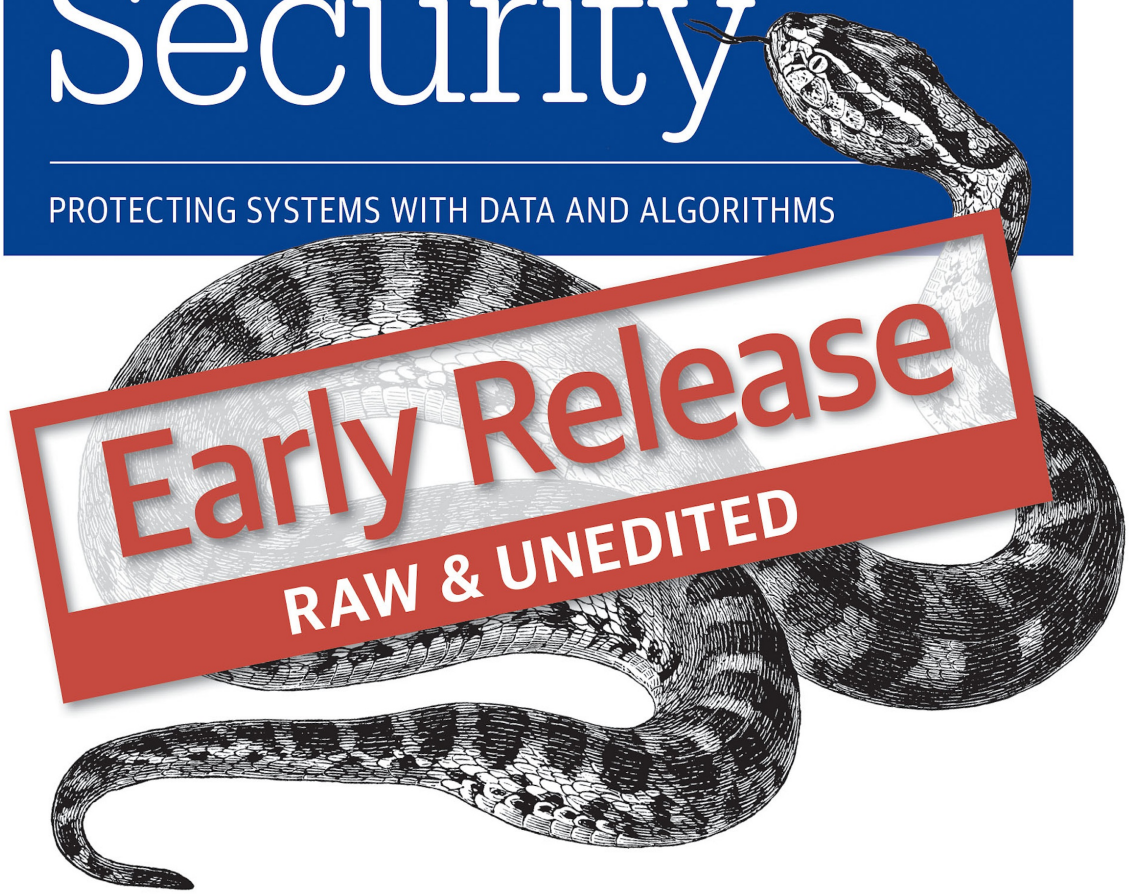
O'REILLY®

Machine Learning & Security

PROTECTING SYSTEMS WITH DATA AND ALGORITHMS



Early Release

RAW & UNEDITED

Clarence Chio & David Freeman

Machine Learning and Security

Clarence Chio and David Freeman

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North,

Sebastopol, CA 95472.

ISBN-13: 9781491979907

6/21/17

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.[1]

In the intervening forty 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. Leveraging email data, content-based models can be built

to create a generalizable approach to recognize spam. Metadata and entity reputations can be extracted from email 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 86% of all emails sent today are spam (according to one study),[2] the best spam filters today block more than 99.9% of all spam,[3] and it is a rarity for users of major email services 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.[4]

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 takes turns at fixing the flaws in design or technique that the other has uncovered. 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 base 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 will demonstrate applications of machine learning and data analysis techniques to various problem domains in security and abuse. We will 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 to leave you not with the answer to every security problem you might face, but to give you a framework for thinking about data and security, and 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 face modern computer and network systems, 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, as above, gives a concrete example of applying machine learning to security that can be generalized to nearly any domain.

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 (fig 1),[5] 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 future chapters.

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 for detection avoidance.

Spyware

Malware installed on a computer system without permission and/or knowledge by the operator, with 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 given up.

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.)

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 knowledge of operators.

Keylogger

A piece of hardware or software that (often covertly) records the keys pressed

on a keyboard or similar input 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, etc. Typically the goal of a login attack, but also can be small scale and highly targeted (e.g. spyware, social engineering).

Phishing (a.k.a. Masquerading)

Communications with a human that pretend to be reputable entities or persons in order to induce the revelation of personal information or the obtaining of private assets.

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 from a human being using non-technical methods such as lying, trickery, bribery, blackmail, etc.

Incendiary speech

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

Denial of Service (DoS) and Distributed Denial of Service (DDoS)

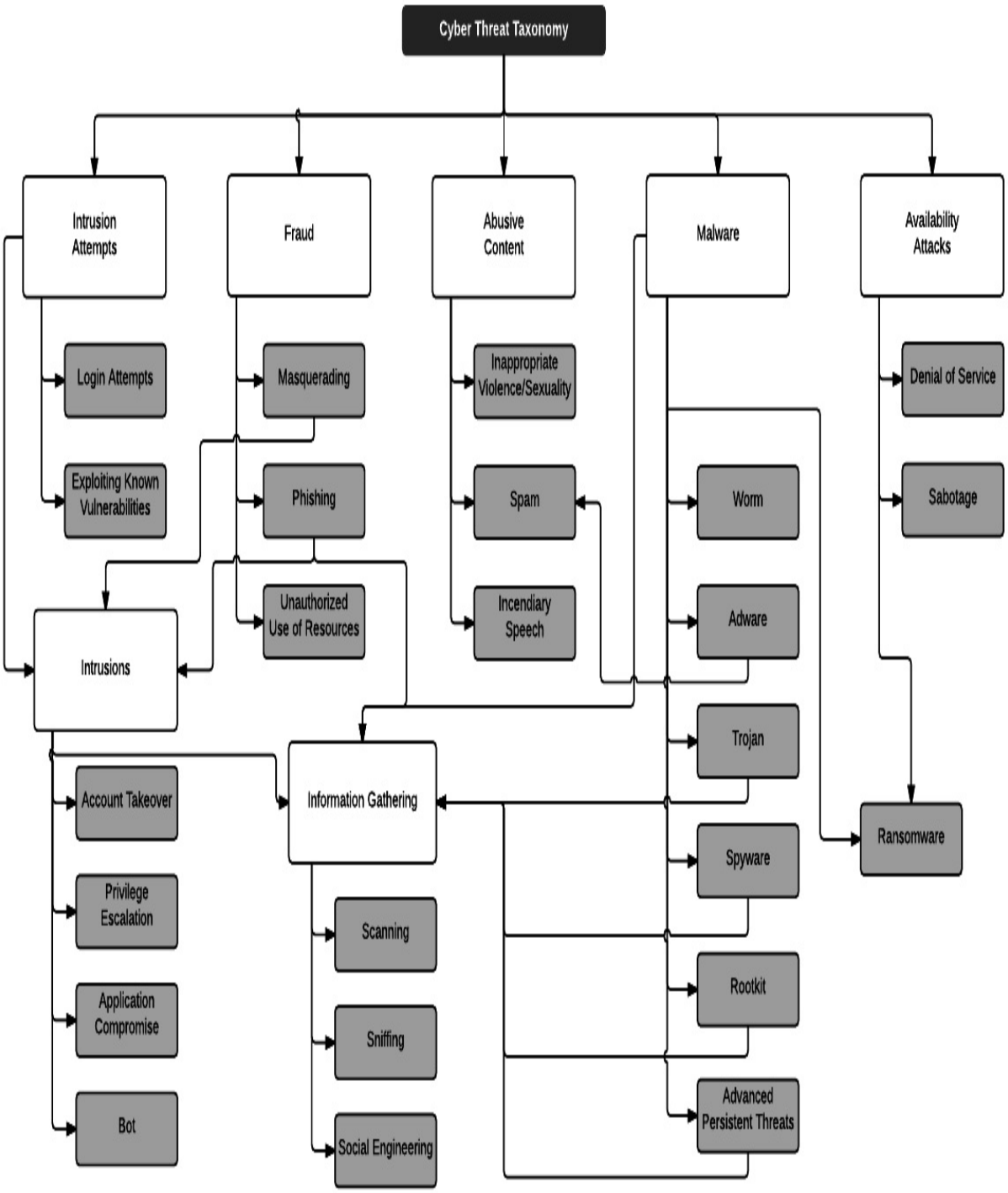
Attacks on the availability of systems through high-volume bombardment and/or malformed requests, often also breaking down system integrity and reliability.

Advanced Persistent Threats (APT)

A highly targeted network 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.



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. Because of the higher stakes at play, these institutions often have more advanced defense mechanisms in place, making the life 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 a more relaxed security measures in place, abusing systems that are open by design, and resorting to more indirect techniques that would eventually still allow them to monetize.

A marketplace to sell 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.[6] 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 in the late 1840s, merchants 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 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 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.[7] 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 (SSNs), email accounts, phone numbers, addresses, and other private information can be sold online to criminals looking to perform identity theft, fake account creation, fraud, etc. In particular, the economy for credit cards and the path to monetization once you have a victim's credit card number is 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 the cashing out of 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 seen as a powerful enabler for technically skilled people, especially those in less wealthy nations and 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?

Artificial intelligence is something that people have fantasized about since the dawn of the technological age. For an autonomous entity to make correct decisions without being explicitly instructed how to do so, to draw

generalizations and distill concepts from complex information sets -- that is the embodiment of intelligence.

Machine learning refers to one aspect of artificial intelligence --- 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 the core, it 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).

Let's say we wish to classify a group of animals into mammals and reptiles. With a supervised method, we would have a set of animals for which we are definitively told their category (e.g. we are given that the dog and elephant are mammals and the alligator and iguana are reptiles). We then try to extract out 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 elephant both give birth to live offspring, unlike most reptiles. The binary property "gives birth to live offspring" is what we call a *feature*, a useful abstraction for observed properties so there can be normalized grounds on which we can perform comparisons between different observations. After extracting a set of features that might help differentiate mammals and reptiles in the labeled data, we can then run the 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 has to classify it as either a mammal or 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 would be 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 has to group the different sets of data points in a way that would result in a binary classification. Seeing that most 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 will 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.

Adversaries using Artificial Intelligence

Note that nothing prevents adversaries from leveraging artificial intelligence to avoid detection and get past 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 to their payloads to circumvent detection,

probing spam filters by performing A/B tests on email content and learning what causes their click-through rates to rise and fall. Machine learning is used by both the good guys and bad guys in fuzzing campaigns to speed up the process of finding vulnerabilities in software.[8] 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.[9]

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 exploit various algorithms to make erroneous predictions or learn the wrong thing.[10] In this growing field of study named *Adversarial Machine Learning*, attackers with varying degrees of access to a machine learning system can execute a range of attacks to achieve their ends. A later chapter of this book is dedicated to this topic, and will paint 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 security machine learning systems.

Real world uses of machine learning in security

In this book, we will explore a range of different computer security applications that machine learning has shown promising results in. Applying machine learning and data science to solve problems is not a straightforward task. While convenient programming libraries remove some complexity from the equation, developers still have 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 the data science techniques discussed can also be found at the core of many computer systems that we may interact with on a daily basis.

Machine learning's use cases in security can be classified 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 face 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 covers a statistical majority of the training dataset. Thereafter, deviations from this normality of any sort will be detected as anomalies.

It is natural to conflate the pattern recognition and anomaly detection, since one may 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 have to be strictly derived from the observed data used to train the algorithm. On the other hand, in anomaly detection 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 a classic example of pattern recognition, since spam typically has a largely predictable set of characteristics, and an algorithm can be trained to recognize those characteristics as a pattern to

classify emails with. 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, then we have succeeded. In actuality, spam detection may not be suitable for the anomaly detection paradigm, since it is not difficult to convince yourself that it is in most contexts easier to find similarities in spam than in 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 have optimizations that makes fuzzing much more efficient than blind iteration.[11] 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. In cases where the threat model is clearly known, it may be more suitable to approach the problem through the lens of pattern recognition. In other cases, anomaly detection may be the answer. In many cases, a system may make use of both approaches to achieve better coverage. Network outlier detection is the classic example of anomaly detection, since 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 mercy of rigid and unforgiving access control policies.[12] Through a combination of unsupervised learning and anomaly detection, such systems can infer information access patterns for certain roles in an organization (or individual users) 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 may 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 will dive deeper into a selection of these real-world applications. We will then 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 above, 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 were asked to solve the problem of rampant email spam affecting employees in an organization. For whatever reason, you were 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 have to spend too much time cleaning the data.[13]

Human beings do a good job at recognizing spam, so you start out by implementing a simple solution that approximates a person's thought process at this task. The theory you have is that the presence or absence of some prominent keywords in the 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 training set 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 if 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*.[14] This is a lightly-cleaned raw email message corpus containing 75,419 messages collected from an email server over a 3 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.[15]

For evaluating how well different approaches work, we go through a simple validation process.[16] We split the dataset into non-overlapping training and testing sets, where the training set consists 70% of the data, (an arbitrarily chosen proportion) and the testing 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*[17] (NLTK) to remove morphological affixes from words for more flexible matching. For instance, this would reduce the words "congratulations" and "congrats" to the same stem word, "congrat". We also remove *stop words* (e.g. "the", "is", "are," etc.) before the token extraction process because they typically do not contain much meaning. We first define a pair of functions to help with loading and preprocessing the data and labels:[18]

```
import string

import email

import nltk


punctuations = list(string.punctuation)

stopwords = set(nltk.corpus.stopwords.words('english'))

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

def extract_email_text(path):
    # Load a single email from an input file.
    with open(path) as f:
        msg = email.message_from_file(f)
    if not msg:
        return ""

    # Read the email subject.
    subject = msg['Subject']
```

```
    if subject:
        subject = subject.decode(encoding='utf-8', errors='ignore')
    else:
        subject = ""

    # Read the email body.
    body = ' '.join(m for m in flatten_to_string(msg.get_payload()) if type(m)
== str)
    if body:
        body = body.decode(encoding='utf-8', errors='ignore')
    else:
        body = ""

    return subject + ' ' + body

def load_message(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]


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

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:

```
import os


DATA_DIR = 'trec07p/data'
LABELS_FILE = '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 train 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


blacklist = spam_words - ham_words
```

Upon inspection of the tokens in blacklist, you will realize that many of the words may seem nonsensical. (i.e. unicode, URLs, filenames, symbols, foreign words) This problem can be remedied with a more thorough data

cleaning process, but these simple results should perform adequately for the purposes of this experiment:

viagra, congrat, pill, greenback, reward, enlarge, nigeria, …

Evaluating your methodology on the 22,626 emails in the testing set, you realise that this simplistic algorithm does not do as well as you hoped. We report the results in a *confusion matrix,* a 2 x 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	5175	1368
Actual SPAM	5280	9493

True positive: actual spam -> predicted spam

True negative: actual ham -> predicted ham

False positive: actual ham -> predicted spam

False positive: actual spam -> predicted ham

Converting this to percentages we get:

	Predicted HAM	Predicted SPAM
Actual HAM	22.9%	6.0%
Actual SPAM	23.3%	42.0%

Classification accuracy: 64.9%

Ignoring the fact that 5.8% of emails did not get classified because of preprocessing errors, we see that the performance of this naive algorithm is actually quite fair. Our spam blacklist technique has a 64.9% classification accuracy (i.e., total proportion of correct labels). However, the blacklist doesn't include many words that spam emails use, since 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 produce 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 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*, may allow you to find approximate matches of emails that have been marked as spam.

Upon doing some research, you come across *datasketch,*[19] a comprehensive Python package that has efficient implementations of the *MinHash* + LSH algorithm[20] 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 tradeoff:

```
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) < 2: continue
    for s in stems:
        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:

	Predicted HAM	Predicted SPAM
Actual HAM	6468	58
Actual SPAM	5353	9247

Converting this to percentages,

	Predicted HAM	Predicted SPAM
Actual HAM	30.6%	0.2%
Actual SPAM	25.3%	43.7%

Classification accuracy: 74.4%

That's almost 10% better than your previous naive blacklisting approach, and significantly better with respect to false positives (i.e., actual ham, predicted spam). However these results are still not quite in the same league as modern spam filters. Digging into the data, you realize that it may not be an issue with the algorithm, but with the nature of 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 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 seen obtained promising results using this technique called *Naive Bayes classification*. After getting a decent understanding of how the algorithm works, you start to prototype a solution. *Scikit-learn* provides a surprisingly simple class, sklearn.naive_bayes.MultinomialNB,[21] that you can use to generate quick results for this experiment. We can reuse a lot of the code that we wrote above for reading and preprocessing the labels into tokens, but we still need to do some further processing of the tokens to 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. *Bag-of-words* first goes through the entire corpus of tokens and generates a vocabulary of tokens used throughout the corpus. Every word in the vocabulary comprises a feature, and each feature value is the count of how many times the word appears the message. For example, consider a hypothetical scenario in which we only have 3 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 discards seemingly important information like the order of words, content structure, and word similarities, it is very simple to generate using the sklearn.feature_extraction.CountVectorizer class.[22] Let’s go ahead and implement this:

```
from sklearn.feature_extraction.text import CountVectorizer

from sklearn.cross_validation import train_test_split


# Loading messages

message_stems = [load_message(os.path.join(DATA_DIR, f) for f in filelist]


vectorizer = CountVectorizer()

X = vectorizer.fit_transform(message_stems)

y = np.array([labels[f] for f in filelist])
```

```
X_train, X_test, y_train, y_test
    = train_test_split(X, y, train_size=TRAINING_SET_RATIO)
```

You can also try using the TF/IDF metric (term frequency/inverse document frequency) 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.

Then, we can train and test our Multinomial Naive Bayes classifier.

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

# Initialize the classifier and make label predictions.
classifier = MultinomialNB()
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)

# Print results.
print(classification_report(y_test, y_pred, target_names=['Spam', 'Ham']))
```

```
print("Accuracy {:.3f}".format(accuracy_score(y_test, y_pred)))
```

```
>              precision    recall  f1-score   support
>
>       Spam       0.99      0.95      0.97     15068
>        Ham       0.91      0.98      0.94      7558
>
> avg / total       0.96      0.96      0.96     22626

> Accuracy: 0.962
```

An accuracy of 96.2% - a whopping 21.8% better than the LSH approach![23] That's not a bad result for a few lines of code, and 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 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 may thus not contain as much incriminating text as other kinds

of spam. Some kinds of spam may then try to circumvent link-spam detection classifiers using complex methods like cloaking and redirection chains. Other kinds of spam may 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 may act to avoid detection. This is nothing out of the ordinary for problems in security. You have to constantly improve your detection algorithms and classifiers and stay one step ahead of your adversaries.

In the following chapters, we will 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.

What not to expect from 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 often other factors to optimize for than precision, recall, or accuracy.

For instance, explainability of classification results may be more important in some applications compared to others. It may be considerably more difficult to extract the reasons for a decision made by a machine learning system compared to a simple rule. Some machine learning systems may also be significantly more resource intensive than other alternatives, which may be a deal-breaker 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. While machine learning methods contribute to the notion of artificial intelligence, their capabilities 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. For instance, take the initial blacklisted-words approach that we used for spam filtering as an example. When a person evaluates the content of an email, her 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 may result in it being a reasonable inclusion in non-spam email. Synonyms of blacklisted words that are used in future emails by the spammers may 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.

While your machine-learned model may work perfectly on a training set, you may find that it performs badly on a testing set. A common reason may be 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 may 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 artificial intelligence.

Chapter 3: Anomaly Detection

This chapter is about detecting unexpected events, or "anomalies," in systems. In the context of network and server security, anomaly detection refers to identifying unexpected intruders or breaches. On average it takes tens of days for a system breach to be detected.[24] Once an attacker gets in, however, the damage is usually done in a few days or less. Whether the nature of the attack is data exfiltration, extortion through ransomware, adware, or advanced persistent threats (APT), it is clear that time is not on the defender's side.

The importance of anomaly detection is not confined to the context of security. In a more general context, anomaly detection is any method for finding events that don't conform to an expectation. In instances where system reliability is of critical importance, anomaly detection can be used to detect early signs of system failure, triggering early or preventative investigations by operators. For example, if early anomalies in an electrical power grid can be found and remedied, the power company can potentially avoid expensive damage that occurs when a power surge causes outages in other system components. Another important application of anomaly detection is in the field of fraud detection. Fraud in the financial industry can often be fished out a vast pool of legitimate transactions by studying patterns of normal events and detecting when deviations occur.

```**sidenote**

*Terminology*

Throughout the course of this chapter, we use the terms outlier and anomaly interchangeably. There is an important distinction between *outlier detection* and *novelty detection*. The task of novelty detection involves learning a representation of "regular" data using data that does not contain any outliers, whereas the task of outlier detection involves learning from data that contains both regular data *and* outliers. The importance of this distinction will be discussed later in the chapter. Both novelty detection and outlier detection are forms of anomaly detection.

We refer to non-anomalous data points as *regular* data. Do not confuse this with any references made to *normal* or *standard* data. The term "*normal*" used in this chapter refers to its meaning in statistics, i.e. a Normal (Gaussian) distribution. The term "*standard*" is also used in the statistical context, referring to a Normal distribution with 0 mean and unit variance.

```

A *time series* is a sequence of data points of an event or process observed at successive points in time. These data points, often collected at regular intervals, constitute a sequence of discrete metrics which characterize changes in the series as time progresses. For example, a stock chart depicts the time series corresponding to the value of the given stock over time. In the same vein, the length of bash commands entered into a command-line shell can also form a time series. In this case, the data points are not likely to be equally spaced in time. Instead, the series is event-driven, where each event is an executed command in the shell. Still, we will consider such a data stream as a time series since each data point is associated with the time of a corresponding event occurrence.

The study of anomaly detection is closely coupled with the concept of time series modeling because an anomaly is often defined as a deviation from what is normal or expected, given what had been observed in the past. Studying anomalies in the context of time thus makes a lot of sense. In the following pages, we will look at what anomaly detection is, examine the process of generating a time series, and discuss the techniques used to identify anomalies from a stream of data.

When to use anomaly detection vs. supervised learning

As discussed in Chapter 1, anomaly detection is often conflated with supervised learning, and it is sometimes unclear which approach to take when looking to develop a solution for a problem. For example, if you were looking for fraudulent credit card transactions, it may make sense to use a supervised learning model if you have a large number of both legitimate and fraudulent transactions to train your model on. Supervised learning would be especially suited for the problem if you expect future instances of fraud to look similar to the examples of fraud you have in your training set. Credit card companies sometimes look for specific patterns that could suggest fraud: for example, large purchases made after small purchases, purchases from an unusual location, or purchases of a product that doesn't fit the customer's spending profile. These patterns can be extracted from a body of positive training examples to be used in supervised learning.

In many other scenarios, it can be hard to find a representative pool of positive examples for the algorithm to get a sense of what positive events are like. Server breaches are sometimes caused by zero-day attacks or newly released vulnerabilities in software. By definition, the method of intrusion cannot be predicted in advance, and it is difficult to build a profile of every possible method of intrusion in a system. Because these events are relatively rare, it also contributes to the class imbalance problem that makes for difficult application of supervised learning. Anomaly detection is perfect for

such problems.

Intrusion detection with heuristics

Intrusion detection systems (IDS) have been around since 1986[25] and are commonplace in security-constrained environments. Even today, thresholds, heuristics, and simple statistical profiles remain a reliable way of detecting intrusions and anomalies. For example, suppose we define 10 queries per hour to be the upper limit of normal use for a certain database. Each time the database is queried, we invoke a function is_anomaly(user) with the user's ID as an argument. If the user queries the database for the eleventh time in an hour, the function will indicate that access as an anomaly.

While threshold-based anomaly detection logic is easy to implement, some questions quickly arise. How do we set the threshold? Could some users have a higher threshold requirement than others? Could there be times when users legitimately need to access the database more often? How frequently do you need to update the threshold? Could an attacker exfiltrate data by taking over many user accounts, thus requiring a smaller number of accesses per account? We shall soon see that using machine learning can help you to avoid having to come up with answers to all of these questions, instead letting the data define the solution to the problem.

The first thing we might want to try to make our detection more robust is to replace the hard-coded threshold of 10 queries per hour with a threshold dynamically generated from the data. For example, we could compute a moving average of the number of queries per user per day, and every time the average is updated set the hourly threshold to be a fixed multiple of the daily average. (A reasonable multiple might be 5/24, i.e., with the hourly threshold set to 5 times the hourly average.)

Further improvements can also be made:

- ● Since data analysts will likely need to query customer data with a greater frequency than receptionists, we could classify users by roles and set different query thresholds for each role.
- ● Instead of updating the threshold with averages that can be easily manipulated, we can use other statistical properties of the dataset. For example, if we use the median or interquartile ranges then thresholds will be more resistant to outliers and deliberate attempts to tamper with the integrity of the system.

The method above uses simple statistical accumulation to avoid having to manually define a threshold, but still retains characteristics of the heuristic method, including having to decide on arbitrary parameters such as avg_multiplier. However, in this adaptive-threshold solution, we begin to see the roots of machine learning anomaly detectors. The query_threshold is reminiscent of a model parameter extracted from a dataset of regular events, and the hourly threshold update cycle is the continuous training process necessary for the system to adapt to changing user requirements.

Still, it is easy to see the flaws in a system like this. In an artificially simple environment as described, maintaining a single threshold parameter that learns from a single feature in the system (query counts per user, per hour) is an acceptable solution. However, in even slightly more complex systems, the number of thresholds to compute can quickly get out of hand. There may even be common scenarios where anomalies are not triggered by a single threshold, but by a combination of different thresholds selected differently in different scenarios. In some situations, it may even be inappropriate to use a deterministic set of conditions. If user A makes 11 queries in the hour and user B makes 99 queries in the hour, shouldn’t we assign a higher risk score to B than to A? A probabilistic approach may make more sense, and allow us to estimate the likelihood that an event is anomalous instead of making binary decisions.

```Snort is a popular open source intrusion detection system that sniffs packets and network traffic for real time anomaly detection`[26]``

Data-driven methods

Before starting to explore alternative solutions for anomaly detection, it is important that we define a set of objectives for an optimal anomaly detection system:

1. 1. Low false positives and false negatives

The term *anomaly* suggests an event that stands out from the rest. By that reasoning, it may seem counterintuitive to suggest that finding anomalies is often akin to locating a white rabbit in a snowstorm. Because of the difficulty of reliably defining normality with a descriptive feature set, anomalies raised by systems can sometimes be fraught with false alarms (false positives) or missed alerts (false negatives).

False negatives occur when the system does not find something that the users intend it to find, immediately degrading the reliability of the system. Imagine you installed a new deadbolt lock on your front door, and it only manages to thwart 9 out of 10 lockpick attempts. How would you feel about the effectiveness of the lock? Conversely, false positives occur when the system erroneously recognizes normal events as anomalous ones. If you tried unlocking the bolt with the key and it refuses to let you in, thinking that you are an intruder, that is a case of a false positive.
```

False positives may seem benign, and having an aggressive detection system that “plays it safe” and raises alerts on even the slightest suspicion of anomalies may not seem like a bad option. However, every alert has a cost associated with it, and every false alarm wastes precious time of human analysts that have to investigate it. Spurious false alarm rates can rapidly degrade the integrity of the system, and analysts will no longer treat anomaly alerts as events that they would carefully investigate and respond to. An optimal anomaly detector would accurately find all anomalies with no false positives.

1. 2. Easy to configure, tune, and maintain

As seen in earlier sections of this chapter, configuring anomaly detection systems can be a non-trivial task. Inadequate configuration of threshold-based systems directly causes false positives or negatives. Once there are more than a handful of parameters to tune, you lose the attention of users, who will often fall back to default values (if available) or random values. System usability is greatly affected by the ease of initial configuration and long-term maintenance. A machine-learning anomaly detector that has been sitting in your network for a long period of time may start producing a high rate of false alarms, causing an operator to have to dive in to investigate. An optimal anomaly detector should provide a clear picture of how changing system parameters will directly cause a change in the quality, quantity, and nature of alert outputs.

1. 3. Adapts to changing trends in the data

Seasonality is the tendency of data to show regular patterns due to natural

cycles of user activity (e.g. low activity on weekends). Seasonality needs to be taken care of in all time series pattern recognition systems, and anomaly detectors are no exception. Different datasets have different characteristics, but many exhibit some type of seasonality across varying periodicities. For example, web traffic that originates from a dominant timezone will have a diurnal pattern that peaks in the day and troughs in the night. Most websites see higher traffic on weekdays compared to weekends, while other sites see the opposite trend. Some seasonality trends play out over longer periods. Online shopping websites expect a spike in traffic every year during the peak shopping seasons, while traffic to the United States Internal Revenue Service (IRS) website builds up between January and April, then drops drastically afterwards.

Anomaly detection algorithms that do not have a mechanism for capturing seasonality will suffer high false positive rates when these trends are observed to be different from previous data. Organic *drift* in the data caused by viral promotions or more a gentle uptick in popularity of certain entities can also cause anomaly detectors to raise alerts for events that do not require human intervention. An ideal anomaly detection system would be able to identify and learn seasonality trends in the data and filter them out from the actual outliers.

1. 4. Works well across datasets of different nature

Even though the Gaussian distribution dominates many areas of statistics, it does not mean that all datasets have a Gaussian distribution. Density estimation is a central concept in modeling normality for anomaly detection, but there are other *kernels*[27] that may be more suitable for modeling the distribution of your dataset. For example, some datasets may be better fitted with the exponential, tophat, cosine, or Epanechnikov, kernels.[28] An ideal anomaly detection system should not make assumptions about the data, and should work well across data with different properties.

1. 5. Resource efficient and suitable for real-time application

Especially in the context of security, anomaly detection is often a time-sensitive task. Operators ideally want to be alerted of potential breaches or system failures within minutes of suspicious signals. Every second counts when dealing with an attacker that is actively exploiting a system. Hence, these anomaly detection systems have to run in a streaming fashion, consuming data and generating insights with minimal latency. This requirement rules out some slow and/or resource-intensive techniques.

1. 6. Explainable alerts

Auditing alerts raised by an anomaly detector is important for evaluating the system, as well as investigating false positives and negatives. Alerts that come from a static threshold-based anomaly detector can easily be audited. Simply running the event through the rule engine again will indicate exactly which conditions triggered the alert. For adaptive systems and machine-learning anomaly detectors the problem is more complex. When there is no explicit decision boundary for the parameters within the system, it can sometimes be hard to point to a specific property of the event that triggered an alert. This makes it difficult to debug and tune systems, and leads to lower confidence with the decisions made by the detection engine. The explainability problem is somewhat of an open research topic in the field of machine learning, and is not exclusive to the anomaly detection paradigm. However, when alerts have to be audited in a time-pressured environment, having clear explanations can make for a much easier decision making process by the human or machine components that react to anomaly alerts.

Feature engineering for anomaly detection

Just like any other task in machine learning, selecting good features for anomaly detection is of paramount importance. Most online (streaming) anomaly detection algorithms require input in the form of one-dimensional time series data. If your data source already spits out metrics in this form, you may not have to do any further feature engineering. For example, to detect when a system process has an abnormally high CPU utilization, all you will need is the *cpu utilization* metric, which can be extracted from most basic system monitoring modules. However, many use cases will require you to generate your own one-dimensional data streams on which to apply anomaly detection algorithms.

In this section, we will focus our feature engineering discussions on three domains: *server intrusion detection*, *network intrusion detection, and web application intrusion detection*. There are notable differences between the three, and each requires a set of unique considerations that are specific to its particular space. We will look at examples of tools that we can use to extract these features, and evaluate the pros and cons of the different methods of feature extraction.

Of course, anomaly detection is not restricted to servers and networks only. Other use cases such as fraud detection and detecting anomalies on calls made to a public API service also rely on good feature extraction to achieve a reliable data source to apply algorithms on. Once we have learned the principles of extracting useful features and time series data from the server and network domains, it will be your job to generalize these principles across more specific application domains.

Server intrusion detection

When developing an intrusion detection agent for servers (sometimes also called *host intrusion detection systems*), you will likely need to generate your own metrics and may even want to perform correlations of signals collected from different sources. Relevancy of metrics vary widely depending on the threat model, but basic system and network level statistics make for a good starting point. Collection of these system metrics can be done in a variety of ways and there is a diversity of tools and frameworks that can help you out with the task. *Osquery*[29] is a popular operating system instrumentation framework that collects and exposes low-level operating system metrics, making them available for querying through a SQL-based interface. Making scheduled queries through *osquery* can allow you to establish a baseline of server and application behavior, thereby allowing the intrusion detector to identify suspicious events that occur unexpectedly.

Malware is the most dominant threat vector for servers in many environments. Of course, malware detection and analysis warrants its own full chapter, and we will go into further detail in a later section of the book. For now, we will base our analysis on the assumption that most malware affects system-level actions, and the malware can be detected by collecting system-level activity and looking for indicators of compromise (IOC) in the signals. An example of some common signals that you can collect are:

- ● Running processes
- ● Active/new user accounts
- ● Kernel modules loaded
- ● DNS lookups
- ● Network connections
- ● System scheduler changes
- ● Daemon/background/persistent processes
- ● Startup operations, *launchd* entries
- ● OS registry databases, *plist* files
- ● Temporary file directories

- ● Browser extensions

This list is far from exhaustive; different types of malware naturally generate different sets of behavior, but collecting a wide range of signals will ensure that you have visibility into parts of the system where the risk of compromise by malware is the highest.

Osquery

In *osquery*, you can schedule queries to be run periodically by the *osqueryd* daemon, populating tables which you can then query for later inspection. For investigative purposes, you can also run queries in an ad hoc fashion by using the command line interface, *osqueryi*. An example query that gives you a list of all users on the system is:

SELECT * FROM users;

If you wanted to locate the top 5 memory-hogging processes:

select pid, name, resident_size from processes order by resident_size desc limit 5;

While *osquery* can be used to monitor system reliability or compliance, one of its principal applications is to detect behavior on the system that could potentially be caused by intruders. A malicious binary will usually try to

reduce its footprint on a system by getting rid of any traces it leaves in the file system, for example by deleting itself after it starts execution. A common query for finding anomalous running binaries is to check for currently running processes that have a deleted executable:

```
SELECT * FROM processes WHERE on_disk = 0;
```

Let's say that this query generated some data that looks like this:

```
2017-06-04T18:24:17+00:00 []

2017-06-04T18:54:17+00:00 []

2017-06-04T19:24:17+00:00 ["/tmp/YBBHNCA8J0"]

2017-06-04T19:54:17+00:00 []
```

A very simple way to convert this into a numerical time series is to use the length of the list as a the value. It should then be clear that the third entry in the example above will register as an anomaly.

Besides tapping into system state, the *osquery* daemon can also listen in on OS-level events such as file system modifications and accesses, disk mounts, process state changes, network setting changes, and more. This allows for event-based OS introspection to monitor file system integrity and audit processes and sockets.

```**side note**

*Osquery* includes convenient *query packs* - sets of queries and metrics grouped by problem domain and use case that users can download and apply to the *osquery* daemon. For example, the *incident-response* pack exposes metrics related to the application firewall, crontab, IP forwarding, iptables, launchd, listening ports, disk mounts, open files and sockets, shell history, startup items, and more. The *osx-attacks* pack looks for specific signals exhibited by a set of common OS X malware, checking for specific plists, process names, or applications that a well-known piece of malware installs.

```

The *osqueryd* can be configured with a config file that defines what queries and packs the system should use.[30] For instance, a scheduled query that detects deleted running binaries can be configured to run every 1800 seconds with this query statement in the configuration:

```
{
 …
  // Define a schedule of queries to run periodically:
  "deleted_running_binary": {
    "query": "SELECT * FROM processes WHERE on_disk = 0;",
    "interval": 1800
  }
  …
}
```

Note that *osquery* can log query results as either snapshots or differentials. Differential logging can be useful in reducing the verbosity of received information, but can also be more complex to parse. Once these metrics are logged by the daemon, extracting time series metrics from them is then simply a matter of analyzing log files or performing more SQL queries on the generated tables.

Limitations of *osquery* for security

It is important to consider that *osquery* was not designed to operate in an untrusted environment. There is no built-in feature to obfuscate osquery operations and logs, so there is a possibility that malware can meddle with the metric collection process or osquery logs/database to hide its tracks. *Osquery* is simple to deploy on a single host, but most operationally mature organizations are likely to have multiple servers of variety of flavors deployed on a variety of environments. There is no built in capability for orchestration or central deployment and control in osquery, so some development effort needs to be made to integrate it into your organization's automation and orchestration frameworks (e.g. Chef,[31] Puppet,[32] Ansible,[33] SaltStack,[34] etc.). Third-party tools intended to make the operationalization of *osquery* easier such as *Kolide*[35] (distributed *osquery* command and control) and *doorman*[36] (*osquery* distributed fleet manager) are also growing in number.

Alternatives to *osquery*

There are many open-source and commercial alternatives to *osquery* that help you achieve the same end result — continuous and detailed introspection of

your servers. Mining the wealth of information that many Unix-based systems provide natively (e.g. in */proc*) is a lightweight solution that may be sufficient for your use-case. The *Linux Auditing System* (*auditd,* etc.) is much more mature than *osquery* and is a tool that forensics experts and operational gurus have sworn by for decades.

Network intrusion detection

Almost all malware and intrusions into servers instigate communication with the outside world. Most breaches are carried out with the objective of stealing some valuable data from the target, so detecting intrusions by focusing on the network makes a lot of sense. For botnets, remote command-and-control servers communicate with the compromised "zombie" machines to give instructions on operations to execute. For APTs, hackers can remotely access the machines through a vulnerable or misconfigured service, allowing them shell and/or root access. For adware, communication with external servers is required for downloading unsolicited ad content served. For spyware, results of the covert monitoring are often transmitted over the network to an external receiving server.

From simple protocol tapping utilities like *tcpdump*[37] to some more complex sniffing tools like *Bro*,[38] the network intrusion detection software ecosystem has many utilities and application suites that help you collect signals from network traffic of all sorts. Network intrusion detection tools operate on the basic concept of inspecting traffic that occur between servers. Just like host intrusion detection, attacks can be identified by either matching traffic to a known signature of malicious traffic, or by anomaly detection, comparing traffic to previously established baselines. Signature matching is something that we will not focus on in this section, but will talk more about in the following chapter on malware analysis.

Snort is the de facto choice for intrusion detection monitoring, providing a good balance of usability and functionality. Furthermore, it is backed by a vibrant open-source community of users and contributors who have created add-ons and GUIs for it. Snort has a relatively simple architecture, allowing users to perform real-time traffic analysis on IP networks, write rules that can be triggered by detected conditions, and compare traffic to an established baseline of the normal network communication profile.

In extracting features for network intrusion detection, there is a noteworthy difference between extracting network traffic metadata and inspecting network traffic content. The former is used in *Stateful Packet Inspection* (SPI), working at the network and transport layers (OSI layers 3 and 4),[39] examining each network packet's header and footer without touching the packet context. It maintains state on previous packets received, and hence is able to associate newly received packets with previously seen packets. SPI systems are able to know if a packet is part of a handshake to establish a new connection, a section of an existing network connection, or if it is an unexpected rogue packet. SPI systems are useful in enforcing access control — the traditional function of *network firewalls* — since they have a clear picture of the IP addresses and ports involved in correspondence. They can also be useful in detecting slightly more complex layer 3/4 attacks such as IP spoofing, TCP/IP attacks (such as ARP cache poisoning or SYN flooding), and denial-of-service (DOS) attacks.[40] However, there are obvious limitations to restricting analysis to just packet headers and footers. For example, SPI cannot detect signs of breaches or intrusions on the application level, since doing so would require a deeper level of inspection.

Deep packet inspection

Deep packet inspection (DPI) is the process of examining the data encapsulated in network packets in addition to just the headers. This allows for the collection of signals and statistics about the network correspondence

originating from the application layer. Because of this, DPI is capable of collecting signals that can help detect spam, malware, intrusions, and subtle anomalies. Real-time streaming DPI is a challenging computer science problem because of the computational requirements necessary to decrypt, disassemble, and analyze packets going through a network intersection.

Bro is one of the earliest systems that implemented a passive network monitoring framework for network intrusion detection. *Bro* consists of an efficient event engine that extracts signals from live network traffic, and a policy engine that consumes events and policy scripts and takes the relevant action in response to different observed signals.

One use of *Bro* is to detect suspicious activity in web applications by inspecting the strings present in the POST body of HTTP requests. For example, SQL injections and XSS reflection attacks can be detected by creating a profile of the POST body content for a particular web application entry point. A suspicion score can be generated by comparing the presence of certain anomalous characters (the " ' " character in the case of SQL injections, and the "<" or ">" script tag symbols in the case of XSS reflections) with the baseline, which can be valuable signals for detecting when a malicious actor is attacking your web application.[41]

The set of features to generate through DPI for anomaly detection is strongly dependent on the nature of the applications that operate inside your network as well as the threat vectors relevant to your infrastructure. If your network does not include any outward-facing web servers, using DPI to detect XSS attacks is irrelevant. If you network contains only point-of-sale systems connected to *PostgreSQL* databases storing customer data, perhaps you need to focus on unexpected network connections that could be indicative of attacker *pivoting* in your network.

``` **side note**

*Pivoting*, or *island hopping*, is a multi-layered attack used by hackers to circumvent firewall restrictions in a network. A properly configured network will not allow external accesses to a sensitive database. However, if there is a publicly accessible and vulnerable component in a network with internal access to the database, attackers can exploit that component and *hop* over to the database servers, indirectly accessing the machines. Depending on the open ports and allowed protocols between the compromised server and the target server, attackers can use different methods for pivoting. For example, the attacker may set up a proxy server on the compromised server, creating a covert tunnel between the target and the outside world.

```

If DPI is used in an environment with Transport Layer Security (TLS/SSL) in place, where the packets to be inspected are encrypted, then the application performing DPI has to terminate SSL. DPI essentially requires the anomaly detection system to operate as a man-in-the-middle, meaning that communication passing through the inspection point is no longer end-to-end secure. This may pose a security and/or performance risk to your environment, especially in cases where SSL termination and re-encryption of packets is improperly implemented. Any feature generation techniques that intercept TLS/SSL traffic should be very carefully reviewed and audited before being deployed in production.

Features for network intrusion detection

The Knowledge Discovery and Data Mining Special Interest Group (SIGKDD) from from the Association of Computing Machinery (ACM) holds the *KDD Cup* every year, posing a different challenge to participants. In 1999, the topic was “computer network intrusion detection”, where the

task was to “learn a predictive model capable of distinguishing between legitimate and illegitimate connections in a computer network”.[42] This artificial dataset is very old and has significant published flaws, but the list of derived features provided by the dataset is a good starting source of inspiration for some example features to extract for network intrusion detection in your own environments. Some research has been built on top of this to find out what the most important features in this list are,[43] which is useful to refer to when considering what types of features to generate for network anomaly and intrusion detection. Aggregating transactions by IP addresses, geolocation, netblocks (/16, /24), BFP prefixes, ASN information etc. can often be good ways to distill complex network captures and generate simple count metrics for anomaly detection.[44]

Web application intrusion detection

We saw earlier that web application attacks like XSS and SQL injections can be detected with deep network packet inspection tools such as *Bro*. Inspecting HTTP server logs can provide you with a similar level of information, and is a more direct way of obtaining features derived from web application user interactions. Standard web servers like Apache, IIS, and Nginx generate logs of a format defined and standardized in the *NCSA Common Log Format*, also called *access logs*.[45] In addition, *NCSA combined logs* and *error logs* also record information about the client’s user-agent, referral URL, and any server errors generated by requests. In these logs, each line represents a separate HTTP request made to the server, and each line is made up of tokens in a well-defined format. Here is an example of a *combined log* format that includes the requestor’s user-agent and referral URL:

123.123.123.123 - jsmith [17/Dec/2016:18:55:05 +0800] "GET /index.html HTTP/1.0" 200 2046 "http://referer.com/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.17.3) AppleWebKit/536.27.14 (KHTML, like Gecko) Chrome/55.0.2734.24 Safari/536.27.14"

Unlike deep packet inspection, the standard web access logs do not log POST body data out-of-the-box. This means that attack vectors embedded in the user input cannot be detected by inspecting standard access logs.

```**side note**

Most popular web servers provide modules and plugins that enable you to log HTTP data payloads. Apache's *mod_dumpio* module[46] logs all input received and output sent by the server to be logged. You can add the *proxy_pass* or *fastcgi_pass* directives to the Nginx configuration file to force Nginx servers populate the $request_body variable with the actual POST request body content. Microsoft provides IIS servers with the *Advanced Logging* extension[47] which can be configured to log POST data.

```

Even with the comparatively limited scope of visibility provided in standard HTTP server log files, there are still some interesting features that can be extracted:

- ● IP-level access statistics
  - ○ High frequency, periodicity, volume by a single IP or subnet is suspicious.
- ● URL string aberrations
  - ○ Self-referencing paths ("/./") or back-references ("/../") are frequently used in path-traversal attacks.
- ● Decoded URL and HTML entities, escaped characters, null-byte string termination
  - ○ These are frequently used by simple signature/rule engines to avoid detection.

- ● Unusual referer patterns
  - ○ Page accesses with an abnormal referer URL are often a signal of an unwelcome access to a HTTP endpoint
- ● Sequence of accesses to endpoints
  - ○ Out-of-order access to HTTP endpoints that do not correspond to the website's logical flow is indicative of fuzzing or malicious explorations.
  - ○ For instance, if a user's typical access to a website is a POST to "/login", then 3 successive GETs to "/a", "/b", and "/c", but a particular IP address is repeatedly making GET requests to "/b" and "/c" without a corresponding "/login" or "/a" request, that could be a sign of bot automation or manual reconnaissance activity.
- ● User-agent patterns
  - ○ Frequency analysis can be done on user-agent strings to alert on never-before-seen user-agent strings or extremely old clients (e.g. "Mosaic/0.9" user-agent from 1993) which are likely spoofed.

Web logs provide enough information to detect different kinds of attacks on web applications, including, but not limited to, the OWASP Top Ten[48] (XSS, Injection, CSRF, Insecure Direct Object References, etc.).[49]

Generating a reliable and comprehensive set of features is critical for the anomaly detection process. The goal of feature engineering is to distil complex information into a compact form that removes unnecessary information, but does not sacrifice any important characteristics of the data. These generated features will then be fed into algorithms, where the data will be consumed and used to train machine learning models. In the next section, we will see how feature sets can be converted into valuable insights that drive anomaly detection systems.

Anomaly detection with data and algorithms

After engineering a set of features and applying it to a raw event stream to generate a time series, it is time to use algorithms to generate insights from this data. Anomaly detection has had a long history of academic study, but like all other application areas in data analysis, there is no one-size-fits-all algorithm that works for all types of time series. Thus, you should expect that the process of finding the best algorithm for your particular application will be a journey of exploration and experimentation.

Before selecting an algorithm, it is important to think about the nature and quality of the data source. Whether or not the data is significantly polluted by anomalies will affect the detection methodology. As defined earlier in the chapter, if the data does not contain anomalies (or has anomalies labeled so we can remove them), we refer to the task as *novelty detection*. Otherwise, we will refer to the task as *outlier detection*. In outlier detection, the chosen algorithm needs to be insensitive to small deviations that will hurt the quality of the trained model. Often, determining which approach to take is a non-trivial decision. Cleaning a data set to remove anomalies is laborious and sometimes downright impossible. If you have no idea whether your data contains any anomalies or not, it may be best to start off assuming that it does, and iteratively move towards a better solution.

In this discussion we attempt to synthesize a large variety of anomaly detection methods[50] from literature and industry into a categorization scheme based on the fundamental principles of each algorithm. In our scheme each category contains one or more specific algorithms, and each algorithm belongs to a maximum of one category. Our categories are as follows:

1. 1. Forecasting (supervised machine learning)
2. 2. Statistical metrics
3. 3. Unsupervised machine learning

4. 4. Goodness-of-fit tests
5. 5. Density-based methods

Each category considers a different approach to the problem of finding anomalies. We will present the strengths and pitfalls of each approach, and how different datasets may be better suited for some than for others. For instance, forecasting is only suitable for one-dimensional time-series data, while density-based methods are more suitable for high-dimensional datasets.

Our survey is not meant to be comprehensive, nor is it meant to be a detailed description of each algorithm's theory and implementation. Rather, it serves to give a broad overview of some of the different options you have for implementing your own anomaly detection systems, which we hope you can then use to arrive at the optimal solution for your use case.

1. 1. Forecasting (supervised machine learning)

Forecasting is a highly intuitive way of performing anomaly detection. Taking the weather to be an example, if it had not been raining for weeks, and there is no visible sign of upcoming rain, the forecast would predict a low chance of rain in the coming days. If it does rain in the coming days, it would be a deviation from the forecast. The act of forecasting is to learn from prior data and make a prediction about the future. Any substantial deviations between the forecasts and observations can be considered anomalous.

This class of anomaly detection algorithms uses past data to predict current data, and measures how different the currently observed data is from the prediction. By this definition, forecasting lies in the realm of supervised

machine learning, since it trains a regression model of data values versus time. Because these algorithms also operate strictly within the notion of *past* and *present*, they are only suitable for single-dimension time-series datasets. Predictions made by a forecasting model will correspond to the expected value that this time series will have in the next time step, so applying forecasting on datasets other than time series data does not make sense.

Time series data is naturally suited for being represented in a line chart. Humans are adept at studying line charts, recognizing trends, and identifying anomalies, but machines have a harder time at it. A major reason for this is the noise embedded within time series data, caused either by measurement inaccuracies, sampling frequency, or other external factors associated with the nature of the data. This results in a choppy and volatile series, which can camouflage outbreaks or spikes that we are interested in identifying. In combination with seasonality and cyclic patterns that can sometimes be complex, attempting to use naive linear-fit methods to detect anomalies would likely not give you great results.

In forecasting, it is important to define the following descriptors of time series:

- **- Trends**

A long-term direction of change in the data, undisturbed by relatively small-scale volatility and perturbations. Trends are sometimes non-linear, but can typically be fitted to a low-order polynomial curve.

- **- Seasons**

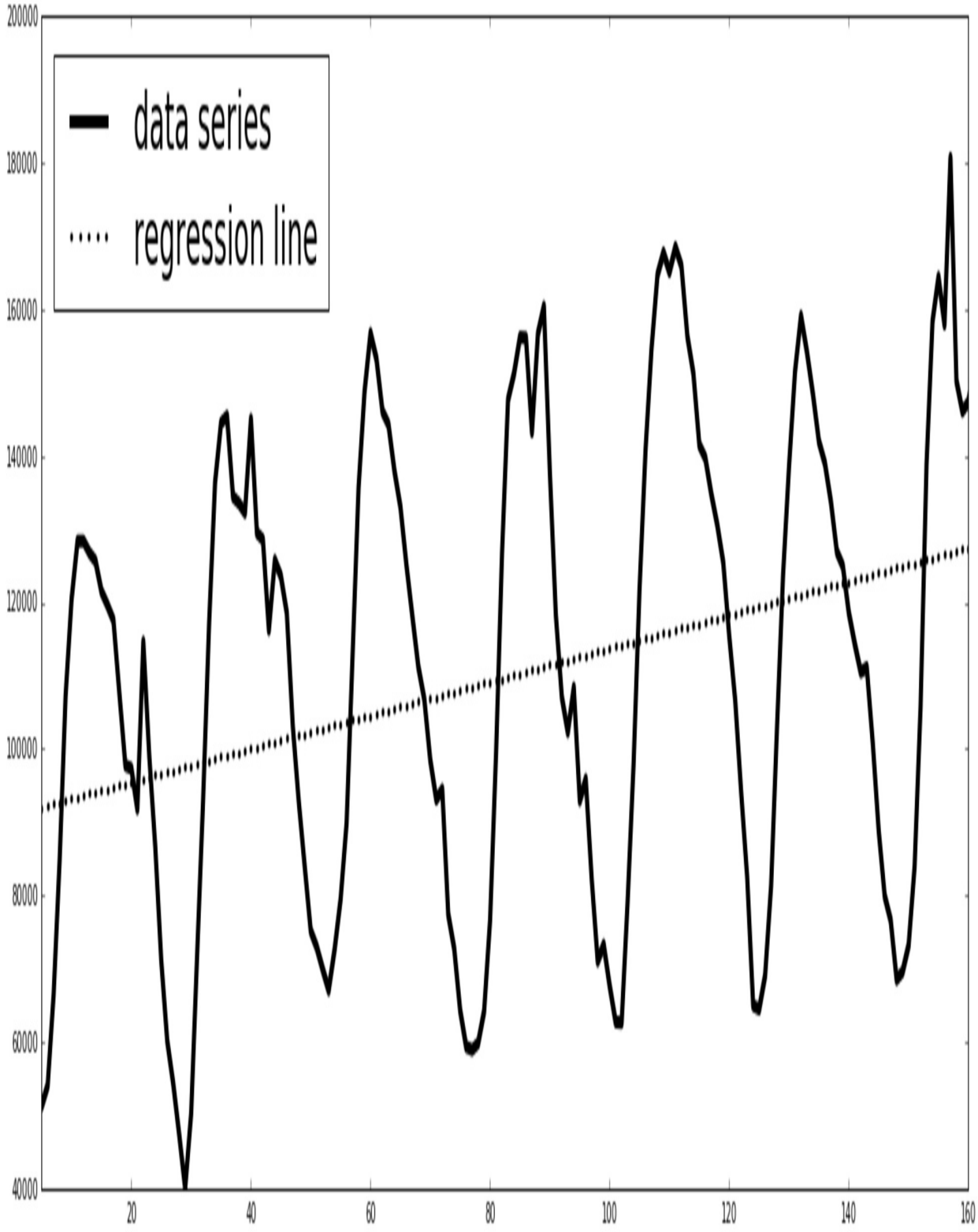
Periodic repetitions of patterns in the data, typically coinciding with factors

closely related to the nature of the data, e.g., day-night patterns, summer-winter differences, moon phases.

- **- Cycles**

General changes in the data that have pattern similarities but vary in periodicity, e.g., long-term stock market cycles.

In the following chart, observe the diurnal-like fluctuating seasonality, with a gentle upward trend illustrated by a regression line fitted to the data.

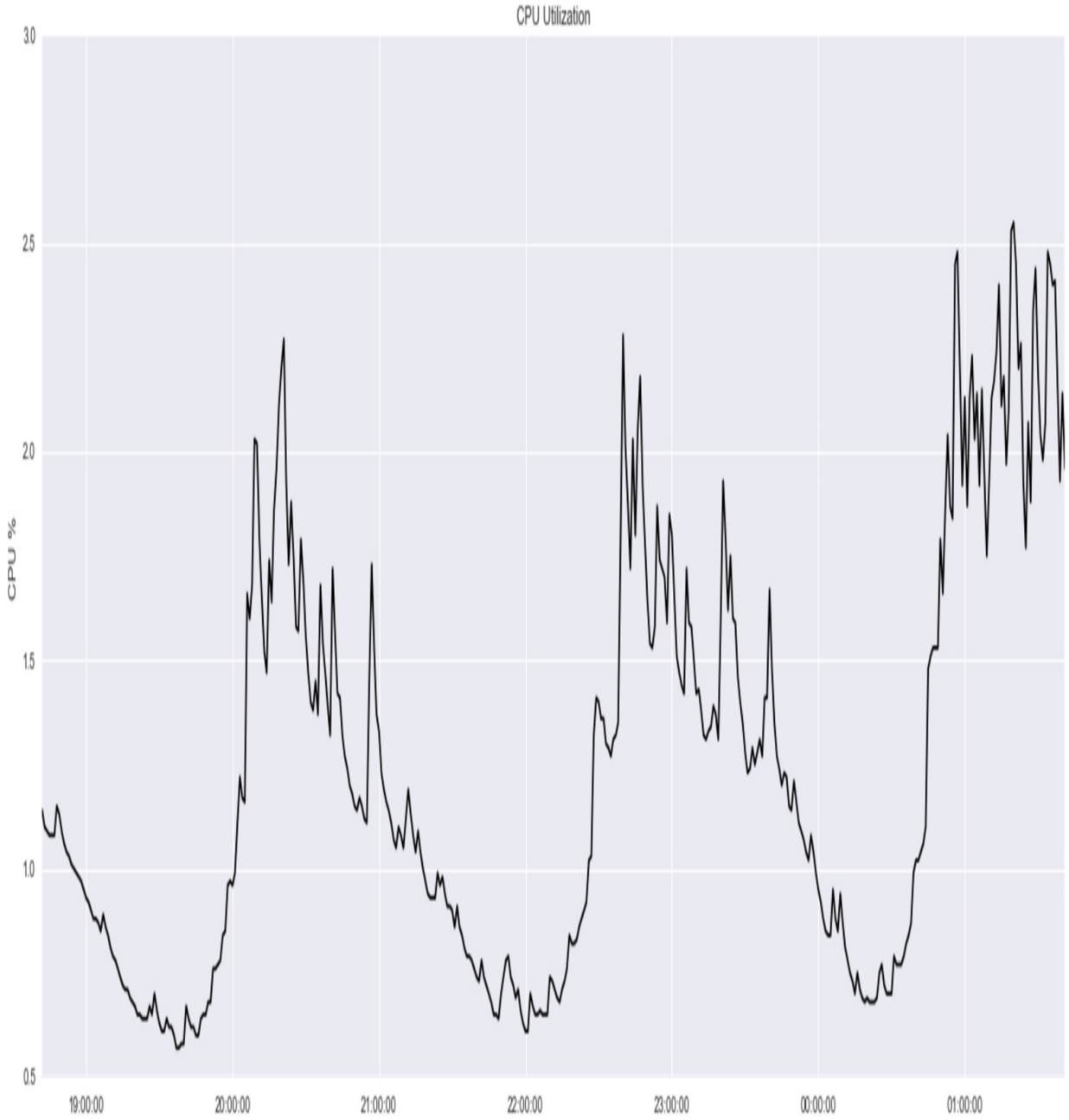

data series
regression line
200000
180000
160000
140000
120000
100000
80000
60000
40000
20
40
60
80
100
120
140
160

- - ARIMA

The *ARIMA (autoregressive integrated moving average)* family of functions is a powerful and flexible way to perform forecasting on time series. Autoregressive models are a class of statistical models that have outputs which are linearly dependent on their own previous values in combination with a stochastic factor.[51] You may have heard of exponential smoothing, which can often be equated/approximated to special cases of *ARIMA* (e.g. Holt-Winters Exponential Smoothing). These operations smoothen jagged line charts, using different variants of weighted moving averages to normalize the data. Seasonal variants of these operations can take periodic patterns into account, helping make more accurate forecasts. For instance, *seasonal ARIMA (SARIMA)* defines both a seasonal and non-seasonal component of the *ARIMA* model, allowing periodic characteristics to be captured.[52]

In choosing an appropriate forecasting model, always visualize your data to identify trends, seasonalities, and cycles. If seasonality is a strong characteristic of the series, consider models with seasonal adjustments such as *SARIMA* and *seasonal Holt-Winters* methods. Forecasting methods learn characteristics of the time series by looking at previous points and making predictions about the future. In exploring the data, a useful metric to learn is the *autocorrelation*, which is the correlation between the series and itself at a previous point in time. A good forecast of the series can be thought of as the future points having high autocorrelation with the previous points. *ARIMA* uses a distributed lag model where regressions are used to predict future values based on lagged values (an *autoregressive* process). Autoregressive and moving average parameters are used to tune the model, along with *polynomial factor differencing* - a process used to make the series *stationary* (i.e., constant statistical properties over time such as mean and variance), a condition that *ARIMA* has on the input series.

In this example, we attempt to perform anomaly detection on per-minute metrics of a server's CPU utilization. <link dataset> The Y-axis shows the percentage CPU utilization, and the X-axis shows time.

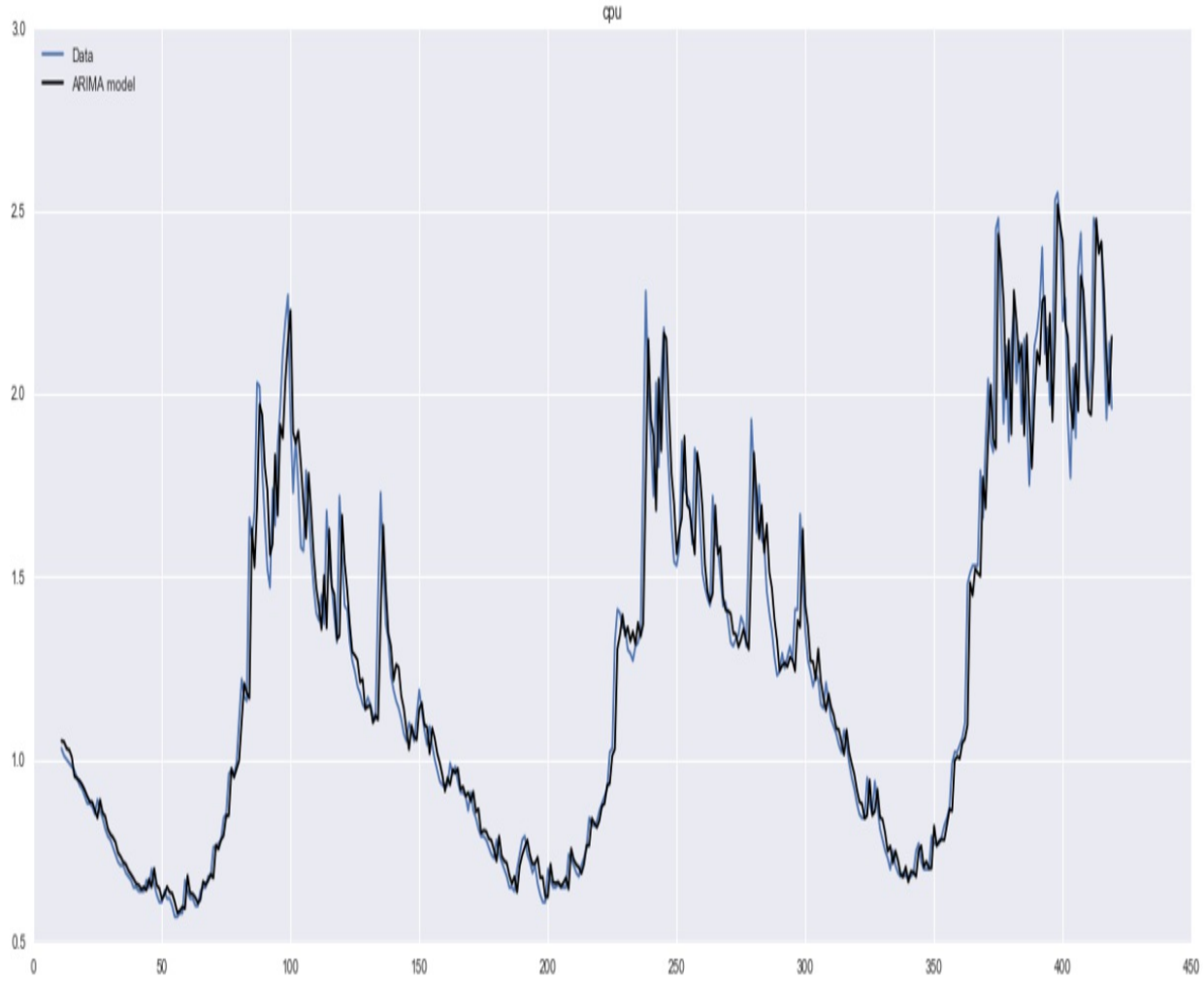


We can observe a clear periodic pattern in this series, with peaks in CPU utilization roughly every 2.5 hours. Using a convenient time series library for Python, *PyFlux*,[53] we apply the *ARIMA* forecasting algorithm with

autoregressive (AR) order 11, moving average (MA) order 11, and differencing order of 0 (since the series looks stationary).[54] There are some tricks to determining the AR and MA orders[55] and the differencing order[56] which we will not elaborate on here. To oversimplify matters, AR and MA orders are needed to correct any residual autocorrelations that remain in the differenced series. (between the time-shifted series and itself) The differencing order is a term used to stationarize the series - an already stationary series (does not have an average changing trend) should have a differencing order of 0, a series with a constant average trend (steadily trending upwards or downwards) should have a differencing order of 1, and a series with a time-varying trend (trend that changes in velocity and direction over the series) should have a differencing order of 2. Plotting the in-sample fit to get an idea of how the algorithm does:

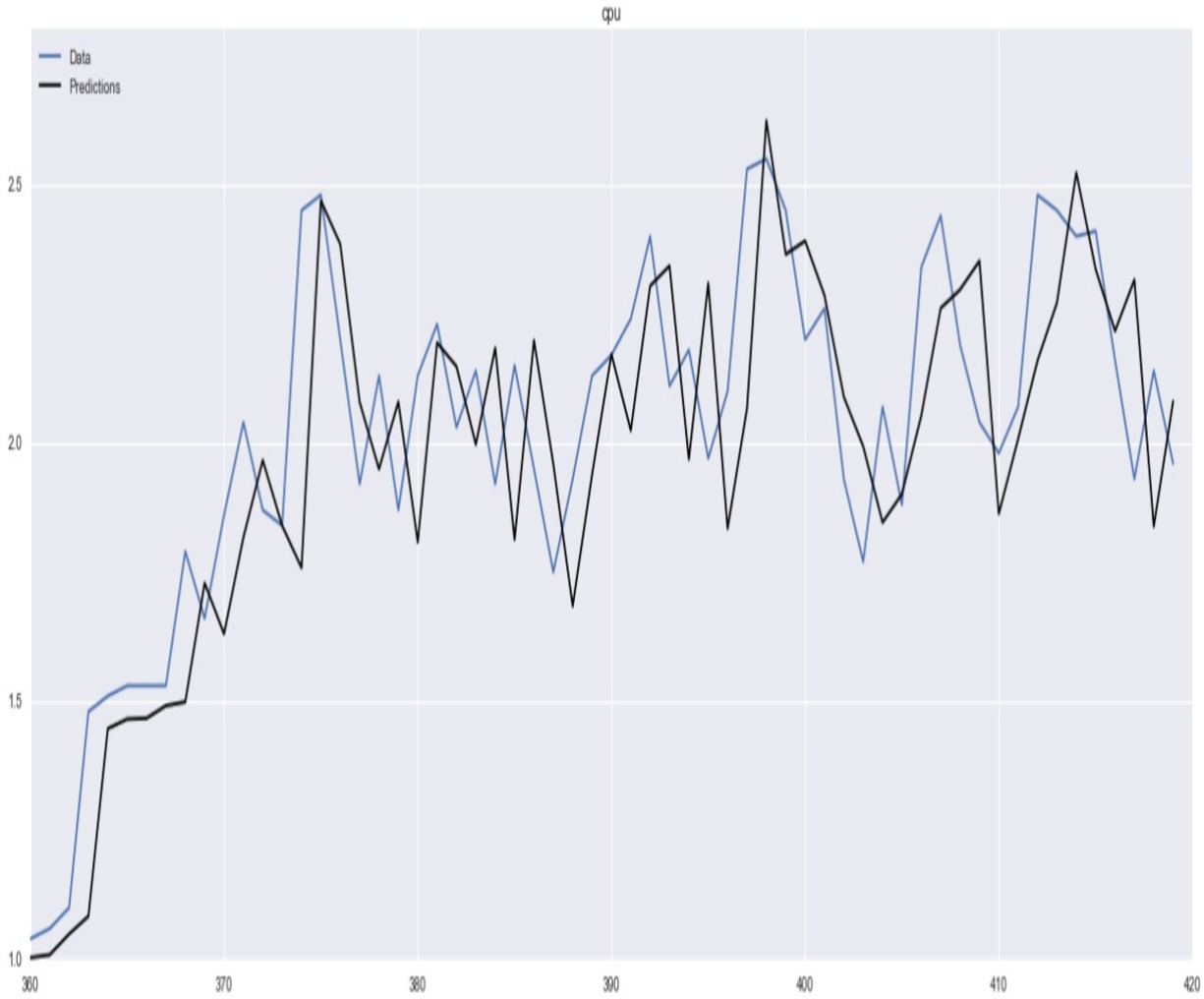
```
import pandas as pd

import pyflux as pf

from datetime import datetime


# Read in the training and testing dataset files

data_train_a = pd.read_csv('cpu-2d-train-a.csv',

parse_dates=[0], infer_datetime_format=True)

data_test_a = pd.read_csv('cpu-2d-test-a.csv',

parse_dates=[0], infer_datetime_format=True)


# Define the model

model_a = pf.ARIMA(data=data_train_a,
```

```
                ar=11, ma=11, integ=0, target='cpu')


# Estimate latent variables for the model using the
# Metropolis-Hastings algorithm as the inference method
x = model_a.fit("M-H")


# Plot the fit of the ARIMA model against the data
model_a.plot_fit()
```



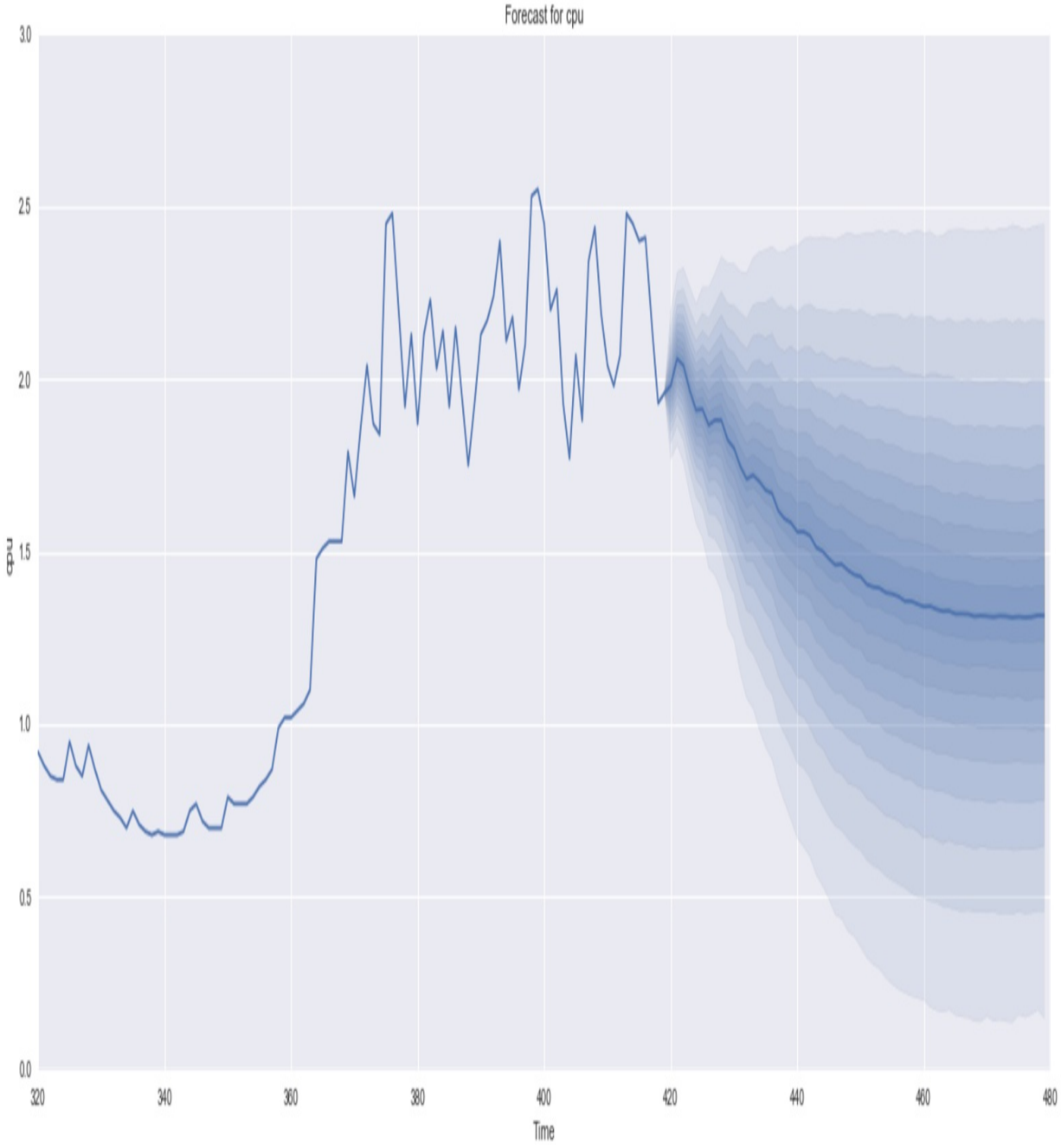
The results fit the observed data quite well. Next, we can do an in-sample test on the last 60 data points of the training data. The in-sample test is a validation step that treats the last subsection of the series as unknown and performs forecasting for those time steps. This allows us to evaluate performance of the model without running tests on future/test data:

> model_a.plot_predict_is(h=60)

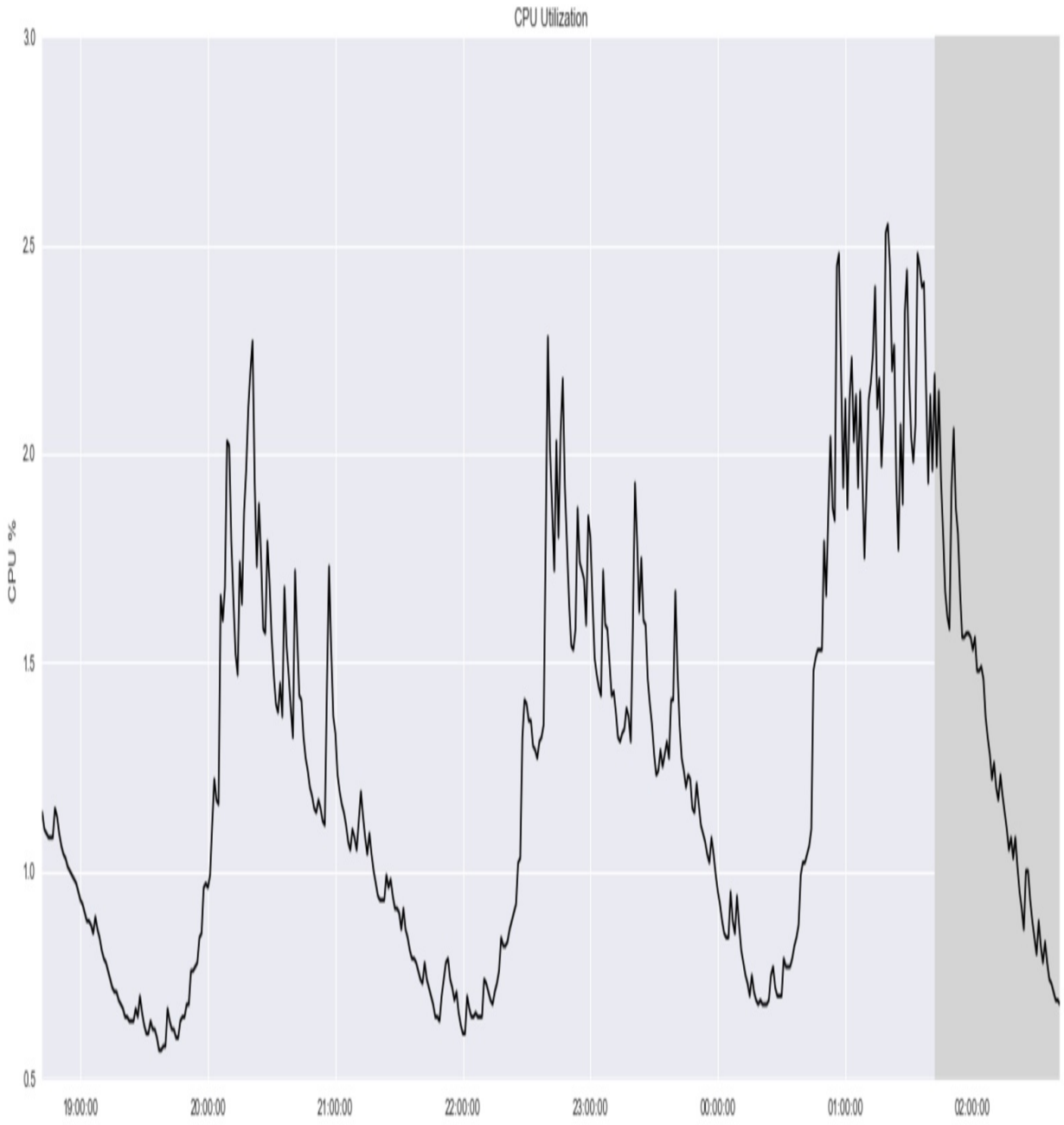


The in-sample prediction test looks pretty good since it does not deviate from the original series significantly in phase and amplitude. Now, let's run the actual forecasting, plotting the most recent observed 100 data points followed by the model's 60 predicted values along with confidence band intervals (bands with a darker shade imply a higher confidence):

> model_a.plot_predict(h=60, past_values=100)



Comparing this to the actual observed points, we see that the prediction is spot-on.



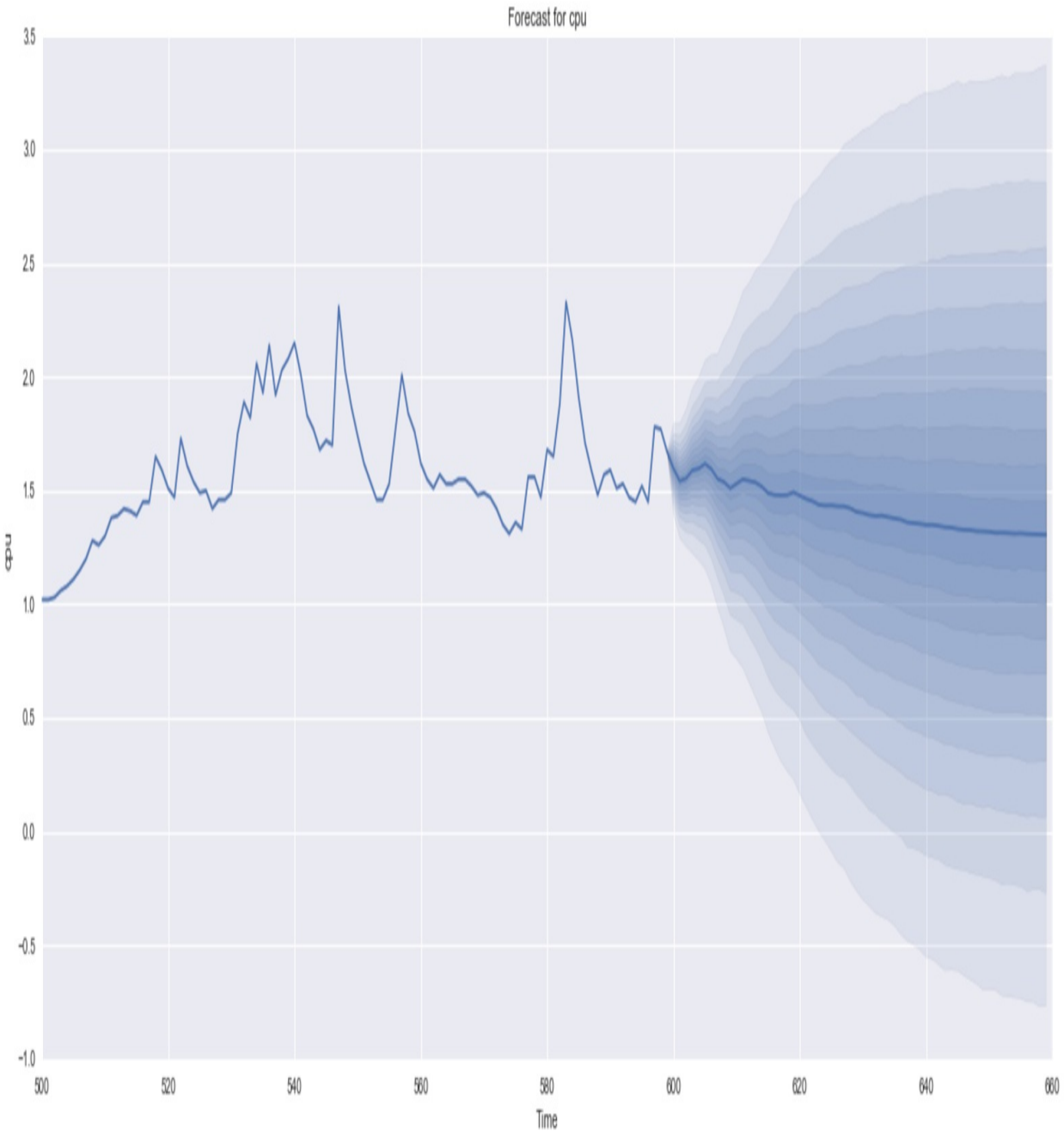
To perform anomaly detection using forecasting, we compare the observed data points with a rolling prediction made periodically. For example, an arbitrary but sensible system might make a new 60 minute forecast every 30 minutes, training a new *ARIMA* model using the previous 24 hours of data. Comparisons between the forecast and observations can be made much more frequently (e.g. every 3 minutes). This method of *incremental learning* can be applied to almost all the methods that we will discuss, approximating

streaming behavior from an algorithm originally designed for batch processing.

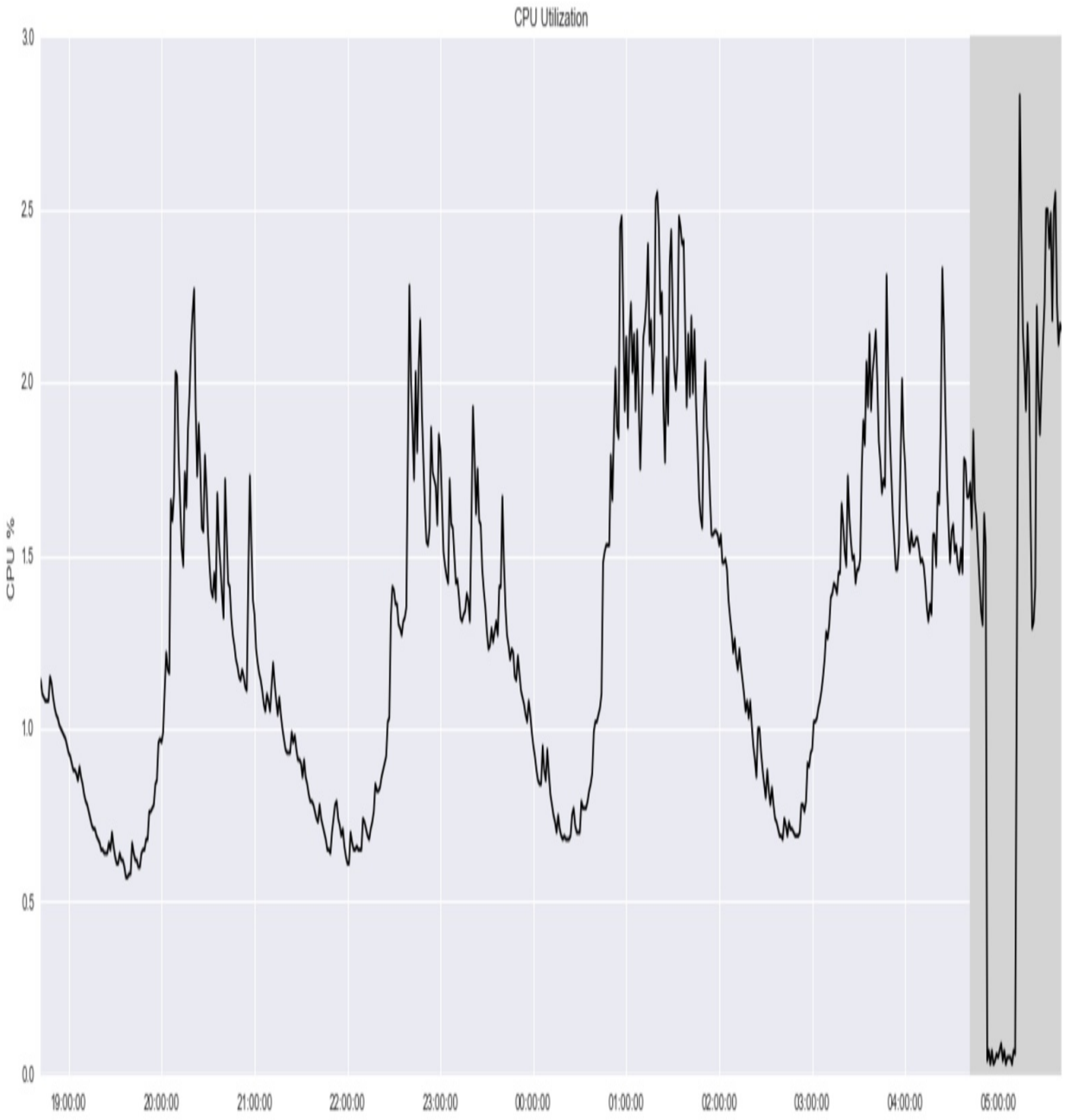
Let’s perform the same forecasting operations on another segment of the CPU utilization dataset captured at a different time:

```
data_train_b = pd.read_csv('cpu-2d-train-b.csv',

parse_dates=[0], infer_datetime_format=True)

data_test_b = pd.read_csv('cpu-2d-test-b.csv',

parse_dates=[0], infer_datetime_format=True)
```

Forecasting using the same *ARIMAX* model trained on *data_train_b*, this is the prediction:



The observed values are however very different from the predictions:



We see a visible anomaly that occurs a short time period after our training period. Since the observed values fall within the low-confidence bands, we will raise an anomaly alert. The specific threshold conditions for how different the forecasted and observed series have to be to raise an anomaly alert is something that is highly application-specific, but should be simple enough to implement on your own.

- - Artificial Neural networks (LSTM)

Another way to perform forecasting on time-series data is to use artificial neural networks. In particular, Long Short Term Memory (LSTM) networks[57] are[58] suitable for this application. LSTMs are a variant of recurrent neural networks (RNNs) that are uniquely architected to learn trends and patterns in time series input for the purposes of classification or prediction. We will not go into the theory or implementation details of neural networks here; instead, we will approach it as a black box that can learn information from time series containing patterns that occur at unknown or irregular periodicities. We will use the Keras LSTM[59] API (backed by TensorFlow)[60] to perform forecasting on the same CPU utilization dataset that we used above, but will not include the full code used to generate these results.[61]

The training methodology for our LSTM network is fairly straightforward. We first extract all continuous length-*n* subsequences of data from the training input, treating the last point in each subsequence as the label for the sample. In other words, we are generating *n-grams* from the input. For example, taking *n=3*:

raw: [0.51, 0.29, 0.14, 1.00, 0.00, 0.13, 0.56]

n-grams: [[0.51, 0.29, 0.14],

[0.29, 0.14, 1.00],

[0.14, 1.00, 0.00],

[1.00, 0.00, 0.13],

[0.00, 0.13, 0.56]]

training set:

samples	labels
0.51, 0.29	0.14
0.29, 0.14	1.00
0.14, 1.00	0.00
1.00, 0.00	0.13
0.00, 0.13	0.56

The model is learning to predict the third value in the sequence following the two already seen values. LSTM networks have a little more complexity built in that deals with remembering patterns and information from previous sequences, but as mentioned before, we will leave the details out. Let's define a 4-layer[62] LSTM network:

```
from keras.models import Sequential

from keras.layers.recurrent import LSTM

from keras.layers.core import Dense, Activation, Dropout
```

```
# Each training data point will be length 100-1,
# since the last value in each sequence is the label
sequence_length = 100

model = Sequential()

# First LSTM layer defining the input sequence length
model.add(LSTM(input_length=sequence_length-1,
               input_dim=1,
               output_dim=32,
               return_sequences=True))
model.add(Dropout(0.2))

# Second LSTM layer with 128 units
model.add(LSTM(128, return_sequences=True))
model.add(Dropout(0.2))

# Third LSTM layer with 100 units
model.add(LSTM(100, return_sequences=False))
```

```
model.add(Dropout(0.2))


# Densely-connected output layer with the linear activation function

model.add(Dense(output_dim=1))

model.add(Activation('linear'))


model.compile(loss='mean_squared_error', optimizer='rmsprop')
```

The precise architecture of the network (number of layers, size of each layer, what kind of layer etc.) is arbitrarily chosen, roughly based off other LSTM networks that work well for similar problems. Notice that we are adding a Dropout(0.2) term after each hidden layer - *dropout*[63] is a regularization technique that is commonly used to prevent neural networks from overfitting. At the end of the model definition, we make a call to the model.compile() method[64] which configures the learning process. We choose the rmsprop optimizer because the documentation states that it is usually a good choice for RNNs.[65] The model fitting process will use the rmsprop optimization algorithm to minimize the loss function, which we have defined to be the mean_squared_error.[66] There are countless other tunable knobs and different architectures that will contribute to model performance, but as usual we opt for simplicity over accuracy.

Let’s prepare our input:

…

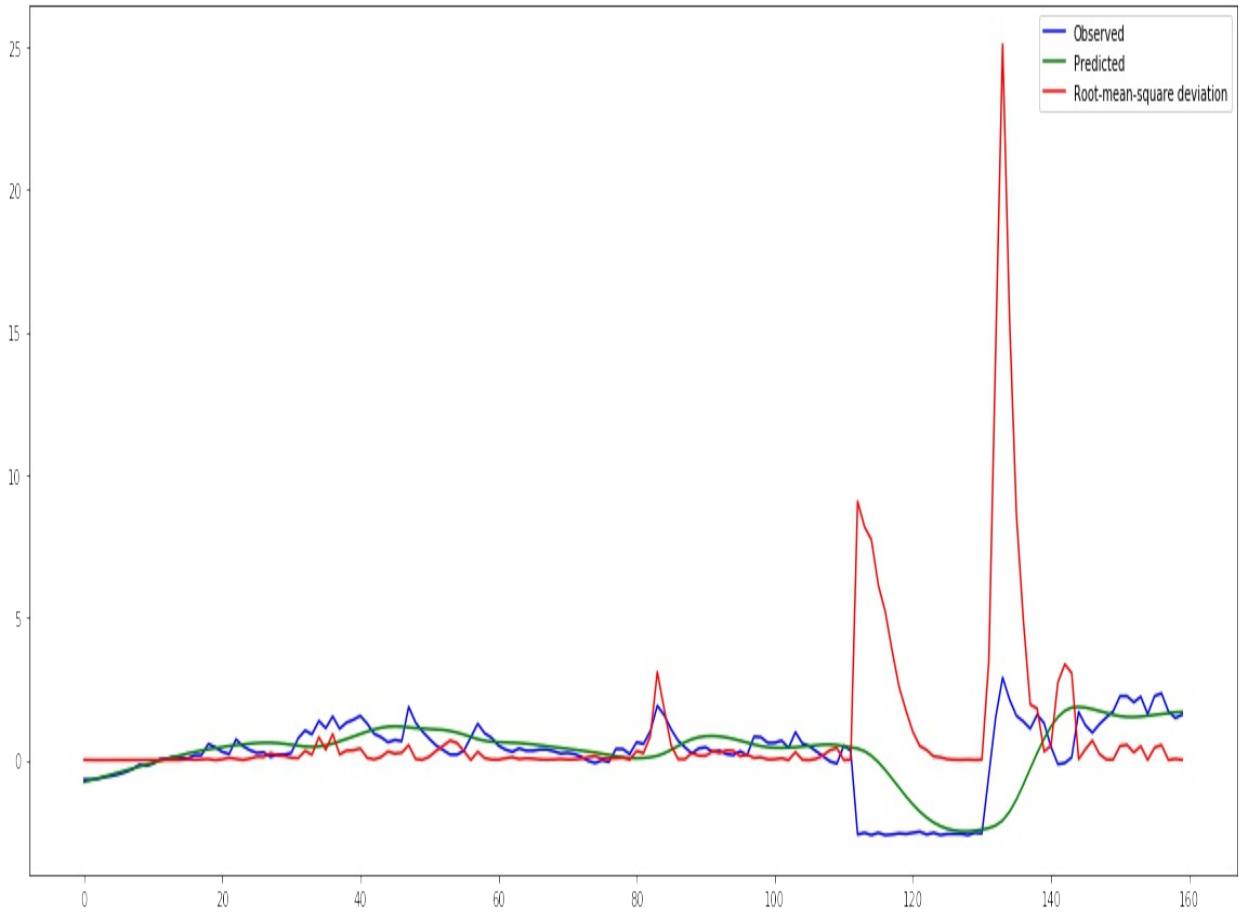
```
# Generating n-grams from the raw training data series
n_grams = []
for ix in range(len(training_data)-sequence_length):
n_grams.append(training_data[ix:ix+sequence_length])

# Normalizing and shuffling the values
n_grams_arr = normalize(np.array(n_grams))
np.random.shuffle(n_grams_arr)

# Separating each sample from its label
x = n_grams_arr[:, :-1]
labels = n_grams_arr[:, -1]
…
```

Then, we can proceed to run the data through the model and make predictions:

```
…
model.fit(x,
  labels,
  batch_size=50,
```

```
    nb_epochs=3,
    validation_split=0.05)

y_pred = model.predict(x_test)
…
```

Plotting the results alongside the root-mean-squared deviation:



We see that the prediction follows the non-anomalous observed series closely (both normalized), which hints to us that the LSTM network is indeed working well. When the anomalous observations occur, we see a large deviation between the predicted and observed series, evident in the root-mean-square plot. Similar to the ARIMA case, such measures of deviations between predictions and observations can be used to signal when anomalies are detected. Thresholding on the observed vs. predicted series divergence is a good way to abstract out the quirks of the data into a simple measure of unexpected deviations.

Summary: Forecasting is an intuitive method of performing anomaly detection. Especially when the time series has predictable seasonality patterns and an observable trend, models such as *ARIMA* can capture the data and reliably make forecasts. For more complex time series data, LSTM networks can work well. There are other methods of forecasting which utilize the same principles and achieve the same goal. Reconstructing time series data from a trained machine learning model (such as clustering models) can be used to generate a forecast, but the validity of such an approach has been disputed in academia.[67]

Note that forecasting does not typically work well on outlier detection, that is, if the training data for your model contains anomalies that you cannot easily filter out, your model will fit to both the inliers *and* outliers, which will make it difficult to detect future outliers. It is well suited for *novelty detection*, which means that the anomalies are only contained in the test data and not the training data. If the time series is highly erratic and does not follow *any* observable trend, or if the amplitude of fluctuations vary widely, forecasting is not likely to perform well. Forecasting works best on one-dimensional series of real-valued metrics, so if your dataset contains multidimensional feature vectors or categorical variables, you will be better off using another method of anomaly detection.

1. 2. Statistical metrics

Statistical tests can be used to determine whether a single new point is similar to the previously seen points. Our example at the beginning of the chapter, where we made a threshold-based anomaly detector adapt to changing data by maintaining an aggregate moving average of the series, falls into this category. Moving averages of time series data can be used as an adaptive metric for comparison in order to determine how well data points conform to this statistic. The moving average (also known as a *low-pass filter* in signal processing terminology) is used as the reference point for statistical comparisons, and significant deviations from the average will be considered anomalies. Here, we will briefly discuss a few noteworthy metrics, but will not dwell too long on each since they are fairly straightforward to use.

- - Median absolute deviation

The standard deviation of a data series is frequently used in adaptive thresholding to detect anomalies. For instance, a sensible definition of anomalies may be any point that lies more than 2 standard deviations from the mean. For example, for a Standard Normal[68] dataset with a mean of 0 and standard deviation of 1, any data points that lie between -2 and 2 will be considered regular. A data point with the value 2.5 would be considered anomalous since it is further than 2 standard deviations from the mean of 0. This is fine in perfect scenarios, but if the training data contains outliers, the calculated mean and standard deviations will be skewed. The median absolute deviation (MAD) is a commonly used alternative to the standard deviation for finding outliers in one-dimensional data. MAD is defined as the median of the absolute deviations from the series median:

```
import numpy as np


# input data series

x = [1, 2, 3, 4, 5, 6]


# calculate median absolute deviation

mad = np.median(np.abs(x - np.median(x)))


# MAD of x is 1.5
```

Since median is much less susceptible to outlier perturbation than mean, MAD is a robust measure suitable for use in scenarios where the training data contains outliers.

- - Grubb’s outlier test

Grubb’s test[69] is algorithm that finds a single outlier in a normally distributed data set, by iteratively considering the current minimum or maximum values in the series. The algorithm is applied iteratively, removing the previously detected outlier between each iteration. We will not go into the details in this section, but a common way to use the Grubb’s outlier test to detect anomalies is to calculate the Grubb’s test statistic and Grubb’s critical value, and mark the point as an outlier if the test statistic is greater than the critical value. This is only suitable for normal distributions, and can be inefficient because it only detects one anomaly in each iteration.

Summary: Statistical metric comparison is a very simple way to perform anomaly detection, and may not be considered by many as a machine learning technique. However, it does check many of the boxes for features of an optimal anomaly detector that we discussed above: anomaly alerts are reproducible and easy to explain, most algorithms adapt to changing trends in the data, it can be very performant because of its simplicity, and it is relatively easy to tune and maintain. Because of this, it may be an optimal choice in some scenarios where statistical measures can perform accurately, or where a lower level of accuracy can be accepted. Because of their simplicity, statistical metrics have limited learning capabilities, and often perform worse than more powerful machine learning algorithms.

1. 3. Goodness-of-fit

In building an anomaly detection system, it is important to consider whether the data used to train the initial model is contaminated with anomalies. As discussed earlier, this question may be difficult to answer, but you can often make an informed guess given a proper understanding of the nature of the data source and threat model. In a perfect world, the expected distributions of a dataset can be accurately modeled with known distributions. For instance, let's imagine that the distribution of API calls to an application server throughout the day (over time) closely fits a normal distribution, and the number of hits to a website in the hours after a promotion is launched may be accurately described by an exponential decay. (clearly, these may not be the most realistic distributions) However, because we do not live in a perfect world, it is rare to find real datasets that conform perfectly to simple distributions. Even if a dataset can be fitted to some hypothetical analytical distribution, accurately determining what this distribution is can be a challenge. Nevertheless, this may be feasible in some cases, especially when dealing with a large dataset for which the expected distribution is well known.[70] In such cases, comparing the divergence between the expected/analytical and observed distributions can be a method of anomaly

detection.

Goodness-of-fit tests such as the Chi-Squared test, the Kolmogorov-Smirnov test, and the Cramér-von Mises criterion can be used to quantify how similar or different two continuous distributions are. These tests are mostly only valid for one-dimensional datasets, which can largely limit their usefulness. We will not dive too deeply into traditional goodness-of-fit tests because of their limited usefulness in real world anomaly detection.

Instead, we will take a closer look at more versatile methods such as the Elliptic Envelope fitting method provided in scikit-learn.

- - Elliptic envelope fitting (covariance estimate fitting)

For normally distributed datasets, elliptic envelope fitting can be a simple and elegant way to perform anomaly detection. Since anomalies are, by definition, points that do not conform to the expected distribution, it is easy for algorithms to apply algorithms to exclude such outliers in the training data. Thus, this method is only minimally affected by the presence of anomalies in the data set.

The use of this method requires that you make a rather strong assumption about your data - that the inliers come from a known analytical distribution. Let's take an example of a hypothetical dataset containing two appropriately scaled and normalized features (e.g. "*peak CPU utilization*" and "*start time*" of user-invoked processes on a server in 24 hours). Note that it is rare to find time-series datasets that correspond to simple and known analytical distributions. More likely than not, this method will be suitable in anomaly

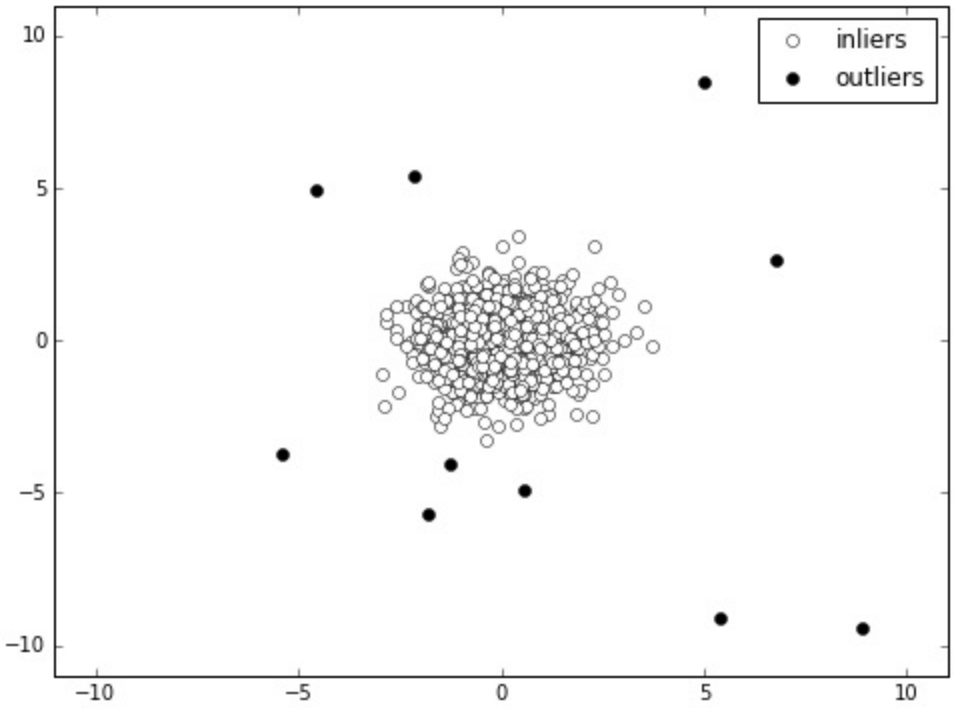
detection problems with the time dimension excluded. We will synthesize this dataset by sampling a Gaussian distribution then including a 0.01 ratio of outliers in the mixture:

```
import numpy as np

num_dimensions = 2

num_samples = 1000

outlier_ratio = 0.01

num_inliers = int(num_samples * (1-outlier_ratio))

num_outliers = num_samples - num_inliers

# Generate the normally-distributed inliers

x = np.random.randn(num_inliers, num_dimensions)

# Add outliers sampled from a random uniform distribution

x_rand = np.random.uniform(low=-10, high=10, size=(num_outliers,
num_dimensions))

x = np.r_[x, x_rand]

# Generate labels, 1 for inliers and -1 for outliers
```

```
labels = np.ones(num_samples, dtype=int)

labels[-num_outliers:] = -1
```

Plotting this as a scatter plot, we see that the outliers are visibly separated from the central mode cluster:

```
import matplotlib.pyplot as plt


plt.plot(x[:num_inliers,0], x[:num_inliers,1], 'wo', label='inliers')

plt.plot(x[-num_outliers:,0], x[-num_outliers:,1], 'ko', label='outliers')

plt.xlim(-11,11)

plt.ylim(-11,11)

plt.legend(numpoints=1)

plt.show()
```

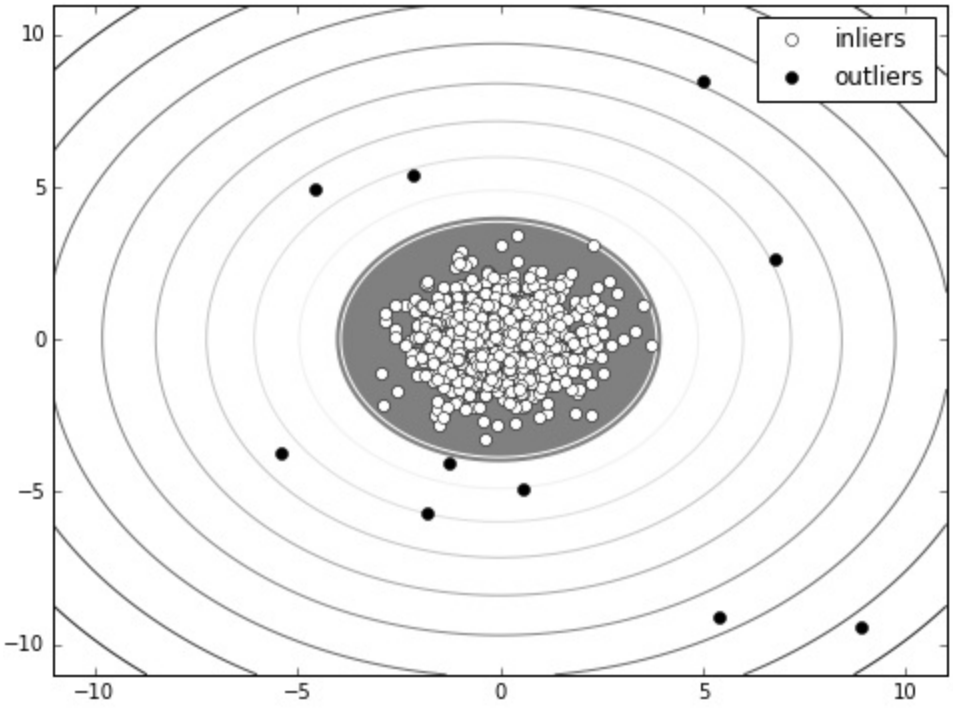


For performing anomaly detection, elliptical envelope fitting does seem like a suitable option since the data looks normally distributed. We use the convenient sklearn.covariance.EllipticEnvelope[71] class in the following analysis:

```
from sklearn.covariance import EllipticEnvelope

classifier = EllipticEnvelope(contamination=outlier_ratio)
classifier.fit(x)
y_pred = classifier.predict(x)
num_errors = sum(y_pred != labels)
print('Number of errors: {}'.format(num_errors))
```

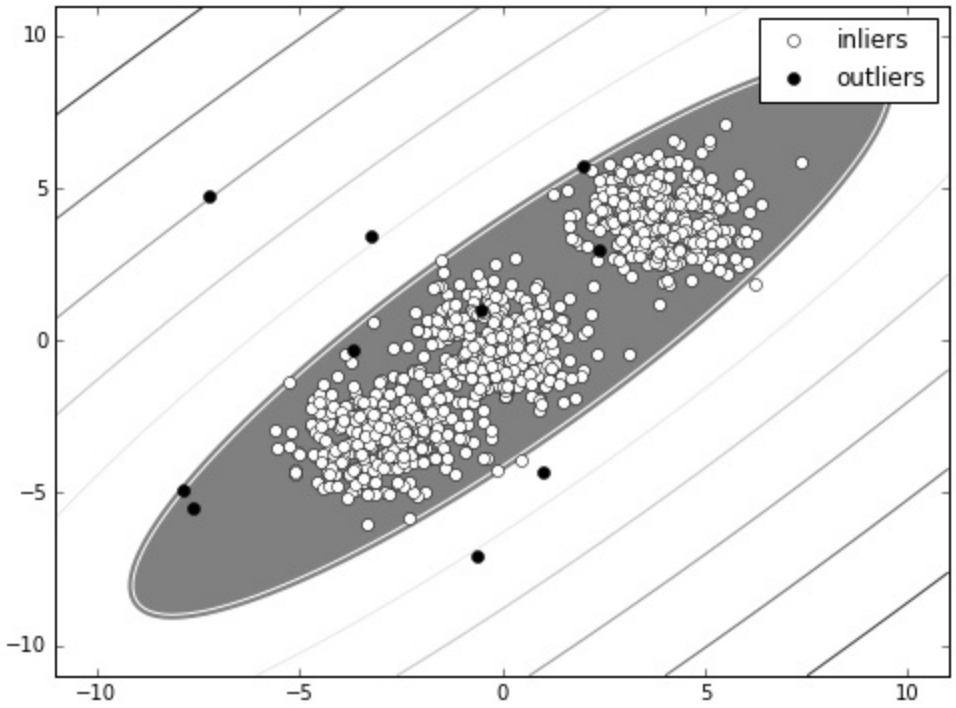
> Number of errors: 0

This method performs very well on this dataset, but that is not surprising at all, given the regularity of the distribution. In this example, we have the accurate value for outlier_ratio as 0.01 since we created the dataset synthetically. The outlier_ratio is an important parameter since it informs the classifier of the number of outliers it should be looking out for. In realistic scenarios where the outlier ratio is not known, a good way to set an initial value is to take a stab in the dark based on your knowledge of the problem. Thereafter, you can iteratively tune the outlier_ratio upwards if you are not detecting some outliers that the algorithm should have found, or tune it downwards if there is a problem with false positives. Let's take a closer look at the decision boundary formed by this classifier:



The center mode is shaded in grey, demarcated by a elliptical decision boundary. Any points lying beyond the decision boundary of the ellipse are considered to be outliers.

We need to keep in mind that this method has varying degrees of effectiveness across different data distributions. Looking at a dataset that *does not* fit a regular Gaussian distribution:



There are now 8 misclassifications - 4 outliers are now classified as inliers, and 4 inliers that fall just outside the decision boundary are flagged as outliers.

Applying this method in streaming anomaly detection system is straightforward. By periodically fitting the elliptical envelope to new data, you will have a constantly updating decision boundary with which to classify incoming data points. In order to remove effects of *drift* and a continually expanding decision boundary over time, it may be a good idea to retire data points after a certain amount of time. However, to ensure that seasonal and cyclical effects are covered, this *sliding window* of fresh data points needs to be wide enough to encapsulate information about daily/weekly patterns etc.

The EllipticEnvelope function in sklearn is located in the sklearn.covariance[72] module. The *covariance* of features in a dataset refers to the joint variability of the features. In other words, it is a measure of the magnitude and direction of the effect that a change in one feature has on another. The covariance is a characteristic of a dataset that can be used to describe distributions, and in turn be used to detect outliers that do not fit within the described distributions. *Covariance estimators* can be used to empirically estimate the covariance of a dataset given some training data, which is exactly how covariance-based fitting for anomaly detection works.

Robust covariance estimates[73] such as the *Minimum Covariance Determinant (MCD)* estimator will minimize influence of training data outliers on the fitted model. The quality of a fitted model can be measured by the distance between outliers and the model's distribution, using a distance function like *Mahalanobis distance*. Compared with non-robust estimates such as the *Maximum Likelihood Estimator (MLE)*, MCD is able to discriminate between outliers and inliers, generating a better fit that results in a inliers having a much smaller distances and outliers having larger distances to the central mode of the fitted model.

The elliptic envelope fitting method makes use of robust covariance estimators to attain covariance estimates that model the distribution of the regular training data, then classifying points that do not meet these estimates as anomalies. We see that the elliptic envelope fitting works reasonably well for a two-dimensional contaminated dataset with a known Gaussian distribution, but not so well on a non-Gaussian dataset. This technique can be applied to higher dimensional datasets as well, but your mileage may vary - elliptic envelopes work better on datasets with a lower dimensionality. When using it on time series data, you may find it useful in some scenarios to remove *time* from the feature set and just fit the model to a subset of other features. In this case, however, note that you will not be able to capture an anomaly that appears statistically regular compared to the aggregate

distribution, but in fact is anomalous relative to the time it appeared. For example, if some anomalous data points from the middle of the night have features that exhibit values that are not out of the ordinary for a midday data point, but is highly anomalous for night time measurements, the outlier may not be detected when the time dimension is not used.

1. 4. Unsupervised machine learning algorithms

We now turn to a class of solutions to the anomaly detection problem that arise from modifications of typical supervised machine learning models. Supervised machine learning classifiers are typically used solve problems that involves two or more classes. However, when used for anomaly detection, the modifications of these algorithms give them characteristics of unsupervised learning. In this section, we look at a couple of them.

- - One-class SVMs

One-class Support Vector Machines (SVM) can be used to detect anomalies with by fitting an SVM with data belonging to only a single class. This data (which is assumed to contain no anomalies) is used to train the model, creating a decision boundary that can be used to classify future incoming data points. There is no robustness mechanism built into standard one-class SVM implementations, which means that the model training is less resilient to outliers in the data set. As such, this method is more suitable for novelty detection than outlier detection - that is, the training data should ideally be thoroughly cleaned and contain no anomalies.

Where the one-class SVM method pulls apart from the pack is in dealing with

non-Gaussian or multi-modal distributions (i.e. when there is more than one "center" mode of regular inliers), as well as high-dimensional datasets. We will apply the one-class SVM classifier to the second dataset we used in the Robust Covariance Estimation section. Note that this is not an ideal dataset for this method since it does contain 1% of outliers, but let's see how much the resulting model is affected by the presence of contaminants:

```
from sklearn import svm

classifier = svm.OneClassSVM(nu=0.99 * outlier_ratio + 0.01,
                             kernel="rbf",
                             gamma=0.1)
classifier.fit(x)
y_pred = classifier.predict(x)
num_errors = sum(y_pred != labels)
print('Number of errors: {}'.format(num_errors))
```

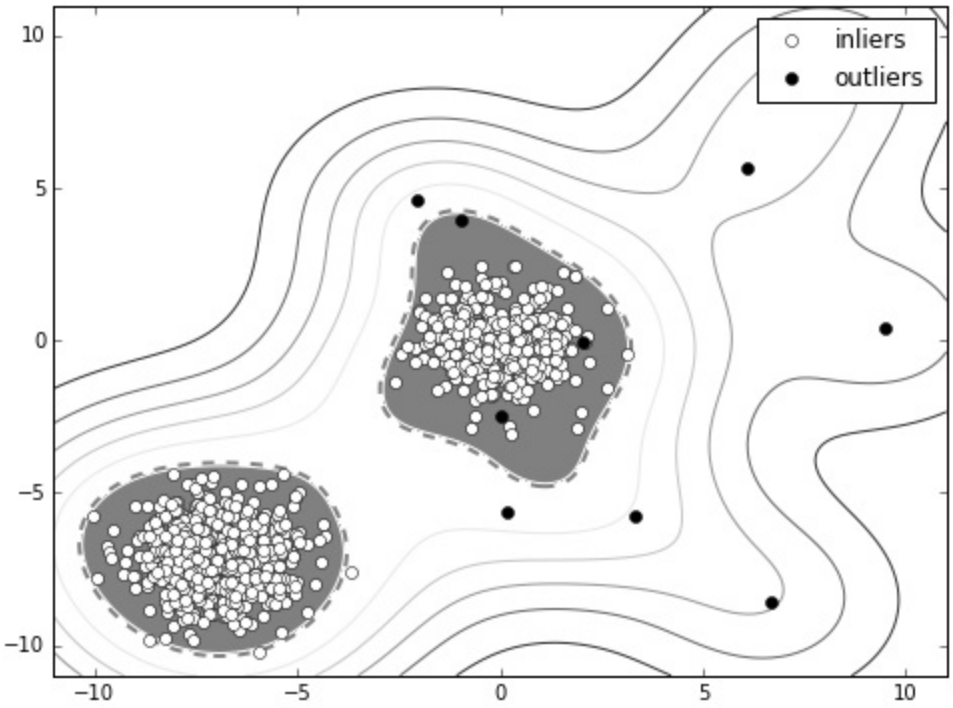
Let's focus on the custom parameters that we specified in the creation of the svm.OneClassSVM[74] object. Note that these parameters are dependent on datasets and usage scenarios; in general, you should always have a good understanding of all tunable parameters offered by a classifier before you use it. In order to deal with a small presence of outliers in the data, we set the nu parameter to be roughly equivalent to the outlier ratio. This parameter controls the "upper bound on the fraction of training errors and the lower bound of the fraction of support vectors". In other words, it represents the acceptable margin of errors generated by the model that can be caused by stray outliers, allowing the model some flexibility to prevent overfitting the

model to outliers in the training set.

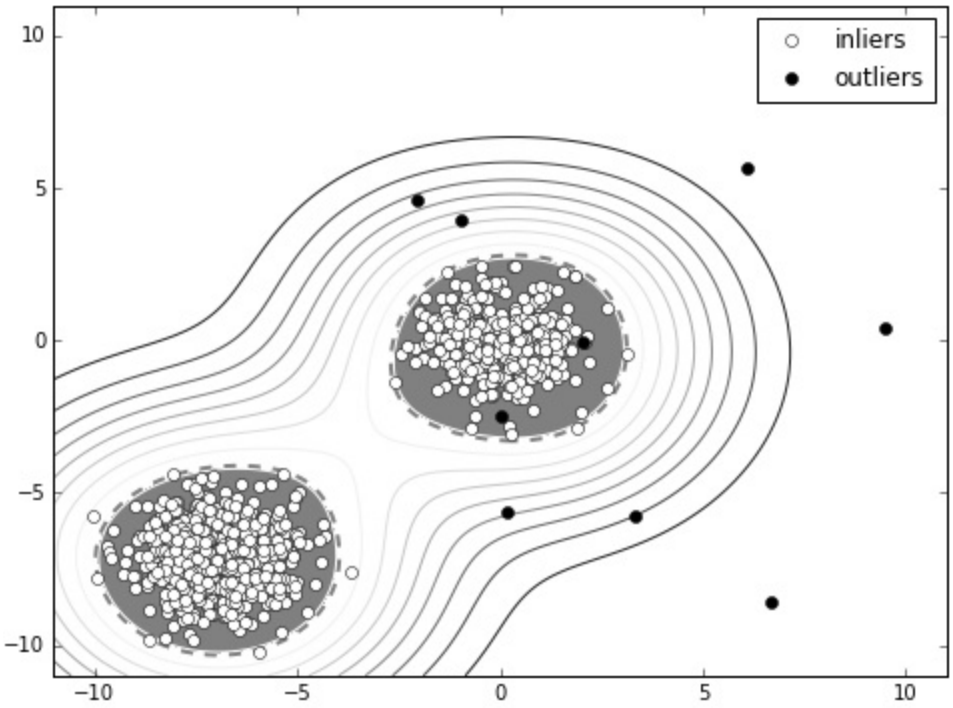
The “rbf” Radial Basis Function (RBF) kernel is selected arbitrarily by visually inspecting the dataset’s distribution. Each cluster in the bimodal distribution has Gaussian characteristics, which suggests that the RBF kernel would be a good fit, since value of the RBF also decreases exponentially as points move radially further away from the center mode.

The “gamma” parameter is used to tune the RBF kernel.[75] The default value of this parameter is 0.5. This parameter defines how much influence any one training sample has on the resulting model. Smaller values of gamma would result in a “smoother” decision boundary, which may not be able to adequately capture the shape of the dataset. Larger values of gamma may result in overfitting. A smaller value of gamma was chosen in this case to prevent overfitting to outliers that are close to the decision boundary.

Inspecting the resulting model, we see that the one-class SVM is able to fit to this strongly-bimodal dataset quite well, generating two mode clusters of inliers. There are 16 misclassifications - the presence of outliers in the training data has some effect on the resulting model.



Let’s retrain the model on purely the inliers and see if it does any better:



Indeed, there are only 3 classification errors now.

One-class SVMs offer a more flexible method for fitting a learned distribution to your dataset compared to Robust Covariance Estimation. If you are thinking of using it as the engine for your anomaly detection system, special attention needs to be made to potential outliers that may slip past detection and cause the gradual degradation of the model's accuracy.

- - Isolation forests

Random forest classifiers have a reputation for working well as anomaly detection engines in high-dimensional datasets. Random forests are algorithmic trees, and stream classification on tree data structures is much more efficient efficient compared to models that involve many cluster or distance function computations. The number of feature value comparisons required to classify an incoming data point is the height of the tree (vertical distance between the root node and the terminating leaf node). This makes it very suitable for real-time anomaly detection on time series data.

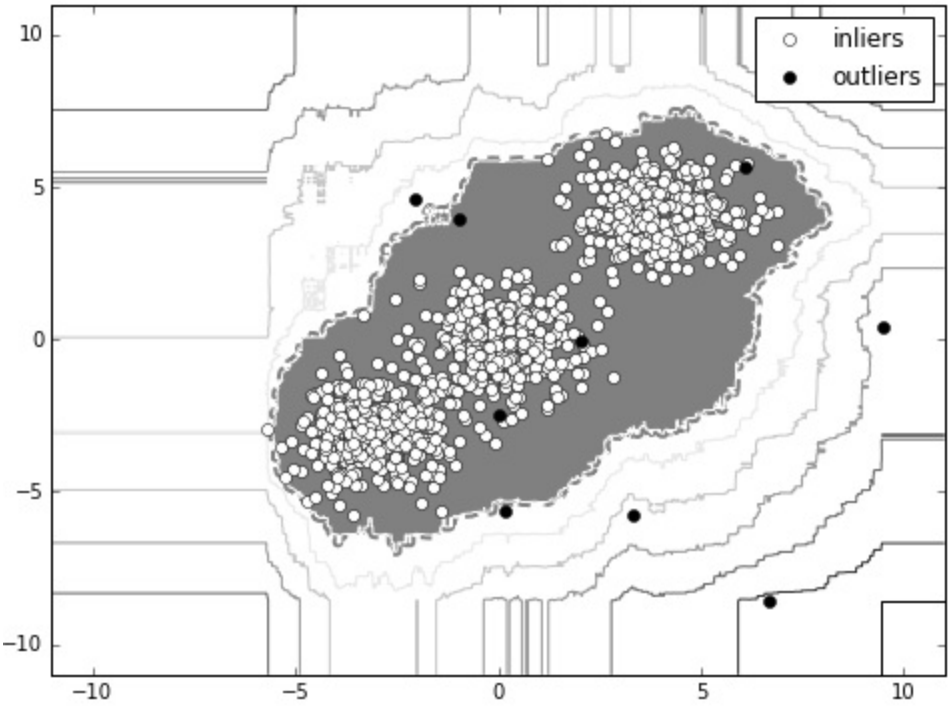
The sklearn.ensemble.IsolationForest class[76] helps determine the anomaly score of a sample using the Isolation Forest algorithm. This algorithm trains a model by iterating through data points in the training set, randomly selecting a feature and randomly selecting a split value between the maximum and minimum values of that feature (across the entire dataset). The algorithm operates in the context of anomaly detection by computing the number of splits required to isolate a single sample; i.e., how many times we need to perform splits on features in the dataset before we end up with a region that only contains the single target sample. The intuition behind this method is that inliers have more feature value similarities between each other, which requires them to go through more splits to be isolated. Outliers, on the other hand, should be easier to isolate with a small number of splits since they will likely have some feature value differences that distinguish them from inliers.

By measuring the “path length” of recursive splits from the root of the tree, we have a metric with which we can attribute an anomaly score to data points. Anomalous data points should have shorter path lengths than regular data points. In the sklearn implementation, the threshold for points considered anomalous is defined by the contamination ratio. With a contamination ratio of 0.01, the shortest 1% of paths will be considered anomalies.

Let’s see it in action by applying the Isolation Forest algorithm on a the non-Gaussian contaminated dataset that we have seen in earlier sections:

```
from sklearn.ensemble import IsolationForest


rng = np.random.RandomState(99)


classifier = IsolationForest(max_samples=num_samples,
                             contamination=outlier_ratio,
                             random_state=rng)
classifier.fit(x)
y_pred = classifier.predict(x)
num_errors = sum(y_pred != labels)
print('Number of errors: {}'.format(num_errors))
```

> Number of errors: 8



Using Isolation Forests in streaming time series anomaly detection is very similar to using one-class SVMs or Robust Covariance Estimations. The anomaly detector simply maintains a tree of Isolation Forest splits and updates the model with new incoming points (as long as they are not deemed anomalies) in newly isolated segments of the feature space.

It is important to note that even though testing/classification is efficient, initially training the model is often more resource- and time-intensive than other methods of anomaly detection discussed earlier. On very low-dimensional data, using Isolation Forest for anomaly detection may not be suitable since the small number of features on which we can perform splits can limit the effectiveness of the algorithm.

1. 5. Density-based methods

Clustering methods such as the *k-means* algorithm are known for their use in unsupervised classification and regression. Similar density-based methods can be used in the context of anomaly detection to identify outliers. Density-based methods are perfect for high-dimensional datasets, which can be difficult to deal with using the other classes of anomaly detection methods described above. Several different density-based methods have been adapted for use in anomaly detection. The idea is to form a cluster representation of the training data, under the hypothesis that outliers or anomalies will be located in low-density regions of this cluster representation. This approach has the convenient property of being resilient to outliers in the training data, since such instances will likely also be found in low-density regions.

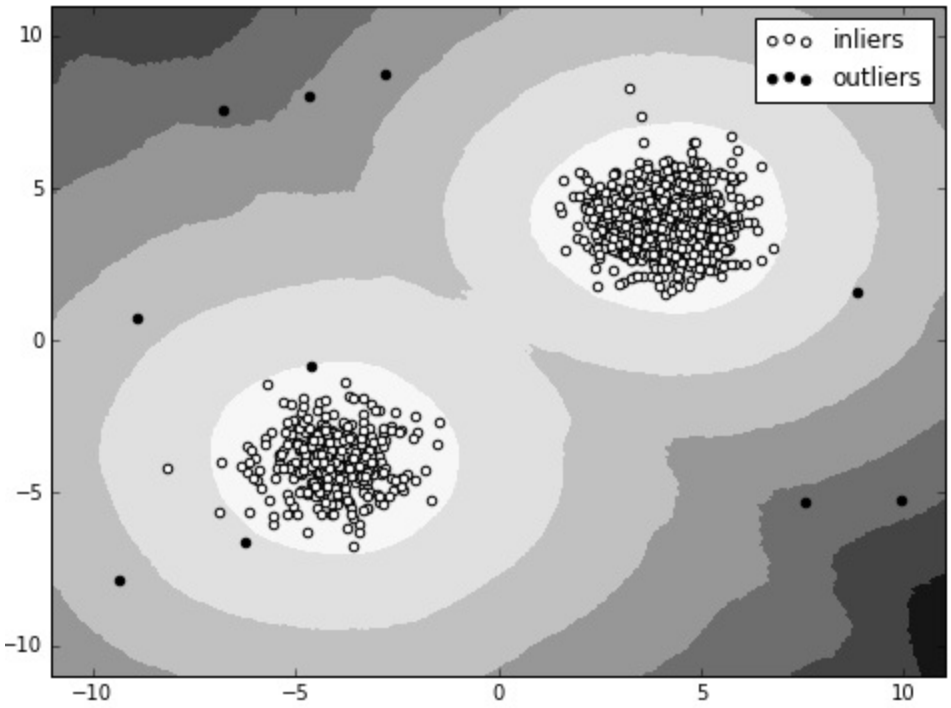
Even though the k-Nearest Neighbors (kNN) algorithm is not a clustering algorithm, it is commonly considered a density-based method, and is actually quite a popular way to measure the probability that a data point is an outlier. In essence, the algorithm can estimate the local sample density of a point by measuring its distance to the k^{th} nearest neighbor. k-means clustering can also be used for anomaly detection in a very similar way, using distances between the point and centroids as a measure of sample density. kNN has the potential to scale well to large datasets by leveraging *k-d trees*, (k-dimensional trees) which can greatly improve computation times for smaller dimensional datasets.[77] In this section, we will focus on a method called the local outlier factor, which is a classic density-based machine learning method for anomaly detection.

- - Local outlier factor

The *Local Outlier Factor* (LOF) is an anomaly score that you can generate

using the scikit-learn class - sklearn.neighbors.LocalOutlierFactor.[78] Similar to the kNN and k-Means anomaly detection methods briefly mentioned above, LOF classifies anomalies using local density around a sample. Local density of a datapoint refers to the concentration of other points in the immediate surrounding region, where the size of this region can either be defined as a fixed distance threshold or a dynamic region containing the closest *n* neighboring points. LOF measures the isolation of a single data point with respect to its closest *n* neighbors. Data points with a significantly lower local density than the density of their closest *n* neighbors are considered to be anomalies. Let's run an example on a similar non-Gaussian, contaminated dataset once again.

```
from sklearn.neighbors import LocalOutlierFactor


classifier = LocalOutlierFactor(n_neighbors=100)

y_pred = classifier.fit_predict(x)


Z = classifier._decision_function(np.c_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)


num_errors = sum(y_pred != labels)

print('Number of errors: {}'.format(num_errors))


> Number of errors: 9
```



LOF works very well even when there is contamination in the training set, and is not very strongly affected by dimensionality of the data. As long as the dataset maintains the property that outliers have a weaker local density than their neighbors in a majority of the training features, LOF can find clusters of inliers well. Because of the algorithm's approach, it is able to distinguish between outliers in datasets with varying cluster densities. For instance, a point in a sparse cluster may have a higher distance to its nearest neighbors compared to another point in a denser cluster (in another area of the same dataset), but because density comparisons are only made with local neighbors, each cluster will have absolute distance conditions for what constitutes an outlier. Lastly, LOF's nonparametric nature means that it can easily be generalized across multiple different dimensions as long as the data is numerical and continuous.

Having analyzed the above 5 categories of anomaly detection algorithms, it should be clear that there is no lack of machine learning methods applicable to this classic data mining problem. Selecting which algorithm to use can

sometimes be daunting and may take a few iterations of trial and error. However, using the provided guidelines and hints for which classes of algorithms work better for the nature of the data you have and for the problem you are solving, you will be in a much better position to leverage the power of machine learning to detect anomalies.

Challenges of using machine learning in anomaly detection

One of the most successful applications of machine learning is in recommendation systems. Using techniques such as collaborative filtering, recommender systems are able to extract latent preferences of users and act as an engine for active demand generation. What if a wrong recommendation is made? If an irrelevant product is recommended to a user browsing through an online shopping site, the repercussions are insignificant. Besides the lost opportunity cost of a potential successful recommendation, the user simply ignores the uninteresting recommendation. If an error is made in the personalized search ranking algorithm, the user experience may be impacted, but there is no big and tangible loss incurred.

Anomaly detection is rooted in a fundamentally different paradigm. The cost of errors in intrusion or anomaly detection is huge. Misclassification of one anomaly can cause a crippling breach in the system. Raising false positive alerts have a less drastic impact, but spurious false positives can quickly degrade confidence in the system, even resulting in alerts being entirely ignored. Because of the high cost of classification errors, fully automated, end-to-end anomaly detection systems that are powered purely by machine learning are very rare -- there is almost always a human in the loop to verify that alerts are relevant before any action is taken on the alert.

The *semantic gap* is a real problem with machine learning in many

environments. Compared with static rule sets or heuristics, it can sometimes be difficult to explain why an event was flagged as an anomaly, leading to longer incident investigation cycles. In practical cases, interpretability or explainability of results is often as important as accuracy of the results. Especially for anomaly detection systems that constantly evolve their decision models over time, it is worthwhile to invest engineering resources for system components that can generate better explanations for alerts generated by a machine learning system. For instance, if an alert is raised by an outlier detection system powered by a one-class SVM using a latent combination of features selected through dimensionality reduction techniques, it can be difficult for humans to figure out what combinations of explicit signals the system is looking for. As much as is possible given the opacity of many machine learning processes, it will be helpful to generate explanations of why the model made the decision it made.

Devising a sound evaluation scheme for anomaly detection systems can be even more difficult that building the system itself. Because performing anomaly detection on time series implies that there is the possibility of unknown data input never seen in the past, there is no comprehensive way of evaluating the system given the vast possibilities of different anomalies that the system may encounter in the wild.

Advanced actors can (and will) spend time and effort to bypass anomaly detection systems if there is a worthwhile payoff on the other side of the wall. The effect of adversarial impact on machine learning systems and algorithms is real, and is a necessary consideration when deploying systems in a potentially hostile environment. A future chapter of this book will explore Adversarial Machine Learning in greater detail, but any security machine learning system should have some safeguards against tampering that will be discussed in a later chapter.

Response and Mitigation

After receiving the anomaly alert, what comes next? Incident response and threat mitigation are fields of practice that are fully deserving of their own publications, and we cannot possibly paint a complete picture of all the nuances and complexities involved. We can, however, consider how machine learning can be infused into traditional security operations workflows to improve efficacy and yield of human effort.

Simple anomaly alerts can come in the form of an email or a mobile notification. In many cases, organizations that maintain a variety of different anomaly detection and security monitoring systems find value in aggregating alerts from multiple sources into a single platform known as security information and event management (SIEM) systems. These platforms offer more than just a convenience. SIEMs can help with the management of the output of fragmented security systems that can quickly grow out hand in volume. Correlation of alerts raised by different systems can also help analysts to gather insights that can only be obtained by juxtaposing the output of different systems. Having a unified location for reporting and alerting can also make a noticeable difference in the value of security alerts raised. Security alerts can often trigger action items for parts of the organization beyond the security team or even the engineering team. Many improvements to an organization's security require coordinated efforts by cross-team management who do not necessarily have low-level knowledge in security operations. Having a platform that can assist with the generation of reports and digestible, human-readable insights into security incidents can be highly valuable for communicating the security needs of an organization to external stakeholders.

Incident response typically involves a human at the receiving end of security alerts, performing manual actions to investigate, verify, and escalate. Incident response is frequently associated with digital forensics (*DFIR - digital forensics and incident response*), which covers a large scope of actions that a security operations analyst has to perform to triage alerts, collect evidence for

investigations, verify authenticity of collected data, and present the information in a format friendly to downstream consumers. Because of the difficulty of automation, incident response has remained a largely manual process. For instance, there are tools that help with inspecting binaries and reading memory dumps, but there is no real substitution for hypothesizing an attacker's probable actions and intentions on a compromised server.

Machine-assisted incident response has shown significant promise in the field. Machine learning can efficiently mine massive data sets for patterns and anomalies. Human analysts can make informed conjectures and perform complex tasks requiring deep contextual and experiential knowledge. Combining these sets of complementary strengths can potentially help improve the efficiency of incident response operations.

Threat mitigation is the process of reacting to intruders and attackers and preventing them from succeeding in their actions. A first reaction to an intrusion alert may be to nip the threat in the bud and prevent risk from spreading any further. However, this prevents you from collecting any further information about the attacker's capabilities, intent, and origin. In an environment where attackers can iterate quickly and pivot their strategies to circumvent detection, banning or blocking them can be counterproductive. The immediate feedback to the attackers can give them information about how they are getting detected, allowing them to iterate to the point where you may find it difficult to detect them. Silently observing attackers while limiting their scope of damage will often give defenders more time to conceive a longer term strategy that can stop attackers for good.

Stealth banning (or shadow banning, hell banning, ghost banning, etc.) is a practice adopted by social networks and online community platforms to block abusive or spam content precisely for the purpose of not giving these actors an immediate feedback loop. A synthetic environment is created such that the attacker initially still thinks his actions are valid, when in fact they cause no

side effects and other users or system components cannot see the results of his actions.

Practical system design concerns

In designing and implementing machine learning systems for security, there are a number of practical system design decisions to make that go beyond improving classification accuracy.

Optimizing for explainability

As mentioned earlier, the *semantic gap* of alert explainability is one of the biggest stumbling blocks of anomaly detectors using machine learning. Many practical machine learning applications value explanations of results. However, true explainability of machine learning is an area of research hasn't yet seen many closed-ended answers.[79]

Simple machine learning classifiers, or even non-machine learning classification engines, are quite transparent in their predictions. For example, a linear regression model on a two-dimensional dataset generates very explainable results, but lacks the ability to learn more complex and nuanced features. More complex machine learning models such as neural networks, random forest classifiers, and ensemble techniques can fit complex real-world data better, but they are as black-box as it can be - decision making processes are completely opaque to an external observer. There are ways to approach the problem that can alleviate the concern that machine learning predictions are difficult to explain, proving that explainability is not in fact at odds with accuracy. Having an external system generate simple, human-readable

explanations for the decisions made by a black-box classifier satisfies the conditions of result explainability, even if the explanations do not describe the actual decision making conditions made by the machine learning system.[80] This can be thought of as an external observer component to the machine learning system, analyzing any output from the system and performing context-aware data analysis to generate the most probable reasons for why the alert was raised.

Performance and scalability in real-time streaming applications

Many applications of anomaly detection in the context of security require a system that can handle real-time streaming classification requests and deal with shifting trends in the data over time. Unlike ad-hoc machine learning processes, classification accuracy is not the only factor to optimize for. Even though they may yield inferior classification results, some algorithms that are less time- and resource-intensive than others may be the optimal choice for designing systems in resource-critical environments (e.g. for performing machine learning on mobile devices or embedded systems).

Parallelization is the classic computer science answer to performance problems. Parallelizing machine learning algorithms and running them in a distributed fashion on MapReduce frameworks such as *Apache Spark* (Streaming)[81] is a good way to improve performance of machine learning systems by orders of magnitude. In designing systems for the real world, keep in mind that some machine learning algorithms cannot easily be parallelized because inter-node communication is required (e.g. simple clustering algorithms). Using distributed machine-learning libraries such as *Apache Spark MLlib*[82] can help you to avoid the pain of having to implement and optimize distributed machine learning systems. We will investigate the use of these frameworks in a later chapter of this book.

Maintainability of anomaly detection systems

The longevity and usefulness of machine learning systems is dictated not by accuracy or efficacy, but by understandability, maintainability, and ease of configuration of the software. Designing a modular system that allows for swapping out, removing, and reimplementing subcomponents is crucial in environments that are in constant flux. The nature of data constantly changes, and a well-performing machine learning model today may no longer be suitable half a year down the road. If the anomaly detection system is designed and implemented on the assumption that Elliptic envelope fitting is to be used, it will be difficult to swap the algorithm out for say, Isolation Forests in the future. Flexible configuration of both system and algorithm parameters is important for the same reason. If tuning model parameters requires recompiling binaries, the system is not configurable enough.

Integrating human feedback

Having a feedback loop in your anomaly detection system can make for a formidable adaptive system. If security analysts were able to report false positives and false negatives directly to a system that adjusts model parameters based on this feedback, the maintainability and flexibility of the system would be vastly elevated. In untrusted environments, however, directly integrating human feedback into the model training can have negative effects.

Mitigating adversarial effects

As mentioned above, deploying machine learning security systems in a

hostile environment implies that the system will be attacked. Attackers of machine learning systems generally use one of two classes of methods to achieve their goals. If the system continually learns from input data and instantaneous feedback labels provided by users (online learning model), attackers can *poison* the model by injecting intentionally misleading *chaff* traffic to skew the decision boundaries of classifiers. Attackers can also target the blind spots of classifiers with *adversarial examples* that are specially crafted to trick specific models and implementations. It is important to put specific processes in place to explicitly prevent these threat vectors from penetrating your system. In particular, designing a system that blindly takes user input to update the model is risky. In an online learning model, inspecting any input that will be converted to model training data is important for detecting attempts at poisoning the system. Using robust statistics that are resilient to poisoning and probing attempts is another way of slowing the attacker down. Maintaining test sets and heuristics that periodically test for abnormalities in the input data, model decision boundary, or classification results can also be useful. We will explore methods and effects of these adversarial effects in a later chapter of this book.

Conclusion

Anomaly detection is an important topic in security, and is an area that machine learning techniques has shown a lot of efficacy. Before diving into complex algorithms and statistical models, take a moment to think carefully about the problem you are trying to solve, and the data available to you. The answer to a better anomaly detection system may not be to use a more fancy and advanced algorithm, but may be to generate a more complete and descriptive set of input. Because of the large scope of threats they are required to mitigate, security systems have a tendency to grow uncontrollably in complexity. In building or improving anomaly detection systems, always keep simplicity as a top priority.

Notes

1. http://www.computerworld.com/article/2539767/cybercrime-hacking/unsung-innovators--gary-thuerk--the-father-of-spam.html

2. https://www.bloomberg.com/news/articles/2016-01-19/e-mail-spam-goes-artisanal

3. https://www.wired.com/2015/07/google-says-ai-catches-99-9-percent-gmail-spam/

4. http://www.paulgraham.com/spam.html

5. Adapted from the "European CSIRT Network project" Security Incidents Taxonomy https://www.enisa.europa.eu/topics/csirt-cert-services/community-projects/existing-taxonomies

6. The Legitimate Vulnerability Market: Inside the Secretive World of 0-day Exploit Sales, Charlie Miller, http://www.econinfosec.org/archive/weis2007/papers/29.pdf

7. Measuring pay-per-install: the commoditization of malware distribution, Juan Caballero, http://software.imdea.org/~juanca/papers/ppi_usenixsec11.pdf

8. http://www.vdiscover.org/OS-fuzzing.html

9. https://www.infosecurity-magazine.com/news/bhusa-researchers-present-phishing/

10. https://people.eecs.berkeley.edu/~tygar/papers/SML2/Adversarial_AISEC.pdf

11. http://lcamtuf.coredump.cx/afl/

12. http://taoxie.cs.illinois.edu/publications/policy06.pdf

13. 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.

14. http://plg.uwaterloo.ca/~gvcormac/treccorpus07/

15. http://trec.nist.gov/pubs/trec16/papers/SPAM.OVERVIEW16.pdf

16. 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 pitfall of information loss from a single train/test split that does not adequately capture the statistical properties of the data. Here, we chose to use conventional validation for simplicity.

17. http://www.nltk.org/

18. In order to run this example, you need to install the "Punkt Tokenizer Models" and the "Stopwords Corpus" libraries in NLTK using the nltk.download() utility

19. https://github.com/ekzhu/datasketch

20. Chapter 3, Mining of Massive Datasets, http://infolab.stanford.edu/~ullman/mmds/ch3.pdf

21. http://scikit-learn.org/stable/modules/naive_bayes.html

22. http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVecto

23. In general, just using accuracy to measure model prediction performance is crude and incomprehensive. Model evaluation is an important topic that deserves dedicated discussion in a later chapter. Here, we opt for simplicity and just use the accuracy as an approximate measure of performance. The sklearn.metrics.classification_report() method provides the *precision, recall, f1-score,* and *support* for each class, which can be used in combination to get a more accurate picture of how the model performs.

24. http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/

25. http://www.dtic.mil/dtic/tr/fulltext/u2/a484998.pdf

26. https://www.snort.org/

27. A *kernel* is a similarity function that is provided to a machine learning algorithm that indicates how similar two inputs are. Kernels offer an alternate approach to feature engineering - instead of extracting individual features from the raw data, kernel functions can be efficiently computed, sometimes in high-dimensional space, to generate implicit features from the data that would otherwise be expensive to generate. The approach of efficiently transforming data into a low-dimensional, implicit feature space is known as the *kernel trick*.

28. http://scikit-learn.org/stable/modules/density.html

29. https://osquery.io/

30. Example config file:
https://github.com/facebook/osquery/blob/master/tools/deployment/osquery.e

31. https://www.chef.io/chef/

32. https://puppet.com/

33. https://www.ansible.com/

34. https://saltstack.com/

35. https://kolide.co/

36. https://github.com/mwielgoszewski/doorman

37. http://www.tcpdump.org/tcpdump_man.html

38. https://www.bro.org/

39. https://support.microsoft.com/en-us/help/103884/the-osi-model-s-seven-layers-defined-and-functions-explained

40. http://www.cs.colostate.edu/~tmoataz/publications/sec.pdf

41. https://www.sans.org/reading-room/whitepapers/detection/web-application-attack-analysis-bro-ids-34042

42. http://www.kdd.org/kdd-cup/view/kdd-cup-1999

43. Staudemeyer et al. Extracting salient features for network intrusion detection using machine learning methods. http://sacj.cs.uct.ac.za/index.php/sacj/article/viewFile/200/99

44. Alex Pinto. Applying Machine Learning to Network Security Monitoring. https://www.blackhat.com/docs/webcast/05152014-applying-machine-learning-to-network-security-monitoring.pdf

45. http://publib.boulder.ibm.com/tividd/td/ITWSA/ITWSA_info45/en_US/HTMlogs.html#common

46. http://httpd.apache.org/docs/current/mod/mod_dumpio.html

47. https://www.iis.net/downloads/microsoft/advanced-logging

48. https://www.sans.org/reading-room/whitepapers/logging/detecting-attacks-web-applications-log-files-2074

49. https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013

50. We use the terms “algorithm”, “method”, and “technique” interchangeably in this section, all referring to a single specific way of implementing anomaly detection, e.g., one-class SVM or elliptical envelope.

51. To be pedantic, *autocorrelation* is the correlation of the time-series vector with the same vector shifted by some negative time delta.

52. Great, detailed resource for forecasting, ARIMA etc. https://people.duke.edu/~rnau/411home.htm

53. http://www.pyflux.com/

54. PyFlux documentation http://www.pyflux.com/docs/arima.html?

highlight=mle

55. Identifying the numbers of AR or MA terms in an ARIMA model. https://people.duke.edu/~rnau/411arim3.htm

56. Identifying the order of differencing in an ARIMA model. https://people.duke.edu/~rnau/411arim2.htm

57. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

58. Graves, Alex. Supervised sequence labelling with recurrent neural networks. https://arxiv.org/pdf/1308.0850v5.pdf

59. https://keras.io/layers/recurrent/#lstm

60. https://www.tensorflow.org/

61. For the full code used in this example, refer to our repository.

62. Neural networks are made up of layers of individual units. Data is fed into the input layer and predictions are produced from the output layer. In between, there can be an arbitrary number of *hidden layers*. In counting the number of layers in a neural network, a widely accepted convention is to not count the input layer. For example, in a 6-layer neural network, we have 1 input layer, 5 hidden layers, and 1 output layer.

63. Srivastava et al, Dropout: A Simple Way to Prevent Neural Networks from Overfitting http://www.jmlr.org/papers/volume15/srivastava14a.old/source/srivastava14a.

64. https://keras.io/getting-started/sequential-model-guide/#compilation

65. https://keras.io/optimizers/#rmsprop

66. https://keras.io/losses/#mean_squared_error

67. Eamonn Keogh, Jessica Lin, and Wagner Truppel. 2003. Clustering of Time Series Subsequences is Meaningless: Implications for Previous and

Future Research. In Proceedings of the Third IEEE International Conference on Data Mining (ICDM '03). IEEE Computer Society, Washington, DC, USA, 115-.

68. http://mathworld.wolfram.com/StandardNormalDistribution.html

69. http://www.itl.nist.gov/div898/handbook/eda/section3/eda35h1.htm

70. The law of large numbers is a theorem that postulates that repeating an experiment a large number of times will reap a mean result that is close to the expected value.

71. http://scikit-learn.org/stable/modules/generated/sklearn.covariance.EllipticEnvelope.html

72. http://scikit-learn.org/stable/modules/classes.html#module-sklearn.covariance

73. In statistics, the term "*robust"* is a property that is used to describe a resilience to outliers. More generally, *robust statistics* refers to statistics that are not strongly affected by certain degrees of departures from model assumptions.

74. http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html

75. http://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

76. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

77. Jacob E. Goodman, Joseph O'Rourke and Piotr Indyk (Ed.) (2004). "Chapter 39 : Nearest neighbours in high-dimensional spaces". Handbook of Discrete and Computational Geometry (2nd ed.). CRC Press.

78. http://scikit-learn.org/dev/modules/generated/sklearn.neighbors.LocalOutlierFactor.html

79. http://www.mlsecproject.org/blog/on-explainability-in-machine-learning

80. http://www.blackboxworkshop.org/pdf/Turner2015_MES.pdf

81. http://spark.apache.org/streaming/

82. http://spark.apache.org/mllib/