of random codebook vectors. These could be randomly selected instances from the training data, or randomly generated vectors with the same scale as the training data. Codebook vectors have the same number of input attributes as the training data. They also have an output class variable.

The instances in the training dataset are processed one at a time. For a given training instance, the most similar codebook vector is selected from the pool. If the codebook vector has the same output as the training instance, the codebook vector is moved closer to the training instance. If it does not match, it is moved further away. The amount that the vector is moved is controlled by an algorithm parameter called the learning rate ($alpha$). For example, the input variable (x) of a codebook vector is moved closer to the training input value (t) by the amount in the learning rate ($alpha$) if the classes match as follows:

$$x = x + alpha \times (t - x) \tag{24.2}$$

The opposite case of moving the input variables of a codebook variable away from a training instance is calculated as:

$$x = x - alpha \times (t - x) \tag{24.3}$$

This would be repeated for each input variable. Because one codebook vector is selected for modification for each training instance the algorithm is referred to as a winner-take-all algorithm or a type of competitive learning. This process is repeated for each instance in the training dataset. One iteration of the training dataset is called an epoch. The process is completed for a number of epochs that you must choose (MaxEpoch), such as 200, 2000 or 20,000.

You must also choose an initial learning rate (such as $alpha = 0.3$). The learning rate is decreased with the epoch, starting at the large value you specify at epoch 1 which makes the most change to the codebook vectors and finishing with a small value near zero on the last

epoch making very minor changes to the codebook vectors. The learning rate for each epoch is calculated as:

$$LearningRate = alpha \times (1 - \frac{Epoch}{MaxEpoch}) \tag{24.4}$$

Where $LearningRate$ is the learning rate for the current epoch (0 to $MaxEpoch$-1), alpha is the learning rate specified to the algorithm at the start of the training run and $MaxEpoch$ is the total number of epochs to run the algorithm also specified at the start of the run. The intuition for the learning process is that the pool of codebook vectors is a compression of the training dataset to the points that best characterize the separation of the classes.

24.4 Preparing Data For LVQ

Generally, it is a good idea to prepare data for LVQ in the same way as you would prepare it for K-Nearest Neighbors.

- **Classification**: LVQ is a classification algorithm that works for both binary (two-class) and multiclass classification algorithms. The technique has been adapted for regression.
- **Multiple-Passes**: Good technique with LVQ involves performing multiple passes of the training dataset over the codebook vectors (e.g. multiple learning runs). The first with a higher learning rate to settle the pool of codebook vectors and the second run with a small learning rate to fine tune the vectors.
- **Multiple Best Matches**: Extensions of LVQ select multiple best matching units to modify during learning, such as one of the same class and one of a different class which are drawn toward and away from a training sample respectively. Other extensions use a custom learning rate for each codebook vector. These extensions can improve the learning process.
- **Normalize Inputs**: Traditionally, inputs are normalized (rescaled) to values between 0 and 1. This is to avoid one attribute from dominating the distance measure. If the input data is normalized, then the initial values for the codebook vectors can be selected as random values between 0 and 1.
- **Feature Selection**: Feature selection that can reduce the dimensionality of the input variables can improve the accuracy of the method. LVQ suffers from the same curse of dimensionality in making predictions as K-Nearest Neighbors.

24.5 Summary

In this chapter you discovered the LVQ algorithm. You learned:

- The representation for LVQ is a small pool of codebook vectors, smaller than the training dataset.
- The codebook vectors are used to make predictions using the same technique as K-Nearest Neighbors.

- The codebook vectors are learned from the training dataset by moving them closer when they are good match and further away when they are a bad match.
- The codebook vectors are a compression of the training data to best separate the classes.
- Data preparation traditionally involves normalizing the input values to the range between 0 and 1.

You now know about the Learning Vector Quantization algorithm for classification. In the next chapter you will discover how you can implement LVQ from scratch.

Chapter 25

Learning Vector Quantization Tutorial

The Learning Vector Quantization (LVQ) algorithm is a lot like the K-Nearest Neighbors algorithm, but it involves learning. In this chapter you will discover how to implement the LVQ algorithm from scratch. After reading this chapter you will know:

- How to initialize an LVQ model.
- How to Update the Best Matching Unit for a training instance.
- How to update the LVQ model for one and multiple epochs.
- How to use a learned LVQ model to make predictions.

Let's get started.

25.1 Tutorial Dataset

The problem is a binary (two-class) classification problem. The problem was contrived for this tutorial. The dataset contains two input variables ($X1$ and $X2$) and the class output variable with the values 0 and 1. The dataset contains 10 records, 5 that belong to each class.

```
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 25.1: LVQ Tutorial Data Set.

You can see that the data for each class is quite separated. This is intentional to make the problem easy to work with so that we can focus on the learning algorithm.

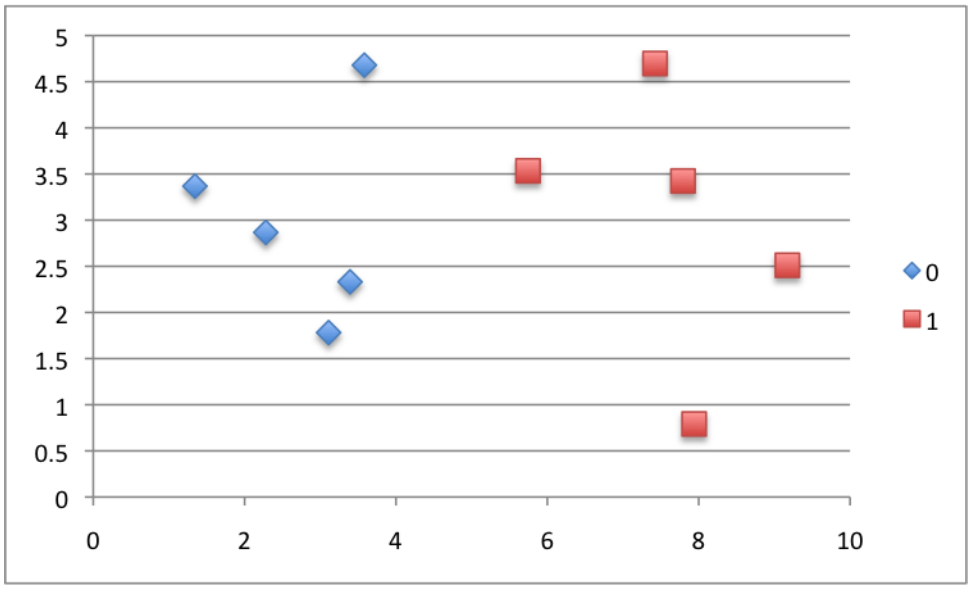

Figure 25.1: LVQ Tutorial Dataset Scatter plot.

25.2 Learn the LVQ Model

LVQ learns a population of codebook vectors from the training data. This section is broken up into 3 parts:

1. Initial Codebook Vectors.
2. Update Codebook Vectors for One Pattern.
3. Update For One Epoch.

25.2.1 Initial Codebook Vectors

The number of codebook vectors often depends on the size and complexity of the problem. Because we are working with a simple problem and for demonstration purposes we will select a small number of codebook vectors of 4, two of each class. The values for the vectors can be chosen randomly or selected from the input data. We will use the latter. The table below lists the selected codebook vectors:

```
X1             X2             Y
3.582294042    0.791637231    0
7.792783481    2.331273381    0
7.939820817    2.866990263    1
3.393533211    4.67917911     1
```

Listing 25.2: Initial LVQ codebook Vectors.

25.2.2 Update Codebook Vectors for One Pattern

In addition to choosing the number of codebook vectors, an initial learning rate must be specified. A good default value is 0.3, but values are typically between 0.1 and 0.5. The learning

rate is used to update the codebook vectors. In this section we will look at the rule used to update the codebook vectors for one training pattern. Let's take the first training pattern: $X1 = 3.393533211$, $X2 = 2.331273381$, $Y = 0$. The update rule for one pattern is as follows:

1. Calculate the distance from the training pattern to each codebook vector.
2. Select the most similar codebook vector, called the Best Matching Unit (BMU).
3. Update the best matching unit to be closer to the training pattern if it has the same class, otherwise further away.

Calculate Distance

We can calculate the distance between a training instance and a codebook vector using Euclidean distance. This is the most common distance measure when all input attributes have the same scale, which they do in this case. Euclidean distance is calculated as the square root of the sum of the squared differences between a point a and point b across all input attributes i.

$$EuclideanDistance(a, b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2} \tag{25.1}$$

Let's calculate the Euclidean distance between each codebook vector and the training instance. The results are listed below.

```
(X1-X1)^2       (X2-X2)^2          Sum             Distance        BMU?
0.035630651     2.370479474        2.406110125     1.551164119     YES
19.35340294     0                  19.35340294     4.39925027
20.668731       0.286992578        20.95572357     4.577742192
0               5.512661312        5.512661312     2.347905729
```

Listing 25.3: Euclidean Distances of Codebook Vectors from Instance.

From the distances, we can see that the codebook vector #1 has the smallest distance and is therefore the BMU.

Update the Best Matching Unit

The best matching unit need to be moved closer to the training pattern if it is the same class or further away if the classes differ. The class for the BMU is 0 and this matches the class of the training instance 0. Therefore we need update the attributes to be closer to the training instance, limited by the learning rate. The learning rate controls how much change can be made to the codebook vectors for a single update. Let's use an initial learning rate of 0.7 which is much larger than normal in order to show fast learning on this simple problem. The update procedure for a single attribute is therefore:

$$bmu_j = bmu_j + alpha \times (training_j - bmu_j) \tag{25.2}$$

Where j is the attribute on the codebook vector being updated (e.g. $X1$), $alpha$ is the learning rate and $training_j$ is the same attribute on the training instance (e.g. $X1$). If the

BMU had a different class, the update would be almost identical except we would use a negative sign to push the BMU further away from the training instance. For example:

$$bmu_j = bmu_j - alpha \times (training_j - bmu_j) \tag{25.3}$$

Let's apply the learning rule and update the attributes of the BMU.

$$\begin{aligned} X1 &= 3.582294042 + 0.7 \times (3.393533211 - 3.582294042) \\ X2 &= 0.791637231 + 0.7 \times (2.331273381 - 0.791637231) \end{aligned} \tag{25.4}$$

or

$$\begin{aligned} X1 &= 3.45016146 \\ X2 &= 1.869382536 \end{aligned} \tag{25.5}$$

We have just completed an update for one training instance.

25.2.3 Update For One Epoch

An epoch is one pass through the entire training dataset (all 10 instances). This involves applying the above procedure for each training instance, locating the BMU and updating it. Before the start of the run we must choose the number of epochs to perform in order to train the model. The number could be hundreds, thousands or even tens of thousands of epochs, depending on the difficulty of the problem. Each epoch, the learning rate is decreased from the starting value. This means that at the start of the run the model is doing a lot of learning and towards the end of the run it is only doing very small adjustments to codebook vectors already learned. The learning rate can be calculated for a given epoch as follows:

$$LearningRate = alpha \times (1 - \frac{Epoch}{MaxEpoch}) \tag{25.6}$$

Where $LearningRate$ is the learning rate for the current epoch (0 to $MaxEpoch$-1), alpha is the learning rate specified to the algorithm at the start of the training run and $MaxEpoch$ is the total number of epochs to run the algorithm also specified at the start of the run.

We end up with the following codebook vectors:

```
X1            X2            Y
2.55988367    2.549260936   0
6.048389028   3.195023766   0
7.343461045   3.512289796   1
5.700572642   6.239052716   1
```

Listing 25.4: Updated Codebook Vector Values.

The process probably needs to be repeated for another 10-to-20 epochs with a lower learning rate (e.g. 0.3) to settle down the codebook vectors.

25.3 Make Predictions with LVQ

Once the codebook vectors are learned, they can be used to make predictions. We can use the same procedure as KNN to make predictions, although K is set to 1. Just like in the learning

process, we use a distance measure to locate the BMU for a new data instance. Instead of updating the BMU, we return the class value which becomes the prediction for our model. Using the codebook vectors prepared above, and the Euclidean distance measure, we can make the following predictions for each instance in the dataset:

```
X1            X2            Prediction Y
3.393533211   2.331273381   0          0
3.110073483   1.781539638   0          0
1.343808831   3.368360954   0          0
3.582294042   4.67917911    0          0
2.280362439   2.866990263   0          0
7.423436942   4.696522875   1          1
5.745051997   3.533989803   0          1
9.172168622   2.511101045   1          1
7.792783481   3.424088941   1          1
7.939820817   0.791637231   1          1
```

Listing 25.5: Predictions with Codebook Vectors.

If we compare this to the actual values in the dataset for Y, we can see some errors. The classification accuracy can be calculated as:

$$\begin{aligned} accuracy &= \frac{count(correct)}{count(instances)} \times 100 \\ accuracy &= \frac{9}{10} \times 100 \\ accuracy &= 90\% \end{aligned} \tag{25.7}$$

With some more training the codebook vectors will become more accurate.

25.4 Summary

In this chapter you discovered how to implement the LVQ machine learning algorithm from scratch for a binary classification problem. You learned:

- How to initialize an LVQ model.
- How to Update the Best Matching Unit for a training instance.
- How to update the LVQ model for one and multiple epochs.
- How to use a learned LVQ model to make predictions.

You now know how to implement Learning Vector Quantization from scratch for classification. In the next chapter you will discover the Support Vector Machine machine learning algorithm for classification.

Chapter 26

Support Vector Machines

Support Vector Machines are perhaps one of the most popular and talked about machine learning algorithms. They were extremely popular around the time they were developed in the 1990s and continue to be the go-to method for a high-performing algorithm with little tuning. In this chapter you will discover the Support Vector Machine (SVM) machine learning algorithm. After reading this chapter you will know:

- How to disentangle the many names used to refer to support vector machines.
- The representation used by SVM when the model is actually stored on disk.
- How a learned SVM model representation can be used to make predictions for new data.
- How to learn an SVM model from training data.
- How to best prepare your data for the SVM algorithm.
- Where you might look to get more information on SVM.

Let's get started.

26.1 Maximal-Margin Classifier

The Maximal-Margin Classifier is a hypothetical classifier that best explains how SVM works in practice. The numeric input variables (x) in your data (the columns) form an n-dimensional space. For example, if you had two input variables, this would form a two-dimensional space. A hyperplane is a line that splits the input variable space. In SVM, a hyperplane is selected to best separate the points in the input variable space by their class, either class 0 or class 1. In two-dimensions you can visualize this as a line and let's assume that all of our input points can be completely separated by this line. For example:

$$B0 + (B1 \times X1) + (B2 \times X2) = 0 \tag{26.1}$$

Where the coefficients ($B1$ and $B2$) that determine the slope of the line and the intercept ($B0$) are found by the learning algorithm, and $X1$ and $X2$ are the two input variables. You can make classifications using this line. By plugging in input values into the line equation, you can calculate whether a new point is above or below the line.

- Above the line, the equation returns a value greater than 0 and the point belongs to the first class (class 0).
- Below the line, the equation returns a value less than 0 and the point belongs to the second class (class 1).
- A value close to the line returns a value close to zero and the point may be difficult to classify.
- If the magnitude of the value is large, the model may have more confidence in the prediction.

The distance between the line and the closest data points is referred to as the margin. The best or optimal line that can separate the two classes is the line that as the largest margin. This is called the Maximal-Margin hyperplane. The margin is calculated as the perpendicular distance from the line to only the closest points. Only these points are relevant in defining the line and in the construction of the classifier. These points are called the support vectors. They support or define the hyperplane. The hyperplane is learned from training data using an optimization procedure that maximizes the margin.

26.2 Soft Margin Classifier

In practice, real data is messy and cannot be separated perfectly with a hyperplane. The constraint of maximizing the margin of the line that separates the classes must be relaxed. This is often called the soft margin classifier. This change allows some points in the training data to violate the separating line. An additional set of coefficients are introduced that give the margin wiggle room in each dimension. These coefficients are sometimes called slack variables. This increases the complexity of the model as there are more parameters for the model to fit to the data to provide this complexity.

A tuning parameter is introduced called simply C that defines the magnitude of the wiggle allowed across all dimensions. The C parameters defines the amount of violation of the margin allowed. A $C = 0$ is no violation and we are back to the inflexible Maximal-Margin Classifier described above. The larger the value of C the more violations of the hyperplane are permitted. During the learning of the hyperplane from data, all training instances that lie within the distance of the margin will affect the placement of the hyperplane and are referred to as support vectors. And as C affects the number of instances that are allowed to fall within the margin, C influences the number of support vectors used by the model.

- The smaller the value of C, the more sensitive the algorithm is to the training data (higher variance and lower bias).
- The larger the value of C, the less sensitive the algorithm is to the training data (lower variance and higher bias).

26.3 Support Vector Machines (Kernels)

The SVM algorithm is implemented in practice using a kernel. The learning of the hyperplane in linear SVM is done by transforming the problem using some linear algebra, which is out of the

scope of this introduction to SVM. A powerful insight is that the linear SVM can be rephrased using the inner product of any two given observations, rather than the observations themselves.

The inner product between two vectors is the sum of the multiplication of each pair of input values. For example, the inner product of the vectors $[2, 3]$ and $[5, 6]$ is $2 \times 5 + 3 \times 6$ or 28. The equation for making a prediction for a new input using the dot product between the input (x) and each support vector (x_i) is calculated as follows:

$$f(x) = B0 + \sum_{i=1}^{n}(a_i \times (x \times x_i)) \tag{26.2}$$

This is an equation that involves calculating the inner products of a new input vector (x) with all support vectors in training data. The coefficients $B0$ and a_i (for each input) must be estimated from the training data by the learning algorithm.

26.3.1 Linear Kernel SVM

The dot-product is called the kernel and can be re-written as:

$$K(x, x_i) = \sum(x \times x_i) \tag{26.3}$$

The kernel defines the similarity or a distance measure between new data and the support vectors. The dot product is the similarity measure used for linear SVM or a linear kernel because the distance is a linear combination of the inputs. Other kernels can be used that transform the input space into higher dimensions such as a Polynomial Kernel and a Radial Kernel. This is called the Kernel Trick. It is desirable to use more complex kernels as it allows lines to separate the classes that are curved or even more complex. This in turn can lead to more accurate classifiers.

26.3.2 Polynomial Kernel SVM

Instead of the dot-product, we can use a polynomial kernel, for example:

$$K(x, x_i) = 1 + \sum(x \times x_i)^d \tag{26.4}$$

Where the degree of the polynomial must be specified by hand to the learning algorithm. When $d = 1$ this is the same as the linear kernel. The polynomial kernel allows for curved lines in the input space.

26.3.3 Radial Kernel SVM

Finally, we can also have a more complex radial kernel. For example:

$$K(x, x_i) = e^{-gamma \times \sum((x - x_i^2)} \tag{26.5}$$

Where gamma is a parameter that must be specified to the learning algorithm. A good default value for gamma is 0.1, where gamma is often $0 < gamma < 1$. The radial kernel is very local and can create complex regions within the feature space, like closed polygons in a two-dimensional space.

26.4 How to Learn a SVM Model

The SVM model needs to be solved using an optimization procedure. You can use a numerical optimization procedure to search for the coefficients of the hyperplane. This is inefficient and is not the approach used in widely used SVM implementations like LIBSVM. If implementing the algorithm as an exercise, you could use a variation of gradient descent called sub-gradient descent.

There are specialized optimization procedures that re-formulate the optimization problem to be a Quadratic Programming problem. The most popular method for fitting SVM is the Sequential Minimal Optimization (SMO) method that is very efficient. It breaks the problem down into sub-problems that can be solved analytically (by calculating) rather than numerically (by searching or optimizing).

26.5 Preparing Data For SVM

This section lists some suggestions for how to best prepare your training data when learning an SVM model.

- **Numerical Inputs**: SVM assumes that your inputs are numeric. If you have categorical inputs you may need to covert them to binary dummy variables (one variable for each category).
- **Binary Classification**: Basic SVM as described in this chapter is intended for binary (two-class) classification problems. Although, extensions have been developed for regression and multiclass classification.

26.6 Summary

In this chapter you discovered the Support Vector Machine Algorithm for machine learning. You learned about:

- The Maximal-Margin Classifier that provides a simple theoretical model for understanding SVM.
- The Soft Margin Classifier which is a modification of the Maximal-Margin Classifier to relax the margin to handle noisy class boundaries in real data.
- Support Vector Machines and how the learning algorithm can be reformulated as a dot-product kernel and how other kernels like Polynomial and Radial can be used.
- How you can use numerical optimization to learn the hyperplane and that efficient implementations use an alternate optimization scheme called Sequential Minimal Optimization.

You now know about the Support Vector Machine algorithm. In the next chapter you will discover how you can implement SVM from scratch using sub-gradient descent.

Chapter 27

Support Vector Machine Tutorial

Support Vector Machines are a flexible nonparametric machine learning algorithm. In this chapter you will discover how to implement the Support Vector Machine algorithm step-by-step using sub-gradient descent. After completing this chapter you will know:

- How to use sub-gradient descent to update the coefficients for an SVM model.
- How to iterate the sub-gradient descent algorithm to learn an SVM model for training data.
- How to make predictions given a learned SVM model.

Let's get started.

27.1 Tutorial Dataset

A test problem was devised so that the classes are linearly separable. This means that a straight line can be drawn to separate the classes. This is intentional so that we can explore how to implement an SVM with a linear kernel (straight line). An assumption made by the SVM algorithm is that first class value is -1 and the second class value is $+1$.

```
X1            X2            Y
2.327868056   2.458016525   -1
3.032830419   3.170770366   -1
4.485465382   3.696728111   -1
3.684815246   3.846846973   -1
2.283558563   1.853215997   -1
7.807521179   3.290132136   1
6.132998136   2.140563087   1
7.514829366   2.107056961   1
5.502385039   1.404002608   1
7.432932365   4.236232628   1
```

Listing 27.1: SVM Tutorial Data Set.

The visualization below provides a scatter plot of the dataset.

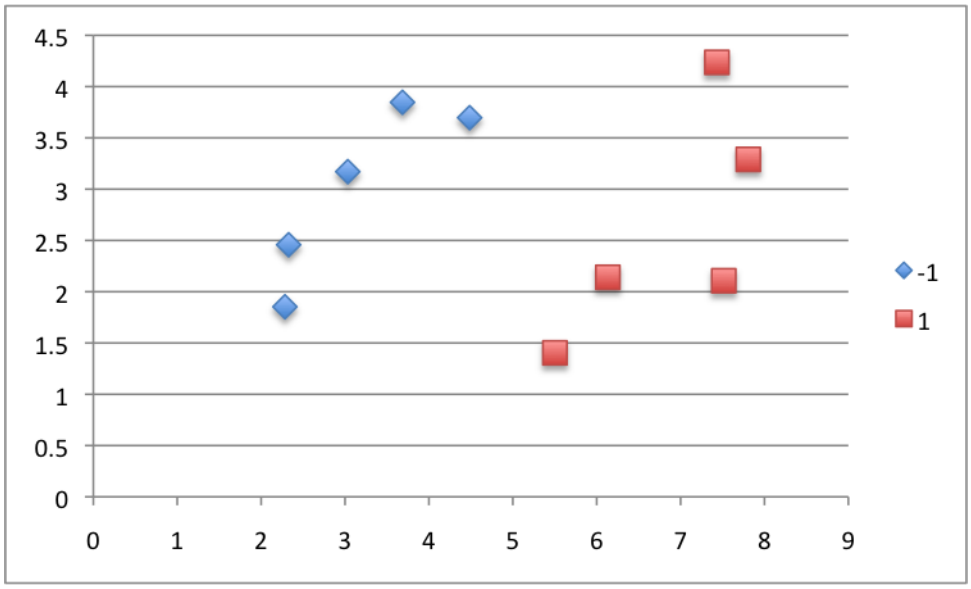

Figure 27.1: SVM Tutorial Dataset Scatter plot.

27.2 Training SVM With Gradient Descent

This section describes the form of the SVM model and how it can be learned using a variant of the gradient descent optimization procedure.

27.2.1 Form of Linear SVM Model

The Linear SVM model is a line and the goal of the learning algorithm is to find values for the coefficients that best separates the classes. The line is typically in the form (grouping the terms for readability):

$$B0 + (B1 \times X1) + (B2 \times X2) = 0 \tag{27.1}$$

Where $B0$, $B1$ and $B2$ are the coefficients and $X1$ and $X2$ are the input variables. This will be the form of the equation that we will be using with one small modification, we will drop the bias term ($B0$) also called the offset or the intercept. For example $B0 = 0$.

$$(B1 \times X1) + (B2 \times X2) = 0 \tag{27.2}$$

This means that the line will pass through the origin ($X1 = 0$ and $X2 = 0$). This is just to make the tutorial easier to follow and because our simple problem does not really need it, you can add the bias term back in if you like.

27.2.2 SVM Optimization Method

The optimization algorithm to find the coefficients can be stated as a quadratic programming problem. This is a type of constraint optimization where fast solvers can be used. We will not be using this approach in this tutorial. Another approach that can be used to discover the coefficient values for Linear SVM is sub-gradient descent. In this method a random training pattern is selected each iteration and used to update the coefficients. After a large number

of iterations (thousands or hundreds of thousands) the algorithm will settle on a stable set of coefficients. The coefficient update equation works as follows. First an output value is calculated as:

$$output = Y \times (B1 \times X1) + (B2 \times X2) \tag{27.3}$$

Two different update procedures are used depending on the output value. If the output value is greater than 1 it suggests that the training pattern was not a support vector. This means that the instance was not directly involved in calculating the output, in which case the weights are slightly decreased:

$$b = (1 - \frac{1}{t}) \times b \tag{27.4}$$

Where b is the weight that is being updated (such as $B1$ or $B2$), t is the current iteration (e.g. 1 for the first update, 2 for the second and so on). If the output is less than 1 then it is assumed that the training instance is a support vector and must be updated to better explain the data.

$$b = (1 - \frac{1}{t}) \times b + \frac{1}{lambda \times t} \times (y \times x) \tag{27.5}$$

Where b is the weight that is being updated, t is the current iteration and lambda is a parameter to the learning algorithm. The lambda is a learning parameter and is often set to very small values such as 0.0001 or smaller. The procedure is repeated until the error rate drops to a desirable level or for a very large fixed number of iterations. Smaller learning rates ofter require much longer training times. The number of iterations is a downside to this learning algorithm.

27.3 Learn an SVM Model from Training Data

In this section we will work through a few updates to the coefficients to demonstrate the SVM learning algorithm. We will use a very large lambda value: $lambda = 0.45$. This is unusually large and will force a lot of change on each update. Normally *lambda* values are very small.

27.3.1 Learning Iteration #1

We will start by setting the coefficients to 0.0.

$$\begin{aligned} B1 &= 0.0 \\ B2 &= 0.0 \end{aligned} \tag{27.6}$$

We also need to keep track of which iteration we are on: $t = 1$. We will train the model using the order of the training patterns. Ideally, the order of the patterns would be randomized to avoid the learning algorithm getting stuck. The first training pattern we will use to update

the coefficients is: Instance: $X1 = 2.327868056$, $X2 = 2.458016525$, $Y = -1$. We can now calculate the output value for this iteration.

$$\begin{aligned} output &= Y \times (B1 \times X1) + (B2 \times X2) \\ output &= -1 \times (0.0 \times 2.327868056) + (0.0 \times 2.458016525) \\ output &= 0.0 \end{aligned} \tag{27.7}$$

Easy enough. The output is less than 1.0, therefore we will use the more complex update procedure that assumes the training pattern is a support vector:

$$\begin{aligned} b &= (1 - \frac{1}{t}) \times b + \frac{1}{lambda \times t} \times (y \times x) \\ B1 &= (1 - \frac{1}{1}) \times 0.0 + \frac{1}{0.5 \times 1} \times (-1 \times 2.327868056) \\ B1 &= -5.173040124 \end{aligned} \tag{27.8}$$

And for $B2$ this is:

$$\begin{aligned} B2 &= (1 - \frac{1}{1}) \times 0.0 + \frac{1}{0.5 \times 1} \times (-1 \times 2.458016525) \\ B2 &= -5.462258944 \end{aligned} \tag{27.9}$$

27.3.2 Learning Iteration #2

We now have updated coefficients that we can use on the next iteration of the learning algorithm with the second instance from the training dataset: Instance: $X1 = 3.032830419$, $X2 = 3.170770366$, $Y = -1$. Again, we must keep track of the iteration: $t = 2$. Let's repeat the process.

$$\begin{aligned} B1 &= -5.173040124 \\ B2 &= -5.462258944 \end{aligned} \tag{27.10}$$

The output value is calculated as:

$$\begin{aligned} output &= -1 \times (-5.173040124 \times 3.032830419) + (-5.462258944 \times 3.170770366) \\ output &= 33.00852224 \end{aligned} \tag{27.11}$$

The output value is larger than 1.0, suggesting that this training instance is not a support vector. We can update the $B1$ coefficient accordingly:

$$\begin{aligned} b &= (1 - \frac{1}{t}) \times b \\ B1 &= (1 - \frac{1}{2}) \times -5.173040124 \\ B1 &= -2.586520062 \end{aligned} \tag{27.12}$$

And for the $B2$ coefficient:

$$\begin{aligned} B2 &= (1 - \frac{1}{2}) \times -5.462258944 \\ B2 &= -2.731129472 \end{aligned} \tag{27.13}$$

27.3.3 More Iterations

Repeat this process for the rest of the dataset. One pass through the dataset is called an epoch. Now repeat the process for a further 15 epochs for a total of 160 iterations (16 epochs × 10 updates per epoch). It is possible to keep track of the loss or the accuracy of the model for each epoch. This is a great way to get insight into whether the algorithm is converging or whether there is a bug in the implementation. If you plot the accuracy for the model at the end of each epoch, you should see something that looks like the following graph:

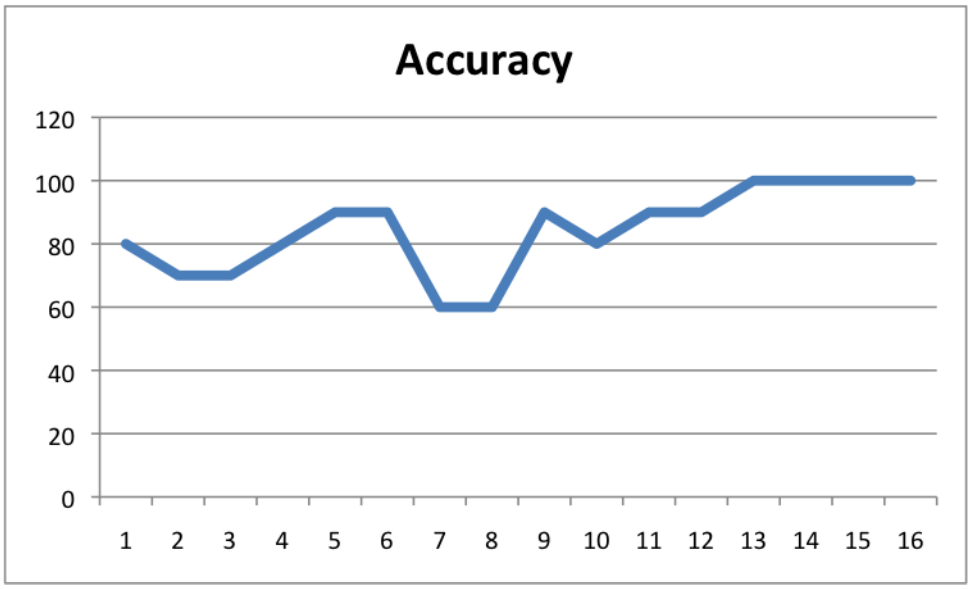

Figure 27.2: Support Vector Machine Model Accuracy.

You can see that after 16 epochs that we achieve an accuracy of 100% on the training data. You should arrive at final values for the coefficients that look like the following:

$$\begin{aligned} B1 &= 0.552391765 \\ B2 &= -0.724533592 \end{aligned} \tag{27.14}$$

The form of the learned hyperplane is therefore:

$$0 + (0.552391765 \times X1) + (-0.724533592 \times X2) = 0 \tag{27.15}$$

27.4 Make Predictions with SVM Model

Now that we have coefficients for the line, we can make predictions. In this section we will make predictions for the training data, but this could just as easily be adapted to make predictions for new data. Predictions can be made using the following equation:

$$\begin{aligned} output &= (B1 \times X1) + (B2 \times X2) \\ Y &= -1 \text{ IF } output < 0 \\ Y &= +1 \text{ IF } output > 0 \end{aligned} \tag{27.16}$$

Using the above coefficients and these prediction rules, we can make a prediction for each instance in the training dataset:

```
Output         Crisp  Y
-0.495020399   -1     -1
-0.622019096   -1     -1
-0.20066956    -1     -1
-0.75170826    -1     -1
-0.081298299   -1     -1
1.928999147    1      1
1.836907801    1      1
2.624496306    1      1
2.022225129    1      1
1.036597783    1      1
```

Listing 27.2: Accuracy of the Learned SVM Model.

Comparing the crisp prediction ($Crisp$) to the expected output column (Y), we can see that our model has achieved 100% accuracy.

27.5 Summary

In this chapter you discovered how to implement the Support Vector Machine algorithm from scratch using the sub-gradient descent optimization technique. You learned:

- How to update coefficient values for SVM using sub-gradient descent.
- How to iterate the sub-gradient descent procedure to find good coefficient values.
- How to make predictions using a learned SVM model.

You now know how to implement the Support Vector Machine algorithm from scratch using sub-gradient descent. This concludes your introduction to nonlinear machine learning algorithms. In the next part you will discover ensemble machine learning algorithms starting the bagging.

Part V

Ensemble Algorithms

Chapter 28

Bagging and Random Forest

Random Forest is one of the most popular and most powerful machine learning algorithms. It is a type of ensemble machine learning algorithm called Bootstrap Aggregation or bagging. In this chapter you will discover the Bagging ensemble algorithm and the Random Forest algorithm for predictive modeling. After reading this chapter you will know about:

- The bootstrap method for estimating statistical quantities from samples.
- The Bootstrap Aggregation algorithm for creating multiple different models from a single training dataset.
- The Random Forest algorithm that makes a small tweak to Bagging and results in a very powerful classifier.

Let's get started.

28.1 Bootstrap Method

Before we get to Bagging, let's take a quick look at an important foundation technique called the bootstrap. The bootstrap is a powerful statistical method for estimating a quantity from a data sample. This is easiest to understand if the quantity is a descriptive statistic such as a mean or a standard deviation. Let's assume we have a sample of 100 values (x) and we'd like to get an estimate of the mean of the sample. We can calculate the mean directly from the sample as:

$$mean(x) = \frac{1}{100} \times \sum_{i=1}^{100} x_i \tag{28.1}$$

We know that our sample is small and that our mean has error in it. We can improve the estimate of our mean using the bootstrap procedure:

1. Create many (e.g. 1000) random sub-samples of our dataset with replacement (meaning we can select the same value multiple times).
2. Calculate the mean of each sub-sample.

3. Calculate the average of all of our collected means and use that as our estimated mean for the data.

For example, let's say we used 3 resamples and got the mean values 2.3, 4.5 and 3.3. Taking the average of these we could take the estimated mean of the data to be 3.367. This process can be used to estimate other quantities like the standard deviation and even quantities used in machine learning algorithms, like learned coefficients.

28.2 Bootstrap Aggregation (Bagging)

Bootstrap Aggregation (or Bagging for short), is a simple and very powerful ensemble method. An ensemble method is a technique that combines the predictions from multiple machine learning algorithms together to make more accurate predictions than any individual model. Bootstrap Aggregation is a general procedure that can be used to reduce the variance for those algorithms that have high variance. An algorithm that has high variance are decision trees, like classification and regression trees (CART).

Decision trees are sensitive to the specific data on which they are trained. If the training data is changed (e.g. a tree is trained on a subset of the training data) the resulting decision tree can be quite different and in turn the predictions can be quite different. Bagging is the application of the Bootstrap procedure to a high-variance machine learning algorithm, typically decision trees. Let's assume we have a dataset of 1000 instances and we are using the CART algorithm. Bagging of the CART algorithm would work as follows.

1. Create many (e.g. 100) random sub-samples of our dataset with replacement.
2. Train a CART model on each sample.
3. Given a new dataset, calculate the average prediction from each model.

For example, if we had 5 bagged decision trees that made the following class predictions for an input instance: `blue`, `blue`, `red`, `blue` and `red`, we would take the most frequent class and predict `blue`. When bagging with decision trees, we are less concerned about individual trees overfitting the training data. For this reason and for efficiency, the individual decision trees are grown deep (e.g. few training samples at each leaf-node of the tree) and the trees are not pruned. These trees will have both high variance and low bias. These are important characteristics of sub-models when combining predictions using bagging.

The only parameters when bagging decision trees is the number of trees to create. This can be chosen by increasing the number of trees on run after run until the accuracy begins to stop showing improvement (e.g. on a cross validation test harness). Creating large numbers of decision trees may take a long time, but will not overfit the training data. Just like the decision trees themselves, Bagging can be used for classification and regression problems.

28.3 Random Forest

Random Forests are an improvement over bagged decision trees. A problem with decision trees like CART is that they are greedy. They choose which variable to split on using a

greedy algorithm that minimizes error. As such, even with Bagging, the decision trees can have a lot of structural similarities and in turn result in high correlation in their predictions. Combining predictions from multiple models in ensembles works better if the predictions from the sub-models are uncorrelated or at best weakly correlated.

Random forest changes the algorithm for the way that the sub-trees are learned so that the resulting predictions from all of the subtrees have less correlation. It is a simple tweak. In CART, when selecting a split point, the learning algorithm is allowed to look through all variables and all variable values in order to select the most optimal split-point. The random forest algorithm changes this procedure so that the learning algorithm is limited to a random sample of features of which to search. The number of features that can be searched at each split point (m) must be specified as a parameter to the algorithm. You can try different values and tune it using cross validation.

- For classification a good default is: $m = \sqrt{p}$.
- For regression a good default is: $m = \frac{p}{3}$.

Where m is the number of randomly selected features that can be searched at a split point and p is the number of input variables. For example, if a dataset had 25 input variables for a classification problem, then:

$$\begin{aligned} m &= \sqrt{25} \\ m &= 5 \end{aligned} \tag{28.2}$$

28.4 Estimated Performance

For each bootstrap sample taken from the training data, there will be samples left behind that were not included. These samples are called Out-Of-Bag samples or OOB. The performance of each model on its left out samples when averaged can provide an estimated accuracy of the bagged models. This estimated performance is often called the OOB estimate. These performance measures are a reliable estimate of test error and correlate well with cross validation estimates of error.

28.5 Variable Importance

As the Bagged decision trees are constructed, we can calculate how much the error function drops for a variable at each split point. In regression problems this may be the drop in sum squared error and in classification this might be the Gini score. These drops in error can be averaged across all decision trees and output to provide an estimate of the importance of each input variable. The greater the drop when the variable was chosen, the greater the importance.

These outputs can help identify subsets of input variables that may be most or least relevant to the problem and suggest at possible feature selection experiments you could perform where some features are removed from the dataset.

28.6 Preparing Data For Bagged CART

Bagged CART does not require any special data preparation other than a good representation of the problem.

28.7 Summary

In this chapter you discovered the Bagging ensemble machine learning algorithm and the popular variation called Random Forest. You learned:

- How to estimate statistical quantities from a data sample.
- How to combine the predictions from multiple high-variance models using bagging.
- How to tweak the construction of decision trees when bagging to de-correlate their predictions, a technique called Random Forests.

You now know about the bagging ensemble algorithm, bagged decision trees and random forests. In the next chapter you will discover how to implement bagged decision trees from scratch.

Chapter 29

Bagged Decision Trees Tutorial

Bagging is a simple ensemble method that almost always results in a lift in performance over the baseline models. In this chapter you will discover how to implement bagged decision trees step-by-step. After reading this chapter you will discover:

- How to create decision trees from bootstrap samples of your training dataset.
- How to aggregate the predictions from multiple bootstrap samples.
- How bagging can create more accurate predictions than individual high-variance models.

Let's get started.

29.1 Tutorial Dataset

In this tutorial we will use dataset with two input variables ($X1$ and $X2$) and one output variable (Y). The input variables are real-valued random numbers drawn from a Gaussian distribution. The output variable has two values, making the problem a binary classification problem. The raw data is listed below.

```
X1            X2            Y
2.309572387   1.168959634   0
1.500958319   2.535482186   0
3.107545266   2.162569456   0
4.090032824   3.123409313   0
5.38660215    2.109488166   0
6.451823468   0.242952387   1
6.633669528   2.749508563   1
8.749958452   2.676022211   1
4.589131161   0.925340325   1
6.619322828   3.831050828   1
```

Listing 29.1: Bagging Tutorial Data Set.

Below is a plot of the dataset. You see that we cannot draw a straight line to separate the classes.

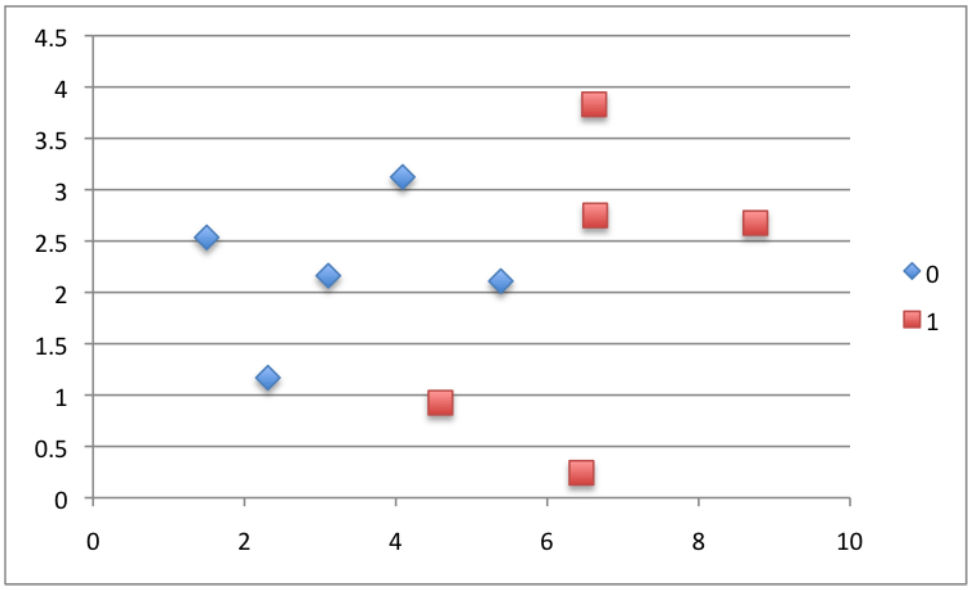

Figure 29.1: Bagging Tutorial Dataset Scatter plot.

29.2 Learn the Bagged Decision Tree Model

Bootstrap Aggregation (or Bagging for short), is a simple and very powerful ensemble method. Bagging works by taking a sub-sample of your training dataset (with replacement) and creating a model, often a decision tree because they have high variance. The process is repeated creating as many trees as you desire. Later when making predictions for new data, the outputs from each tree are combined by taking the average.

Decision trees like CART select split points using a greedy algorithm that seeks to minimize a cost function, like the Gini index for classification. When the tree is created on a random sample of the training data, the algorithm is likely to choose different split points, resulting in different trees and in turn different predictions. This is where the power for bagging comes from, in combining the predictions from models that have very different perspectives on the problem.

The interesting part of bagged decision trees is how they are combined to make ensemble predictions.With this in mind, we will contrive 3 decision trees from the training data, each with sub-par accuracy. The split points were chosen manually to demonstrate exactly how different perspectives on the same problem can be combined to provide increased performance. A decision tree with one split point is called a decision stump, this is because there is little tree to speak of. We will use three decision stumps. The split points for each are as follows:

$$\begin{aligned} Model1 &: X1 \leq 5.38660215 \\ Model2 &: X1 \leq 4.090032824 \\ Model3 &: X2 \leq 0.925340325 \end{aligned} \tag{29.1}$$

If you are adapting this example to your own problem, you can apply the CART algorithm to each bootstrap sample from the training dataset. Because this is a classification problem, you could use the Gini index cost function to minimize in order to choose split points. Also, in this example we are only using 3 models, often you will create tens, hundreds or even thousands of models when bagging. Now that we have our bootstrap models, let's look at how we can aggregate them.

29.3 Make Predictions with Bagged Decision Trees

The predictions from the bootstrap models can be combined to produce more accurate predictions. In this section we look at making predictions with each bootstrapped model, then finally aggregate the sub-model predictions into more accurate ensemble predictions.

29.3.1 Decision Stump Model 1

The split point for the first model is $X1 \leq 5.38660215$. Using this split point we can separate the training data into two groups:

```
Y   Group
0   LEFT
0   LEFT
0   LEFT
0   LEFT
0   LEFT
1   RIGHT
1   RIGHT
1   RIGHT
1   LEFT
1   RIGHT
```

Listing 29.2: Grouping of Instances for Model 1.

We can see that the LEFT group contains mostly instances for class 0 and the RIGHT group contains mostly instances of class 1. Therefore, instances assigned to the two groups will be classified:

- LEFT: class 0
- RIGHT: class 1

We can therefore calculate the prediction for each training instance.

```
Prediction Error  Accuracy
0          0      90
0          0
0          0
0          0
0          0
1          0
1          0
1          0
0          1
1          0
```

Listing 29.3: Prediction of Instances for Model 1.

We can see that the model only got one training instance incorrect (the second last one). This gives the model an accuracy of 90%. Not bad at all, but we can do better.

29.3.2 Decision Stump Model 2

The second model uses the split point $X1 \leq 4.090032824$. We can separate the training data into two groups:

```
Y   Group
0   LEFT
0   LEFT
0   LEFT
0   LEFT
0   RIGHT
1   RIGHT
1   RIGHT
1   RIGHT
1   RIGHT
1   RIGHT
```

Listing 29.4: Grouping of Instances for Model 2.

As above, the LEFT group will classify instances as class 0 and the RIGHT as class 1. Using this model, we can make predictions for all instances in the training dataset:

```
Prediction Error  Accuracy
0          0      90
0          0
0          0
0          0
1          1
1          0
1          0
1          0
1          0
1          0
```

Listing 29.5: Prediction of Instances for Model 2.

Again, like the first model, we can see the accuracy is 90% with only one instance misclassified. This time the 5th instance. Combining just these two models results in some ambiguous predictions where the models conflict in their predictions. Adding a third model will clear things up.

29.3.3 Decision Stump Model 3

The split point used by the third model is $X2 \leq 0.925340325$. Using this split point, we can again sort the training dataset into two groups.

```
Y   Group
0   RIGHT
0   RIGHT
0   RIGHT
0   RIGHT
0   RIGHT
1   LEFT
1   RIGHT
1   RIGHT
1   LEFT
```

```
1   RIGHT
```

Listing 29.6: Grouping of Instances for Model 3.

In this model the RIGHT group will classify instances as class 0 and the LEFT group as class 1. Using this simple model, we can again make predictions for each training instance.

```
Prediction Error  Accuracy
0          0      70
0          0
0          0
0          0
0          0
1          0
0          1
0          1
1          0
0          1
```

Listing 29.7: Prediction of Instances for Model 3.

This model has slightly worse accuracy of 70%. Importantly it correctly predicts both the second last instance and the 5th instance, clearing up any ambiguity if the previous two models were combined.

29.4 Final Predictions

Now that we have predictions from the bootstrap models we can aggregate the predictions into an ensemble prediction. We can do that by taking the mode of each models prediction for each training instance. The mode is a statistical function that select the most common value. In this case, the most common class value predicted. Using this simple procedure we get the following predictions:

```
Prediction Y   Error  Accuracy
0          0   0      100
0          0   0
0          0   0
0          0   0
0          0   0
1          1   0
1          1   0
1          1   0
1          1   0
1          1   0
```

Listing 29.8: Prediction of Instances for Bagged Model.

You can see that all 10 training instances were classified correctly at 100% accuracy.

29.5 Summary

In this chapter you discovered bagged decision trees. You learned.

- How to create multiple decision stump models from bootstrap data samples.

- How to aggregate the predictions from multiple decision tree models.

You now know how to implement bagged decision trees from scratch. In the next chapter you will discover the boosting ensemble method and the AdaBoost machine learning algorithm.

Chapter 30

Boosting and AdaBoost

Boosting is an ensemble technique that attempts to create a strong classifier from a number of weak classifiers. In this chapter you will discover the AdaBoost Ensemble method for machine learning. After reading this chapter, you will know:

- What the boosting ensemble method is and generally how it works.
- How to learn to boost decision trees using the AdaBoost algorithm.
- How to make predictions using the learned AdaBoost model.
- How to best prepare your data for use with the AdaBoost algorithm.

Let's get started.

30.1 Boosting Ensemble Method

Boosting is a general ensemble method that creates a strong classifier from a number of weak classifiers. This is done by building a model from the training data, then creating a second model that attempts to correct the errors from the first model. Models are added until the training set is predicted perfectly or a maximum number of models are added. AdaBoost was the first really successful boosting algorithm developed for binary classification. It is the best starting point for understanding boosting. Modern boosting methods build on AdaBoost, most notably stochastic gradient boosting machines.

30.2 Learning An AdaBoost Model From Data

AdaBoost is best used to boost the performance of decision trees on binary classification problems. AdaBoost was originally called AdaBoost.M1 by the developers of the technique. More recently it may be referred to as discrete AdaBoost because it is used for classification rather than regression. AdaBoost can be used to boost the performance of any machine learning algorithm. It is best used with weak learners.

These are models that achieve accuracy just above random chance on a classification problem. The most suited and therefore most common algorithm used with AdaBoost are decision trees with one level. Because these trees are so short and only contain one decision for classification,

they are often called decision stumps. Each instance in the training dataset is weighted. The initial weight is set to:

$$weight(x_i) = \frac{1}{n} \tag{30.1}$$

Where x_i is the i'th training instance and n is the number of training instances.

30.3 How To Train One Model

A weak classifier (decision stump) is prepared on the training data using the weighted samples. Only binary (two-class) classification problems are supported, so each decision stump makes one decision on one input variable and outputs a +1.0 or -1.0 value for the first or second class value. The misclassification rate is calculated for the trained model. Traditionally, this is calculated as:

$$error = \frac{correct - N}{N} \tag{30.2}$$

Where *error* is the misclassification rate, correct are the number of training instance predicted correctly by the model and N is the total number of training instances. For example, if the model predicted 78 of 100 training instances correctly the error or misclassification rate would be $\frac{78-100}{100}$ or 0.22. This is modified to use the weighting of the training instances:

$$error = \frac{\sum_{i=1}^{n}(w_i \times perror_i)}{\sum_{i=1}^{n} w} \tag{30.3}$$

Which is the weighted sum of the misclassification rate, where w is the weight for training instance i and *perror* is the prediction error for training instance i which is 1 if misclassified and 0 if correctly classified. For example, if we had 3 training instances with the weights 0.01, 0.5 and 0.2. The predicted values were -1, -1 and -1, and the actual output variables in the instances were -1, 1 and -1, then the *perror* values would be 0, 1, and 0. The misclassification rate would be calculated as:

$$\begin{aligned} error &= \frac{0.01 \times 0 + 0.5 \times 1 + 0.2 \times 0}{0.01 + 0.5 + 0.2} \\ error &= 0.704 \end{aligned} \tag{30.4}$$

A stage value is calculated for the trained model which provides a weighting for any predictions that the model makes. The stage value for a trained model is calculated as follows:

$$stage = ln(\frac{1 - error}{error}) \tag{30.5}$$

Where stage is the stage value used to weight predictions from the model, $ln()$ is the natural logarithm and error is the misclassification error for the model. The effect of the stage weight is that more accurate models have more weight or contribution to the final prediction. The training weights are updated giving more weight to incorrectly predicted instances, and less weight to correctly predicted instances. For example, the weight of one training instance (w) is updated using:

$$w = w \times e^{stage \times perror} \tag{30.6}$$

Where w is the weight for a specific training instance, e is the numerical constant Euler's number raised to a power, stage is the misclassification rate for the weak classifier and *perror* is the error the weak classifier made predicting the output variable for the training instance, evaluated as:

- perror = 0 **IF** y == p
- perror = 1 **IF** y != p

Where y is the output variable for the training instance and p is the prediction from the weak learner. This has the effect of not changing the weight if the training instance was classified correctly and making the weight slightly larger if the weak learner misclassified the instance.

30.4 AdaBoost Ensemble

Weak models are added sequentially, trained using the weighted training data. The process continues until a pre-set number of weak learners have been created (a user parameter) or no further improvement can be made on the training dataset. Once completed, you are left with a pool of weak learners each with a stage value.

30.5 Making Predictions with AdaBoost

Predictions are made by calculating the weighted average of the weak classifiers. For a new input instance, each weak learner calculates a predicted value as either +1.0 or -1.0. The predicted values are weighted by each weak learners stage value. The prediction for the ensemble model is taken as a the sum of the weighted predictions. If the sum is positive, then the first class is predicted, if negative the second class is predicted.

For example, 5 weak classifiers may predict the values 1.0, 1.0, -1.0, 1.0, -1.0. From a majority vote, it looks like the model will predict a value of 1.0 or the first class. These same 5 weak classifiers may have the stage values 0.2, 0.5, 0.8, 0.2 and 0.9 respectively. Calculating the weighted sum of these predictions results in an output of -0.8, which would be an ensemble prediction of -1.0 or the second class.

30.6 Preparing Data For AdaBoost

This section lists some heuristics for best preparing your data for AdaBoost.

- **Quality Data**: Because the ensemble method continues to attempt to correct misclassification's in the training data, you need to be careful that the training data is of a high-quality.
- **Outliers**: Outliers will force the ensemble down the rabbit hole of working hard to correct for cases that are unrealistic. These could be removed from the training dataset.
- **Noisy Data**: Noisy data, specifically noise in the output variable can be problematic. If possible, attempt to isolate and clean these from your training dataset.

30.7 Summary

In this chapter you discovered the Boosting ensemble method for machine learning. You learned about:

- Boosting and how it is a general technique that keeps adding weak learners to correct classification errors.
- AdaBoost as the first successful boosting algorithm for binary classification problems.
- Learning the AdaBoost model by weighting training instances and the weak learners themselves.
- Predicting with AdaBoost by weighting predictions from weak learners.
- Where to look for more theoretical background on the AdaBoost algorithm.

You now know about the boosting ensemble method and the AdaBoost machine learning algorithm. In the next chapter you will discover how to implement the AdaBoost algorithm from scratch.

Chapter 31

AdaBoost Tutorial

AdaBoost is one of the first boosting ensemble machine learning algorithms. In this chapter you will discover how AdaBoost for machine learning works step-by step. After reading this chapter you will know:

- How to create decision stumps from training data using weighted training data.
- How to update the weights in the training data so that more attention is put on difficult to classify instances.
- How to create a sequence of 3 models one after the other.
- How to use an AdaBoost model with three weak models to make predictions.

Let's get started.

31.1 Classification Problem Dataset

In this tutorial we will use a dataset with two input variables ($X1$ and $X2$) and one output variable (Y). The input variables are real-valued random numbers drawn from a Gaussian distribution. The output variable has two values, making the problem a binary classification problem. The raw data is listed below.

```
X1            X2            Y
3.64754035    2.996793259   0
2.612663842   4.459457779   0
2.363359679   1.506982189   0
4.932600453   1.299008795   0
3.776154753   3.157451378   0
8.673960793   2.122873405   1
5.861599451   0.003512817   1
8.984677361   1.768161009   1
7.467380954   0.187045945   1
4.436284412   0.862698005   1
```

Listing 31.1: Boosting Tutorial Data Set.

Plotting the raw data, you can see that the classes are not clearly separated.

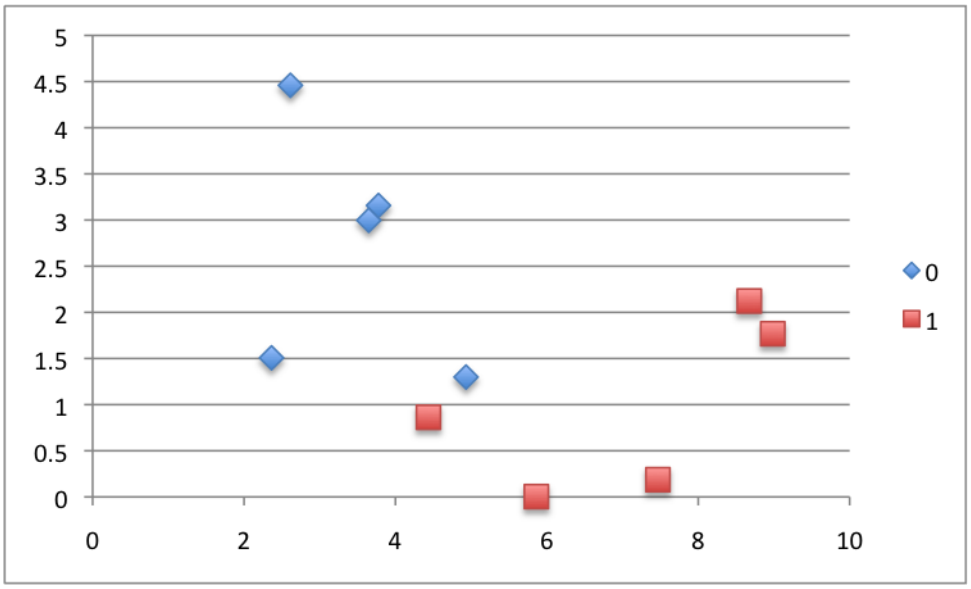

Figure 31.1: Boosting Tutorial Dataset Scatter plot.

31.2 Learn AdaBoost Model From Data

In this section we are going to learn an AdaBoost model from the training data. This will involve learning 3 models, one after the other so that we can observe the effect on the weightings to the training instances and how the predictions from each of the two models are combined. AdaBoost uses decision stump (one node decision trees) as the internal model. Rather than using the CART algorithm or similar to choose the split points for these decision trees, we will select split points manually. Again, these split points will be picked poorly to create classification errors and to demonstrate how the second model can correct the first model, and so on.

31.3 Decision Stump: Model #1

The first model will be a decision stump for the $X1$ input variable split at the value 3.64754035.

- **IF** $X1 \leq 4.932600453$ **THEN** LEFT
- **IF** $X1 > 4.932600453$ **THEN** RIGHT

This is a poor split point for this data and was chosen intentionally for this tutorial to create some misclassified instances. Ideally, you would use the CART algorithm to choose a split point for this dataset with the Gini index as a cost function. If we apply this split point to the training data we can see the data split into the following two groups:

```
Y   Group
0   LEFT
0   LEFT
0   LEFT
0   LEFT
0   LEFT
1   RIGHT
```

```
1   RIGHT
1   RIGHT
1   RIGHT
1   LEFT
```

Listing 31.2: Splitting of the Training Dataset by Model 1.

Looking at the composition of this group, we can see that the right group is predominately in class 1 and the left group is predominately class 0. These class values will be the predictions made for each group.

- LEFT: Class 0
- RIGHT: Class 1

Now that we have a decision stump model trained, we can use it to make predictions for the training dataset. The error in the prediction can be calculated using:

- error = 0 **IF** Prediction == Y
- error = 1 **IF** Prediction != Y

We can summarize the predictions using the decision stump and their errors in the table below.

```
Y       Prediction      Error
0       0               0
0       0               0
0       0               0
0       0               0
0       0               0
1       1               0
1       1               0
1       1               0
1       1               0
1       0               1
```

Listing 31.3: Predictions of the Training Dataset by Model 1.

We can see 0 errors in 10 predictions or an accuracy of 0.9 or 90%. Not bad, but not ideal. When using AdaBoost, we calculate the misclassification rate for a weak model like the above decision stump using the following equation:

$$MisclassificationRate = \frac{\sum_{i=1}^{n}(w_i \times error_i)}{\sum_{i=1}^{n} w} \tag{31.1}$$

Each instance in the training dataset has the starting weight of $\frac{1}{N}$ where N is the number of training instance, in this case 10 so $\frac{1}{10} = 0.1$.

$$\begin{aligned} weight &= \frac{1}{N} \\ weight &= \frac{1}{10} \\ weight &= 0.1 \end{aligned} \tag{31.2}$$

Using this starting weight and the prediction errors above, we can calculate the weighted error for each prediction.

$$WeightedError = weight \times error \tag{31.3}$$

The results are listed below.

```
Weight  Error   Weighted Error
0.1     0       0
0.1     0       0
0.1     0       0
0.1     0       0
0.1     0       0
0.1     0       0
0.1     0       0
0.1     0       0
0.1     0       0
0.1     1       0.1
```

Listing 31.4: Weight Errors For Model 1.

Now we can calculate the misclassification rate as:

$$\begin{aligned} MisclassificationRate &= \frac{\sum(WeightedError)}{\sum(weight)} \\ MisclassificationRate &= \frac{0.2}{1.0} \\ MisclassificationRate &= 0.2 \end{aligned} \tag{31.4}$$

Finally, we can use the misclassification rate to calculate the stage for this weak model. The stage is the weight applied to any prediction made by this model later when we use it to actually make predictions. The stage is calculated as:

$$\begin{aligned} stage &= ln(\frac{1 - MisclassificationRate}{MisclassificationRate}) \\ stage &= ln(\frac{1 - 0.1}{0.1}) \\ stage &= 2.197224577 \end{aligned} \tag{31.5}$$

This classifier will have a lot of weight on predictions, which is good because it is 90% accurate.

31.3.1 Update Instance Weights

Before we can prepare a second boosted model, we must update the instance weights. This is the very core of boosting. The weights are updated so that the next model that is created pays more attention to the training instances that the previous models got wrong and less attention to the instances that it got right. The weight for a training instance is updated using:

$$weight = weight \times e^{stage \times error} \tag{31.6}$$

We know the current weight for each training instance (0.1) and the errors made on each training instance. We now also know the stage. Updating the weights for each training instance

is therefore quite straightforward. Below are the updated weights for each training instance. You will notice that only the weights for the one instance that the first model got wrong are difference. In fact it is larger so that the next model that is created pays more attention to them.

```
Weight
0.1
0.1
0.1
0.1
0.1
0.1
0.1
0.1
0.1
0.124573094
```

Listing 31.5: Updated Instance Weights Training Model 1.

31.4 Decision Stump: Model #2

Now we can create our second weak model from the weighted training instances. This second model will also be a decision stump, but this time it will make a split on the $X2$ variable at the value 2.122873405. Again, this is a contrived model intended to have poor accuracy. Ideally, you would use something like the CART algorithm and the Gini index to choose a good quality split point. It should be noted that the CART algorithm would take the instance weight into account and focus on splitting at instances with a larger weight to result in a different decision stump. If not, the algorithm will create the same decision stump in model after model and there would be no chance to correct the predictions made from prior models. We will go through the same process using this new split point.

- **IF** $X2 \leq 2.122873405$ **THEN** LEFT
- **IF** $X2 > 2.122873405$ **THEN** RIGHT

This results in the following groups:

```
Y   Group
0   RIGHT
0   RIGHT
0   LEFT
0   LEFT
0   RIGHT
1   LEFT
1   LEFT
1   LEFT
1   LEFT
1   LEFT
```

Listing 31.6: Splitting of the Training Dataset by Model 2.

Looking at the composition for each group, the LEFT group has the class value 1 and the RIGHT group has the class value 0.

- LEFT: Class 1
- RIGHT: Class 0

Using this simple decision tree, we can now calculate a prediction for each training instance and the error and the weighted error for those predictions.

```
Y   Prediction Error  Weighted Error
0   0          0      0
0   0          0      0
0   1          1      0.1
0   1          1      0.1
0   0          0      0
1   1          0      0
1   1          0      0
1   1          0      0
1   1          0      0
1   1          0      0
```

Listing 31.7: Predictions and Weighted Error for Model 2.

A quick back of the envelope check indicates that this model was 80% accurate on the training data. Slightly worse than the first model. It also looks like it made different mistakes and correctly predicted the instances that the first model predicted incorrectly. We can calculate the stage value for this classifier:

$$\begin{aligned} &\sum weight = 1.024573094 \\ &\sum WeightedError = 0.2 \\ &MisclassificationRate = 0.1952032521 \\ &stage = 1.416548424 \end{aligned} \tag{31.7}$$

We are not done yet. If we stop there, we will not have a very accurate AdaBoost model. In fact a quick check suggests the model will only have an accuracy of 60% on the training data. You can check this yourself later by changing the next section to only combine the predictions from the first two models. We need one more model to make some final corrections.

31.5 Decision Stump: Model #3

Again, we can update the weights for the instances using the same update procedure described above. This gives us the following weights:

```
Weight
0.1
0.1
0.11521789
0.11521789
0.1
0.1
0.1
0.1
0.1
0.124573094
```

Listing 31.8: Weight Values Prior to Training Model 3.

This model will choose a split point for $X2$ at 0.862698005.

- **IF** $X2 \leq 0.862698005$ **THEN** LEFT
- **IF** $X2 > 0.862698005$ **THEN** RIGHT

This split point separates the dataset into the following groups:

```
Y   Group
0   RIGHT
0   RIGHT
0   RIGHT
0   RIGHT
0   RIGHT
1   RIGHT
1   LEFT
1   RIGHT
1   LEFT
1   LEFT
```

Listing 31.9: Splitting of the Training Dataset by Model 3.

Again, looking at the composition for each group, the LEFT group has the class value 1 and the RIGHT group has the class value 0.

- LEFT: Class 1
- RIGHT: Class 0

Using this final simple decision tree, we can now calculate a prediction for each training instance and the error and the weighted error for those predictions.

```
Y   Prediction Error  Weighted Error
0   0          0      0
0   0          0      0
0   0          0      0
0   0          0      0
0   0          0      0
1   0          1      0.1
1   1          0      0
1   0          1      0.1
1   1          0      0
1   1          0      0
```

Listing 31.10: Predictions and Weighted Error for Model 3.

This model too makes 2 errors (or has an accuracy of 80%). Again, we can calculate the stage value for this classifier:

$$\begin{aligned} &\sum(weight) = 1.024573094 \\ &\sum WeightedError = 0.2 \\ &MisclassificationRate = 0.1952032521 \\ &stage = 1.416548424 \end{aligned} \tag{31.8}$$

Now we are done. Let's look at how we can use this series of boosted models to make predictions.

31.6 Make Predictions with AdaBoost Model

We can now make predictions for the training dataset using the AdaBoost model. Predictions for a single classifier are either +1 or -1 and weighted by the stage value for the model. For example for this problem we will use:

$$prediction = stage \times (\text{IF } (output == 0) \text{ THEN } -1 \text{ ELSE } +1) \tag{31.9}$$

Using the three models above, we can make weighted predictions given the input values from the training data. This could just as easily be new data for which we would like to make predictions.

X1	X2	Model 1	Model 2	Model 3
3.64754035	2.996793259	-2.197224577	-1.416548424	-1.45279448
2.612663842	4.459457779	-2.197224577	-1.416548424	-1.45279448
2.363359679	1.506982189	-2.197224577	1.416548424	-1.45279448
4.932600453	1.299008795	-2.197224577	1.416548424	-1.45279448
3.776154753	3.157451378	-2.197224577	-1.416548424	-1.45279448
8.673960793	2.122873405	2.197224577	1.416548424	-1.45279448
5.861599451	0.003512817	2.197224577	1.416548424	1.45279448
8.984677361	1.768161009	2.197224577	1.416548424	-1.45279448
7.467380954	0.187045945	2.197224577	1.416548424	1.45279448
4.436284412	0.862698005	-2.197224577	1.416548424	1.45279448

Listing 31.11: Predictions from All 3 Models.

We can sum the predictions for each model to give a final outcome. If an outcome is less than 0 then the 0 class is predicted, and if an outcome is greater than 0 the 1 class is predicted. We can now calculate the final predictions from the AdaBoost model.

Sum	Prediction Y	Error	Accuracy
-5.066567482	0	0	100
-5.066567482	0	0	
-2.233470634	0	0	
-2.233470634	0	0	
-5.066567482	0	0	
2.160978521	1	0	
5.066567482	1	0	
2.160978521	1	0	
5.066567482	1	0	
0.672118327	1	0	

Listing 31.12: Ensemble Prediction from AdaBoost Model.

We can see that the predictions match the expected Y values perfectly. Or stated another way, the AdaBoost model achieved an accuracy of 100% on the training data.

31.7 Summary

In this chapter you discovered how to work through implementing an AdaBoost model from scratch step-by-step. You learned:

- How to create decision stumps from training data using weighted training data.
- How to update the weights in the training data so that more attention is put on difficult to classify instances.
- How to create a sequence of 3 models one after the other.
- How to use an AdaBoost model with three weak models to make predictions.

You now know how to implement the AdaBoost machine learning algorithm from scratch. This concludes your introduction to ensemble machine learning algorithms.

Part VI

Conclusions

Chapter 32

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come.

1. You started off with an interest in machine learning algorithms.
2. You learned the underlying principle for all supervised machine learning algorithms that they are estimating a mapping function from input to output variables.
3. You discovered the difference between parametric and nonparametric algorithms, supervised and unsupervised algorithms and error introduced from bias and variance.
4. You discovered and implemented linear machine learning algorithms including linear regression, logistic regression and linear discriminant analysis.
5. You implemented from scratch nonlinear machine learning algorithms including classification and regression trees, naive Bayes, k-nearest neighbors, learning vector quantization and support vector machines.
6. Finally, you discovered and implemented two of the most popular ensemble algorithms bagging with decision trees and boosting with adaboost.

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 machine learning algorithms from scratch in a spreadsheet. There is nowhere to hide in a spreadsheet, you either understand the algorithm and it works or you don't and you don't get results.

I want to take a moment and sincerely thank you for letting me help you start your journey with machine learning algorithms. I hope you keep learning and have fun as you continue to master machine learning.

Chapter 33

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.

33.1 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 are 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

33.2 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.

33.3 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

Errata

This section lists changes between different revisions of this book.

Revision 1.0

- First edition of this book.

Revision 1.1

- Fixed formatting Chapter 2, pages 7-8.