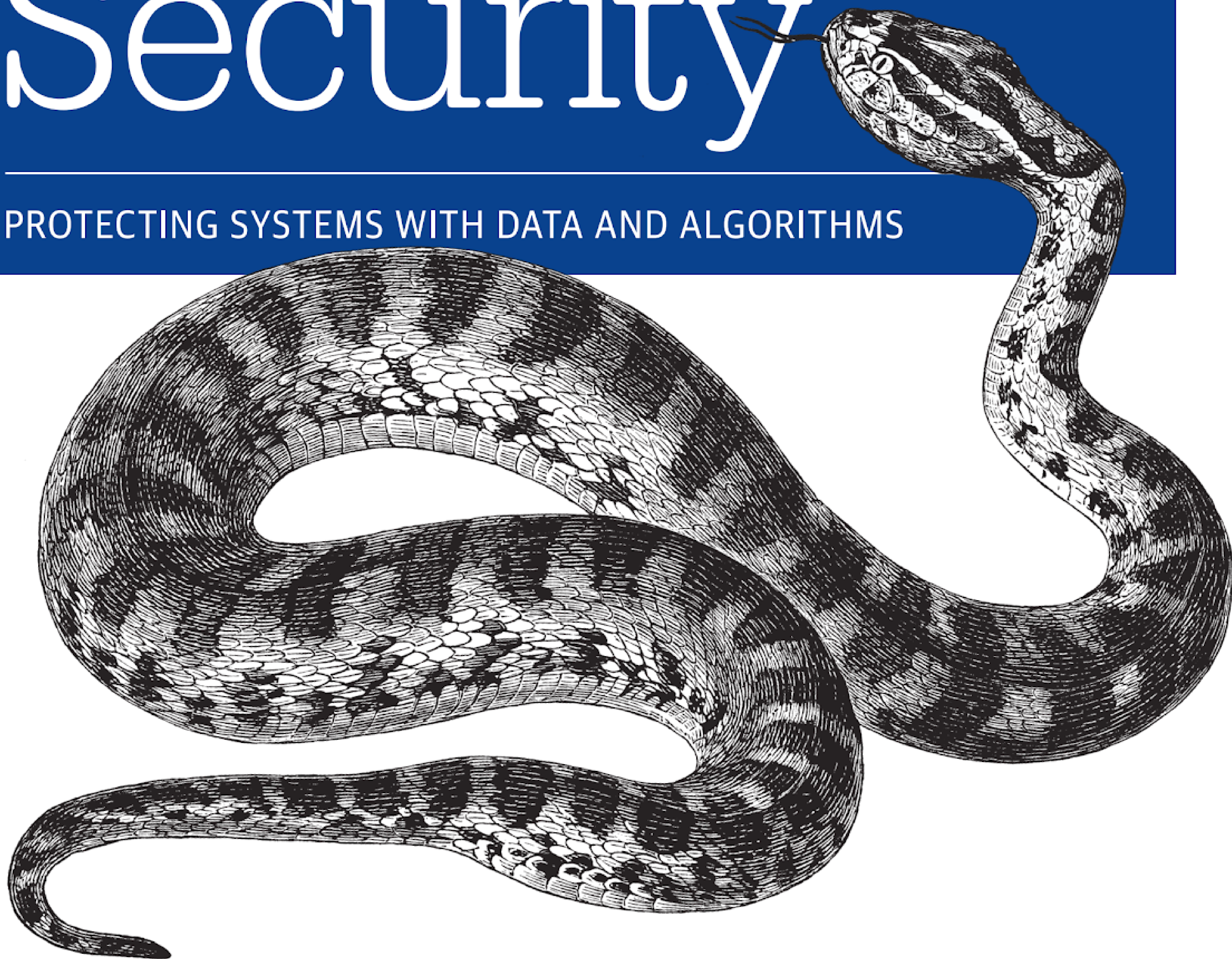
O'REILLY®

Machine Learning & Security

PROTECTING SYSTEMS WITH DATA AND ALGORITHMS



Clarence Chio & David Freeman

Copyrighted material

Machine Learning and Security

Protecting Systems with Data and Algorithms

Clarence Chio and David Freeman

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Copyrighted material

Machine Learning and Security

by Clarence Chio and David Freeman

Copyright © 2018 Clarence Chio and David Freeman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Courtney Allen
Production Editor: Kristen Brown
Copyeditor: Octal Publishing, Inc.
Proofreader: Rachel Head
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest
Tech Reviewers: Joshua Saxe, Hyrum Anderson, Jess Males, and Alex Pinto

February 2018: First Edition

Revision History for the First Edition

2018-01-26: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491979907* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Machine Learning and Security*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-97990-7

[LSI]

Copyrighted material

Table of Contents

Preface xi

1. Why Machine Learning and Security? 1
Cyber Threat Landscape 3
The Cyber Attacker's Economy 7
A Marketplace for Hacking Skills 7
Indirect Monetization 8
The Upshot 8
What Is Machine Learning? 9
What Machine Learning Is Not 10
Adversaries Using Machine Learning 11
Real-World Uses of Machine Learning in Security 12
Spam Fighting: An Iterative Approach 14
Limitations of Machine Learning in Security 23

2. Classifying and Clustering 25
Machine Learning: Problems and Approaches 25
Machine Learning in Practice: A Worked Example 27
Training Algorithms to Learn 32
Model Families 33
Loss Functions 35
Optimization 36
Supervised Classification Algorithms 40
Logistic Regression 40
Decision Trees 42
Decision Forests 45
Support Vector Machines 47
Naive Bayes 49

Copyrighted material

k-Nearest Neighbors 52
Neural Networks 53
Practical Considerations in Classification 55
Selecting a Model Family 55
Training Data Construction 56
Feature Selection 59
Overfitting and Underfitting 61
Choosing Thresholds and Comparing Models 62
Clustering 65
Clustering Algorithms 65
Evaluating Clustering Results 75
Conclusion 77

3. Anomaly Detection. 79
When to Use Anomaly Detection Versus Supervised Learning 80
Intrusion Detection with Heuristics 81
Data-Driven Methods 82
Feature Engineering for Anomaly Detection 85
Host Intrusion Detection 85
Network Intrusion Detection 89
Web Application Intrusion Detection 92
In Summary 93
Anomaly Detection with Data and Algorithms 93
Forecasting (Supervised Machine Learning) 95
Statistical Metrics 106
Goodness-of-Fit 107
Unsupervised Machine Learning Algorithms 112
Density-Based Methods 116
In Summary 118
Challenges of Using Machine Learning in Anomaly Detection 119
Response and Mitigation 120
Practical System Design Concerns 121
Optimizing for Explainability 121
Maintainability of Anomaly Detection Systems 123
Integrating Human Feedback 123
Mitigating Adversarial Effects 123
Conclusion 124

4. Malware Analysis. 125
Understanding Malware 126
Defining Malware Classification 128
Malware: Behind the Scenes 131

Copyrighted material

Feature Generation 145
Data Collection 146
Generating Features 147
Feature Selection 171
From Features to Classification 174
How to Get Malware Samples and Labels 178
Conclusion 179

5. Network Traffic Analysis. 181
Theory of Network Defense 183
Access Control and Authentication 183
Intrusion Detection 184
Detecting In-Network Attackers 185
Data-Centric Security 185
Honeypots 186
Summary 186
Machine Learning and Network Security 187
From Captures to Features 187
Threats in the Network 193
Botnets and You 197
Building a Predictive Model to Classify Network Attacks 203
Exploring the Data 205
Data Preparation 210
Classification 214
Supervised Learning 216
Semi-Supervised Learning 222
Unsupervised Learning 223
Advanced Ensembling 228
Conclusion 233

6. Protecting the Consumer Web. 235
Monetizing the Consumer Web 236
Types of Abuse and the Data That Can Stop Them 237
Authentication and Account Takeover 237
Account Creation 243
Financial Fraud 248
Bot Activity 251
Supervised Learning for Abuse Problems 256
Labeling Data 256
Cold Start Versus Warm Start 258
False Positives and False Negatives 258
Multiple Responses 259

Copyrighted material

Large Attacks 259
Clustering Abuse 260
Example: Clustering Spam Domains 261
Generating Clusters 262
Scoring Clusters 266
Further Directions in Clustering 271
Conclusion 272

7. Production Systems. 275
Defining Machine Learning System Maturity and Scalability 275
What's Important for Security Machine Learning Systems? 277
Data Quality 277
Problem: Bias in Datasets 277
Problem: Label Inaccuracy 279
Solutions: Data Quality 279
Problem: Missing Data 280
Solutions: Missing Data 281
Model Quality 284
Problem: Hyperparameter Optimization 285
Solutions: Hyperparameter Optimization 285
Feature: Feedback Loops, A/B Testing of Models 289
Feature: Repeatable and Explainable Results 293
Performance 297
Goal: Low Latency, High Scalability 297
Performance Optimization 298
Horizontal Scaling with Distributed Computing Frameworks 300
Using Cloud Services 305
Maintainability 307
Problem: Checkpointing, Versioning, and Deploying Models 307
Goal: Graceful Degradation 308
Goal: Easily Tunable and Configurable 309
Monitoring and Alerting 310
Security and Reliability 311
Feature: Robustness in Adversarial Contexts 312
Feature: Data Privacy Safeguards and Guarantees 312
Feedback and Usability 313
Conclusion 314

8. Adversarial Machine Learning. 315
Terminology 316
The Importance of Adversarial ML 317
Security Vulnerabilities in Machine Learning Algorithms 318

Copyrighted material

Attack Transferability 320
Attack Technique: Model Poisoning 322
Example: Binary Classifier Poisoning Attack 325
Attacker Knowledge 330
Defense Against Poisoning Attacks 331
Attack Technique: Evasion Attack 333
Example: Binary Classifier Evasion Attack 334
Defense Against Evasion Attacks 339
Conclusion 340

A. Supplemental Material for Chapter 2. 343

B. Integrating Open Source Intelligence. 351

Index. 355

Copyrighted material

Copyrighted material

Preface

Machine learning is eating the world. From communication and finance to transportation, manufacturing, and even agriculture,[1] nearly every technology field has been transformed by machine learning and artificial intelligence, or will soon be.

Computer security is also eating the world. As we become dependent on computers for an ever-greater proportion of our work, entertainment, and social lives, the value of breaching these systems increases proportionally, drawing in an increasing pool of attackers hoping to make money or simply wreak mischief. Furthermore, as systems become increasingly complex and interconnected, it becomes harder and harder to ensure that there are no bugs or backdoors that will give attackers a way in. Indeed, as this book went to press we learned that pretty much every microprocessor currently in use is insecure.[2]

With machine learning offering (potential) solutions to everything under the sun, it is only natural that it be applied to computer security, a field which intrinsically provides the robust data sets on which machine learning thrives. Indeed, for all the security threats that appear in the news, we hear just as many claims about how A.I. can "revolutionize" the way we deal with security. Because of the promise that it holds for nullifying some of the most complex advances in attacker competency, machine learning has been touted as the technique that will finally put an end to the cat-and-mouse game between attackers and defenders. Walking the expo floors of major security conferences, the trend is apparent: more and more companies are embracing the use of machine learning to solve security problems.

Mirroring the growing interest in the marriage of these two fields, there is a corresponding air of cynicism that dismisses it as hype. So how do we strike a balance?

1 Monsanto, "How Machine Learning is Changing Modern Agriculture," *Modern Agriculture*, September 13, 2017, *https://modernag.org/innovation/machine-learning-changing-modern-agriculture/*.

2 "Meltdown and Spectre," Graz University of Technology, accessed January 23, 2018, *https://spectreattack.com/*.

Copyrighted material

What is the true potential of A.I. applied to security? How can you distinguish the marketing fluff from promising technologies? What should I actually use to solve my security problems? The best way we can think of to answer these questions is to dive deep into the science, understand the core concepts, do lots of testing and experimentation, and let the results speak for themselves. However, doing this requires a working knowledge of both data science and computer security. In the course of our work building security systems, leading anti-abuse teams, and speaking at conferences, we have met a few people who have this knowledge, and many more who understand one side and want to learn about the other.

This book is the result.

What's In This Book?

We wrote this book to provide a framework for discussing the inevitable marriage of two ubiquitous concepts: machine learning and security. While there is some literature on the intersection of these subjects (and multiple conference workshops: CCS's AISec, AAAI's AICS, and NIPS's Machine Deception), most of the existing work is academic or theoretical. In particular, we did not find a guide that provides concrete, worked examples with code that can educate security practitioners about data science and help machine learning practitioners think about modern security problems effectively.

In examining a broad range of topics in the security space, we provide examples of how machine learning can be applied to augment or replace rule-based or heuristic solutions to problems like intrusion detection, malware classification, or network analysis. In addition to exploring the core machine learning algorithms and techniques, we focus on the challenges of building maintainable, reliable, and scalable data mining systems in the security space. Through worked examples and guided discussions, we show you how to think about data in an adversarial environment and how to identify the important signals that can get drowned out by noise.

Who Is This Book For?

If you are working in the security field and want to use machine learning to improve your systems, this book is for you. If you have worked with machine learning and now want to use it to solve security problems, this book is also for you.

We assume you have some basic knowledge of statistics; most of the more complex math can be skipped upon your first reading without losing the concepts. We also assume familiarity with a programming language. Our examples are in Python and we provide references to the Python packages required to implement the concepts we discuss, but you can implement the same concepts using open source libraries in Java, Scala, C++, Ruby, and many other languages.

Copyrighted material

Conventions Used in This Book

The following typographical conventions are used in this book:

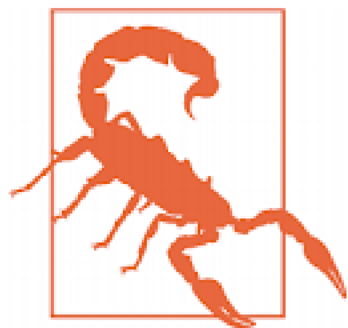
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords. Also used for commands and command-line output.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user. Also used for emphasis in command-line output.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip, suggestion, or general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreilly-mlsec/book-resources*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a signifi-

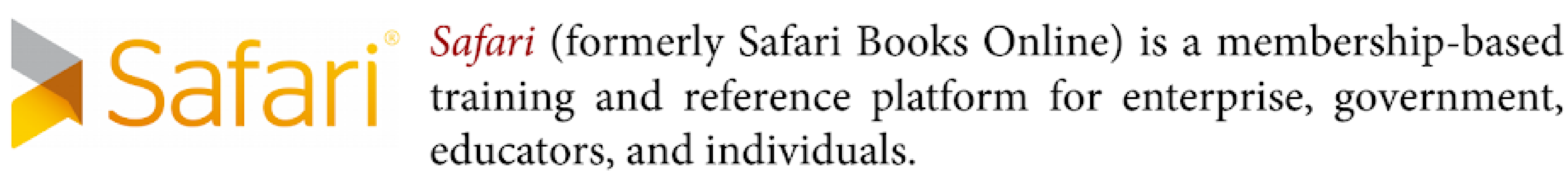
Copyrighted material

cant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Machine Learning and Security* by Clarence Chio and David Freeman (O'Reilly). Copyright 2018 Clarence Chio and David Freeman, 978-1-491-97990-7."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Safari



Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

O'Reilly Media has a web page for this book, where they list errata, examples, and any additional information. You can access this page at *http://bit.ly/machineLearningAndSecurity*. The authors have created a website for the book at *https://mlsec.net*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

Copyrighted material

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

The authors thank Hyrum Anderson, Jason Craig, Nwokedi Idika, Jess Males, Andy Oram, Alex Pinto, and Joshua Saxe for thorough technical reviews and feedback on early drafts of this work. We also thank Virginia Wilson, Kristen Brown, and all the staff at O'Reilly who helped us take this project from concept to reality.

Clarence thanks Christina Zhou for tolerating the countless all-nighters and weekends spent on this book, Yik Lun Lee for proofreading drafts and finding mistakes in my code, Jarrod Overson for making me believe I could do this, and Daisy the Chihuahua for being at my side through the toughest of times. Thanks to Anto Joseph for teaching me security, to all the other hackers, researchers, and training attendees who have influenced this book in one way or another, to my colleagues at Shape Security for making me a better engineer, and to Data Mining for Cyber Security speakers and attendees for being part of the community that drives this research. Most of all, thanks to my family in Singapore for supporting me from across the globe and enabling me to chase my dreams and pursue my passion.

David thanks Deepak Agarwal for convincing me to undertake this effort, Dan Boneh for teaching me how to think about security, and Vicente Silveira and my colleagues at LinkedIn and Facebook for showing me what security is like in the real world. Thanks also to Grace Tang for feedback on the machine learning sections as well as the occasional penguin. And the biggest thanks go to Torrey, Elodie, and Phoebe, who put up with me taking many very late nights and a few odd excursions in order to complete this book, and never wavered in their support.

Copyrighted material

Copyrighted material

CHAPTER 1

Why Machine Learning and Security?

In the beginning, there was spam.

As soon as academics and scientists had hooked enough computers together via the internet to create a communications network that provided value, other people realized that this medium of free transmission and broad distribution was a perfect way to advertise sketchy products, steal account credentials, and spread computer viruses.

In the intervening 40 years, the field of computer and network security has come to encompass an enormous range of threats and domains: intrusion detection, web application security, malware analysis, social network security, advanced persistent threats, and applied cryptography, just to name a few. But even today spam remains a major focus for those in the email or messaging space, and for the general public spam is probably the aspect of computer security that most directly touches their own lives.

Machine learning was not invented by spam fighters, but it was quickly adopted by statistically inclined technologists who saw its potential in dealing with a constantly evolving source of abuse. Email providers and internet service providers (ISPs) have access to a wealth of email content, metadata, and user behavior. Using email data, content-based models can be built to create a generalizable approach to recognize spam. Metadata and entity reputations can be extracted from emails to predict the likelihood that an email is spam without even looking at its content. By instantiating a user behavior feedback loop, the system can build a collective intelligence and improve over time with the help of its users.

Email filters have thus gradually evolved to deal with the growing diversity of circumvention methods that spammers have thrown at them. Even though 85% of all emails sent today are spam (according to one research group), the best modern spam filters block more than 99.9% of all spam, and it is a rarity for users of major email services

Copyrighted material

to see unfiltered and undetected spam in their inboxes. These results demonstrate an enormous advance over the simplistic spam filtering techniques developed in the early days of the internet, which made use of simple word filtering and email metadata reputation to achieve modest results.

The fundamental lesson that both researchers and practitioners have taken away from this battle is the importance of using data to defeat malicious adversaries and improve the quality of our interactions with technology. Indeed, the story of spam fighting serves as a representative example for the use of data and machine learning in any field of computer security. Today, almost all organizations have a critical reliance on technology, and almost every piece of technology has security vulnerabilities. Driven by the same core motivations as the spammers from the 1980s (unregulated, cost-free access to an audience with disposable income and private information to offer), malicious actors can pose security risks to almost all aspects of modern life. Indeed, the fundamental nature of the battle between attacker and defender is the same in all fields of computer security as it is in spam fighting: a motivated adversary is constantly trying to misuse a computer system, and each side races to fix or exploit the flaws in design or technique before the other uncovers it. The problem statement has not changed one bit.

Computer systems and web services have become increasingly centralized, and many applications have evolved to serve millions or even billions of users. Entities that become arbiters of information are bigger targets for exploitation, but are also in the perfect position to make use of the data and their user bases to achieve better security. Coupled with the advent of powerful data crunching hardware and the development of more powerful data analysis and machine learning algorithms, there has never been a better time for exploiting the potential of machine learning in security.

In this book, we demonstrate applications of machine learning and data analysis techniques to various problem domains in security and abuse. We explore methods for evaluating the suitability of different machine learning techniques in different scenarios, and focus on guiding principles that will help you use data to achieve better security. Our goal is not to leave you with the answer to every security problem you might face, but rather to give you a framework for thinking about data and security as well as a toolkit from which you can pick the right method for the problem at hand.

The remainder of this chapter sets up context for the rest of the book: we discuss what threats modern computer and network systems face, what machine learning is, and how machine learning applies to the aforementioned threats. We conclude with a detailed examination of approaches to spam fighting, which provides a concrete example of applying machine learning to security that can be generalized to nearly any domain.

Copyrighted material

Cyber Threat Landscape

The landscape of adversaries and miscreants in computer security has evolved over time, but the general categories of threats have remained the same. Security research exists to stymie the goals of attackers, and it is always important to have a good understanding of the different types of attacks that exist in the wild. As you can see from the Cyber Threat Taxonomy tree in Figure 1-1,[1] the relationships between threat entities and categories can be complex in some cases.

We begin by defining the principal threats that we will explore in the chapters that follow:

Malware (or virus)
: Short for "malicious software," any software designed to cause harm or gain unauthorized access to computer systems.

Worm
: Standalone malware that replicates itself in order to spread to other computer systems.

Trojan
: Malware disguised as legitimate software to avoid detection.

Spyware
: Malware installed on a computer system without permission and/or knowledge by the operator, for the purposes of espionage and information collection. *Keyloggers* fall into this category.

Adware
: Malware that injects unsolicited advertising material (e.g., pop ups, banners, videos) into a user interface, often when a user is browsing the web.

Ransomware
: Malware designed to restrict availability of computer systems until a sum of money (ransom) is paid.

Rootkit
: A collection of (often) low-level software designed to enable access to or gain control of a computer system. ("Root" denotes the most powerful level of access to a system.)

1 Adapted from the European CSIRT Network project's Security Incidents Taxonomy.

Copyrighted material

Backdoor
: An intentional hole placed in the system perimeter to allow for future accesses that can bypass perimeter protections.

Bot
: A variant of malware that allows attackers to remotely take over and control computer systems, making them zombies.

Botnet
: A large network of bots.

Exploit
: A piece of code or software that exploits specific vulnerabilities in other software applications or frameworks.

Scanning
: Attacks that send a variety of requests to computer systems, often in a brute-force manner, with the goal of finding weak points and vulnerabilities as well as information gathering.

Sniffing
: Silently observing and recording network and in-server traffic and processes without the knowledge of network operators.

Keylogger
: A piece of hardware or software that (often covertly) records the keys pressed on a keyboard or similar computer input device.

Spam
: Unsolicited bulk messaging, usually for the purposes of advertising. Typically email, but could be SMS or through a messaging provider (e.g., WhatsApp).

Login attack
: Multiple, usually automated, attempts at guessing credentials for authentication systems, either in a brute-force manner or with stolen/purchased credentials.

Account takeover (ATO)
: Gaining access to an account that is not your own, usually for the purposes of downstream selling, identity theft, monetary theft, and so on. Typically the goal of a login attack, but also can be small scale and highly targeted (e.g., spyware, social engineering).

Phishing (aka masquerading)
: Communications with a human who pretends to be a reputable entity or person in order to induce the revelation of personal information or to obtain private assets.

Copyrighted material

Spear phishing
: Phishing that is targeted at a particular user, making use of information about that user gleaned from outside sources.

Social engineering
: Information exfiltration (extraction) from a human being using nontechnical methods such as lying, trickery, bribery, blackmail, and so on.

Incendiary speech
: Discriminatory, discrediting, or otherwise harmful speech targeted at an individual or group.

Denial of service (DoS) and distributed denial of service (DDoS)
: Attacks on the availability of systems through high-volume bombardment and/or malformed requests, often also breaking down system integrity and reliability.

Advanced persistent threats (APTs)
: Highly targeted networks or host attack in which a stealthy intruder remains intentionally undetected for long periods of time in order to steal and exfiltrate data.

Zero-day vulnerability
: A weakness or bug in computer software or systems that is unknown to the vendor, allowing for potential exploitation (called a zero-day attack) before the vendor has a chance to patch/fix the problem.

Copyrighted material

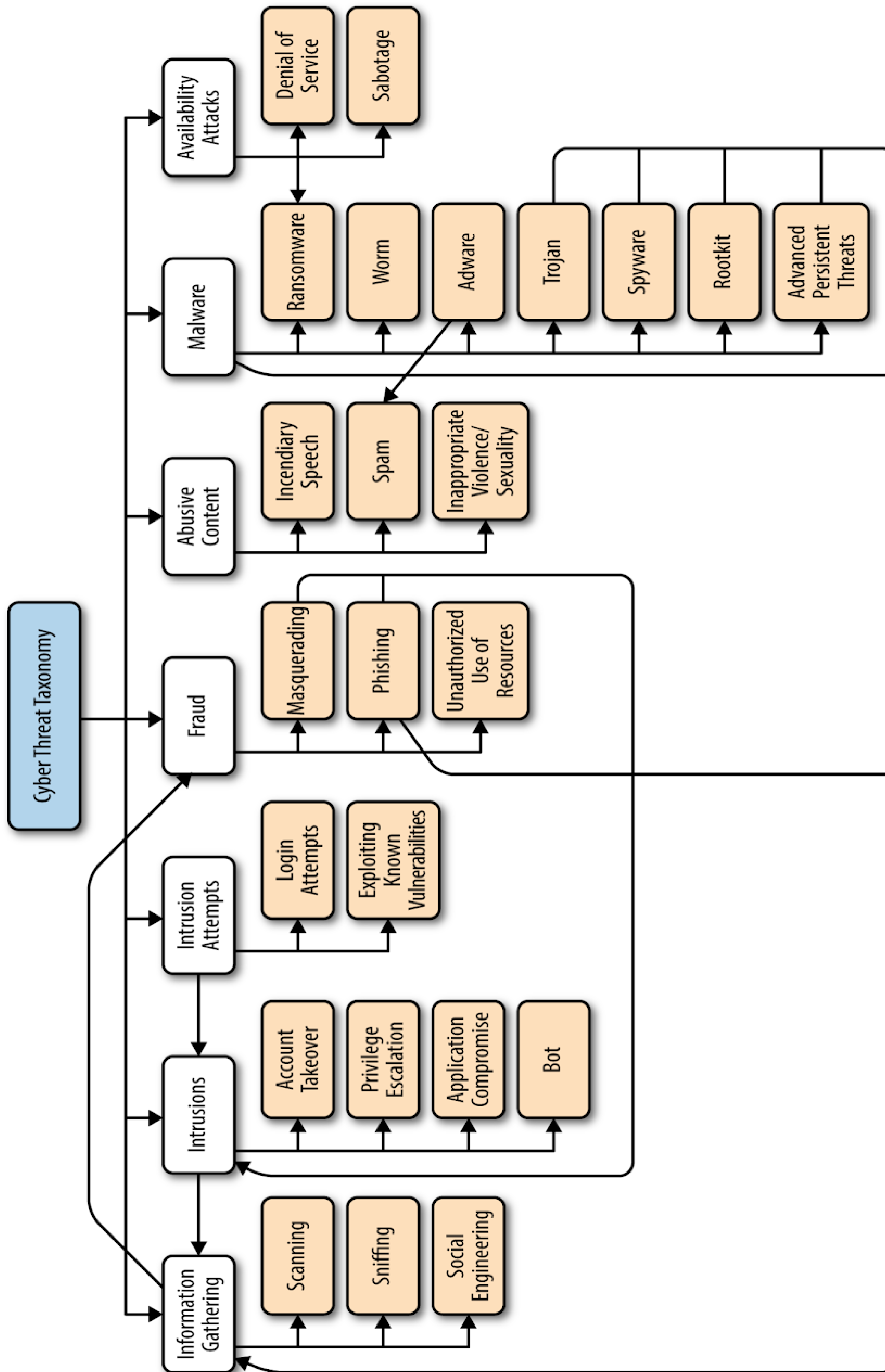


Figure 1-1. Cyber Threat Taxonomy tree

Copyrighted material

The Cyber Attacker's Economy

What drives attackers to do what they do? Internet-based criminality has become increasingly commercialized since the early days of the technology's conception. The transformation of cyber attacks from a reputation economy ("street cred," glory, mischief) to a cash economy (direct monetary gains, advertising, sale of private information) has been a fascinating process, especially from the point of view of the adversary. The motivation of cyber attackers today is largely monetary. Attacks on financial institutions or conduits (online payment platforms, stored value/gift card accounts, Bitcoin wallets, etc.) can obviously bring attackers direct financial gains. But because of the higher stakes at play, these institutions often have more advanced defense mechanisms in place, making the lives of attackers tougher. Because of the allure of a more direct path to financial yield, the marketplace for vulnerabilities targeting such institutions is also comparatively crowded and noisy. This leads miscreants to target entities with more relaxed security measures in place, abusing systems that are open by design and resorting to more indirect techniques that will eventually still allow them to monetize.

A Marketplace for Hacking Skills

The fact that *darknet* marketplaces and illegal hacking forums exist is no secret. Before the existence of organized underground communities for illegal exchanges, only the most competent of computer hackers could partake in the launching of cyber attacks and the compromising of accounts and computer systems. However, with the commoditization of hacking and the ubiquitization of computer use, lower-skilled "hackers" can participate in the ecosystem of cyber attacks by purchasing vulnerabilities and user-friendly hacking scripts, software, and tools to engage in their own cyber attacks.

The zero-day vulnerability marketplace has variants that exist both legally and illegally. Trading vulnerabilities and exploits can become a viable source of income for both security researchers and computer hackers.[2] Increasingly, the most elite computer hackers are not the ones unleashing zero-days and launching attack campaigns. The risks are just too high, and the process of monetization is just too long and uncertain. Creating software that empowers the common *script-kiddy* to carry out the actual hacking, selling vulnerabilities on marketplaces, and in some cases even providing boutique hacking consulting services promises a more direct and certain path to financial gain. Just as in the California Gold Rush of the late 1840s, merchants

2 Charlie Miller, "The Legitimate Vulnerability Market: Inside the Secretive World of 0-day Exploit Sales," *Proceedings of the 6th Workshop on the Economics of Information Security* (2007).

Copyrighted material

providing amenities to a growing population of wealth-seekers are more frequently the receivers of windfalls than the seekers themselves.

Indirect Monetization

The process of monetization for miscreants involved in different types of computer attacks is highly varied, and worthy of detailed study. We will not dive too deep into this investigation, but we will look at a couple of examples of how indirect monetization can work.

Malware distribution has been commoditized in a way similar to the evolution of cloud computing and Infrastructure-as-a-Service (IaaS) providers. The *pay-per-install* (PPI) marketplace for malware propagation is a complex and mature ecosystem, providing wide distribution channels available to malware authors and purchasers.[3] Botnet rentals operate on the same principle as on-demand cloud infrastructure, with per-hour resource offerings at competitive prices. Deploying malware on remote servers can also be financially rewarding in its own different ways. Targeted attacks on entities are sometimes associated with a bounty, and ransomware distributions can be an efficient way to extort money from a wide audience of victims.

Spyware can assist in the stealing of private information, which can then be sold in bulk on the same online marketplaces where the spyware is sold. Adware and spam can be used as a cheap way to advertise dodgy pharmaceuticals and financial instruments. Online accounts are frequently taken over for the purposes of retrieving some form of stored value, such as gift cards, loyalty points, store credit, or cash rewards. Stolen credit card numbers, Social Security numbers, email accounts, phone numbers, addresses, and other private information can be sold online to criminals intent on identity theft, fake account creation, fraud, and so on. But the path to monetization, in particular when you have a victim's credit card number, can be a long and complex one. Because of how easily this information is stolen, credit card companies, as well as companies that operate accounts with stored value, often engineer clever ways to stop attackers from monetizing. For instance, accounts suspected of having been compromised can be invalidated, or cashing out gift cards can require additional authentication steps.

The Upshot

The motivations of cyber attackers are complex and the paths to monetization are convoluted. However, the financial gains from internet attacks can be a powerful motivator for technically skilled people, especially those in less-wealthy nations and

3 Juan Caballero et al., "Measuring Pay-per-Install: The Commoditization of Malware Distribution," *Proceedings of the 20th USENIX Conference on Security* (2011).

Copyrighted material

communities. As long as computer attacks can continue to generate a non-negligible yield for the perpetrators, they will keep coming.

What Is Machine Learning?

Since the dawn of the technological age, researchers have dreamed of teaching computers to reason and make "intelligent" decisions in the way that humans do, by drawing generalizations and distilling concepts from complex information sets without explicit instructions.

Machine learning refers to one aspect of this goal—specifically, to algorithms and processes that "learn" in the sense of being able to generalize past data and experiences in order to predict future outcomes. At its core, machine learning is a set of mathematical techniques, implemented on computer systems, that enables a process of information mining, pattern discovery, and drawing inferences from data.

At the most general level, *supervised* machine learning methods adopt a Bayesian approach to knowledge discovery, using probabilities of previously observed events to infer the probabilities of new events. *Unsupervised* methods draw abstractions from unlabeled datasets and apply these to new data. Both families of methods can be applied to problems of *classification* (assigning observations to categories) or *regression* (predicting numerical properties of an observation).

Suppose that we want to classify a group of animals into mammals and reptiles. With a supervised method, we will have a set of animals for which we are definitively told their category (e.g., we are told that the dog and elephant are mammals and the alligator and iguana are reptiles). We then try to extract some features from each of these labeled data points and find similarities in their properties, allowing us to differentiate animals of different classes. For instance, we see that the dog and the elephant both give birth to live offspring, unlike the alligator and the iguana. The binary property "gives birth to live offspring" is what we call a *feature*, a useful abstraction for observed properties that allows us to perform comparisons between different observations. After extracting a set of features that might help differentiate mammals and reptiles in the labeled data, we then can run a learning algorithm on the labeled data and apply what the algorithm learned to new, unseen animals. When the algorithm is presented with a meerkat, it now must classify it as either a mammal or a reptile. Extracting the set of features from this new animal, the algorithm knows that the meerkat does not lay eggs, has no scales, and is warm-blooded. Driven by prior observations, it makes a category prediction that the meerkat is a mammal, and it is exactly right.

In the unsupervised case, the premise is similar, but the algorithm is not presented with the initial set of labeled animals. Instead, the algorithm must group the different sets of data points in a way that will result in a binary classification. Seeing that most

Copyrighted material

animals that don't have scales do give birth to live offspring and are also warm-blooded, and most animals that have scales lay eggs and are cold-blooded, the algorithm can then derive the two categories from the provided set and make future predictions in the same way as in the supervised case.

Machine learning algorithms are driven by mathematics and statistics, and the algorithms that discover patterns, correlations, and anomalies in the data vary widely in complexity. In the coming chapters, we go deeper into the mechanics of some of the most common machine learning algorithms used in this book. This book will not give you a complete understanding of machine learning, nor will it cover much of the mathematics and theory in the subject. What it will give you is critical intuition in machine learning and practical skills for designing and implementing intelligent, adaptive systems in the context of security.

What Machine Learning Is Not

Artificial intelligence (AI) is a popular but loosely defined term that indicates algorithmic solutions to complex problems typically solved by humans. As illustrated in Figure 1-2, machine learning is a core building block for AI. For example, self-driving cars must classify observed images as people, cars, trees, and so on; they must predict the position and speed of other cars; they must determine how far to rotate the wheels in order to make a turn. These classification and prediction problems are solved using machine learning, and the self-driving system is a form of AI. There are other parts of the self-driving AI decision engine that are hardcoded into rule engines, and that would not be considered machine learning. Machine learning helps us create AI, but is not the only way to achieve it.

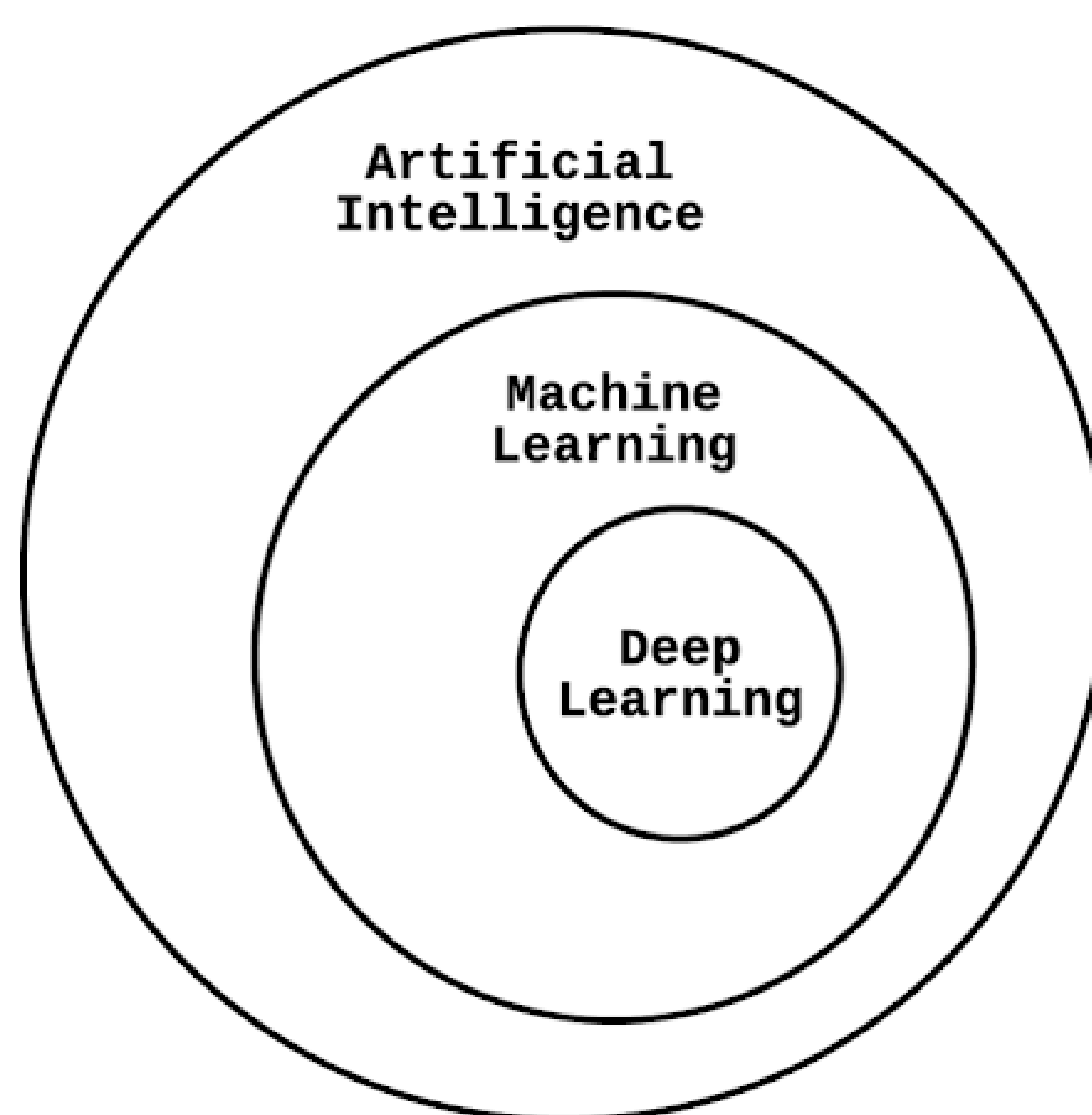


Figure 1-2. Artificial intelligence as it relates to machine learning and deep learning

Copyrighted material

Deep learning is another popular term that is commonly conflated with machine learning. Deep learning is a strict subset of machine learning referring to a specific class of multilayered models that use layers of simpler statistical components to learn representations of data. "Neural network" is a more general term for this type of layered statistical learning architecture that might or might not be "deep" (i.e., have many layers). For an excellent discussion of this topic, see *Deep Learning* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (MIT Press).

Statistical analysis is a core part of machine learning: outputs of machine learning algorithms are often presented in terms of probabilities and confidence intervals. We will touch on some statistical techniques in our discussion of anomaly detection, but we will leave aside questions regarding experimentation and statistical hypothesis testing. For an excellent discussion of this topic, see *Probability & Statistics for Engineers & Scientists* by Ronald Walpole et al. (Prentice Hall).

What Is AI?

The definition of AI is a slightly more contentious topic than the definition of machine learning. *Machine learning* refers to statistical learning algorithms that are able to create generalizable abstractions (models) by seeing and dissecting a dataset. AI systems have been loosely defined to be machine-driven decision engines that can achieve *near*-human-level intelligence. How *near* does this intelligence have to be to human intelligence before we consider it to be AI? As you might imagine, differing expectations and definitions of the term make it quite difficult to draw universally agreeable boundaries around this.

Adversaries Using Machine Learning

Note that nothing prevents adversaries from taking advantage of machine learning to avoid detection and evade defenses. As much as the defenders can learn from the attacks and adjust their countermeasures accordingly, attackers can also learn the nature of defenses to their own benefit. Spammers have been known to apply polymorphism (i.e., changing the appearance of content without changing its meaning) to their payloads to circumvent detection, or to probe spam filters by performing A/B tests on email content and learning what causes their click-through rates to rise and fall. Both good guys and bad guys use machine learning in fuzzing campaigns to speed up the process of finding vulnerabilities in software. Adversaries can even use machine learning to learn about your personality and interests through social media in order to craft the perfect phishing message for you.

Finally, the use of dynamic and adaptive methods in the area of security always contains a certain degree of risk. Especially when explainability of machine learning predictions is often lacking, attackers have been known to cause various algorithms to

Copyrighted material

make erroneous predictions or learn the wrong thing.[4] In this growing field of study called *adversarial machine learning*, attackers with varying degrees of access to a machine learning system can execute a range of attacks to achieve their ends. Chapter 8 is dedicated to this topic, and paints a more complete picture of the problems and solutions in this space.

Machine learning algorithms are often not designed with security in mind, and are often vulnerable in the face of attempts made by a motivated adversary. Hence, it is important to maintain an awareness of such threat models when designing and building machine learning systems for security purposes.

Real-World Uses of Machine Learning in Security

In this book, we explore a range of different computer security applications for which machine learning has shown promising results. Applying machine learning and data science to solve problems is not a straightforward task. Although convenient programming libraries remove some complexity from the equation, developers still need to make many decisions along the way.

By going through different examples in each chapter, we will explore the most common issues faced by practitioners when designing machine learning systems, whether in security or otherwise. The applications described in this book are not new, and you also can find the data science techniques we discuss at the core of many computer systems that you might interact with on a daily basis.

We can classify machine learning's use cases in security into two broad categories: *pattern recognition* and *anomaly detection*. The line differentiating pattern recognition and anomaly detection is sometimes blurry, but each task has a clearly distinguished goal. In pattern recognition, we try to discover explicit or latent characteristics hidden in the data. These characteristics, when distilled into feature sets, can be used to teach an algorithm to recognize other forms of the data that exhibit the same set of characteristics. Anomaly detection approaches knowledge discovery from the other side of the same coin. Instead of learning specific patterns that exist within certain subsets of the data, the goal is to establish a notion of normality that describes most (say, more than 95%) of a given dataset. Thereafter, deviations from this normality of any sort will be detected as anomalies.

It is common to erroneously think of anomaly detection as the process of recognizing a set of normal patterns and differentiating it from a set of abnormal patterns. Patterns extracted through pattern recognition must be strictly derived from the observed data used to train the algorithm. On the other hand, in anomaly detection

4 Ling Huang et al., "Adversarial Machine Learning," *Proceedings of the 4th ACM Workshop on Artificial Intelligence and Security* (2011): 43–58.

Copyrighted material

there can be an infinite number of anomalous patterns that fit the bill of an outlier, even those derived from hypothetical data that do not exist in the training or testing datasets.

Spam detection is perhaps the classic example of pattern recognition because spam typically has a largely predictable set of characteristics, and an algorithm can be trained to recognize those characteristics as a pattern by which to classify emails. Yet it is also possible to think of spam detection as an anomaly detection problem. If it is possible to derive a set of features that describes normal traffic well enough to treat significant deviations from this normality as spam, we have succeeded. In actuality, however, spam detection might not be suitable for the anomaly detection paradigm, because it is not difficult to convince yourself that it is in most contexts easier to find similarities between spam messages than within the broad set of normal traffic.

Malware detection and botnet detection are other applications that fall clearly in the category of pattern recognition, where machine learning becomes especially useful when the attackers employ polymorphism to avoid detection. *Fuzzing* is the process of throwing arbitrary inputs at a piece of software to force the application into an unintended state, most commonly to force a program to crash or be put into a vulnerable mode for further exploitation. Naive fuzzing campaigns often run into the problem of having to iterate over an intractably large application state space. The most widely used fuzzing software has optimizations that make fuzzing much more efficient than blind iteration. Machine learning has also been used in such optimizations, by learning patterns of previously found vulnerabilities in similar programs and guiding the fuzzer to similarly vulnerable code paths or idioms for potentially quicker results.

For user authentication and behavior analysis, the delineation between pattern recognition and anomaly detection becomes less clear. For cases in which the threat model is clearly known, it might be more suitable to approach the problem through the lens of pattern recognition. In other cases, anomaly detection can be the answer. In many cases, a system might make use of both approaches to achieve better coverage. Network outlier detection is a classic example of anomaly detection because most network traffic follows strict protocols and normal behavior matches a set of patterns in form or sequence. Any malicious network activity that does not manage to masquerade well by mimicking normal traffic will be caught by outlier detection algorithms. Other network-related detection problems, such as malicious URL detection, can also be approached from the angle of anomaly detection.

Access control refers to any set of policies governing the ability of system users to access certain pieces of information. Frequently used to protect sensitive information from unnecessary exposure, access control policies are often the first line of defense against breaches and information theft. Machine learning has gradually found its way into access control solutions because of the pains experienced by system users at the

Copyrighted material

mercy of rigid and unforgiving access control policies.[5] Through a combination of unsupervised learning and anomaly detection, such systems can infer information access patterns for certain users or roles in an organization and engage in retaliatory action when an unconventional pattern is detected.

Imagine, for example, a hospital's patient record storage system, where nurses and medical technicians frequently need to access individual patient data but don't necessarily need to do cross-patient correlations. Doctors, on the other hand, frequently query and aggregate the medical records of multiple patients to look for case similarities and diagnostic histories. We don't necessarily want to prevent nurses and medical technicians from querying multiple patient records because there might be rare cases that warrant such actions. A strict rule-based access control system would not be able to provide the flexibility and adaptability that machine learning systems can provide.

In the rest of this book, we dive deeper into a selection of these real-world applications. We then will be able to discuss the nuances around applying machine learning for pattern recognition and anomaly detection in security. In the remainder of this chapter, we focus on the example of spam fighting as one that illustrates the core principles used in any application of machine learning to security.

Spam Fighting: An Iterative Approach

As discussed earlier, the example of spam fighting is both one of the oldest problems in computer security and one that has been successfully attacked with machine learning. In this section, we dive deep into this topic and show how to gradually build up a sophisticated spam classification system using machine learning. The approach we take here will generalize to many other types of security problems, including but not limited to those discussed in later chapters of this book.

Consider a scenario in which you are asked to solve the problem of rampant email spam affecting employees in an organization. For whatever reason, you are instructed to develop a custom solution instead of using commercial options. Provided with administrator access to the private email servers, you are able to extract a body of emails for analysis. All the emails are properly tagged by recipients as either "spam" or "ham" (non-spam), so you don't need to spend too much time cleaning the data.[6]

Human beings do a good job at recognizing spam, so you begin by implementing a simple solution that approximates a person's thought process while executing this task. Your theory is that the presence or absence of some prominent keywords in an

---

5 Evan Martin and Tao Xie, "Inferring Access-Control Policy Properties via Machine Learning," *Proceedings of the 7th IEEE International Workshop on Policies for Distributed Systems and Networks* (2006): 235–238.

6 In real life, you will spend a large proportion of your time cleaning the data in order to make it available to and useful for your algorithms.

Copyrighted material

email is a strong binary indicator of whether the email is spam or ham. For instance, you notice that the word "lottery" appears in the spam data a lot, but seldom appears in regular emails. Perhaps you could come up with a list of similar words and perform the classification by checking whether a piece of email contains any words that belong to this blacklist.

The dataset that we will use to explore this problem is the 2007 TREC Public Spam Corpus. This is a lightly cleaned raw email message corpus containing 75,419 messages collected from an email server over a three-month period in 2007. One-third of the dataset is made up of spam examples, and the rest is ham. This dataset was created by the Text REtrieval Conference (TREC) Spam Track in 2007, as part of an effort to push the boundaries of state-of-the-art spam detection.

For evaluating how well different approaches work, we will go through a simple validation process.[7] We split the dataset into nonoverlapping training and test sets, in which the training set consists of 70% of the data (an arbitrarily chosen proportion) and the test set consists of the remaining 30%. This method is standard practice for assessing how well an algorithm or model developed on the basis of the training set will generalize to an independent dataset.

The first step is to use the Natural Language Toolkit (NLTK) to remove morphological affixes from words for more flexible matching (a process called *stemming*). For instance, this would reduce the words "congratulations" and "congrats" to the same stem word, "congrat." We also remove *stopwords* (e.g., "the," "is," and "are,") before the token extraction process, because they typically do not contain much meaning. We define a set of functions[8] to help with loading and preprocessing the data and labels, as demonstrated in the following code:[9]

```
import string
import email
import nltk

punctuations = list(string.punctuation)
stopwords = set(nltk.corpus.stopwords.words('english'))
```

7 This validation process, sometimes referred to as *conventional validation*, is not as rigorous a validation method as *cross-validation*, which refers to a class of methods that repeatedly generate all (or many) different possible splits of the dataset (into training and testing sets), performing validation of the machine learning prediction algorithm separately on each of these. The result of cross-validation is the average prediction accuracy across these different splits. Cross-validation estimates model accuracy better than conventional validation because it avoids the pitfall of information loss from a single train/test split that might not adequately capture the statistical properties of the data (this is typically not a concern if the training set is sufficiently large). Here we chose to use conventional validation for simplicity.

8 These helper functions are defined in the file *chapter1/email_read_util.py* in our code repository.

9 To run this code, you need to install the Punkt Tokenizer Models and the stopwords corpus in NLTK using the `nltk.download()` utility.

Copyrighted material

```
stemmer = nltk.PorterStemmer()

# Combine the different parts of the email into a flat list of strings
def flatten_to_string(parts):
    ret = []
    if type(parts) == str:
        ret.append(parts)
    elif type(parts) == list:
        for part in parts:
            ret += flatten_to_string(part)
    elif parts.get_content_type == 'text/plain':
        ret += parts.get_payload()
    return ret

# Extract subject and body text from a single email file
def extract_email_text(path):
    # Load a single email from an input file
    with open(path, errors='ignore') as f:
        msg = email.message_from_file(f)
    if not msg:
        return ""

    # Read the email subject
    subject = msg['Subject']
    if not subject:
        subject = ""

    # Read the email body
    body = ' '.join(m for m in flatten_to_string(msg.get_payload())
                    if type(m) == str)
    if not body:
        body = ""

    return subject + ' ' + body

# Process a single email file into stemmed tokens
def load(path):
    email_text = extract_email_text(path)
    if not email_text:
        return []

    # Tokenize the message
    tokens = nltk.word_tokenize(email_text)

    # Remove punctuation from tokens
    tokens = [i.strip("".join(punctuations)) for i in tokens
              if i not in punctuations]

    # Remove stopwords and stem tokens
    if len(tokens) > 2:
        return [stemmer.stem(w) for w in tokens if w not in stopwords]
    return []
```


Copyrighted material

Next, we proceed with loading the emails and labels. This dataset provides each email in its own individual file (*inmail.1*, *inmail.2*, *inmail.3*, …), along with a single label file (*full/index*) in the following format:

```
spam   ../data/inmail.1
ham    ../data/inmail.2
spam   ../data/inmail.3
...
```

Each line in the label file contains the "spam" or "ham" label for each email sample in the dataset. Let's read the dataset and build a blacklist of spam words now:[10]

```
import os

DATA_DIR = 'datasets/trec07p/data/'
LABELS_FILE = 'datasets/trec07p/full/index'
TRAINING_SET_RATIO = 0.7

labels = {}
spam_words = set()
ham_words = set()

# Read the labels
with open(LABELS_FILE) as f:
    for line in f:
        line = line.strip()
        label, key = line.split()
        labels[key.split('/')[-1]] = 1 if label.lower() == 'ham' else 0

# Split corpus into training and test sets
filelist = os.listdir(DATA_DIR)
X_train = filelist[:int(len(filelist)*TRAINING_SET_RATIO)]
X_test = filelist[int(len(filelist)*TRAINING_SET_RATIO):]

for filename in X_train:
    path = os.path.join(DATA_DIR, filename)
    if filename in labels:
        label = labels[filename]
        stems = load(path)
        if not stems:
            continue
        if label == 1:
            ham_words.update(stems)
        elif label == 0:
            spam_words.update(stems)
        else:
            continue
```

10 This example can be found in the Python Jupyter notebook *chapter1/spam-fighting-blacklist.ipynb* in our code repository.

Copyrighted material

```
blacklist = spam_words - ham_words
```

Upon inspection of the tokens in `blacklist`, you might feel that many of the words are nonsensical (e.g., Unicode, URLs, filenames, symbols, foreign words). You can remedy this problem with a more thorough data-cleaning process, but these simple results should perform adequately for the purposes of this experiment:

```
greenback, gonorrhea, lecher, ...
```

Evaluating our methodology on the 22,626 emails in the testing set, we realize that this simplistic algorithm does not do as well as we had hoped. We report the results in a *confusion matrix*, a 2 × 2 matrix that gives the number of examples with given predicted and actual labels for each of the four possible pairs:

	Predicted HAM	Predicted SPAM
Actual HAM	6,772	714
Actual SPAM	5,835	7,543

True positive: predicted spam + actual ham

True negative: predicted ham + actual ham

False positive: predicted spam + actual ham

False negative: predicted ham + actual spam

Converting this to percentages, we get the following:

	Predicted HAM	Predicted SPAM
Actual HAM	32.5%	3.4%
Actual SPAM	28.0%	36.2%

Classification accuracy: 68.7%

Ignoring the fact that 5.8% of emails were not classified because of preprocessing errors, we see that the performance of this naive algorithm is actually quite fair. Our spam blacklist technique has a 68.7% classification accuracy (i.e., total proportion of correct labels). However, the blacklist doesn't include many words that spam emails use, because they are also frequently found in legitimate emails. It also seems like an impossible task to maintain a constantly updated set of words that can cleanly divide spam and ham. Maybe it's time to go back to the drawing board.

Next, you remember reading that one of the popular ways that email providers fought spam in the early days was to perform fuzzy hashing on spam messages and filter emails that produced a similar hash. This is a type of *collaborative filtering* that relies on the wisdom of other users on the platform to build up a collective intelligence that

Copyrighted material

will hopefully generalize well and identify new incoming spam. The hypothesis is that spammers use some automation in crafting spam, and hence produce spam messages that are only slight variations of one another. A fuzzy hashing algorithm, or more specifically, a *locality-sensitive hash* (LSH), can allow you to find approximate matches of emails that have been marked as spam.

Upon doing some research, you come across *datasketch*, a comprehensive Python package that has efficient implementations of the MinHash + LSH algorithm[11] to perform string matching with sublinear query costs (with respect to the cardinality of the spam set). MinHash converts string token sets to short signatures while preserving qualities of the original input that enable similarity matching. LSH can then be applied on MinHash signatures instead of raw tokens, greatly improving performance. MinHash trades the performance gains for some loss in accuracy, so there will be some false positives and false negatives in your result. However, performing naive fuzzy string matching on every email message against the full set of n spam messages in your training set incurs either $O(n)$ query complexity (if you scan your corpus each time) or $O(n)$ memory (if you build a hash table of your corpus), and you decide that you can deal with this trade-off:[12,13]

```
from datasketch import MinHash, MinHashLSH

# Extract only spam files for inserting into the LSH matcher
spam_files = [x for x in X_train if labels[x] == 0]

# Initialize MinHashLSH matcher with a Jaccard
# threshold of 0.5 and 128 MinHash permutation functions
lsh = MinHashLSH(threshold=0.5, num_perm=128)

# Populate the LSH matcher with training spam MinHashes
for idx, f in enumerate(spam_files):
    minhash = MinHash(num_perm=128)
    stems = load(os.path.join(DATA_DIR, f))
    if len(stems) &lt; 2: continue
    for s in stems:
```

11 See Chapter 3 in *Mining of Massive Datasets*, 2nd ed., by Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman (Cambridge University Press).

12 This example can be found in the Python Jupyter notebook *chapter1/spam-fighting-lsh.ipynb* in our code repository.

13 Note that we specified the `MinHashLSH` object's `threshold` parameter as 0.5. This particular LSH implementation uses Jaccard similarities between the MinHashes in your collection and the query MinHash, returning the list of objects that satisfy the threshold condition (i.e., Jaccard similarity score > 0.5). The MinHash algorithm generates short and unique signatures for a string by passing random permutations of the string through a hash function. Configuring the `num_perm` parameter to `128` means that 128 random permutations of the string were computed and passed through the hash function. In general, the more random permutations used in the algorithm, the higher the accuracy of the hash.

Copyrighted material

```
        minhash.update(s.encode('utf-8'))
    lsh.insert(f, minhash)
```

Now it's time to have the LSH matcher predict labels for the test set:

```
def lsh_predict_label(stems):
    '''
    Queries the LSH matcher and returns:
        0 if predicted spam
        1 if predicted ham
       -1 if parsing error
    '''
    minhash = MinHash(num_perm=128)
    if len(stems) < 2:
        return -1
    for s in stems:
        minhash.update(s.encode('utf-8'))
    matches = lsh.query(minhash)
    if matches:
        return 0
    else:
        return 1
```

Inspecting the results, you see the following:

	Predicted HAM	Predicted SPAM
Actual HAM	7,350	136
Actual SPAM	2,241	11,038

Converting this to percentages, you get:

	Predicted HAM	Predicted SPAM
Actual HAM	35.4%	0.7%
Actual SPAM	10.8%	53.2%

Classification accuracy: 88.6%

That's approximately 20% better than the previous naive blacklisting approach, and significantly better with respect to false positives (i.e., predicted spam + actual ham). However, these results are still not quite in the same league as modern spam filters. Digging into the data, you realize that it might not be an issue with the algorithm, but with the nature of the data you have—the spam in your dataset just doesn't seem all that repetitive. Email providers are in a much better position to make use of collaborative spam filtering because of the volume and diversity of messages that they see. Unless a spammer were to target a large number of employees in your organization, there would not be a significant amount of repetition in the spam corpus. You need to

Copyrighted material

go beyond matching stem words and computing Jaccard similarities if you want a breakthrough.

By this point, you are frustrated with experimentation and decide to do more research before proceeding. You see that many others have obtained promising results using a technique called *Naive Bayes classification*. After getting a decent understanding of how the algorithm works, you begin to create a prototype solution. Scikit-learn provides a surprisingly simple class, `sklearn.naive_bayes.MultinomialNB`, that you can use to generate quick results for this experiment. You can reuse a lot of the earlier code for parsing the email files and preprocessing the labels. However, you decide to try passing in the entire email subject and plain text body (separated by a new line) without doing any stopword removal or stemming with NLTK. You define a small function to read all the email files into this text form:[14,15]

```
def read_email_files():
    X = []
    y = []
    for i in xrange(len(labels)):
        filename = 'inmail.' + str(i+1)
        email_str = extract_email_text(os.path.join(DATA_DIR, filename))
        X.append(email_str)
        y.append(labels[filename])
    return X, y
```

Then you use the utility function `sklearn.model_selection.train_test_split()` to randomly split the dataset into training and testing subsets (the argument `random_state=123` is passed in for the sake of result reproducibility):

```
from sklearn.model_selection import train_test_split

X, y = read_email_files()

X_train, X_test, y_train, y_test, idx_train, idx_test = \
    train_test_split(X, y, range(len(y)),
    train_size=TRAINING_SET_RATIO, random_state=2)
```

Now that you have prepared the raw data, you need to do some further processing of the tokens to convert each email to a vector representation that `MultinomialNB` accepts as input.

One of the simplest ways to convert a body of text into a feature vector is to use the *bag-of-words* representation, which goes through the entire corpus of documents and generates a vocabulary of tokens used throughout the corpus. Every word in the

---

14 This example can be found in the Python Jupyter notebook *chapter1/spam-fighting-naivebayes.ipynb* in our code repository.

15 It is a loose convention in machine learning code to choose lowercase variable names for single columns of values and uppercase variable names for multiple columns of values.

Copyrighted material

vocabulary comprises a feature, and each feature value is the count of how many times the word appears in the corpus. For example, consider a hypothetical scenario in which you have only three messages in the entire corpus:

```
tokenized_messages: {
    'A': ['hello', 'mr', 'bear'],
    'B': ['hello', 'hello', 'gunter'],
    'C': ['goodbye', 'mr', 'gunter']
}

# Bag-of-words feature vector column labels:
# ['hello', 'mr', 'doggy', 'bear', 'gunter', 'goodbye']
vectorized_messages: {
    'A': [1,1,0,1,0,0],
    'B': [2,0,0,0,1,0],
    'C': [0,1,0,0,1,1]
}
```

Even though this process discards seemingly important information like the order of words, content structure, and word similarities, it is very simple to implement using the `sklearn.feature_extraction.CountVectorizer` class:

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
X_train_vector = vectorizer.fit_transform(X_train)
X_test_vector = vectorizer.transform(X_test)
```

You can also try using the term frequency/inverse document frequency (TF/IDF) vectorizer instead of raw counts. TF/IDF normalizes raw word counts and is in general a better indicator of a word's statistical importance in the text. It is provided as `sklearn.feature_extraction.text.TfidfVectorizer`.

Now you can train and test your multinomial Naive Bayes classifier:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

# Initialize the classifier and make label predictions
mnb = MultinomialNB()
mnb.fit(X_train_vector, y_train)
y_pred = mnb.predict(X_test_vector)

# Print results
print('Accuracy {:.3f}'.format(accuracy_score(y_test, y_pred)))

> Accuracy: 0.956
```


Copyrighted material

An accuracy of 95.6%—a whole 7% better than the LSH approach![16] That's not a bad result for a few lines of code, and it's in the ballpark of what modern spam filters can do. Some state-of-the-art spam filters are in fact actually driven by some variant of Naive Bayes classification. In machine learning, combining multiple independent classifiers and algorithms into an *ensemble* (also known as *stacked generalization* or *stacking*) is a common way of taking advantage of each method's strengths. So, you can imagine how a combination of word blacklists, fuzzy hash matching, and a Naive Bayes model can help to improve this result.

Alas, spam detection in the real world is not as simple as we have made it out to be in this example. There are many different types of spam, each with a different *attack vector* and method of avoiding detection. For instance, some spam messages rely heavily on tempting the reader to click links. The email's content body thus might not contain as much incriminating text as other kinds of spam. This kind of spam then might try to circumvent link-spam detection classifiers using complex methods like cloaking and redirection chains. Other kinds of spam might just rely on images and not rely on text at all.

For now, you are happy with your progress and decide to deploy this solution. As is always the case when dealing with human adversaries, the spammers will eventually realize that their emails are no longer getting through and might act to avoid detection. This response is nothing out of the ordinary for problems in security. You must constantly improve your detection algorithms and classifiers and stay one step ahead of your adversaries.

In the following chapters, we explore how machine learning methods can help you avoid having to be constantly engaged in this whack-a-mole game with attackers, and how you can create a more adaptive solution to minimize constant manual tweaking.

Limitations of Machine Learning in Security

The notion that machine learning methods will always give good results across different use cases is categorically false. In real-world scenarios there are usually factors to optimize for other than precision, recall, or accuracy.

As an example, explainability of classification results can be more important in some applications than others. It can be considerably more difficult to extract the reasons for a decision made by a machine learning system compared to a simple rule. Some

16 In general, using only accuracy to measure model prediction performance is crude and incomprehensive. Model evaluation is an important topic that we discuss further in Chapter 2. Here we opt for simplicity and use accuracy as an approximate measure of performance. The `sklearn.metrics.classification_report()` method provides the *precision*, *recall*, f_1*-score*, and *support* for each class, which can be used in combination to get a more accurate picture of how the model performs.

Copyrighted material

machine learning systems might also be significantly more resource intensive than other alternatives, which can be a dealbreaker for execution in constrained environments such as embedded systems.

There is no silver bullet machine learning algorithm that works well across all problem spaces. Different algorithms vary vastly in their suitability for different applications and different datasets. Although machine learning methods contribute to the notion of artificial intelligence, their capabilities can still only be compared to human intelligence along certain dimensions.

The human decision-making process is informed by a vast body of context drawn from cultural and experiential knowledge. This process is very difficult for machine learning systems to emulate. Take the initial blacklisted-words approach that we used for spam filtering as an example. When a person evaluates the content of an email to determine if it's ham or spam, the decision-making process is never as simple as looking for the existence of certain words. The context in which a blacklisted word is being used can result in it being a reasonable inclusion in non-spam email. Also, spammers might use synonyms of blacklisted words in future emails to convey the same meaning, but a simplistic blacklist would not adapt appropriately. The system simply doesn't have the context that a human has—it does not know what relevance a particular word bears to the reader. Continually updating the blacklist with new suspicious words is a laborious process, and in no way guarantees perfect coverage.

Even though your machine-learned model may work perfectly on a training set, you might find that it performs badly on a testing set. A common reason for this problem is that the model has *overfit* its classification boundaries to the training data, learning characteristics of the dataset that do not generalize well across other unseen datasets. For instance, your spam filter might learn from a training set that all emails containing the words "inheritance" and "Nigeria" can immediately be given a high suspicion score, but it does not know about the legitimate email chain discussion between employees about estate inheritances in Nigerian agricultural insurance schemes.

With all these limitations in mind, we should approach machine learning with equal parts of enthusiasm and caution, remembering that not everything can instantly be made better with AI.

Copyrighted material

CHAPTER 2
Classifying and Clustering

In this chapter, we discuss the most useful machine learning techniques for security applications. After covering some of the basic principles of machine learning, we offer up a toolbox of machine learning algorithms that you can choose from when approaching any given security problem. We have tried to include enough detail about each technique so that you can know when and how to use it, but we do not attempt to cover all the nuances and complexities of the algorithms.

This chapter has more mathematical detail than the rest of the book; if you want to skip the details and begin trying out the techniques, we recommend you read the sections "Machine Learning in Practice: A Worked Example" on page 27 and "Practical Considerations in Classification" on page 55 and then look at a few of the most popular supervised and unsupervised algorithms: logistic regression, decision trees and forests, and *k*-means clustering.

Machine Learning: Problems and Approaches

Suppose that you are in charge of computer security for your company. You install firewalls, hold phishing training, ensure secure coding practices, and much more. But at the end of the day, all your CEO cares about is that you don't have a breach. So, you take it upon yourself to build systems that can detect and block malicious traffic to any attack surface. Ultimately, these systems must decide the following:

- For every file sent through the network, does it contain malware?
- For every login attempt, has someone's password been compromised?
- For every email received, is it a phishing attempt?
- For every request to your servers, is it a denial-of-service (DoS) attack?

Copyrighted material

- For every outbound request from your network, is it a bot calling its command-and-control server?

These tasks are all *classification* tasks—binary decisions about the nature of the observed event.

Your job can thus be rephrased as follows:

> Classify all events in your network as malicious or legitimate.

When phrased in this manner, the task seems almost hopeless; how are you supposed to classify *all* traffic? But not to fear! You have a secret weapon: *data*.

Specifically, you have historical logs of binary files, login attempts, emails received, and inbound and outbound requests. In some cases, you might even know of attacks in the past and be able to associate these attacks with the corresponding events in your logs. Now, to begin solving your problem, you look for patterns in the past data that seem to indicate malicious attacks. For example, you observe that when a single IP address is making more than 20 requests per second to your servers over a period of 5 minutes, it's probably a DoS attack. (Maybe your servers went down under such a load in the past.)

After you have found patterns in the data, the next step is to encode these patterns as an *algorithm*—that is, a function that takes as input data about whatever you're trying to classify and outputs a binary response: "malicious" or "legitimate." In our example, this algorithm would be very simple:[1] it takes as input the number of requests from an IP address over the 5 minutes prior to the request, and outputs "legitimate" if the number is less than 6,000 and "malicious" if it is greater than 6,000.

At this point, you have learned from the data and created an algorithm to block bad traffic. Congratulations! But there should be something nagging at you: what's special about the number 20? Why isn't the limit 19 or 21? Or 19.77? Ideally you should have some principled way of determining which one of these options, or in fact which real number, is best. And if you use an algorithm to scan historical data and find the best classification rule according to some mathematical definition of "best," this process is called *machine learning*.

More generally, machine learning is the process of using historical data to create a prediction algorithm for future data. The task we just considered was one of *classification*: determine which class a new data point (the request) falls into. Classification can be *binary*, as we just saw, in which there are only two classes, or *multiclass*; for example, if you want to determine whether a piece of malware is ransomware, a keylogger, or a remote access trojan.

1 Simple algorithms like this one are usually called "rules."

Copyrighted material

Machine learning can also be used to solve *regression* problems, in which we try to predict the value of a real-number variable. For example, you might want to predict the number of phishing emails an employee receives in a given month, given data about their position, access privileges, tenure in the company, security hygiene score, and so on. Regression problems for which the inputs have a time dimension are sometimes called *time series analysis*; for example, predicting the value of a stock tomorrow given its past performance, or the number of account sign-ins from the Seattle office given a known history. *Anomaly detection* is a layer on top of regression: it refers to the problem of determining when an observed value is sufficiently different from a predicted value to indicate that something unusual is going on.

Machine learning is also used to solve *clustering* problems: given a bunch of data points, which ones are similar to one another? For example, if you are trying to analyze a large dataset of internet traffic to your site, you might want to know which requests group together. Some clusters might be botnets, some might be mobile providers, and some might be legitimate users.

Machine learning can be *supervised*, in which case you have labels on historical data and you are trying to predict labels on future data. For example, given a large corpus of emails labeled as spam or ham, you can train a spam classifier that tries to predict whether a new incoming message is spam. Alternatively, machine learning can be *unsupervised*, in which case you have no labels on the historical data; you might not even know what the labels are that you're trying to predict, for example if you have an unknown number of botnets attacking your network that you want to disambiguate from one another. Classification and regression tasks are examples of supervised learning, and clustering is a typical form of unsupervised learning.

Machine Learning in Practice: A Worked Example

As we said earlier, machine learning is the process of using historical data to come up with a prediction algorithm for previously unseen data. Let's examine how this process works, using a simple dataset as an example. The dataset that we are using is transaction data for online purchases collected from an ecommerce retailer.[2] The dataset contains 39,221 transactions, each comprising 5 properties that can be used to describe the transaction, as well as a binary "label" indicating whether this transaction is an instance of fraud—"1" if fraudulent, and "0" if not. The comma-separated values (CSV) format that this data is in is a standard way of representing data for analytics. Observing that the first row in the file indicates the names for each positional value in each subsequent line, let's consider what each value means by examining a randomly selected row of data:

2 You can find the dataset in *chapter2/datasets/payment_fraud.csv* in our code repository.

Copyrighted material

```
accountAgeDays,numItems,localTime,paymentMethod,paymentMethodAgeDays,label
...
196, 1, 4.962055, creditcard, 5.10625, 0
```

Putting this in a more human-readable form:

```
accountAgeDays:       196
numItems:             1
localTime:            4.962055
paymentMethod:        creditcard
paymentMethodAgeDays: 5.10625
label:                0
```

We see that this transaction was made through a user account that was created 196 days ago (`accountAgeDays`), and that the user purchased 1 item (`numItems`) at around 4:58 AM in the consumer's local time (`localTime`). Payment was made through credit card (`paymentMethod`), and this method of payment was added about 5 days before the transaction (`paymentMethodAgeDays`). The `label` is 0, which indicates that this transaction is not fraudulent.

Now, you might ask how we came to learn that a certain transaction was fraudulent. If someone made an unauthorized transaction using your credit card, assuming that you were vigilant, you would file a *chargeback* for this transaction, indicating that the transaction was not made by you and you want to get your money back. Similar processes exist for payments made through other payment methods, such as PayPal or store credit. The chargeback is a strong and clear indication that the transaction is fraudulent, allowing us to collect data about fraudulent transactions.

However, the reason we can't use chargeback data in real time is that merchants receive chargeback details many months after the transaction has gone through—after they have shipped the items out to the attackers, never to be seen again. Typically, the retailer absorbs all losses in situations like this, which could translate to potentially enormous losses in revenue. This financial loss could be mitigated if we had a way to predict how likely a transaction is to be fraudulent *before* we ship the items out. Now, we could examine the data and come up with some rules, such as "If the payment method was added in the last day and the number of items is at least 10, the transaction is fraudulent." But such a rule might have too many false positives. How can we use data to find the *best* prediction algorithm? This is what machine learning does.

Each property of a transaction is called a *feature* in machine learning parlance. What we want to achieve is to have a machine learning algorithm learn how to identify a fraudulent transaction from the five features in our dataset. Because the dataset contains a label for what we are aiming to predict, we call this a "labeled dataset" and can perform supervised learning on it. (If there had been no label, we could only have performed semi-supervised learning or unsupervised learning.) The ideal fraud detection system will take in features of a transaction and return a *probability score*

Copyrighted material

for how likely this transaction is to be fraudulent. Let's see how we can create a prototype system by using machine learning.

Similar to how we approached the spam classification problem in Chapter 1, we'll take advantage of the functionality in the Python machine learning library scikit-learn. In addition, we'll use Pandas, a popular data analysis library for Python, to perform some lightweight data wrangling. First, we'll use the `pandas.read_csv()` utility to read the dataset in the CSV file:

```
import pandas as pd

df = pd.read_csv('ch1/payment_fraud.csv')
```

Notice that the result of `read_csv()` is stored into the variable `df`, short for *DataFrame*. A DataFrame is a Pandas data structure that represents datasets in a two-dimensional table-like form, allowing for operations to be applied on rows or columns. DataFrame objects allow you to perform a plethora of manipulations on the data, but we will not dive into the specifics here.[3] Let's use the `DataFrame.sample()` function to retrieve a snippet of three rows from `df`:

```
df.sample(3)
```

	accountAgeDays	numItems	localTime	paymentMethod	paymentMethodAgeDays	label
31442	2000	1	4.748314	storecredit	0.000000	0
27232	1	1	4.886641	storecredit	0.000000	1
8687	878	1	4.921349	paypal	0.000000	0

This command returns a tabular view of three random rows. The left column indicates the numerical index of each selected row, and the top row indicates the name of each column. Note that one column stands out because it is of non-numerical type: `paymentMethod`. There are three possible values that this feature takes on in our dataset: `creditcard`, `paypal`, and `storecredit`. This feature is called a *categorical variable* because it takes on a value indicating the category it belongs to. Many machine learning algorithms require all features to be numeric.[4] We can use `pandas.get_dummies()` to convert variables from categorical to numeric:[5]

3 For more information on Pandas DataFrames, see the documentation.

4 This is not true for *all* machine learning algorithms. For example, decision trees do not require any or all features to be numeric. The advantage of expressing features in numeric terms is that each data point can be expressed as a vector in a real vector space, and we can apply all the techniques of linear algebra and multivariable calculus to the problem.

5 Typically, we'll set the `pd.get_dummies()` argument `drop_first` to `True` to avoid the so-called "dummy variable trap," in which independent variables being closely correlated violates assumptions of independence in regression. We chose to keep things simple to avoid confusion, but elaborate on this problem in Chapter 5.

Copyrighted material

```
df = pd.get_dummies(df, columns=['paymentMethod'])
```

Upon inspection of the new DataFrame object, we notice that three new columns have been added to the table—`paymentMethod_creditcard`, `paymentMethod_paypal`, and `paymentMethod_storecredit`:

```
df.sample(3)
```

	accountAgeDays	...	paymentMethod_creditcard	paymentMethod_paypal	paymentMethod_storecredit
23393	57	...	1	0	0
3355	1,366	...	0	1	0
34248	19	...	1	0	0

Each of these features is a binary feature (i.e., they take on a value of either 0 or 1), and each row has exactly one of these features set to 1, hence the name of this method of categorical variable encoding: *one-hot encoding*. These variables are called *dummy variables* in statistics terminology.

Now, we can divide the dataset into training and test sets (as we did in Chapter 1):

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    df.drop('label', axis=1), df['label'],
    test_size=0.33, random_state=17)
```

The `sklearn.model_selection.train_test_split()` function helps us split our dataset into training and test sets. Notice that in the first argument to the function, we passed in `df.drop('label', axis=1)`. This will be split into `X_train` and `X_test` to the ratio of 0.67:0.33 because we passed in `test_size=0.33`, which means that we want two-thirds of the dataset to be used for training the machine learning algorithm, and the remaining third, the test set, to be used to see how well the algorithm performs. We are dropping the `label` column from `X` before splitting it into `X_train` and `X_test`, and passing in the `label` column as `y`—`df['label']`. The labels will then be split in the same ratio into `y_train` and `y_test`.

Now let's apply a standard supervised learning algorithm, *logistic regression*, to this data:

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
clf.fit(X_train, y_train)
```

In the first line, we import the `sklearn.linear_model.LogisticRegression` class. Then, in the second line, we initialize the `LogisticRegression` object by invoking the constructor. In the third line, we feed `X_train` and `y_train` (i.e., the training set) into the `fit()` function, resulting in a trained classifier model, which is stored in the `clf`

Copyrighted material

object. This classifier has taken the training data and used logistic regression (which we elaborate on further in the next section) to distill some generalizations about fraudulent and nonfraudulent transactions into a model.

To make predictions using this model, all we need to do now is to pass some unlabeled features into this classifier object's `predict()` function:

```
y_pred = clf.predict(X_test)
```

Inspecting `y_pred`, we can see the label predictions made for each row in `X_test`. Note that at training time the classifier did not have any access to `y_test` at all; the predictions made, contained in `y_pred`, are thus purely a result of the generalizations learned from the training set. We use the `sklearn.metrics.accuracy_score()` function (that we also used in Chapter 1) to get a feel of how good these predictions are:

```
from sklearn.metrics import accuracy_score

print(accuracy_score(y_pred, y_test))

> 0.99992273816
```

A 99.992% accuracy is pretty good! However, we discussed in Chapter 1 that the accuracy score can often be a misleading oversimplification and is quite a bad metric for evaluating results like this. Let's generate a confusion matrix, instead:

```
from sklearn.metrics import confusion_matrix

print(confusion_matrix(y_test, y_pred))
```

0	Predicted NOT FRAUD	Predicted FRAUD
Actual NOT FRAUD	12,753	0
Actual FRAUD	1	189

There appears to only be a single misclassification in the entire test set. 189 transactions are correctly flagged as fraud, and there is 1 false negative in which the fraudulent transaction was not detected. There are zero false positives.

As a recap, here is the entire piece of code that we used to train and test our logistic regression payment fraud detection model:[6]

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix
```

6 You can find this example as a Python Jupyter notebook in our repo at *chapter2/logistic-regression-fraud-detection.ipynb*.

Copyrighted material

```
# Read in the data from the CSV file
df = pd.read_csv('ch1/payment_fraud.csv')

# Convert categorical feature into dummy variables with one-hot encoding
df = pd.get_dummies(df, columns=['paymentMethod'])

# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    df.drop('label', axis=1), df['label'],
    test_size=0.33, random_state=17)

# Initialize and train classifier model
clf = LogisticRegression().fit(X_train, y_train)

# Make predictions on test set
y_pred = clf.predict(X_test)

# Compare test set predictions with ground truth labels
print(accuracy_score(y_pred, y_test))
print(confusion_matrix(y_test, y_pred))
```

We can apply this model to any given incoming transaction and get a probability score for how likely this transaction is to be fraudulent:

```
clf.predict_proba(df_real)

  # Array that represents the probability of the transaction
  # having a label of 0 (in position 0) or 1 (in position 1)
> [[  9.99999994e-01   5.87025707e-09]]
```

Taking `df_real` to be a DataFrame that contains a single row representing an incoming transaction received by the online retailer, the classifier predicts that this transaction is 99.9999994% likely to *not* be fraudulent (remember that `y = 0` means not fraudulent).

You might have noticed that all the work of machine learning—i.e., the part where we learn the prediction algorithm—has been abstracted out into the single scikit-learn API call, `LogisticRegression.fit()`. So, what *actually* goes on in this black box that allows this model to learn how to predict fraudulent transactions? We will now open up the box and find out.

Training Algorithms to Learn

At its core, a machine learning algorithm takes in a *training dataset* and outputs a *model*. The model is an algorithm that takes in new data points in the same form as the training data and outputs a prediction. All machine learning algorithms are defined by three interdependent components:

Copyrighted material

- A *model family*, which describes the universe of models from which we can choose
- A *loss function*, which allows us to quantitatively compare different models
- An *optimization procedure*, which allows us to choose the best model in the family

Let's now consider each of these components.

Model Families

Recall that we expressed our fraud dataset in terms of seven numerical features: four features from the raw data and three from the one-hot encoding of the payment method. We can thus think of each transaction as a point in a seven-dimensional real vector space, and our goal is to divide up the space into areas of fraud and nonfraud transactions. The "model" output by our machine learning algorithm is a description of this division of the vector space.

In theory, the division of our vector space into fraud and nonfraud areas can be infinitely complex; in practice, most algorithms produce a *decision boundary*, which is a surface in the vector space.[7] One side of the decision boundary consists of the points labeled as fraud, and the other side consists of the points labeled as nonfraud. The boundary can be as simple as a line (or hyperplane in higher dimensions) or as complex as a union of nonlinear disconnected regions. Figure 2-1 presents some examples.

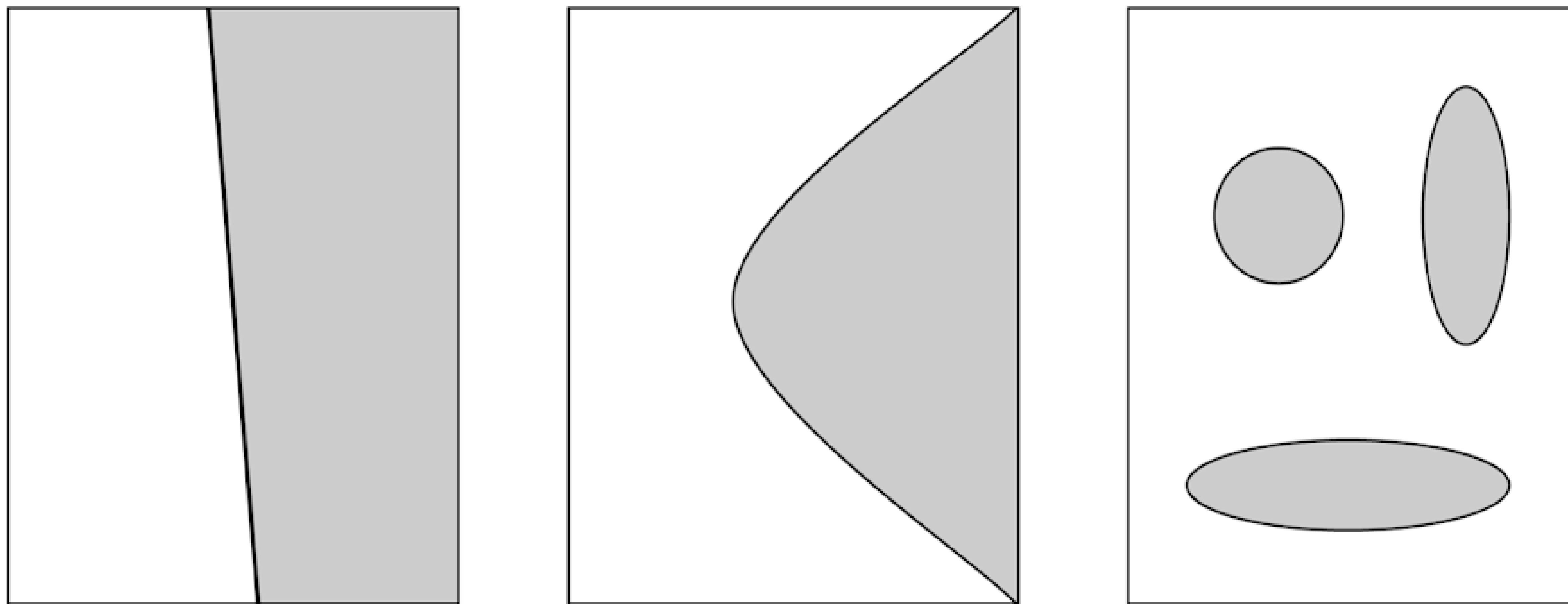


Figure 2-1. Examples of two-dimensional spaces divided by a decision boundary

7 Technically, a differentiable, oriented surface.

Copyrighted material

If we want to be more granular, instead of mapping each point in the vector space to either "fraud" or "nonfraud," we can map each point to a *probability* of fraud. In this case our machine learning algorithm outputs a function that assigns each point in the vector space a value between 0 and 1, to be interpreted in our example as the probability of fraud.

Any given machine learning algorithm restricts itself to finding a certain type of decision boundary or probability function that can be described by a finite number of *model parameters*. The simplest decision boundary is a *linear* decision boundary—that is, a hyperplane in the vector space. An oriented hyperplane H in an n-dimensional vector space can be described by an n-dimensional vector $\boldsymbol{\theta}$ orthogonal to the hyperplane, plus another vector $\boldsymbol{\beta}$ indicating how far the hyperplane is from the origin:

$$H:\ \boldsymbol{\theta} \cdot (\boldsymbol{x} - \boldsymbol{\beta}) = 0$$

This description allows us to divide the vector space in two; to assign probabilities we want to look at the distance of the point $\boldsymbol{x}$ from the hyperplane H. We can thus compute a real-valued "score":

$$s(\boldsymbol{x}) = \boldsymbol{\theta} \cdot (\boldsymbol{x} - \boldsymbol{\beta}) = \boldsymbol{\theta} \cdot \boldsymbol{x} + b$$

where we have let $b = -\boldsymbol{\theta} \cdot \boldsymbol{\beta}$. Our model to compute the score can thus be described by $n + 1$ model parameters: n parameters to describe the vector $\boldsymbol{\theta}$, and one "offset" parameter b. To turn the score into a classification, we simply choose a threshold t above which all scores indicate fraud, and below which all scores indicate nonfraud.

If we want to map the real-valued score $s(\boldsymbol{x})$ to a probability, we must apply a function that maps the real numbers to the interval [0,1]. The standard function to apply is known as the *logistic function* or *sigmoid function*,[8] as illustrated in Figure 2-2. It is formulated as:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

8 The full background for why this particular function is selected as the hypothesis function for binary logistic regression is slightly more involved, and we will not go into much more detail here. For further details, see section 4.4 of *The Elements of Statistical Learning*, 2nd ed., by Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman (Springer).

Copyrighted material

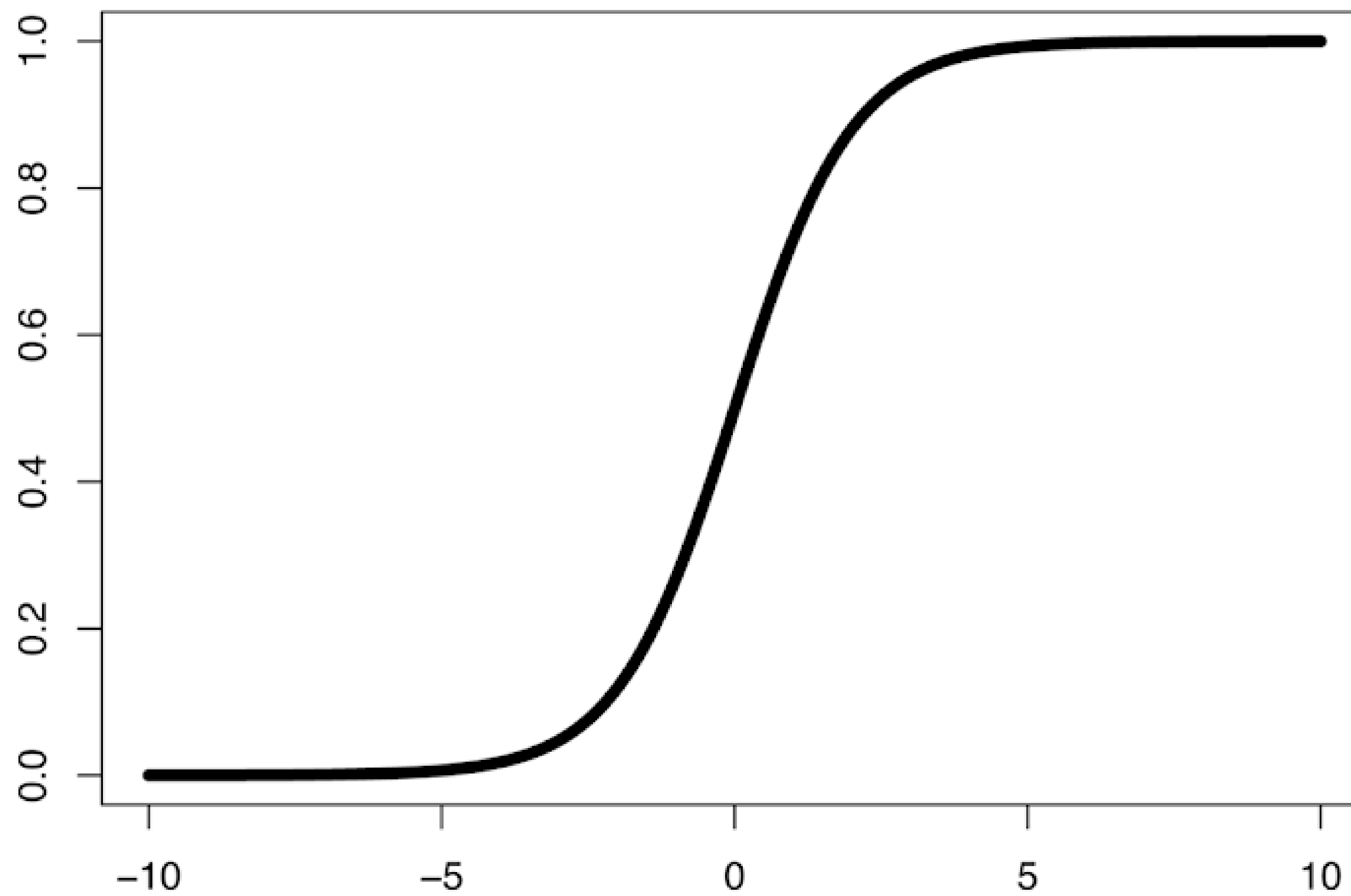


Figure 2-2. The sigmoid function

The output of the logistic function can be interpreted as a probability, allowing us to define the likelihood of the dependent variable taking on a particular value given some input feature vector $\boldsymbol{x}$.

Loss Functions

Now that we have restricted our choice of prediction algorithms to a certain parametrized family, we must choose the best one for the given training data. How do we know when we have found the best algorithm? We define the best algorithm to be one that optimizes some quantity computed from the data. This quantity is called an *objective function*. In the case of machine learning, the objective function is also known as a *cost function* or *loss function*, because it measures the "cost" of wrong predictions or the "loss" associated with them.

Mathematically, a loss function is a function that maps a set of pairs of (*predicted label*, *truth label*) to a real number. The goal of a machine learning algorithm is to find the model parameters that produce predicted labels for the training set that minimize the loss function.

In regression problems, for which the prediction algorithm outputs a real number instead of a label, the standard loss function is the *sum of squared errors*. If y_i is the true value and $\hat{y}_i$ is the predicted value, the loss function is as follows:

$$C(Y) = \sum_i \left(\hat{y}_i - y_i\right)^2$$

Copyrighted material

We can use this loss function for classification problems as well, where y_i is either 0 or 1, and $\hat{y}_i$ is the probability estimate output by the algorithm.

For logistic regression, we use *negative log likelihood* as the loss function. The *likelihood* of a set of probability predictions $\{p_i\}$ for a given set of ground truth labels $\{y_i\}$ is defined to be the probability that these truth labels would have arisen if sampled from a set of binomial distributions according to the probabilities $\{p_i\}$. Concretely, if the truth label y_i is 0, the likelihood of 0 is $1 - p_i$—the probability that 0 would have been sampled from a binomial distribution with mean p_i. If the truth label y_i is 1, the likelihood is p_i.

The likelihood of the entire set of predictions is the product of the individual likelihoods:

$$\mathscr{L}(\{p_i\}, \{y_i\}) = \prod_{y_i = 0} (1 - p_i) \cdot \prod_{y_i = 1} p_i$$

The goal of logistic regression is to find parameters that produce probabilities $\{p_i\}$ that maximize the likelihood.

To make computations easier, and, in particular, because most optimization methods require computing derivatives of the loss function, we take the negative log of the likelihood. As maximizing the likelihood is equivalent to minimizing the negative log likelihood, we call negative log likelihood the loss function:

$$\ell(\{p_i\}, \{y_i\}) = -\sum_i \left((1 - y_i) \log (1 - p_i) + y_i \log p_i\right)$$

Here we have used the fact that y_i is always 0 or 1 to combine the two products into a single term.

Optimization

The last step in the machine learning procedure is to search for the optimal set of parameters that minimizes the loss function. To carry out this search we use an *optimization algorithm*. There may be many different optimization algorithms available to you when fitting your machine learning model.[9] Most scikit-learn estimators (e.g.,

9 A seminal book that defined the field of convex optimization (note that not all optimization problems we speak of may be *convex* in nature) is *Convex Optimization* by Stephen P. Boyd and Lieven Vandenberghe (Cambridge University Press).

Copyrighted material

`LogisticRegression`) allow you to specify the *numerical solver* to use, but what are the differences between the different options, and how do you go about selecting one?

The job of an optimization algorithm is to minimize (or maximize) an objective function. In the case of machine learning, the objective function is expressed in terms of the model's learnable parameters ($\boldsymbol{\theta}$ and b in the previous example), and the goal is to find the values of $\boldsymbol{\theta}$ and b that optimize the objective function.

Optimization algorithms mainly come in two different flavors:

First-order algorithms
: These algorithms optimize the objective function using the first derivatives of the function with respect to the learnable parameters. *Gradient descent methods* are the most popular types of first-order optimization algorithms; we can use them to find the inputs to a function that give the minimum (or maximum) value. Computing the gradient of a function (i.e., the partial derivatives with respect to each variable) allows us to determine the instantaneous direction that the parameters need to move in order to achieve a more optimal outcome.

Second-order algorithms
: As the name suggests, these algorithms use the second derivatives to optimize the objective function. Second-order algorithms will not fall victim to paths of slow convergence. For example, second-order algorithms are good at detecting saddle points, whereas first-order algorithms are likely to become stuck at these points. However, second-order methods are often slower and more expensive to compute.

First-order methods tend to be much more frequently used because of their relative efficiency. Picking a suitable optimization algorithm depends on the size of the dataset, the nature of the cost function, the type of learning problem, and speed/resource requirements for the operation. In addition, some regularization techniques can also have compatibility issues with certain types of optimizers. First-order algorithms include the following:

- *LIBLINEAR*[10] is the default solver for the linear estimators in scikit-learn. This algorithm tends to not do well on larger datasets; as suggested by the scikit-learn documentation, the Stochastic Average Gradient (SAG) or SAGA (improvement to SAG) methods work better for large datasets.[11]

10 Rong-En Fan et al., "LIBLINEAR: A Library for Large Linear Classification," *Journal of Machine Learning Research* 9 (2008): 1871–1874.

11 Francis Bach, "Stochastic Optimization: Beyond Stochastic Gradients and Convexity." INRIA - *Ecole Normale Supérieure*, Paris, France. Joint tutorial with Suvrit Sra, MIT - NIPS - 2016.

Copyrighted material

Different optimization algorithms deal with multiclass classification differently. LIBLINEAR works only on binary classification. For it to work in a multiclass scenario, it has to use the one-versus-rest scheme; we discuss this scheme more fully in Chapter 5.

- Stochastic Gradient Descent (SGD) is a very simple and efficient algorithm for optimization that performs a parameter update for each separate training example. The stochastic nature of the gradient descent means that the algorithm is more likely to discover new and possibly better local minima as compared to standard gradient descent. However, it typically results in high-variance oscillations, which can result in a delay in convergence. This can be solved with a decreasing learning rate (i.e., exponentially decrease the learning rate) that results in smaller fluctuations as the algorithm approaches convergence.

  The technique called *momentum* also helps accelerate SGD convergence by navigating the optimization movement only in the relevant directions and softening any movement in irrelevant directions, which stabilizes SGD.

- Optimization algorithms such as AdaGrad, AdaDelta, and Adam (Adaptive Moment Estimation) allow for separate and adaptive learning rates for each parameter that solve some problems in the other simpler gradient descent algorithms.

- When your training dataset is large, you will need to use a distributed optimization algorithm. One popular algorithm is the Alternating Direction Method of Multipliers (ADMM).[12]

Example: Gradient descent

To conclude this section, we go briefly into the details of gradient descent, a powerful optimization algorithm that has been applied to many different machine learning problems.

The standard algorithm for gradient descent is as follows:

1. Select random starting parameters for the machine learning model. In the case of a linear model, this means selecting a random normal vector $\boldsymbol{\theta}$ and offset $\boldsymbol{\beta}$, which results in a random hyperplane in n-dimensional space.
2. Compute the value of the gradient of the loss function for this model at the point described by these parameters.

12 Stephen Boyd et al., “Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers,” Foundations and Trends in Machine Learning 3 (2011): 1–122.

Copyrighted material

3. Change the model parameters in the direction of greatest gradient decrease by a certain small magnitude, typically referred to as the α or learning rate.
4. Iterate: repeat steps 2 and 3 until *convergence* or a satisfactory optimization result is attained.

Figure 2-3 illustrates the intermediate results of a gradient descent optimization process of a linear regression. At zero iterations, observe that the regression line, formed with the randomly chosen parameters, does not fit the dataset at all. As you can imagine, the value of the *sum-of-squares* cost function is quite large at this point. At three iterations, notice that the regression line has very quickly moved to a more sensible position. Between 5 and 20 iterations the regression line slowly adjusts itself to more optimal positions where the cost function is minimized. If performing any more iterations doesn't decrease the cost function significantly, we can say that the optimization has *converged*, and we have the final learned parameters for the trained model.

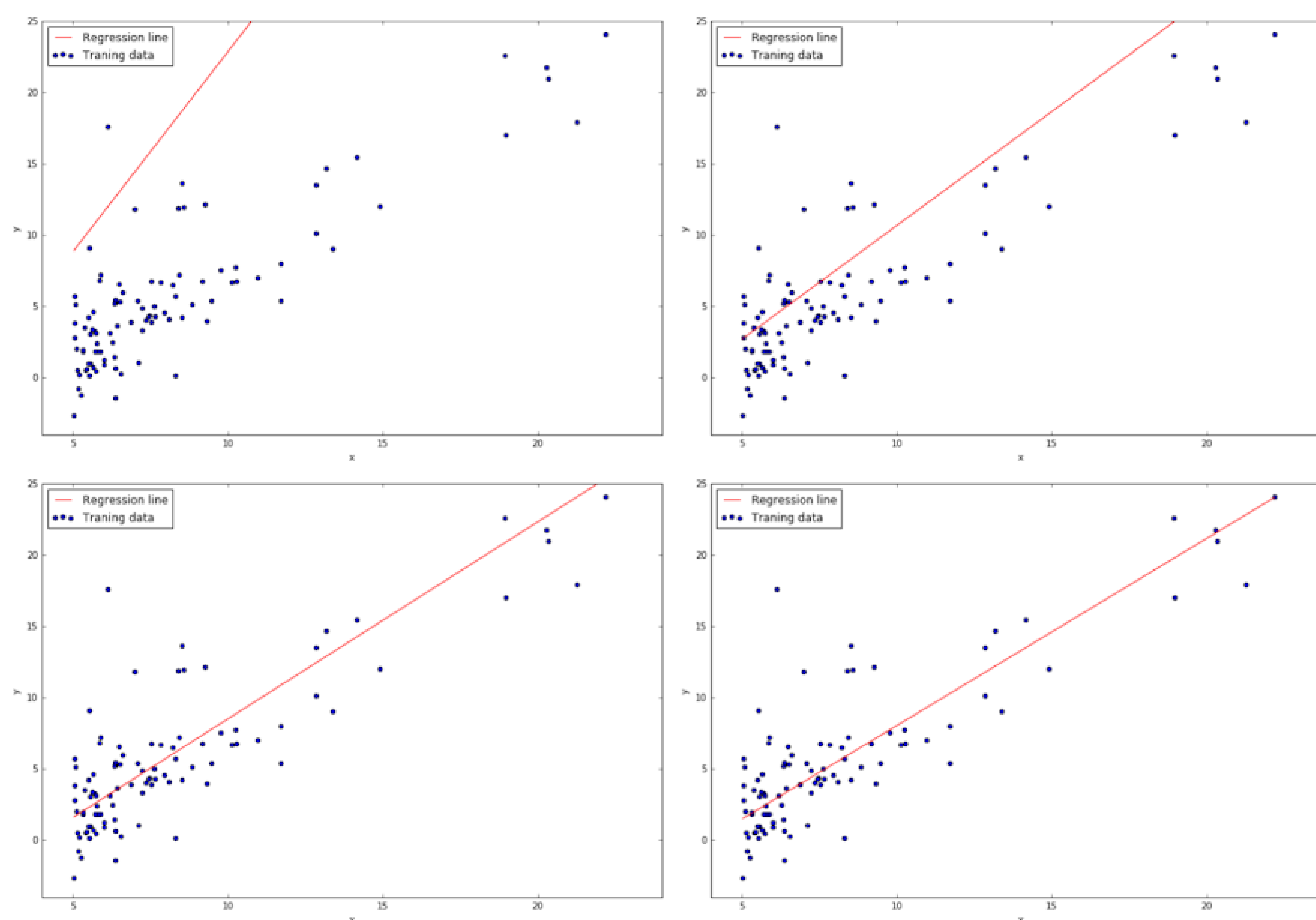


Figure 2-3. Regression line after a progressive number of iterations of gradient descent optimization (0, 3, 5, and 20 iterations of gradient descent, shown at the upper left, upper right, lower left, and lower right, respectively)

Which optimization algorithm?

As with many things in data science, no optimization algorithm is one-size-fits-all, and there are no clear rules for which algorithm definitely performs better for certain

Copyrighted material

types of problems. A certain amount of trial-and-error experimentation is often needed to find an algorithm that suits your requirements and meets your needs. There are many considerations other than convergence or speed that you should take into account when selecting an optimizer. Starting with the default or the most sensible option and iterating when you see clues for improvements is generally a good strategy.

Supervised Classification Algorithms

Now that we know how machine learning algorithms work in principle, we will briefly describe some of the most popular supervised learning algorithms for classification.

Logistic Regression

Although we discussed logistic regression in some detail earlier, we go over its key properties here. Logistic regression takes as input numerical feature vectors and attempts to predict the *log odds*[13] of each data point occurring; we can convert the log odds to probabilities by using the sigmoid function discussed earlier. In the log odds space, the decision boundary is linear, so increasing the value of a feature monotonically increases or decreases (depending on the sign of the coefficient) the score output by the model.

Why Not Linear Regression?

Linear regression, taught in every introductory statistics course, is a powerful tool for predicting future outcomes based on past data. The algorithm takes data consisting of input variables (expressed as vectors in a vector space) and a response variable (a real number) and produces a "best fit" linear model that maps each point in the vector space to its predicted response. Why can't we use it to solve classification problems? The issue is that linear regression predicts a *real-valued* variable, and in classification we want to predict a *categorical* variable. If we try to map the two categories to 0 and 1 and perform linear regression, we end up with a line that maps input variables to some output, as demonstrated in Figure 2-4. But what does this output mean? We can't interpret it as a probability, because it can take values below 0 or above 1, as in the figure. We could interpret it as a score and choose a threshold for the class boundary, but while this approach works technically it does not produce a good classifier. The reason is that the squared-error loss function used in linear regression does not accurately reflect how far points are from the classification boundary: in the example

13 For an event X that occurs with probability p, the *odds* of X are $p/(1-p)$, and the *log odds* are $\log(p/(1-p))$.

Copyrighted material

of Figure 2-4, the point at $X = 1$ has larger error than the points around $X = 10$, even though this point will be farther from the classification boundary (e.g., $X = 50$).

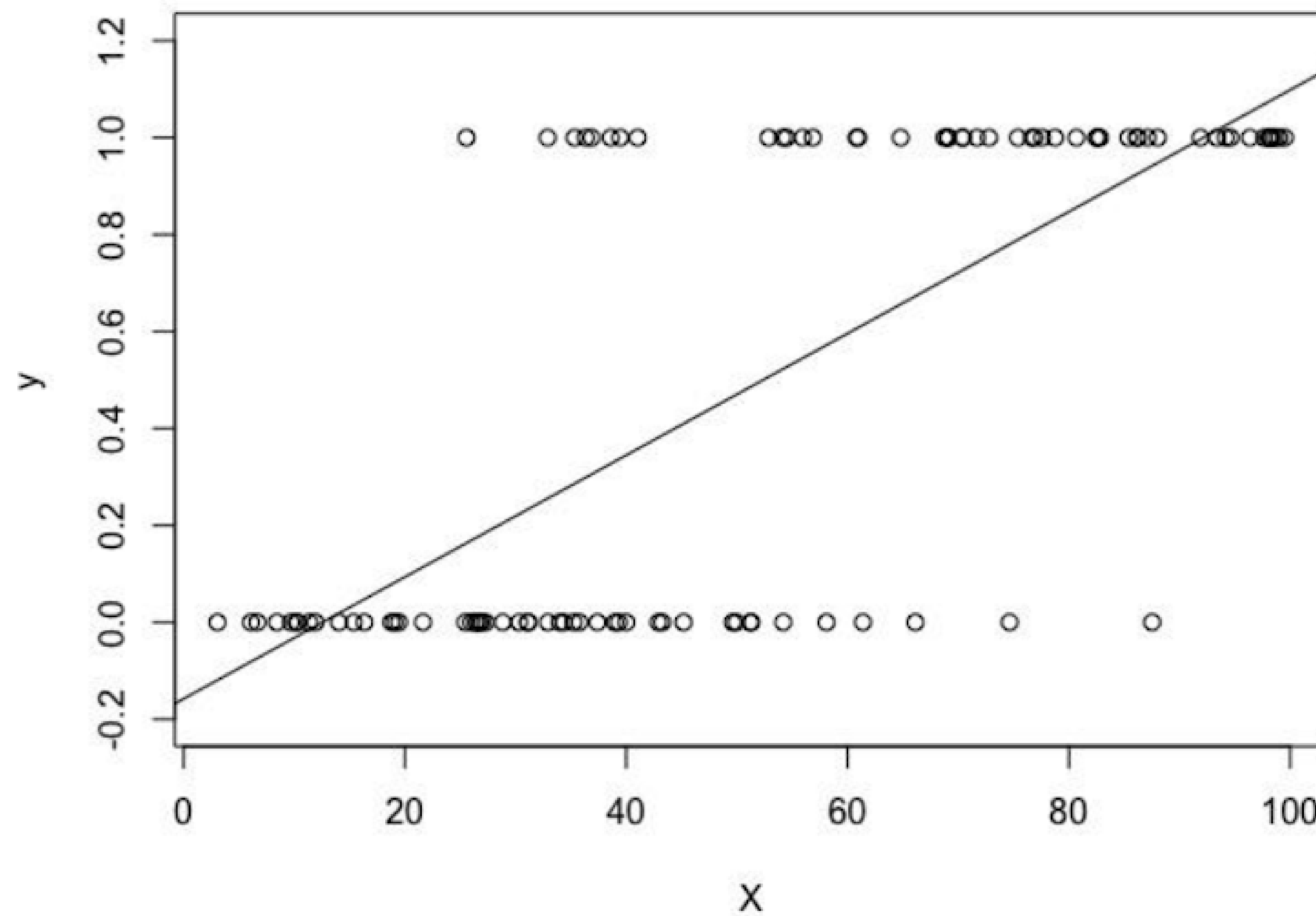


Figure 2-4. Linear regression for classification

Logistic regression is one of the most popular algorithms in practice due to a number of properties: it can be trained very efficiently and in a distributed manner, it scales well to millions of features, it admits a concise description and fast scoring algorithm (a simple dot product), and it is explainable—each feature's contribution to the final score can be computed.

However, there are a few important things to be aware of when considering logistic regression as a supervised learning technique:

- Logistic regression assumes linearity of features (independent variables) and log odds, requiring that *features are linearly related to the log odds*. If this assumption is broken the model will perform poorly.
- Features should have *little to no multicollinearity*;[14] that is, independent variables should be truly independent from one another.
- Logistic regression typically *requires a larger sample size* compared to other machine learning algorithms like linear regression. Maximum likelihood esti-

14 This assumption is not exclusive to logistic regression. Most other machine learning algorithms also require that features be uncorrelated.

Copyrighted material

mates (used in logistic regression) are less powerful than ordinary least squares (used in linear regression), which results in requiring more training samples to achieve the same statistical learning power.[15]

Decision Trees

Decision trees are very versatile supervised learning models that have the important property of being easy to interpret. A decision tree is, as its name suggests, a binary tree data structure that is used to make a decision. Trees are a very intuitive way of displaying and analyzing data and are popularly used even outside of the machine learning field. With the ability to predict both categorical values (classification trees) and real values (regression trees) as well as being able to take in numerical and categorical data without any normalization or dummy variable creation,[16] it's not difficult to see why they are a popular choice for machine learning.

Let's see how a typical (top-down) learning decision tree is constructed:

1. Starting at the root of the tree, the full dataset is split based on a binary condition into two child subsets. For example, if the condition is "age ≥ 18," all data points for which this condition is true go to the left child and all data points for which this condition is false go to the right child.
2. The child subsets are further recursively partitioned into smaller subsets based on other conditions. Splitting conditions are automatically selected at each step based on what condition best splits the set of items. There are a few common metrics by which the quality of a split is measured:

   Gini impurity
      If samples in a subset were randomly labeled according to the distribution of labels in the set, the proportion of samples incorrectly labeled would be the *Gini impurity*. For example, if a subset were made up of 25% samples with label 0 (and 75% with label 1), assigning label 0 to a random 25% of all samples (and label 1 to the rest) would give 37.5% incorrect labels: 75% of the

---

15 For performing logistic regression on smaller datasets, consider using *exact logistic regression*.

16 Note that as of late 2017, scikit-learn's implementation of decision trees (`sklearn.tree.DecisionTreeClassifier` and other tree-based learners) does not properly handle categorical data. Categorical variables encoded with integer labels (i.e., with `sklearn.preprocessing.LabelEncoder` and *not* `sklearn.preprocessing.OneHotEncoder` or the `pandas.get_dummies()` function) will be incorrectly treated as numerical variables. Even though a scikit-learn maintainer claimed that models like `sklearn.tree.RandomForestClassifier` tend to be "very robust to categorical features abusively encoded as integer features in practice,", it is still highly recommended that you convert categorical variables to dummy/one-hot variables before feeding them into sklearn decision trees. There should be a new feature for tree-based learners to have support for categorical splits (up to 64 categories per feature) in 2018.

Copyrighted material

label-0 samples and 25% of the label-1 samples would be incorrect. A higher-quality decision tree split would split the set into subsets cleanly separated by their label, hence resulting in a lower Gini impurity; that is, the rate of misclassification would be low if most points in a set belong to the same class.

Variance reduction
: Often used in regression trees, where the dependent variable is continuous. Variance reduction is defined as the total reduction in a set's variance as a result of the split into two subsets. The best split at a node in a decision tree would be the split that results in the greatest variance reduction.

Information gain
: Information gain is a measure of the *purity* of the subsets resulting from a split. It is calculated by subtracting the weighted sum of each decision tree child node's entropy from the parent node's entropy. The smaller the entropy of the children, the greater the information gain, hence the better the split.

3. There are a few different methods for determining when to stop splitting nodes:
    - When all leaves of the tree are *pure*—that is, all leaf nodes each only contain samples belonging to the same class—stop splitting.
    - When a branch of the tree has reached a certain predefined *maximum depth*, the branch stops being split.
    - When either of the child nodes will contain fewer than the *minimum number of samples*, the node will not be partitioned.
4. Ultimately the algorithm outputs a tree structure where each node represents a binary decision, the children of each node represent the two possible outcomes of that decision, and each leaf represents the classification of data points following the path from the root to that leaf. (For impure leaves, the decision is determined by majority vote of the training data samples at that leaf.)

An important quality of decision trees is the relative ease of explaining classification or regression results, since every prediction can be expressed in a series of Boolean conditions that trace a path from the root of the tree to a leaf node. For example, if a decision tree model predicted that a malware sample belongs to malware family A, we know it is because the binary was signed before 2015, does not hook into the window manager framework, does make multiple network calls out to Russian IP addresses, etc. Because each sample traverses at most the height of the binary tree (time complexity $O(\log n)$), decision trees are also efficient to train and make predictions on. As a result, they perform favorably for large datasets.

Copyrighted material

Nevertheless, decision trees have some limitations:

- Decision trees often suffer from the problem of *overfitting*, wherein trees are overly complex and don't generalize well beyond the training set. Pruning is introduced as a regularization method to reduce the complexity of trees.
- Decision trees are more *inefficient at expressing some kinds of relationships* than others. For example, Figures 2-5 and 2-6 present the minimal decision tree required to represent the AND, OR, and XOR relationships. Notice how XOR requires one more intermediate node and split to be appropriately represented, even in this simple example. For realistic datasets, this can quickly result in exploding model complexity.

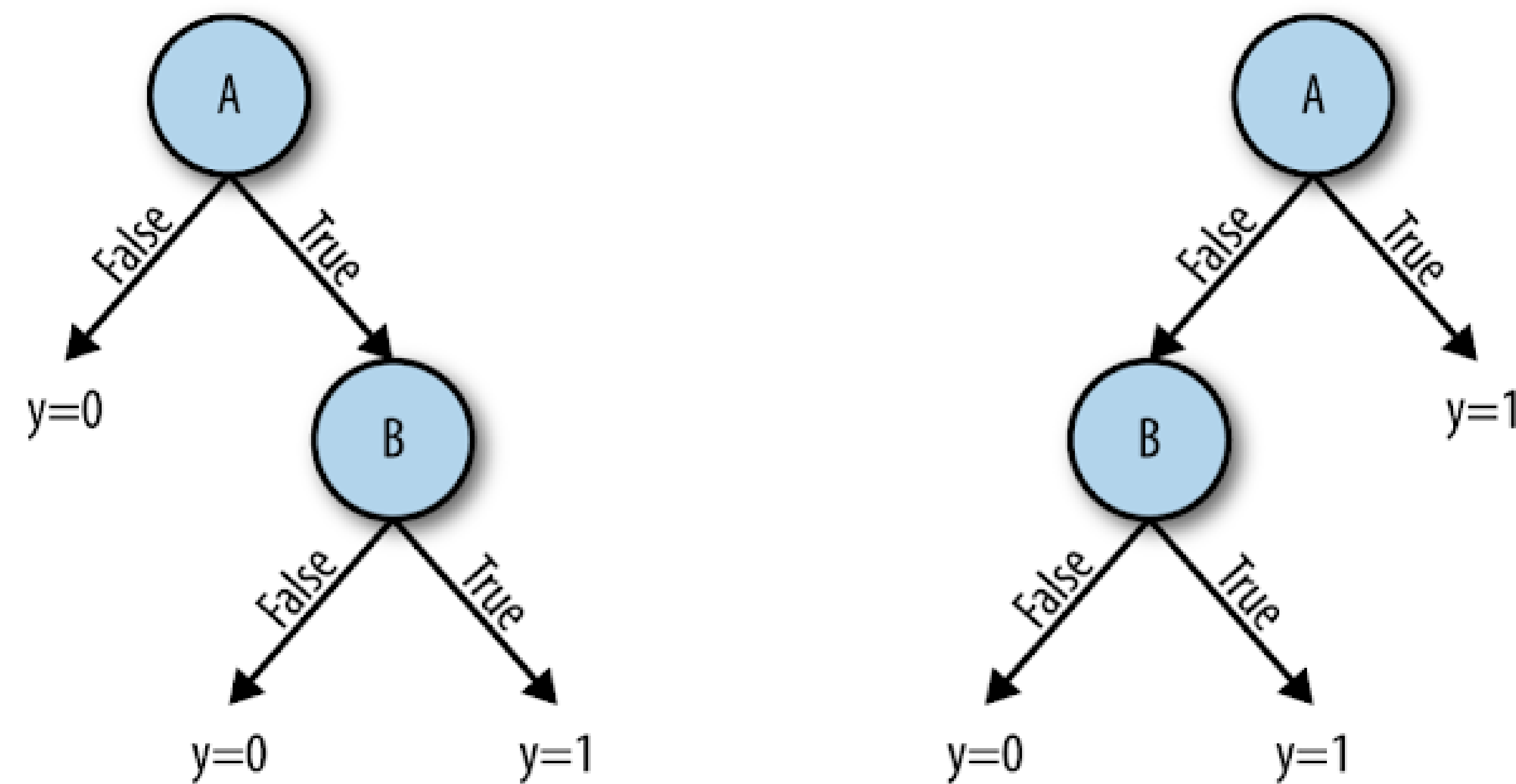


Figure 2-5. Decision tree for A-AND-B → y=1 (left), A-OR-B → y=1 (right)

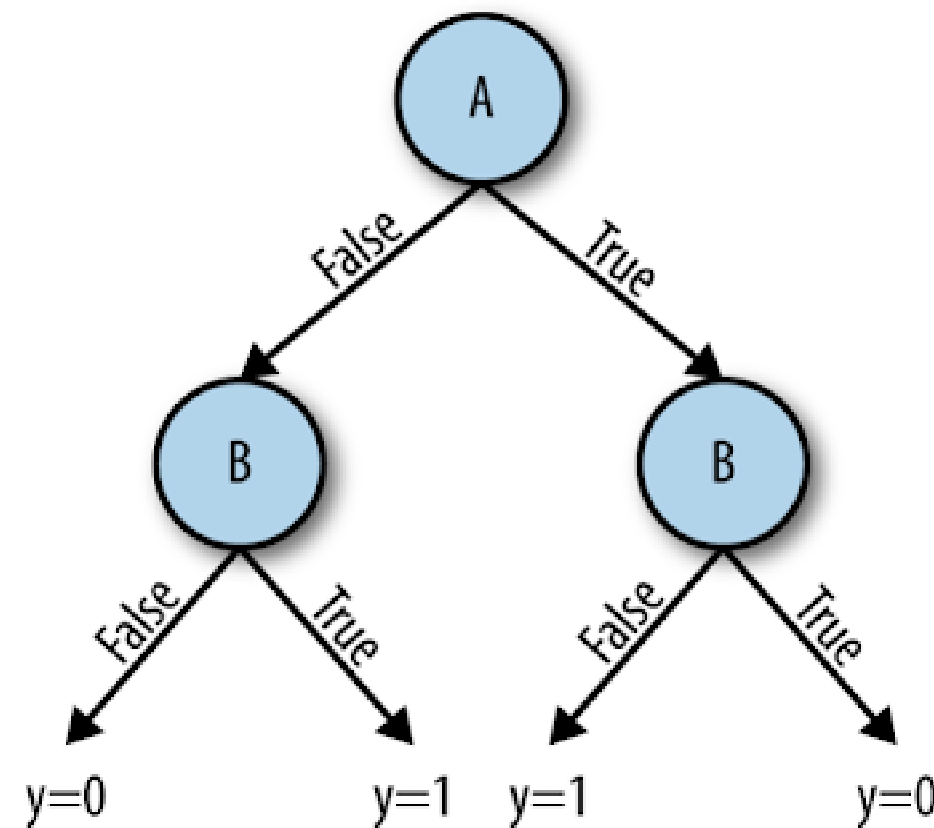


Figure 2-6. Decision tree for A-XOR-B → y=1 (bottom)

Copyrighted material

- Decision trees tend to be *less accurate and robust* than other supervised learning techniques. Small changes to the training dataset can result in large changes to the tree, which in turn result in changes to model predictions. This means that decision trees (and most other related models) are unsuitable for use in online learning or incremental learning.
- Split-quality metrics for categorical variables in decision trees are *biased toward variables with more possible values*; that is, splits on continuous variables or categorical variables with three or more categories will be chosen with a greater probability than binary variables.
- Greedy training of decision trees (as it is almost always done) *does not guarantee an optimal decision tree* because locally optimal and not globally optimal decisions are made at each split point. In fact, the training of a globally optimal decision tree is an NP-complete problem.[17]

Decision Forests

An *ensemble* refers to a combination of multiple classifiers that creates a more complex, and often better performing, classifier. Combining decision trees into ensembles is a proved technique for creating high-quality classifiers. These ensembles are aptly named *decision forests*. The two most common types of forests used in practice are *decision forests* and *gradient-boosted decision trees*:

- *Random forests* are formed by simple ensembling of multiple decision trees, typically ranging from tens to thousands of trees. After training each individual decision tree, overall random forest predictions are made by taking the statistical mode of individual tree predictions for classification trees (i.e., each tree "votes"), and the statistical mean of individual tree predictions for regression trees.

  You might notice that simply having many decision trees in the forest will result in highly similar trees and a lot of repeated splits across different trees, especially for features that are strong predictors of the dependent variable. The random forest algorithm addresses this issue using the following training algorithm:

  1. For the training of each individual tree, randomly draw a subset of *N* samples from the training dataset.

17 Laurent Hyafil and R.L. Rivest, "Constructing Optimal Binary Decision Trees is NP-Complete," *Information Processing Letters* 5:1 (1976): 15–17.

Copyrighted material

2. At each split point, we randomly select m features from the p available features, where $m \leq p$,[18] and pick the optimal split point from these m features.[19]
3. Repeat step 2 until the individual tree is trained.
4. Repeat steps 1, 2, and 3 until all trees in the forest are trained.

Single decision trees tend to overfit to their training sets, and random forests mitigate this effect by taking the average of multiple decision trees, which usually improves model performance. In addition, because each tree in the random forest can be trained independently of all other trees, it is straightforward to parallelize the training algorithm and therefore random forests are very efficient to train. However, the increased complexity of random forests can make them much more *storage intensive*, and it is *much harder to explain predictions* than with single decision trees.

- *Gradient-boosted decision trees* (GBDTs) make use of smarter combinations of individual decision tree predictions to result in better overall predictions. In gradient boosting, multiple weak learners are selectively combined by performing gradient descent optimization on the loss function to result in a much stronger learning model.

  The basic technique of *gradient boosting* is to add individual trees to the forest one at a time, using a *gradient descent* procedure to minimize the loss when adding trees. Addition of more trees to the forest stops either when a fixed limit is hit, when validation set loss reaches an acceptable level, or when adding more trees no longer improves this loss.

  Several improvements to basic GBDTs have been made to result in better performing, better generalizing, and more efficient models. Let's look at a handful of them:

1. Gradient boosting requires weak learners. Placing *artificial constraints* on trees, such as limits on tree depth, number of nodes per tree, or minimum number of samples per node, can help constrain these trees without overly diminishing their learning ability.
2. It can happen that the decision trees added early on in the additive training of gradient-boosted ensembles contribute much more to the overall prediction than the trees added later in the process. This situation results in an imbal-

---

18 For classification problems with p total features, $m = \sqrt{p}$ is recommended. For regression tasks, $m = p/3$ is recommended. Refer to section 15.2 of *The Elements of Statistical Learning*, 2nd ed., by Trevor Hastie, Robert Tibshirani, and Jerome Friedman.

19 There also exist variants of random forests that limit the set of features available to an individual decision tree; for example, if the total feature set is {A,B,C,D,E,F,G}, all split points made in decision tree 1 might randomly select only three features out of the subset of features {A,B,D,F,G}.

Copyrighted material

anced model that limits the benefits of ensembling. To solve this problem, the *contribution of each tree is weighted* to slow down the learning process, using a technique called *shrinkage*, to reduce the influence of individual trees and allow future trees to further improve the model.

3. We can combine the stochasticity of random forests with gradient boosting by *subsampling the dataset* before creating a tree and *subsampling the features* before creating a split.
4. We can use standard and popular regularization techniques such as L_1 and L_2 *regularization* to smooth final learned weights to further avoid overfitting.

XGBoost[20] is a popular GBDT flavor that achieves state-of-the-art results while scaling well to large datasets. As the algorithm that was responsible for many winning submissions to machine learning competitions, it garnered the attention of the machine learning community and has become the decision forest algorithm of choice for many practitioners. Nevertheless, GBDTs are more *prone to overfitting* than random forests, and also *more difficult to parallelize* because they use additive training, which relies on the results of a given tree to update gradients for the subsequent tree. We can mitigate overfitting of GBDTs by using *shrinkage*, and we can parallelize training within a single tree instead of across multiple trees.

Support Vector Machines

Like logistic regression, a *support vector machine* (SVM) is (in its simplest form) a linear classifier, which means that it produces a hyperplane in a vector space that attempts to separate the two classes in the dataset. The difference between logistic regression and SVMs is the loss function. Logistic regression uses a log-likelihood function that penalizes all points proportionally to the error in the probability estimate, even those on the correct side of the hyperplane. An SVM, on the other hand, uses a *hinge loss*, which penalizes only those points on the wrong side of the hyperplane or very near it on the correct side.

More specifically, the SVM classifier attempts to find the *maximum-margin* hyperplane separating the two classes, where "margin" indicates the distance from the separating plane to the closest data points on each side. For the case in which the data is not linearly separable, points within the margin are penalized proportionately to their distance from the margin. Figure 2-7 shows a concrete example: the two classes are represented by white and black points, respectively. The solid line is the separating

20 Tianqi Chen and Carlos Guestrin, "XGBoost: A Scalable Tree Boosting System," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 785–794.

Copyrighted material

plane and the dashed lines are the margins. The square points are the *support vectors*; that is, those that provide nonzero contribution to the loss function. This loss function is expressed mathematically as:

$$\beta + C \sum_{i=1}^{N} \xi_i$$

where β is the margin, ξ_i is the distance from the ith support vector to the margin, and C is a model hyperparameter that determines the relative contribution of the two terms.

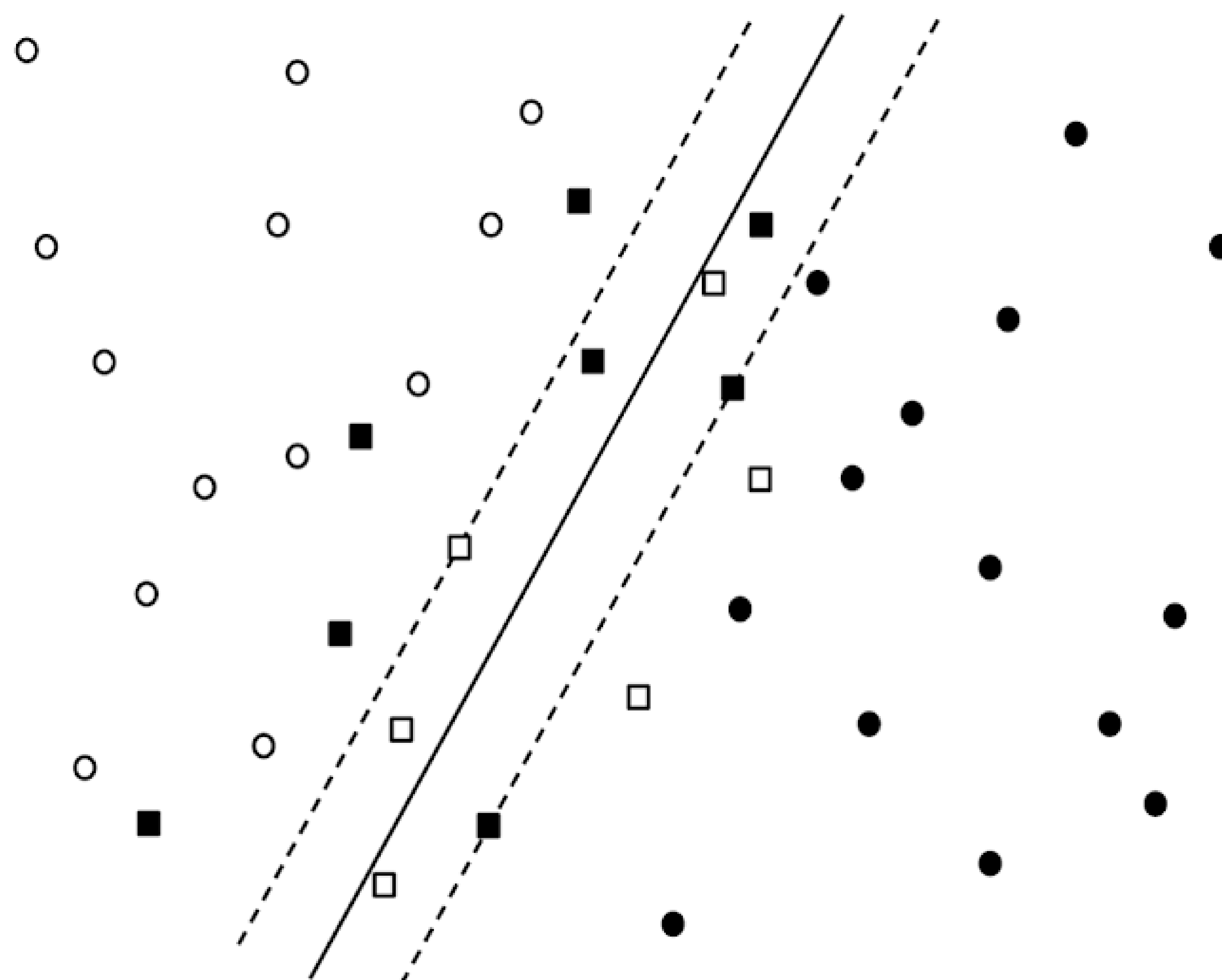


Figure 2-7. Classification boundary (dark line) and margins (dashed lines) for linear SVM separating two classes (black and white points); squares represent support vectors

To classify a new data point x, we simply determine which side of the plane x falls on. If we want to get a real-valued score we can compute the distance from x to the separating plane and then apply a sigmoid to map to [0,1].

The real power of SVMs comes from the *kernel trick*, which is a mathematical transformation that takes a linear decision boundary and produces a nonlinear boundary. At a high level, a kernel transforms one vector space, V_1, to another space, V_2. Mathematically, the kernel is a function on $V_1 \times V_1$ defined by $K(x, y)$, and each $x \in V_1$ is mapped to the *function* $K(x, \cdot)$; V_2 is the space spanned by all such functions. For example, we can recover the linear SVM by defining K to be the usual dot product $K(x, y) = x \cdot y$.

Copyrighted material

If the kernel is nonlinear, a linear classifier in V_1 will produce a nonlinear classifier in V_2. The most popular choice is the *radial basis function* $K(x, y) = e^{-\gamma|x - y|}$. Even though we omit the mathematical details here,[21] you can think of an SVM with the RBF kernel as producing a sort of smoothed linear combination of spheres around each point x, where the inside of each sphere is classified the same as x, and the outside is assigned the opposite class. The parameter γ determines the radius of these spheres; i.e., how close to each point you need to be in order to be classified like that point. The parameter C determines the "smoothing"; a large value of C will lead to the classifier consisting of a union of spheres, while a small value will yield a wigglier boundary influenced somewhat by each sphere. We can therefore see that too large a value of C will produce a model that overfits the training data, whereas too small a value will give a poor classifier in terms of accuracy. (Note that γ is unique to the RBF kernel, whereas C exhibits the same properties for any kernel, including the linear SVM. Optimal values of these parameters are usually found using *grid search*.)

SVMs have shown very good performance in practice, especially in high-dimensional spaces, and the fact that they can be described in terms of support vectors leads to efficient implementations for scoring new data points. However, the complexity of training a kernelized SVM grows quadratically with the number of training samples, so that for training set sizes beyond a few million, kernels are rarely used and the decision boundary is linear. Another disadvantage is that the scores output by SVMs are not interpretable as probabilities; converting scores to probabilities requires additional computation and cross-validation, for example using Platt scaling or isotonic regression. The scikit-learn documentation has further details.

Naive Bayes

The *Naive Bayes* classifier is one of the oldest statistical classifiers. The classifier is called "naive" because it makes a very strong statistical assumption, namely that features are chosen independently from some (unknown) distribution. This assumption never actually holds in real life. For example, consider a spam classifier for which the features are the words in the message. The Naive Bayes assumption posits that a spam message is composed by sampling words independently, where each word w has a probability $p_{w, spam}$ of being sampled, and similarly for good messages. This assumption is clearly ludicrous; for one thing, it completely ignores word ordering. Yet despite the fact that the assumption doesn't hold, Naive Bayes classifiers have been shown to be quite effective for problems such as spam classification.

---

21 See section 5.8 and Chapter 12 of *The Elements of Statistical Learning* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman.

Copyrighted material

The main idea behind Naive Bayes is as follows: given a data point with feature set $X = x_1, ..., x_n$, we want to determine the probability that the label Y for this point is the class C. In Equation 2-1, this concept is expressed as a conditional probability.

Equation 2-1.

$$\Pr\left[Y = C \mid X = \left(x_1, ..., x_n\right)\right]$$

Now using *Bayes' Theorem*, this probability can be reexpressed as in Equation 2-2.

Equation 2-2.

$$\frac{\Pr\left[X = \left(x_1, \ldots, x_n\right) \middle| Y = C\right] \cdot \Pr\left[Y = C\right]}{\Pr\left[X = \left(x_1, \ldots, x_n\right)\right]}$$

If we make the very strong assumption that for samples in each class the features are chosen independently of one another, we get Equation 2-3.

Equation 2-3.

$$\Pr\left[X = \left(x_1, ..., x_n\right) \mid Y = C\right] = \prod_{i=1}^{n} \Pr\left[X_i = x_i \mid Y = C\right]$$

we can estimate the numerator of Equation 2-2 from labeled data: $\Pr\left[X_i = x_i \mid Y = C\right]$ is simply the fraction of samples with the ith feature equal to x_i out of all the samples in class C, whereas $\Pr\left[Y = C\right]$ is the fraction of samples in class C out of all the labeled samples.

What about the denominator of Equation 2-2? It turns out that we don't need to compute this, because in two-class classification it is sufficient to compute the *ratio* of the probability estimates for the two classes C_1 and C_2. This ratio (Equation 2-4) gives a positive real number score.

Equation 2-4.

$$\begin{aligned}\theta &= \frac{\Pr\left[Y = C_1 \mid X = \left(x_1, ..., x_n\right)\right]}{\Pr\left[Y = C_2 \mid X = \left(x_1, ..., x_n\right)\right]} \\ &\approx \frac{\Pr\left[Y = C_1\right]\Pi_{i=1}^{n} \Pr\left[X_i = x_i \mid Y = C_1\right]}{\Pr\left[Y = C_2\right]\Pi_{i=1}^{n} \Pr\left[X_i = x_i \mid Y = C_2\right]}\end{aligned}$$

Copyrighted material

A score of $\theta > 1$ indicates that C_1 is the more likely class, whereas $\theta < 1$ indicates that C_2 is more likely. (If optimizing for one of precision or recall you might want to choose a different threshold for the classification boundary.)

The astute observer will notice that we obtained our score θ without reference to a loss function or an optimization algorithm. The optimization algorithm is actually hidden in the estimate of $\Pr\left[X_i = x_i \mid Y = C\right]$; using the fraction of samples observed in the training data gives the *maximum likelihood estimate*, the same loss function used for logistic regression. The similarity to logistic regression doesn't stop there: if we take logarithms of Equation 2-4 the righthand side becomes a linear function of the features, so we can view Naive Bayes as a linear classifier, as well.

A few subtleties arise when trying to use Naive Bayes in practice:

- What happens when all of the examples of some feature are in the same class (e.g. brand names of common sex enhancement drugs appear only in spam messages)? Then, one of the terms of Equation 2-4 will be zero, leading to a zero or infinity estimate for θ, which doesn't make sense. To get around this problem we use *smoothing*, which means adding "phantom" samples to the labeled data for each feature. For example, if we are smoothing by a factor of α, we would calculate the value for a feature as follows:

$$\Pr\left[X_i = x_i \mid Y = C\right] = \frac{(\#\text{ samples in class } C \text{ with feature } x_i) + \alpha}{(\#\text{ samples in class } C\ ) + \alpha \cdot (\#\text{ of features})}$$

  The choice $\alpha = 1$ is called *Laplace smoothing*, whereas $\alpha < 1$ is called *Lidstone smoothing*.

- What happens if a feature x_i appears in our validation set (or worse, in real-life scoring) that did not appear in the training set? In this case we have no estimate for $\Pr\left[X_i = x_i \mid Y = C\right]$ at all. A naive estimate would be to set the probability to $\Pr\left[Y = C\right]$; for more sophisticated approaches see the work of Freeman.[22]

As a final note, we can map the score $\theta \in (0, \infty)$ to a probability in (0,1) using the mapping $\theta \rightarrow \frac{\theta}{1+\theta}$; however, this probability estimate will not be properly calibrated. As with SVMs, to obtain better probability estimates we recommend techniques such as *Platt scaling* or *isotonic regression*.

22 David Freeman, "Using Naive Bayes to Detect Spammy Names in Social Networks," *Proceedings of the 2013 ACM Workshop on Artificial Intelligence in Security* (2013): 3–12.

Copyrighted material

k-Nearest Neighbors

The *k*-nearest neighbors (*k*-NN) algorithm is the most well-known example of a *lazy learning* algorithm. This type of machine learning technique puts off most computations to classification time instead of doing the work at training time. Lazy learning models don't learn generalizations of the data during the training phase. Instead, they record all of the training data points they are passed and use this information to make the local generalizations around the test sample during classification. *k*-NN is one of the simplest machine learning algorithms:

- The training phase simply consists of storing all the feature vectors and corresponding sample labels in the model.
- The classification prediction[23] is simply the most common label out of the test sample's *k* nearest neighbors (hence the name).

The distance metrics for determining how "near" points are to each other in an *n*-dimensional feature space (where *n* is the size of the feature vectors) are typically the *Euclidean distance* for continuous variables and the *Hamming distance* for discrete variables.

As you might imagine, with such a simple algorithm the training phase of *k*-NN is typically very fast compared to other learning algorithms, at the cost of classification processing time. Also, the fact that all feature vectors and labels need to be stored within the model results in a very *space-inefficient model*. (A *k*-NN model that takes in 1 GB of training feature vectors will at least be 1 GB in size.)

The simplicity of *k*-NN makes it a popular example for teaching the concept of machine learning to novices, but it is rarely seen in practical scenarios because of the serious drawbacks it has. These include:

- *Large model sizes*, because models must store (at least) all training data feature vectors and labels.
- *Slow classification speeds*, because all generalization work is pushed off until classification time. Searching for the nearest neighbors can be time consuming, especially if the model stores training data points in a manner that is not optimized for spatial search. *k-d trees* (explained in "k-d trees" on page 72) are often used as an optimized data structure to speed up neighbor searches.[24]

23 *k*-NN can also be used for regression—typically, the average of a test sample's *k* nearest neighboring sample labels is taken to be the prediction result.

24 Jon Louis Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM* 18 (1975): 509–517.

Copyrighted material

- *High sensitivity to class imbalance*[25] in the dataset. Classifications will be skewed towards the classes with more samples in the training data since there is a greater likelihood that samples of these classes will make it into the *k*-NN set of any given test sample.
- *Diminished classification accuracy* due to noisy, redundant, or unscaled features. (Choosing a larger *k* reduces the effect of noise in the training data, but also can result in a weaker learner.)
- *Difficulty in choosing the parameter* k. Classification results are highly dependent on this parameter, and it can be difficult to choose a *k* that works well across all parts of the feature space because of differing densities within the dataset.
- *Breaks down in high dimensions* due to the "curse of dimensionality." In addition, with more dimensions in the feature space, the "neighborhood" of any arbitrary point becomes larger, which results in noisier neighbor selection.

Neural Networks

Artificial neural networks (ANNs) are a class of machine learning techniques that have seen a resurgence in popularity recently. One can trace the origins of neural networks all the way back to 1942, when McCulloch and Pitts published a groundbreaking paper postulating how neurons in the human nervous system might work.[26] Between then and the 1970s, neural network research advanced at a slow pace, in large part due to von Neumann computing architectures (which are quite in opposition to the idea of ANNs) being in vogue. Even after interest in the field was renewed in the 1980s, research was still slow because the computational requirements of training these networks meant that researchers often had to wait days or weeks for the results of their experiments. What triggered the recent popularity of neural networks was a combination of hardware advancements—namely graphics processing units (GPUs) for "almost magically" parallelizing and speeding up ANN training—and the availability of the huge amounts of data that ANNs need to get good at complex tasks like image and speech recognition.

The human brain is composed of a humongous number of neurons (on the order of 10 billion), each with connections to tens of thousands of other neurons. Each neuron receives electrochemical inputs from other neurons, and if the sum of these electrical inputs exceeds a certain level, the neuron then triggers an output transmission of another electrochemical signal to its attached neurons. If the input does not exceed this level, the neuron does not trigger any output. Each neuron is a very simple

25 We explain the concept of *class imbalance* in greater detail in Chapter 5.

26 W.S. McCulloch and W.H. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics* 5 (1942): 115–133.

Copyrighted material

In any case, if you are artificially sampling your training data, be sure to either leave the validation and test sets unsampled or use a performance metric that is invariant under sampling (such as ROC AUC, described in "Choosing Thresholds and Comparing Models" on page 62). Otherwise, sampling the validation set will skew your performance metrics.

Missing features

In an ideal world every event is logged perfectly with exactly the data you need to classify. In real life, things go wrong: bugs appear in the logging; you only realize partway through data collection that you need to log a certain feature; some features are delayed or purged. As a result, some of your samples might have *missing features*. How do you incorporate these samples into your training set?

One approach is to simply remove any event with missing features. If the features are missing due to sporadic random failures this might be a good choice; however, if the data with missing features is clustered around a certain event or type of data, throwing out this data will change your distribution.

To use a sample with a missing feature you will need to *impute* the value of the missing feature. There is a large literature on imputation that we won't attempt to delve into here; it suffices to say that the simplest approach is to assign the missing feature the average or median value for that feature. More complex approaches involve using existing features to predict the value of the missing feature.

Large events

In an adversarial setting, you might have large-scale attacks from relatively unsophisticated actors that you are able to stop easily. If you naïvely include these events in your training data your model might learn how to stop these attacks but not how to address smaller, more sophisticated attacks. Thus, for better performance you might need to downsample large-scale events.

Attacker evolution

In an adversarial environment the attackers will rarely give up after you deploy a new defense—instead, they will modify their methods to try to circumvent your defenses, you will need to respond, and so on, and so forth, and so on. Thus, not only does the distribution of attacks change over time, but it changes *directly in response to your actions*. To produce a model that is robust against current attacks, your training set should thus weight recent data more heavily, either by relying only on the past *n* days or weeks, or by using some kind of decay function to downsample historical data.

On the other hand, it may be dangerous for your model to "forget" attacks from the past. As a concrete example, suppose that each day you train a new model on the past seven days' worth of data. An attack happens on Monday that you are not able to stop

Copyrighted material

(though you do find and label it quickly). On Tuesday your model picks up the new labeled data and is able to stop the attack. On Wednesday the attacker realizes they are blocked and gives up. Now consider what happens the next Wednesday: there have been no examples of this attack in the past seven days, so the model you produce might not be tuned to stop it—and if the attacker finds the hole, the entire cycle will repeat.

All of the preceding considerations illustrate what could go wrong with certain choices of training data. It is up to you to weigh the trade-offs between data freshness, historical robustness, and system capacity to produce the best solution for your needs.

Feature Selection

If you have a reasonably efficient machine learning infrastructure, most of your time and energy will be spent on feature engineering—figuring out signals that you can use to identify attacks, and then building them into your training and scoring pipeline. To make best use of your effort, you want to use only features that provide high discriminatory power; the addition of each feature should noticeably improve your model.

In addition to requiring extra effort to build and maintain, redundant features can hurt the quality of your model. If the number of features is greater than the number of data points, your model will be overfit: there are enough model parameters to draw a curve through all the training data. In addition, highly correlated features can lead to instability in model decisions. For example, if you have a feature that is "number of logins yesterday" and one that is "number of logins in the last two days," the information you are trying to collect will be split between the two features essentially arbitrarily, and the model might not learn that either of these features is important.

You can solve the feature correlation problem by computing covariance matrices between your features and combining highly correlated features (or projecting them into orthogonal spaces; in the previous example, "number of logins in the day before yesterday" would be a better choice than "number of logins in the last two days").

There are number of techniques to address the feature selection problem:

- Logistic regression, SVMs, and decision trees/forests have methods of determining relative feature importance; you can run these and keep only the features with highest importance.
- You can use L_1 regularization (see the next section) for feature selection in logistic regression and SVM classifiers.
- If the number n of features is reasonably small (say, $n < 100$) you can use a "build it up" approach: build n one-feature models and determine which is best on your

Copyrighted material

validation set; then build $n-1$ two-feature models, and so on, until the gain of adding an additional feature is below a certain threshold.

- Similarly, you can use a "leave one out" approach: build a model on n features, then n models on $n-1$ features and keep the best, and so on until the loss of removing an additional feature is too great.

scikit-learn implements the `sklearn.feature_selection.SelectFromModel` helper utility that assists operators in selecting features based on importance weights. As long as a trained estimator has the `feature_importances_` or `coef_` attribute after fitting,[28] it can be passed into `SelectFromModel` for feature selection importance ranking. Assuming that we have a pretrained `DecisionTreeClassifier` model (variable name `clf`) and an original training dataset (variable name `train_x`) with 119 features, here is a short code snippet showing how to use `SelectFromModel` to keep only features with a `feature_importance` that lies above the mean:[29,30]

```
from sklearn.feature_selection import SelectFromModel

sfm = SelectFromModel(clf, prefit=True)

# Generate new training set, keeping only the selected features
train_x_new = sfm.transform(train_x)

print("Original num features: {}, selected num features: {}"
      .format(train_x.shape[1], train_x_new.shape[1]))

> Original num features: 119, selected num features: 7
```

28 Most tree-based estimators like `DecisionTreeClassifier` and `RandomForestClassifier`, as well as some ensemble estimators like `GradientBoostingClassifier`, have the `feature_importances_` attribute. Generalized linear models such as `LinearRegression` and `LogisticRegression` and support vector machines such as SVC have the `coef_` attribute, allowing `SelectFromModel` to compare magnitudes of the coefficients or importances corresponding to each feature.

29 Using the mean as a feature importance threshold is the default strategy of `SelectFromModel` unless you specify the `threshold` parameter as something else; for example, `median` or a static value.

30 You can find an example of applying `sklearn.feature_selection.SelectFromModel` to a real problem in a Python Jupyter notebook in *chapter2/select-from-model-nslkdd.ipynb* from our code repository.

Copyrighted material

Overfitting and Underfitting

One problem that can occur with any machine learning algorithm is *overfitting*: the model you construct matches the training data so thoroughly that it does not generalize well to unseen data. For example, consider the decision boundary shown for a two-dimensional dataset in the left of Figure 2-9. All points are classified correctly, but the shape of the boundary is very complex, and it is unlikely that this boundary can be used to effectively separate new points.

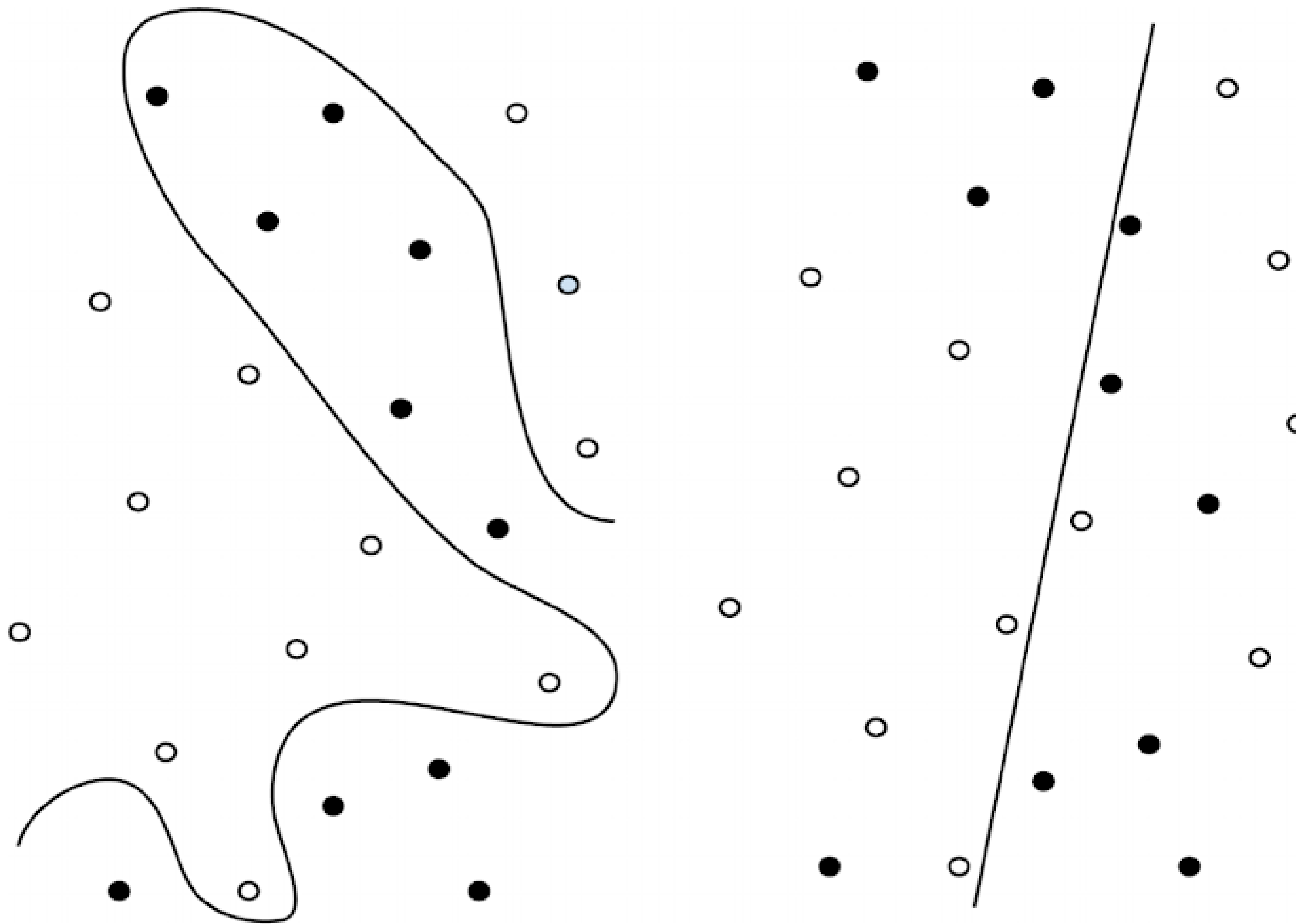


Figure 2-9. Left: overfit decision boundary; right: underfit decision boundary

On the other hand, too simple of a model might also result in poor generalization to unseen data; this problem is called *underfitting*. Consider, for example, the decision boundary shown on the right side of Figure 2-9; this simple line is correct on the training data a majority of the time, but makes a lot of errors and will likely also have poor performance on unseen data.

The most common approach to minimizing overfitting and underfitting is to incorporate model complexity into the training procedure. The mathematical term for this is *regularization*, which means adding a term to the loss function that represents model complexity quantitatively. If ϕ represents the model, y_i the training labels, and $\hat{y}_i$ the predictions (either labels or probabilities), the regularized loss function is:

$$\mathscr{L}(\phi) = \sum_i \ell(\hat{y}_i, y_i) + \lambda \cdot \Omega(\phi)$$

Copyrighted material

where ℓ is the aforementioned ordinary loss function, and Ω is a penalty term. For example, in a decision tree Ω could be the number of leaves; then trees with too many leaves would be penalized. You can adjust the parameter λ to balance the trade-off between the ordinary loss function and the regularization term. If λ is too small, you might get an overfit model; if it is too large, you might get an underfit model.

In logistic regression the standard regularization term is the norm of the coefficient vector $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_n)$. There are two different ways to compute the norm: L_2 regularization uses the standard Euclidean norm $|\boldsymbol{\beta}| = \Sigma_i \beta_i^2$, whereas L_1 regularization uses the so-called "Manhattan distance" norm $|\boldsymbol{\beta}| = \Sigma_i |\beta_i|$. The L_1 norm has the property that local minima occur when feature coefficients are zero; L_1 regularization thus selects the features that contribute most to the model.

When using regularized logistic regression, you must take care to normalize the features before training the model, for example by applying a linear transformation that results in each feature having mean 0 and standard deviation 1. If no such transformation is applied, the coefficients of different features are not comparable. For example, the feature that is account age in seconds will in general be much larger than the feature that is number of friends in the social graph. Thus, the coefficient for age will be much smaller than the coefficient for friends in order to have the same effect, and regularization will penalize the friends coefficient much more strongly.

Regardless of the model you are using, you should choose your regularization parameters based on experimental data from the validation set. But be careful not to overfit to your validation set! That's what the test set is for: if performance on the test set is much worse than on the validation set, you have overfit your parameters.

Choosing Thresholds and Comparing Models

The supervised classification algorithm you choose will typically output a real-valued score, and you will need to choose a threshold or thresholds[31] above which to block the activity or show additional friction (e.g., require a phone number). How do you choose this threshold? This choice is ultimately a business decision, based on the trade-off between security and user friction. Ideally you can come up with some cost function—for example, 1 false positive is equal to 10 false negatives—and minimize the total cost on a representative sample of the data. Another option is to fix a precision or recall target—for example, 98%—and choose a threshold that achieves that target.

31 The default method that most machine learning libraries use to deal with this is to pick the class that has the highest score and use that as the prediction result. For instance, for a binary classification problem this simply translates to a threshold of 50%, but for a three-class (or more) classification problem the class with the highest probability/confidence is selected as the classifier's prediction result.

Copyrighted material

Now suppose that you have two versions of your model with different parameters (e.g., different regularization) or even different model families (e.g., logistic regression versus random forest). Which one is better? If you have a cost function this is easy: compute the cost of the two versions on the same dataset and choose the lower-cost option. If fixing a precision target, choose the version that optimizes recall, and vice versa.

Another common method for model comparison is to plot the *receiver operating characteristic* (ROC) curve and compute the *area under the curve* (AUC). The ROC curve plots false positive rate (FP / (FP + TN)) on the x-axis and true positive rate (TP / (TP + FN), also known as recall) on the y-axis. Each point on the curve corresponds to a score threshold and represents the (FPR, TPR) pair at that threshold. The AUC can be interpreted as the probability that a randomly chosen positive example has a higher score than a randomly chosen negative example; under this interpretation it's easy to see that the worst case is AUC 0.5, which is equivalent to a random ordering of samples.[32]

Figure 2-10 shows an example ROC curve, with the line $y = x$ plotted for comparison. Because the AUC is very high, we have used a log scale to zoom in on the lefthand side, which is where differences between high-performance models will appear; if you are operating only in the high-precision region of the curve, you may want to calculate up to a threshold false positive rate, such as 1%.

One nice property of AUC is that it is unaffected by sampling bias. Thus, if you sample two classes with different weights, the AUC you get on the resulting dataset will be representative of the AUC on the unsampled dataset.

32 If the AUC is less than 0.5, by reversing the classifier labels you can produce a classifier with AUC > 0.5.

Copyrighted material

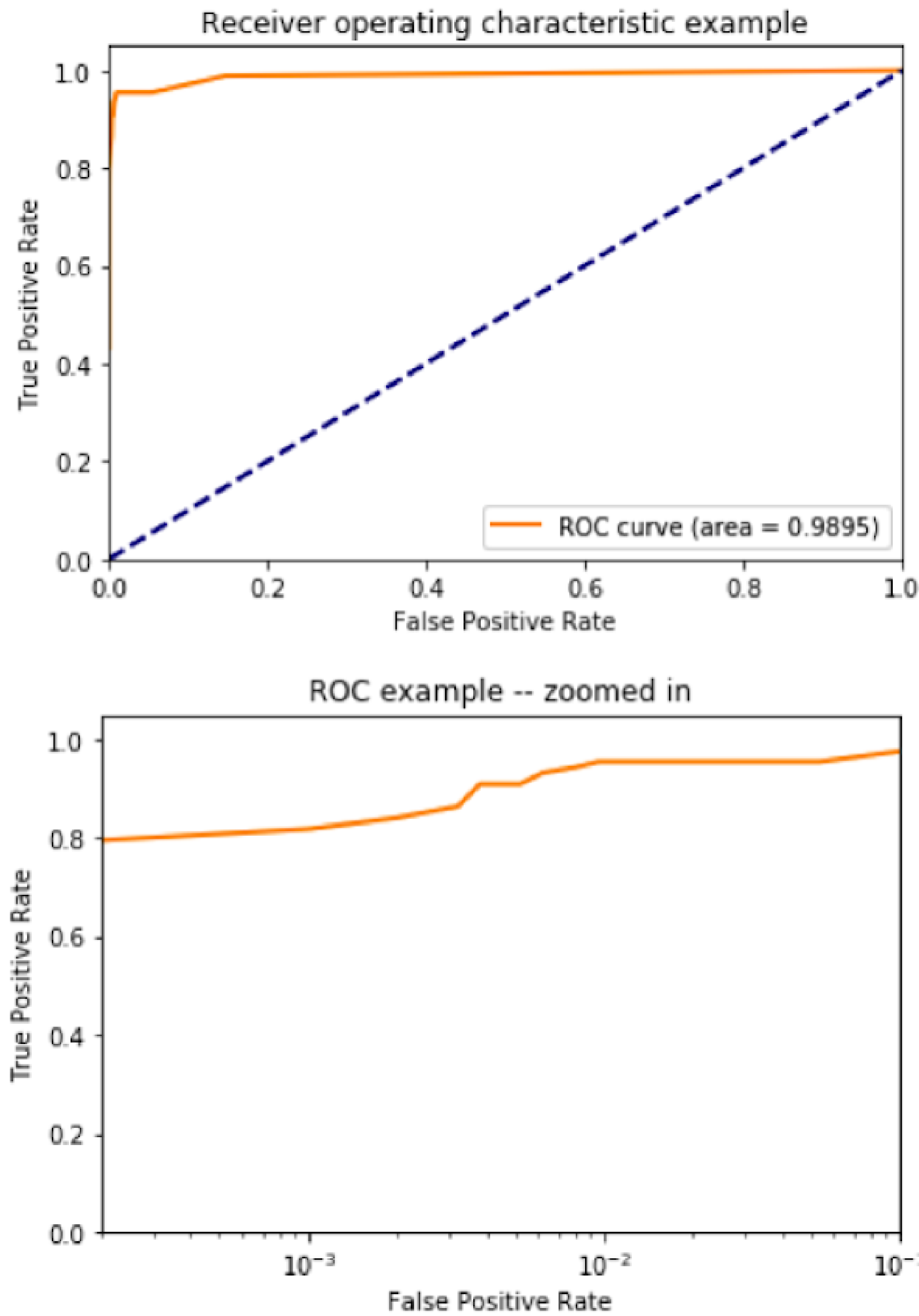


Figure 2-10. ROC curves

One common metric that is limited in real-world usefulness is the *F-score*, which is defined as follows:

$$F_{\alpha} = \frac{1 + \alpha}{\frac{1}{precision} + \frac{\alpha}{recall}}$$

The F-score combines precision and recall and harshly penalizes extremes; however, it requires choosing a threshold and a relative weighting of precision and recall (parametrized by α).

Copyrighted material

Clustering

Bad things often happen in bunches. For example, if someone is trying to breach your network, there is a good chance that they will try many times before actually getting through. Or, if someone is sending pharmaceutical spam, they will need to send a lot of emails in order to get enough people to fall for the scam. Thus, your job as a defender will be made easier if you can segment your traffic into groups belonging to the same actor, and then block traffic from malicious actors. This process of segmentation is called *clustering*.

In this section, we survey some common techniques for clustering data. Of course, grouping your data is not an end in and of itself—your ultimate goal is to determine *which* clusters consist of malicious activity. Thus, we will also discuss various techniques for labeling the clusters generated by the different algorithms.

Clustering Algorithms

The geometric intuition behind clustering is straightforward: you want to group together data points that are "close together" in some sense. Thus, for any algorithm to work you need to have some concrete way to measure "closeness"; such a measurement is called a *metric*. The metric and clustering algorithm you use will depend on the form your data is in; for example, your data might consist of real-valued vectors, lists of items, or sequences of bits. We now consider the most popular algorithms.

Grouping

The most basic clustering method is so simple that it is not even usually thought of as a clustering method: namely, pick one or more dimensions and define each cluster to be the set of items that share values in that dimension. In SQL syntax, this is the `GROUP BY` statement, so we call this technique "grouping." For example, if you group on IP address, you will define one cluster per IP address, and the elements of the cluster will be entities that share the same IP address.

We already saw the grouping technique at the beginning of this chapter, when we considered high-volume requests coming in on the same IP address; this approach is equivalent to clustering on IP address and labeling as malicious any cluster with more than 20 queries per second. This example illustrated the power of clustering via simple grouping, and you will find that you can go pretty far without resorting to more complex algorithms.

k-means

k-means is usually the first algorithm that comes to mind when you think of clustering. *k*-means applies to real-valued vectors, when you know how many clusters you expect; the number of clusters is denoted by *k*. The goal of the algorithm is to assign

Copyrighted material

each data point to a cluster such that the sum of the distances from each point to its cluster centroid is minimized. Here the notion of distance is the usual Euclidean distance in a vector space:

$$d(x, y) = \sqrt{\sum_i (x_i - y_i)^2}$$

In mathematical terms, the k-means algorithm computes a cluster assignment $f: X \to \{1, \ldots k\}$ that minimizes the *loss function*:

$$L(X) = \sum_i d(x_i, c_{f(x_i)})$$

where $X = \{x_1, \ldots, x_n\}$ is your dataset, c_j is the jth centroid, and d is the distance between two points. The value $L(X)$ is called "inertia."

The standard algorithm for computing k-means clusters is as follows:

1. Choose k centroids $c_1, \ldots, c_k$ at random.
2. Assign each data point x_i to its nearest centroid.
3. Recompute the centroids c_j by taking the average of all the data points assigned to the jth cluster.
4. Repeat (2) and (3) until the algorithm *converges*; that is, the difference between $L(X)$ on successive iterations is below a predetermined threshold.

k-means is a simple and effective clustering algorithm that scales well to very large datasets. However, there are some things for which you need to be on the lookout:

- Since k is a fixed parameter of the algorithm, you must choose it appropriately. If you know how many clusters you are looking for (e.g., if you are trying to cluster different families of malware), you can simply choose k to be that number. Otherwise, you will need to experiment with different values of k. It is also common to choose values of k that are between one to three times the number of classes (labels) in your data, in case some categories are discontinuous. Warning: loss functions computed using different values of k are not comparable to each other!
- You must *normalize your data* before using k-means. A typical normalization is to map the jth coordinate x_{ij} to $(x_{ij}-\mu_j)/\sigma_j$, where μ_j is the mean of the jth coordinates, and σ_j is the standard deviation.

Copyrighted material

To see why normalization is necessary, consider a two-dimensional dataset whose first coordinate ranges between 0 and 1, and whose second coordinate ranges between 0 and 100. Clearly the second coordinate will have a much greater impact on the loss function, so you will lose information about how close together points are in the first coordinate.

- *Do not use k-means with categorical features*, even if you can represent them as a number. For example, you could encode "red," "green," and "blue" as 0, 1, and 2, respectively, but these numbers don't make sense in a vector space—there is no reason that blue should be twice as far from red as green is. This problem can be addressed by one-hot encoding the categorical features as multiple binary features (as discussed in the worked example earlier in this chapter), but…
- *Beware when using k-means with binary features*. *k*-means can sometimes be used with binary features, encoding the two responses as 0 and 1, or −1 and 1, but results here can be unpredictable; the binary feature might become the dominant feature determining the cluster, or its information might be lost entirely.
- *k*-means *loses effectiveness in high dimensions*, due to the "curse of dimensionality"—all points are roughly equally distant from each other. For best results use *k*-means in low dimensions or after applying a dimensionality reduction algorithm such as principal component analysis (PCA). Another option is to use the *L-infinity distance*, where the distance between two points is taken to be the maximum of the difference of any coordinate:

$$d(x, y) = max_i(|x_i - y_i|)$$

- *k*-means works best when the initial centroids are chosen at random; however, this choice can make reproducing results difficult. *Try different choices of initial centroids* to see how the results depend on the initialization.
- *k*-means assumes that the clusters are *spherical* (*globular*) in nature. As you can imagine, it *does not work well on non-spherical distributions*, such as the one illustrated in Figure 2-11.

Copyrighted material

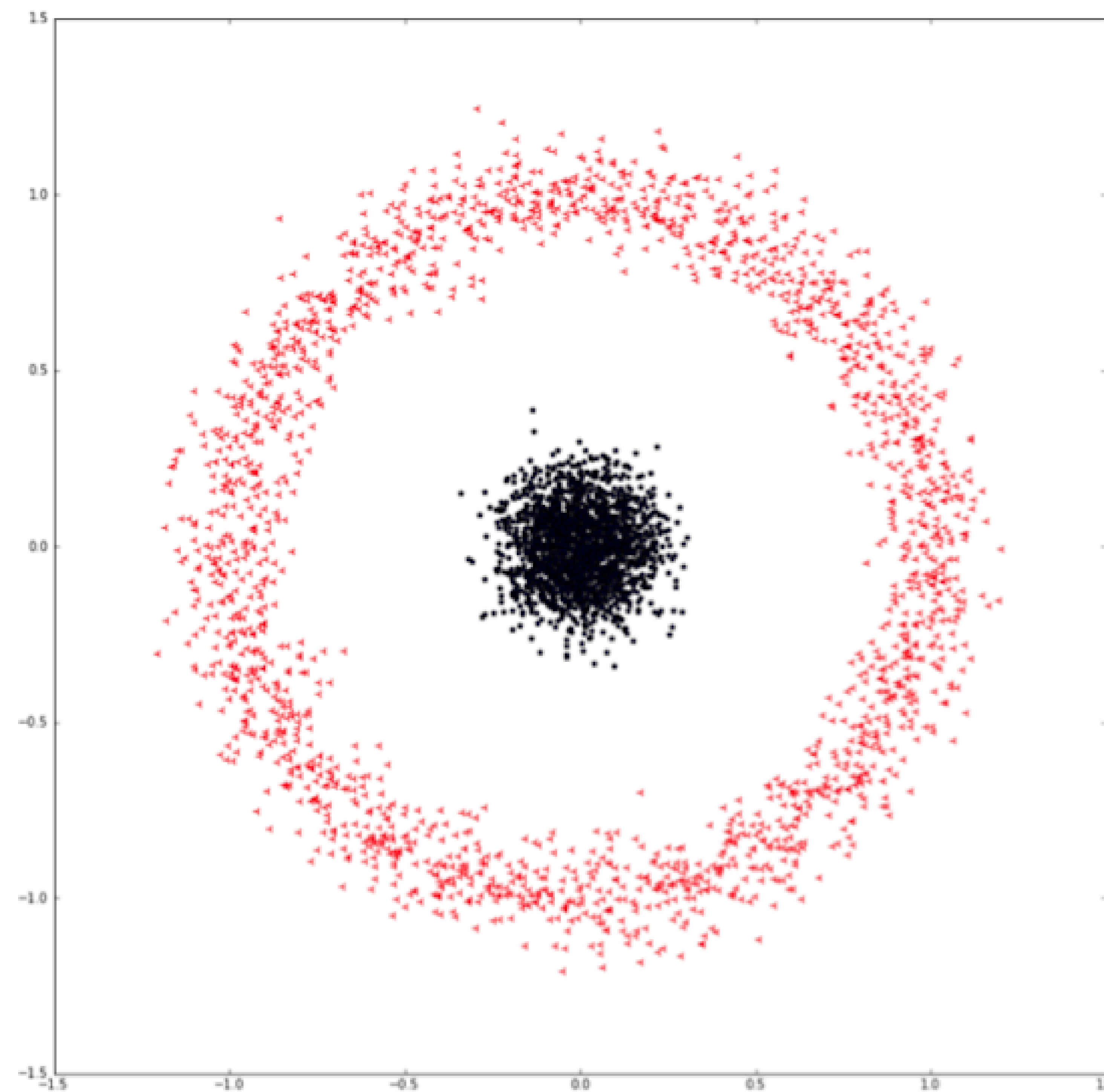


Figure 2-11. Nonspherical data distribution

Hierarchical clustering

Unlike the *k*-means algorithm, hierarchical clustering methods are not parametrized by an operator-selected value *k* (the number of clusters you want to create). Choosing an appropriate *k* is a nontrivial task, and can significantly affect clustering results. *Agglomerative (bottom-up) hierarchical clustering* builds clusters as follows (illustrated in Figure 2-12):

1. Assign each data point to its own cluster (Figure 2-12, bottom layer).
2. Merge the two clusters that are the most similar, where "most similar" is determined by a distance metric such as the *Euclidean distance* or *Mahalanobis distance*.
3. Repeat step 2 until there is only one cluster remaining (Figure 2-12, top layer).
4. Navigate the layers of this tree (*dendrogram*) and select the layer that gives you the most appropriate clustering result.

Divisive (top-down) hierarchical clustering is another form of hierarchical clustering that works in the opposite direction. Instead of starting with as many clusters as there are data points, we begin with a single cluster consisting of *all* data points and start *dividing* clusters based on the distance metric, stopping when each data point is in its own separate cluster.

Copyrighted material

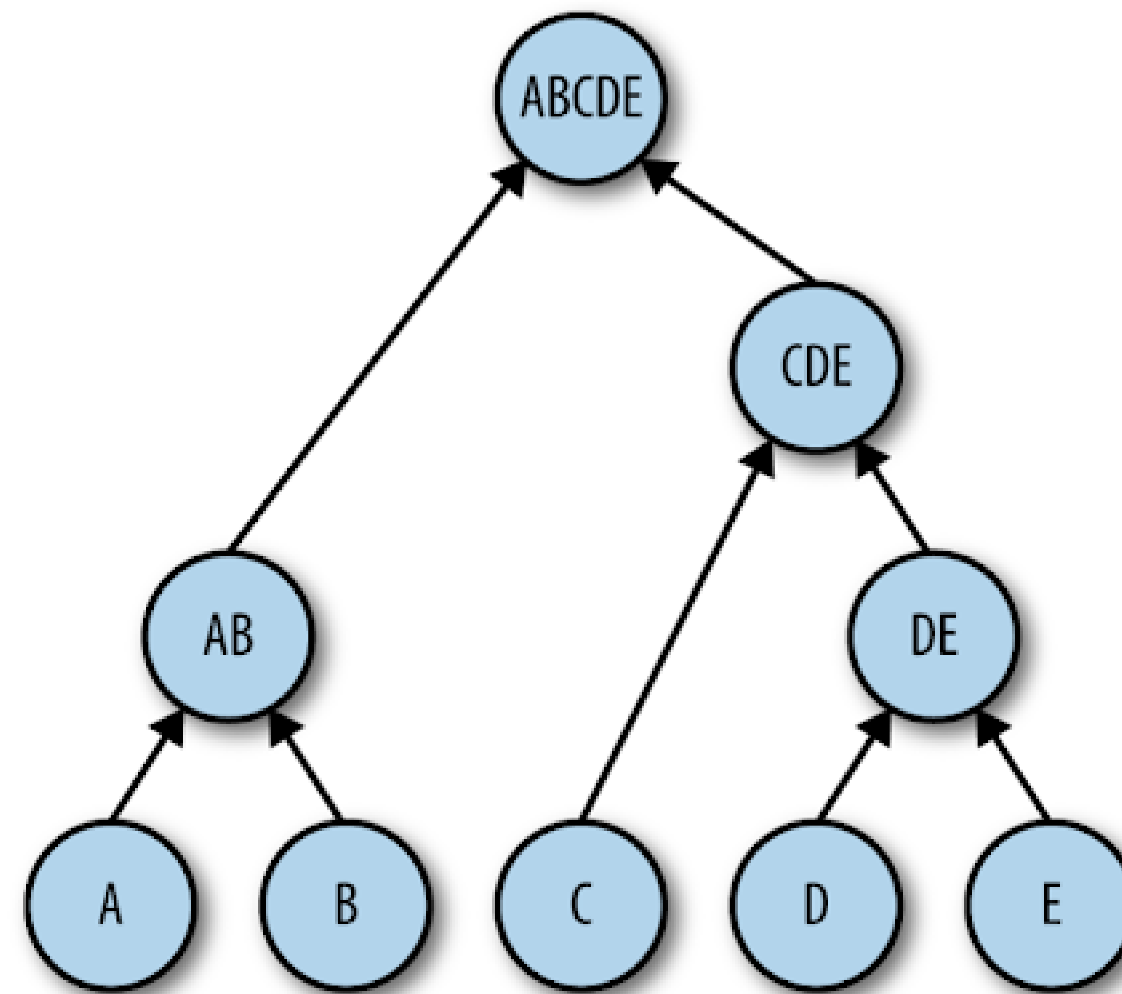


Figure 2-12. Agglomerative hierarchical clustering dendrogram

There are some important points to take note of when considering whether to use hierarchical clustering:

- Hierarchical clustering produces a dendrogram tree model, as illustrated in Figure 2-12. This model can be more complex to analyze and *takes up more storage space* than the centroids produced by *k*-means, but also conveys more information about the underlying structure of the data. If model compactness or ease of analysis is a priority, hierarchical clustering might not be your best option.
- *k*-means works with only a small selection of distance metrics (mostly Euclidean distance) and requires numerical data to work. In contrast, hierarchical clustering works with almost any kind of distance metric or similarity function, as long as it produces a result that can be numerically compared (e.g., C is more similar to A than to B). You *can use it with categorical data*, mixed type data, strings, images, and so on as long as an appropriate distance function is provided.
- Hierarchical clustering has *high time complexity*, which makes it *unsuitable for large datasets*. Taking n to be the number of data points, agglomerative hierarchical clustering has a time complexity of $O(n^2 \log(n))$, and naive divisive clustering has a time complexity of $O(2^n)$.

Locality-sensitive hashing

k-means is good for determining which items are close together when each item can be represented as a sequence of numbers (i.e., a vector in a vector space). However, many items that you would want to cluster do not easily admit such a representation. The classic example is text documents, which are of variable length and admit essentially an infinite choice of words and word orders. Another example is lists, such as

Copyrighted material

the set of IP addresses accessed by a given user, or the set of all of a user's friends in a social graph.

One very common similarity metric for unordered sets is *Jaccard similarity*. Jaccard similarity is defined to be the proportion of common items between two sets, out of all the items in the two sets. More precisely, for two sets X and Y, the Jaccard similarity is defined as follows:

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

To generate clusters when your items are sets, all you need to do is find the groups of items whose Jaccard similarities are very high. The problem here is that this computation is quadratic in the number of items—so as your dataset grows, finding the clusters will quickly become impossible. *Locality-sensitive hashing* (LSH) attempts to solve this problem. LSH is not normally considered a clustering algorithm, but you can use it as a method for grouping similar items together according to some notion of "distance," effectively achieving a similar effect to other more typical clustering algorithms.

If the items you want to cluster are not unordered sets (e.g., a text document), the first step is to convert them into sets. For text documents, the most straightforward conversion is into a *bag of words*—simply, a list of all the words that are included in the document. Depending on your implementation, repeated words might or might not be included multiple times in your list, and/or "stopwords" such as "a," "the," and "of" might be excluded.

However, the bag-of-words conversion loses an important aspect of a text document, namely the ordering of the words. To retain information about the ordering, we generalize the conversion into *shingling*: taking our list to be (overlapping) sequences of consecutive words in the document. For example, if the document was "the quick brown fox jumps over the lazy dog," the three-word shingles would be:

> {(the, quick, brown), (quick, brown, fox), (brown, fox, jumps), (fox, jumps, over), (jumps, over, the), (over, the lazy), (the, lazy, dog)}

You also can perform shingling at the character level, which can be useful for short documents or text strings that can't be parsed into words.

Now, given a dataset consisting of unordered sets, the question is how to efficiently determine which ones are similar to one another according to the Jaccard metric. The first step is to convert each set into a short "signature," in such a way that documents that are similar have similar signatures. The standard algorithm for this task is MinHash, which works as follows:

Copyrighted material