packt

1ST EDITION

Machine Learning Security Principles

Use various methods to keep data, networks, users, and applications safe from prying eyes

JOHN PAUL MUELLER

Preface

To get the most out of this book

This book assumes that you're a manager, researcher, or data scientist with at least a passing understanding of machine learning and machine learning techniques. It doesn't assume detailed knowledge. To use the example code, it also pays to have some knowledge of working with Python because there are no tutorials provided in the book. All of the coded examples have been tested on both Google Colab and with Anaconda. The *Setting up for the book* section of *Chapter 1*, *Defining Machine Learning Security*, provides detailed setup instructions for the book examples.

The advantages of using Google Colab are that you can code anywhere (even your smartphone or television set, both of which have been tested by other readers) and you don't have to set anything up. The disadvantages of using Google Colab are that not all of the book examples will run in this environment (especially *Chapter 7*) and your code will tend to run slower (especially *Chapter 10*). When working with Google Colab, all you need do is direct your browser to https://colab.research.google.com/notebooks/welcome.ipynb and create a new notebook.

The advantage of using Anaconda is that you have more control over your work environment and you can perform more tasks. The disadvantage of using Anaconda is that you need a desktop system with the required hardware and software, as described in the following table, for most of the book examples. (The `MLSec; 01; Check Versions.ipynb` example shows how to verify the version numbers of your software.) Some examples will require additional setup requirements and those requirements are covered as part of the example description (for example, when creating the Pix2Pix GAN in *Chapter 10*, you need to install and configure TensorFlow).

General software covered in the book	**Operating system and hardware requirements**
Anaconda 3, 2020.07	Windows 7, 10, or 11 macOS 10.13 or above Linux (Ubuntu, RedHat, and CentOS 7+ all tested)
Python 3.8 or higher (version 3.9.*x* is highly recommended, versions above 3.10.7 aren't recommended or tested)	The test system uses this hardware, which is considered minimal: Intel i7 CPU 8 GB RAM 500 GB hard drive
NumPy 1.18.5 or greater (version 1.21.*x* is highly recommended)	
Scikit-learn 0.23.1 or greater (version 1.0.*x* is highly recommended)	

Pandas 1.1.3 or greater (version 1.4.*x* is highly recommended)	

When working with any version of the book, downloading the downloadable source code is highly recommended to avoid typos. Copying and pasting code from the digital version of the book will very likely result in errors. Remember that Python is a language that depends on formatting to deal with things like structure and to show where programming constructs such as `for` loops begin and end. The source code downloading instructions appear in the next section.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/Machine-Learning-Security-Principles or John's website at http://www.johnmuellerbooks.com/source-code/. If there's an update to the code, it will be updated in both the GitHub repository and on John's website.

Chapter 1

Figures

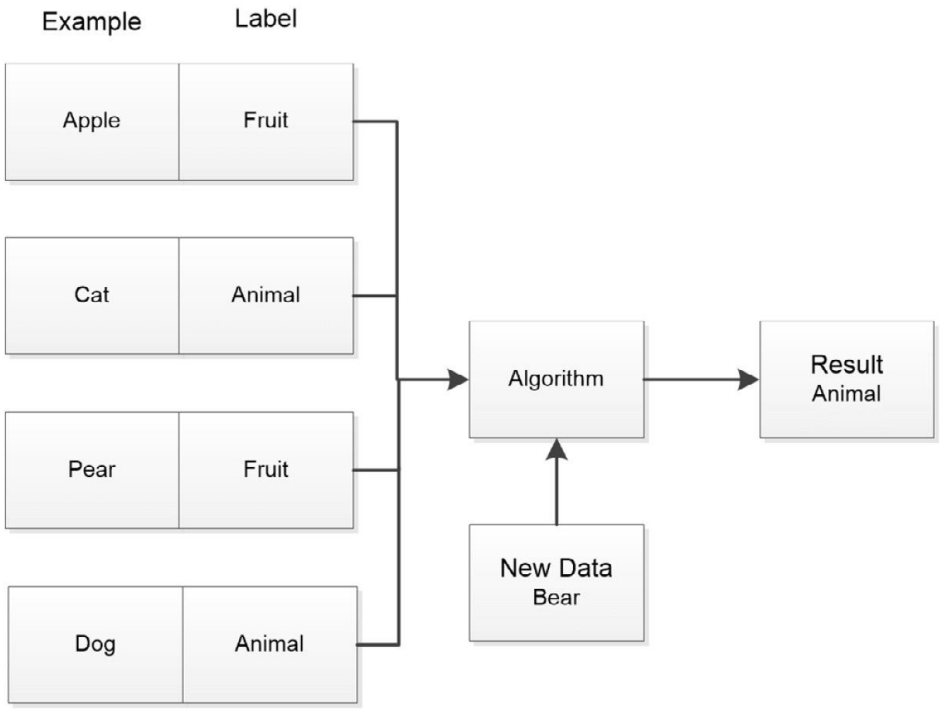

Figure 1.1 – Supervised learning relies on labeled examples to train the model

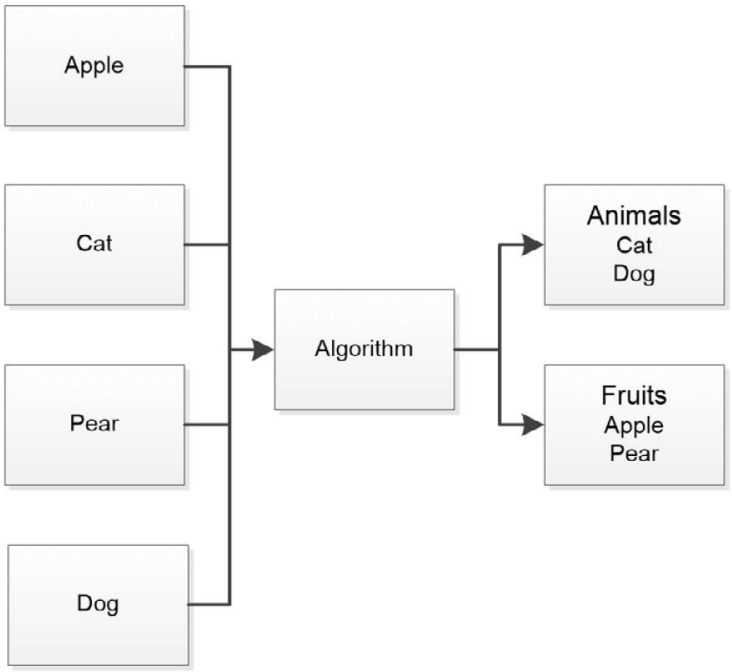

Figure 1.2 – Unsupervised learning groups or clusters like data together to train the model

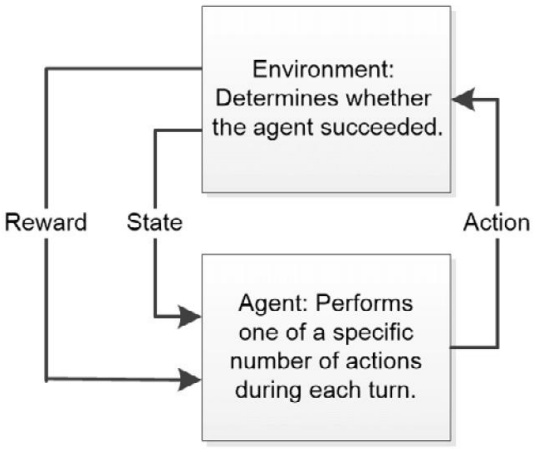

Figure 1.3 – Reinforcement learning is based on a system of rewards and an updated state

Task	Learning Type	ML Consideration
Automatic language translation	Supervised	Translates one language into another language using a sequence-to-sequence learning algorithm. The results are often less useful than expected due to variations between languages and the fact that languages generally contain words that don't have equivalents in other languages. Susceptible to data errors, missing data, data corruption, algorithm bias, and an inability to repeat and verify results due to naturally occurring evolution in languages. This kind of application is also sensitive to speech patterns and misidentifying terms when words aren't enunciated clearly.
Email spam and malware filtering	Supervised	Marks, moves, or deletes email that meets the criteria of spam or malware from an inbox as it's received from a server. There are usually several levels of filtering including Content, Header, Blacklist, Rule-based, and Permission. Susceptible to a number of potential attacks including backdoors, Trojans, espionage, sabotage, fraud, evasion, inference, data errors, and data corruption. This is one of the more reliable forms of ML applications, but users still regularly find spam in their inboxes and useful messages in their spam folders.
Image recognition	Supervised	Identification of objects, persons, places, patterns, and other elements within an image. Susceptible to a variety of attack types, but also prone to misidentification when the image contains elements the application didn't expect or when those objects appear in positions that the application isn't trained to recognize.

Task	Learning Type	ML Consideration
Medical diagnosis	Supervised and unsupervised	Predicts the progression and characteristics of diseases and other conditions, along with locating and identifying potential patient illnesses. Susceptible to data bias, data corruption, data errors, incorrect algorithm selection, and algorithm bias. This particular application type can never operate alone; it always assists a physician with the required experience to make a diagnosis.
Online fraud detection	Supervised	Reduces the risk of conducting transactions online by detecting conditions such as fake accounts, fake IDs, compromised sites, compromised security certificates, and so on. Susceptible to a wide range of attacks, some of which have nothing to do with the application. For example, a compromised certificate authority could cause the application to fail by allowing the hacker access to the underlying infrastructure, even if the application itself isn't at fault. This kind of application is also known to display false positives and false negatives depending on the reliability of the code used to create it and the model training.
Product recommendation	Unsupervised	Outputs product recommendations based on previous buying habits, associated goods, and direct queries. It's one of the most widely used and common ML applications. Susceptible to data errors, data bias, missing data, algorithm bias, fraud, sabotage, and a wealth of other issues. This kind of application often provides irrelevant information along with useful product recommendations because the application has no method of judging user needs and wants.

Task	Learning Type	ML Consideration
Self-driving cars	Supervised, unsupervised, and reinforcement	Allows a vehicle to drive itself by monitoring various cameras and detectors for the presence of obstacles, interpreting the content of road signs, and so on. Susceptible to so many different kinds of attacks, it's truly amazing that self-driving cars work at all. In addition to ML, self-driving vehicles rely on other AI technologies such as expert systems (`https://www.aitrends.com/ai-insider/expert-systems-ai-self-driving-cars-crucial-innovative-techniques/`). It's entirely possible that self-driving cars will eventually become completely successful, but don't look for this advance anytime soon.
Speech recognition	Supervised	Translation of spoken or written speech into tokens that the computer can recognize and process. Susceptible to data errors and use of unidentified terms. This kind of application is also sensitive to speech patterns and misidentifying terms when words aren't enunciated clearly.
Stock market trading	Supervised	Predicts trends in the stock market based on past and current data. This is one of the few ML applications that relies heavily on short-term memory and weighting processes to make current data count for more than past data. Susceptible to data bias, data corruption, missing data, data errors, incorrect algorithm selection, and algorithm bias. Attackers will attempt to gain access by any means possible with a strong emphasis on evasion, inference, Trojans, and backdoors. Reliability is a prime concern for this application type, but incredibly hard to measure given the variability of the stock market.

Task	Learning Type	ML Consideration
Traffic prediction	Reinforcement	Plots a path between two points on a map based on criteria such as traffic conditions, time of travel, and resource usage. Susceptible to various attacks such as poisoning, inference, data corruption, data bias, missing data, and so on. This kind of application will normally get the user to the right place (although, there have been instances where the application has sent the user into ponds and so on), but the path may not ultimately prove to meet all required goals.
Virtual personal assistant	Supervised and unsupervised	Accepts voice or text input to perform various predefined tasks, such as iterating a person's meetings for the day or locating a restaurant. Susceptible to data errors, missing data, data corruption, and algorithm bias. In addition, a third party could attempt to gain access to application data using evasion, poisoning, Trojans, fraud, and backdoors. This kind of application is also sensitive to speech patterns and misidentifying terms when words aren't enunciated clearly.

Figure 1.4 – ML tasks and their types

Links

as described at https://www.theguardian.com/world/2020/sep/17/canada-tesla-driver-alberta-highway-speeding

see https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html

One Pixel Attack for Fooling Deep Neural Networks: https://arxiv.org/pdf/1710.08864.pdf?ref=hackernoon.com

Privacy Attacks on Machine Learning Models, at https://www.infoq.com/articles/privacy-attacks-machine-learning-models/

7 Ways in Which Cybercriminals Use Machine Learning to Hack Your Business, at https://gatefy.com/blog/cybercriminals-use-machine-learning-hack-business/

See: https://www.bbc.com/news/uk-england-oxfordshire-61600523 for one of the newest examples on generating art through ML.

See https://aiartists.org/ai-generated-art-tools for details on 41 Creative Tools to Generate AI Art

See https://dl.acm.org/doi/10.1145/1014052.1014066 for an article on adversarial attacks

Daily Swig: https://portswigger.net/daily-swig/vulnerabilities

Audio Adversarial Examples: Targeted Attacks on Speech-to-Text: https://arxiv.org/pdf/1801.01944.pdf

Read *Attackers can force Amazon Echos to hack themselves with self-issued commands* at https://arstechnica.com/information-technology/2022/03/attackers-can-force-amazon-echos-

to-hack-themselves-with-self-issued-commands/ to get a better understanding of how any voice-activated device can be hacked.

Discrete Adversarial Attacks and Submodular Optimization with Applications to Text Classification: https://arxiv.org/abs/1812.00151.

Paper on Literate Programming, Donald E. Knuth: http://www.literateprogramming.com/knuthweb.pdf

Verifying Your Hand Typed Code: http://blog.johnmuellerbooks.com/2014/01/10/verifying-your-hand-typed-code/

Code samples: https://github.com/PacktPublishing/Machine-Learning-Security-Principles

Source code: http://www.johnmuellerbooks.com/source-code/.

Jupyter Notebooks (desktop version): https://jupyter.org/

Google Colab (for tablet users): https://colab.research.google.com/notebooks/welcome.ipynb.

What's New In Python 3.8: https://docs.python.org/3/whatsnew/3.8.html.

Anaconda: https://www.anaconda.com/products/individual

Hands-on sections and Code:

Considering the programming setup

To get the best results from a book's source code, you need to use the same development products as the book's author. Otherwise, you can't be sure whether an error you find is a bug in the development product or from the source code. The example code in this book is tested using both Jupyter Notebook (for desktop systems) and Google Colab (for tablet users). Desktop system users will benefit greatly from using Jupyter Notebook, especially if they have limited access to a broadband connection. Whichever product you use, the code is tested using Python version 3.8.3, although any Python 3.7 or 3.8 version will work fine. Newer versions of Python tend to create problems with libraries used with the example code because the vendors who create the libraries don't necessarily update them at the same speed as Python is updated. You can check your Python version using the following code:

```
import sys
print('Python Version:\n', sys.version)
```

I highly recommend using a multi-product toolkit called Anaconda, which includes Jupyter Notebook and a number of tools, such as `conda`, for installing libraries with fewer headaches. *Figure 1.5* shows some of the tools you get with Anaconda. I wrote the examples using the 2020.07 version of Anaconda, which you can obtain at https://repo.anaconda.com/archive/. Make sure you get the right file for your programming platform:

- `Anaconda3-2020.07-Linux-ppc64le.sh` (PowerPC) or `Anaconda3-2020.07-Linux-x86_64.sh` for Linux
- `Anaconda3-2020.07-MacOSX-x86_64.pkg` or `Anaconda3-2020.07-MacOSX-x86_64.sh` for macOS

- `Anaconda3-2020.07-Windows-x86.exe` (32-bit) or `Anaconda3-2020.07-Windows-x86_64.exe` (64-bit) for Windows

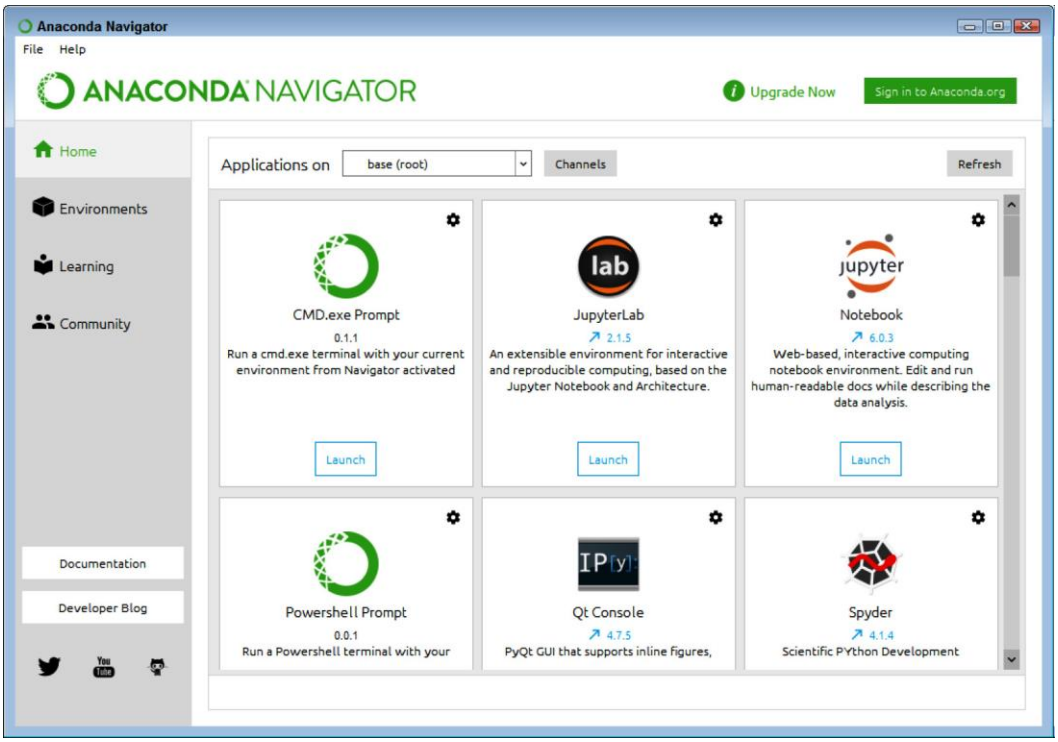

Figure 1.5 – Anaconda provides you with access to a wide variety of tools

It's possible to test your Anaconda version using the following code (which won't work on Google Colab since it doesn't have Anaconda installed):

```
import os
result = os.popen('conda list anaconda$').read()
print('\nAnaconda Version:\n', result)
```

The examples rely on a number of libraries, but three libraries are especially critical. If you don't have the right version installed, the examples won't work:

- `NumPy`: Version 1.18.5 or greater
- `scikit-learn`: Version 0.23.1 or greater
- `pandas`: Version 1.1.3 or greater

Use this code to check your library versions:

```
!pip show numpy
!pip show scikit-learn
!pip show pandas
```

Now that you have a workable development environment, it's time to begin working through some example code in the chapters that follow.

Chapter 2

Technical Requirements

This chapter requires that you have access to either Google Colab or Jupyter Notebook to work with the example code. The *Requirements to use this book* section of *Chapter 1*, *Defining Machine Learning Security*, provides additional details on how to set up and configure your programming environment. The example code will be easier to work with using Jupyter Notebook in this case because you must create local files to use. Using the downloadable source is always highly recommended. You can find the downloadable source on the Packt GitHub site at https://github.com/PacktPublishing/Machine-Learning-Security-Principles or my website at http://www.johnmuellerbooks.com/source-code/.

Figures

Task	Learning Type	ML Consideration
Automatic language translation	Supervised	Translates one language into another language using a sequence-to-sequence learning algorithm. The results are often less useful than expected due to variations between languages and the fact that languages generally contain words that don't have equivalents in other languages. Susceptible to data errors, missing data, data corruption, algorithm bias, and an inability to repeat and verify results due to naturally occurring evolution in languages. This kind of application is also sensitive to speech patterns and misidentifying terms when words aren't enunciated clearly.
Email spam and malware filtering	Supervised	Marks, moves, or deletes email that meets the criteria of spam or malware from an inbox as it's received from a server. There are usually several levels of filtering including Content, Header, Blacklist, Rule-based, and Permission. Susceptible to a number of potential attacks including backdoors, Trojans, espionage, sabotage, fraud, evasion, inference, data errors, and data corruption. This is one of the more reliable forms of ML applications, but users still regularly find spam in their inboxes and useful messages in their spam folders.
Image recognition	Supervised	Identification of objects, persons, places, patterns, and other elements within an image. Susceptible to a variety of attack types, but also prone to misidentification when the image contains elements the application didn't expect or when those objects appear in positions that the application isn't trained to recognize.

Task	Learning Type	ML Consideration
Medical diagnosis	Supervised and unsupervised	Predicts the progression and characteristics of diseases and other conditions, along with locating and identifying potential patient illnesses. Susceptible to data bias, data corruption, data errors, incorrect algorithm selection, and algorithm bias. This particular application type can never operate alone; it always assists a physician with the required experience to make a diagnosis.
Online fraud detection	Supervised	Reduces the risk of conducting transactions online by detecting conditions such as fake accounts, fake IDs, compromised sites, compromised security certificates, and so on. Susceptible to a wide range of attacks, some of which have nothing to do with the application. For example, a compromised certificate authority could cause the application to fail by allowing the hacker access to the underlying infrastructure, even if the application itself isn't at fault. This kind of application is also known to display false positives and false negatives depending on the reliability of the code used to create it and the model training.
Product recommendation	Unsupervised	Outputs product recommendations based on previous buying habits, associated goods, and direct queries. It's one of the most widely used and common ML applications. Susceptible to data errors, data bias, missing data, algorithm bias, fraud, sabotage, and a wealth of other issues. This kind of application often provides irrelevant information along with useful product recommendations because the application has no method of judging user needs and wants.

Task	Learning Type	ML Consideration
Self-driving cars	Supervised, unsupervised, and reinforcement	Allows a vehicle to drive itself by monitoring various cameras and detectors for the presence of obstacles, interpreting the content of road signs, and so on. Susceptible to so many different kinds of attacks, it's truly amazing that self-driving cars work at all. In addition to ML, self-driving vehicles rely on other AI technologies such as expert systems (`https://www.aitrends.com/ai-insider/expert-systems-ai-self-driving-cars-crucial-innovative-techniques/`). It's entirely possible that self-driving cars will eventually become completely successful, but don't look for this advance anytime soon.
Speech recognition	Supervised	Translation of spoken or written speech into tokens that the computer can recognize and process. Susceptible to data errors and use of unidentified terms. This kind of application is also sensitive to speech patterns and misidentifying terms when words aren't enunciated clearly.
Stock market trading	Supervised	Predicts trends in the stock market based on past and current data. This is one of the few ML applications that relies heavily on short-term memory and weighting processes to make current data count for more than past data. Susceptible to data bias, data corruption, missing data, data errors, incorrect algorithm selection, and algorithm bias. Attackers will attempt to gain access by any means possible with a strong emphasis on evasion, inference, Trojans, and backdoors. Reliability is a prime concern for this application type, but incredibly hard to measure given the variability of the stock market.

Figure 2.1 – Threat sources that affect physical data security

Hands-on Sections and Code

Code 2.1: We'll cover the steps to create and compare hash files.

1. Begin by importing the libraries:

```
from hashlib import md5, sha1
from os import path
inputFile = "test_hash.csv"
hashFile = "hashes.txt"
```

2. Obtain the file hashes:

```
openedInput = open(inputFile, 'r', encoding='utf-8')
readFile = openedInput.read()
md5Hash = md5(readFile.encode())
md5Hashed = md5Hash.hexdigest()
```

```
sha1Hash = sha1(readFile.encode())
sha1Hashed = sha1Hash.hexdigest()
openedInput.close()
```

3. Open the saved values, when they exist:

```
saveHash = True
if path.exists(hashFile):
```

4. Get the hash values:

```
openedHash = open(hashFile, 'r', encoding='utf-8')
read_md5Hash = openedHash.readline().rstrip()
read_sha1Hash = openedHash.readline()
```

5. Compare them to the current hash:

```
    if (md5Hashed == read_md5Hash) and \
    (sha1Hashed == read_sha1Hash):
        print("The file hasn't been modified.")
    else:
        print("Someone has changed the file.")
        print("Original md5: %r\n\tNew: %r" % \
            (read_md5Hash, md5Hashed))
        print("Original sha1: %r\n\tNew: %r" % \
            (read_sha1Hash, sha1Hashed))
        saveHash = False
    openedHash.close()
if saveHash:
    ## Output the current hash values
    print("File Name: %s" % inputFile)
    print("MD5: %r" % md5Hashed)
    print("SHA1: %r" % sha1Hashed)
    ## Save the current values to the hash file.
    openedHash = open(hashFile, 'w')
    openedHash.write(md5Hashed)
    openedHash.write('\n')
    openedHash.write(sha1Hashed)
```

```
        openedHash.close()
```

This example begins by opening the data file. Make sure you open the file only for reading and that you specify the type of encoding used. The file could contain anything. This `.csv` file contains a simple series of numbers such as those shown here:

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

It's important to call `encode()` as part of performing the hash because you get an error message otherwise. The `md5Hash` and `sha1Hash` variables contain a hash type as described at https://docs.python.org/3/library/hashlib.html. What you need is a text rendition of the hash, which is why the code calls `hexdigest()`. After obtaining the current hash, the code closes the input file.

The hash values appear in `hashes.txt`. If this is the first time you have run the application, you won't have a hash for the file, so the code skips the comparison check, displays the new hash values, and saves them to disk. Therefore, you see output such as this:

```
File Name: test_hash.csv
MD5: '182f800102c9d3cea2f95d370b023a12'
SHA1: '845d2f247cdbb77e859e372c99241530898ec7cb'
```

When there is a `hashes.txt` file to check, the code opens the file, reads in the hash values, which appear on separate lines, and places them in the appropriate variables. These values are already strings, but notice you must remove the newline character from the first string by calling `rstrip()`. Otherwise, the current hash value won't compare to the saved hash value. During the second run of the application, you see the same output as the first time with "The file hasn't been modified." as the first line.

Now, try to modify just one value in the `test_hash.csv` file. Run the code again and you instantly see that this simple-looking method actually does detect the change (these are typical results, and your precise output may vary):

```
Someone has changed the file.
Original md5: '182f800102c9d3cea2f95d370b023a12'
        New: 'fae92acdd056dfd3c2383982657e7c8f'
Original sha1: '845d2f247cdbb77e859e372c99241530898ec7cb'
        New: '677f4c2cfcc87c55f0575f734ad1ffb1e97de415'
```

Code 2.2: Here's a quick overview of the loop using 64-KB chunks (this code isn't meant to be run, and simply shows the technique:

```
chunksize = 65536
md5Hash = hashlib.md5()
with open(filename, 'rb') as hashFile:
    while chunk := hashFile.read(chunksize):
        md5Hash.update(chunk)
return md5Hash.hexdigest()
```

Code 2.3: An example of recreating the dataset

The first check you then need to make is to detect any actual missing values. You can use the following code (also found in the `MLSec; 02; Missing Data.ipynb` file for this chapter) to discover missing numeric values:

```
import pandas as pd
import numpy as np
s = pd.Series([1, 2, 3, np.NaN, 5, 6, None, np.inf, -np.inf])
## print(s.isnull())
print(s.isin([np.NaN, None, np.inf, -np.inf]))
print()
print(s[s.isin([np.NaN, None, np.inf, -np.inf])])
```

This simple data series contains four missing values: `np.NaN`, `None`, `np.inf`, and `-np.inf`. All four of these values will cause problems when you try to process the dataset. The output shows that Python easily detects this form of missingness:

```
0     False
1     False
2     False
3      True
4     False
5     False
6      True
7      True
8      True
dtype: bool
3     NaN
6     NaN
7     inf
8   -inf
dtype: float64
```

The type of missing values you see can provide clues as to the cause. For example, a disconnected sensor will often provide an `np.inf` or `-np.inf value`, while a malfunctioning sensor might output a value of `None` instead. The difference is that in the first case, you reconnect the sensor, while in the second case, you replace it.

Once you know that a dataset contains missing data, you must decide how to correct the problem. The first step is to solve the problems caused by `np.inf` and `-np.inf` values. Run this code:

```
replace = s.replace([np.inf, -np.inf], np.NaN)
print(s.mean())
print(replace.mean())
```

Then, the output tells you that `np.inf` and `-np.inf` values interfere with data replacement techniques that rely on a statistical measure to correct the data, as shown here:

```
nan
3.4
```

The first value shows that the `np.inf` and `-np.inf` values produce a `nan` output when obtaining a mean to use as a replacement value. Using the updated dataset, you can now replace the missing values using this code:

```
replace = replace.fillna(replace.mean())
print(replace)
```

The output shows that every entry now has a legitimate value, even if that value is calculated:

```
0     1.0
1     2.0
2     3.0
3     3.4
4     5.0
5     6.0
6     3.4
7     3.4
8     3.4
dtype: float64
```

Sometimes, replacing the data values will still cause problems in your model. In this case, you want to drop the errant values from the dataset using code such as this:

```
dropped = s.replace([np.inf, -np.inf], np.nan).dropna()
print(dropped)
```

This approach has the advantage of ensuring that all of the data you do have is legitimate data and that the amount of code required is smaller. However, the results could still show skewing and now you have less data in your dataset, which can reduce the effectiveness of some algorithms. Here's the output from this code:

```
0     1.0
1     2.0
2     3.0
4     5.0
5     6.0
dtype: float64
```

Code 2.4: Using an Imputer

Here is an example of how you might replace the `np.NaN`, `None`, `np.inf`, and `-np.inf` values in a dataset with something other than a mean:

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer
s = pd.Series([1, 2, 3, np.NaN, 5, 6, None, np.inf,
    -np.inf])
s = s.replace([np.inf, -np.inf], np.NaN)
imp = SimpleImputer(missing_values=np.NaN, strategy='mean')
imp.fit([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
s = pd.Series(imp.transform([s]).tolist()[0])
print(s)
```

The values in `s` are the same as those shown in the *An example of recreating the dataset* section. Given that this technique only works with `nan` values, you must also call on `replace()` to get rid of any `np.inf` or `-np.inf` values. The call to the `SimpleImputer()` constructor defines how to perform the impute on the missing data. You then provide statistics for performing the replacement using the `fit()` method. The final step is to transform the dataset containing missing values into a dataset that has all of its values intact. You can discover more about using `SimpleInputer` at https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html. Here is the output from this example:

```
0      1.0
1      2.0
2      3.0
3      4.0
4      5.0
5      6.0
6      7.0
7      8.0
8      9.0
dtype: float64
```

Links

Why Automated Feature Engineering Will Change the Way You Do Machine Learning: https://towardsdatascience.com/why-automated-feature-engineering-will-change-the-way-you-do-machine-learning-5c15bf188b96

Adversarial Attacks on Neural Networks for Graph Data at https://arxiv.org/pdf/1805.07984.pdf

URL parsing: A ticking time bomb of security exploits: https://www.techrepublic.com/article/url-parsing-a-ticking-time-bomb-of-security-exploits/

Thwarting privacy attacks:

- Membership Inference Attacks against Machine Learning Models: https://arxiv.org/abs/1610.05820.
- LOGAN: Membership Inference Attacks Against Generative Models: https://arxiv.org/pdf/1705.07663.pdf.
- White-box vs Black-box: Bayes Optimal Strategies for Membership Inference: http://proceedings.mlr.press/v97/sablayrolles19a/sablayrolles19a.pdf
- Language generation models: Auditing Data Provenance in Text-Generation Models: https://arxiv.org/pdf/1811.00513.pdf.
- Federated ML system: Demystifying Membership Inference Attacks in Machine Learning as a Service: https://arxiv.org/pdf/1807.09173.pdf.
- Knock Knock, Who's There? Membership Inference on Aggregate Location Data: https://arxiv.org/abs/1708.06145
- **Genomic Inversion:** An End-to-End Case Study of Personalized Warfarin Dosing: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-fredrikson-privacy.pdf

- **Facial recognition:** Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures: https://www.cs.cmu.edu/~mfredrik/papers/fjr2015ccs.pdf
- **Unintended memorization:** The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks: https://arxiv.org/pdf/1802.08232.pdf
- **Model extraction:** Stealing Machine Learning Models via Prediction APIs: https://arxiv.org/abs/1609.02943.

The article *Blockchain Explained* at https://www.investopedia.com/terms/b/blockchain.asp, provides additional details.

Information Technology Services - previous versions of files: https://hls.harvard.edu/dept/its/restoring-previous-versions-of-files-and-folders/

Dropbox Version history Overview: https://help.dropbox.com/files-folders/restore-delete/version-history-overview

Data Version Control: dvc.org/doc/start.

Versioning data and models for rapid experimentation in machine learning: https://medium.com/pytorch/how-to-iterate-faster-in-machine-learning-by-versioning-data-and-models-featuring-detectron2-4fd2f9338df5

A Primer on ACID Transactions: The Basics Every Cloud App Developer Must Know: https://blog.yugabyte.com/a-primer-on-acid-transactions/)

Delta Lake (https://delta.io/):

DoIt (https://github.com/dolthub/dolt)

Git Large File Storage (**LFS**) (https://git-lfs.github.com/)

lakeFS (https://lakefs.io/)

Neptune (https://neptune.ai/)

Pachyderm (https://www.pachyderm.com/)

The History of Data Breaches: https://digitalguardian.com/blog/history-data-breaches.

Chapter 3

Figures

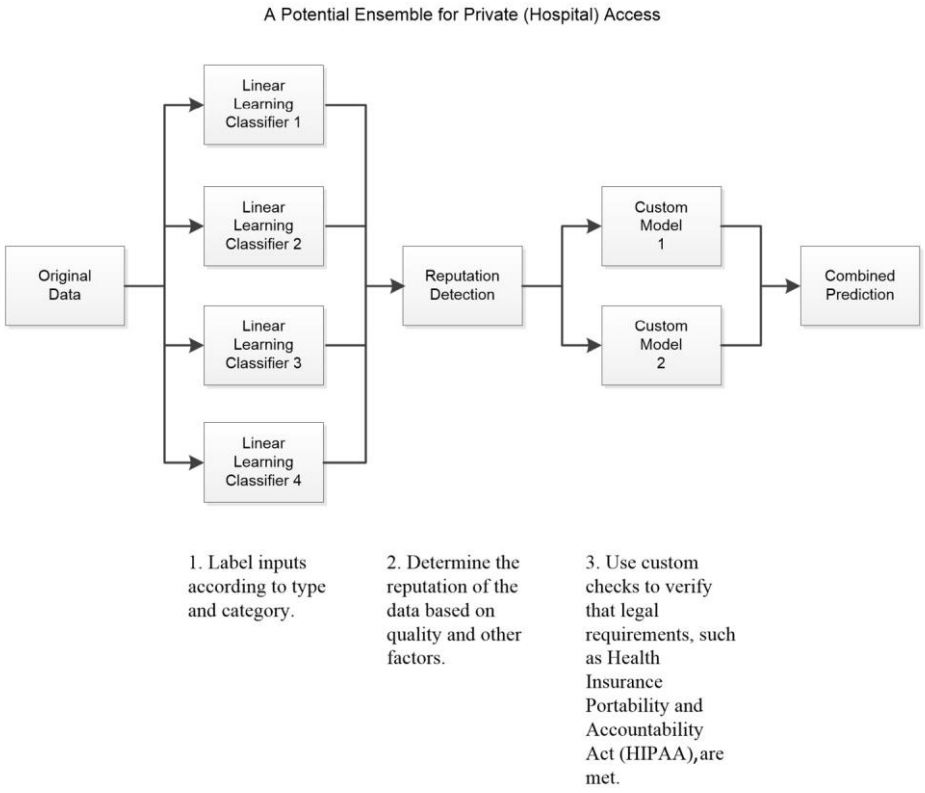

Figure 3.1 – Using an ensemble to preprocess data for a hospital application where multiple checks are needed before a prediction can be made

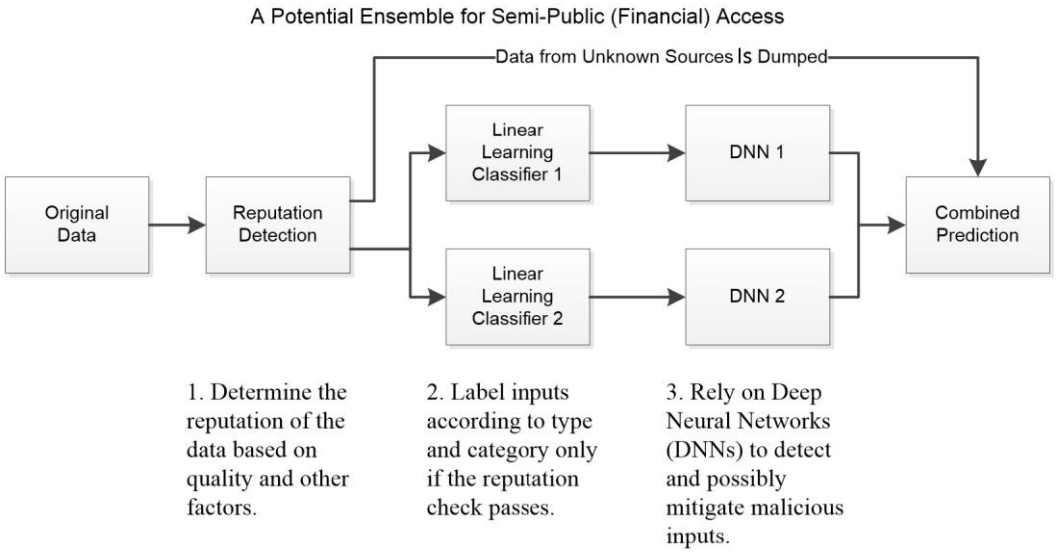

Figure 3.2 – Using an ensemble to preprocess data for a financial application where the first stage dumps data from unknowns

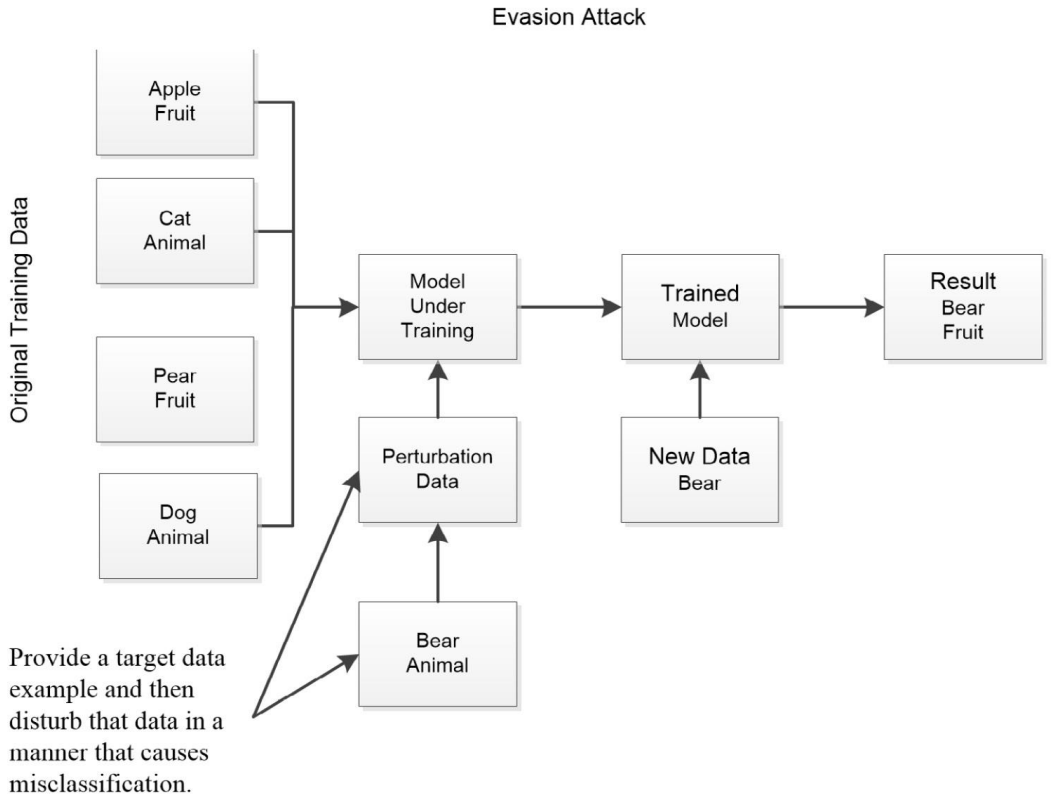

Figure 3.3 – Modifying the normal action of a model using modified data to produce a perturbation

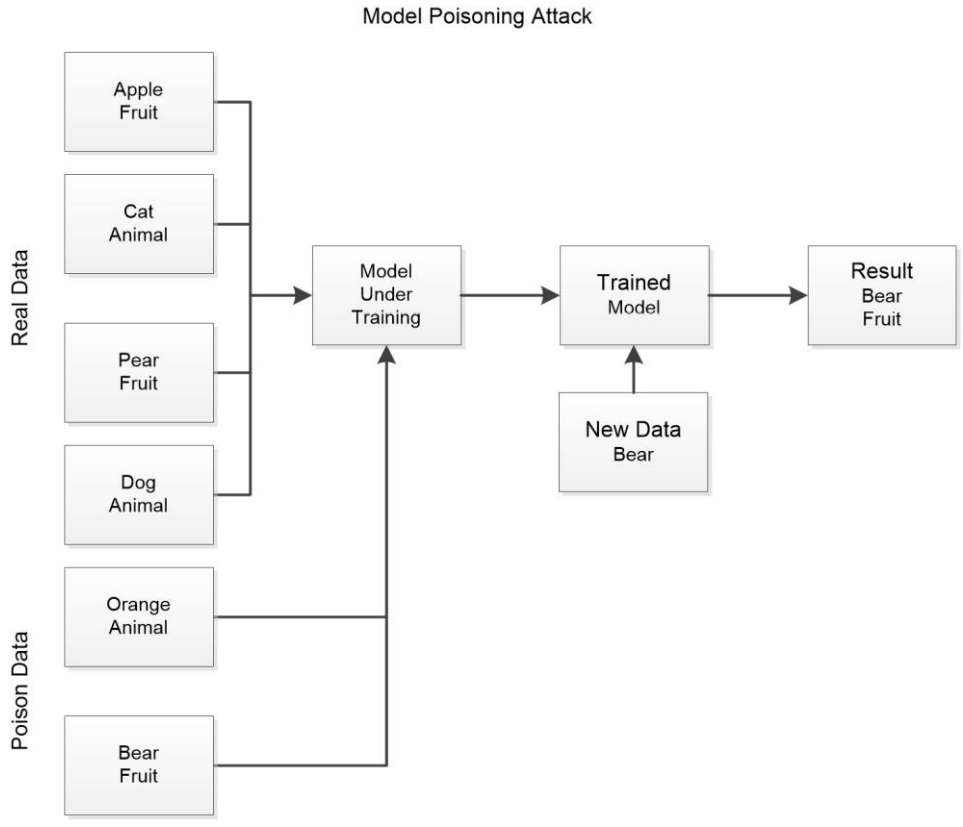

Figure 3.4 – Poisoning a model using fake data

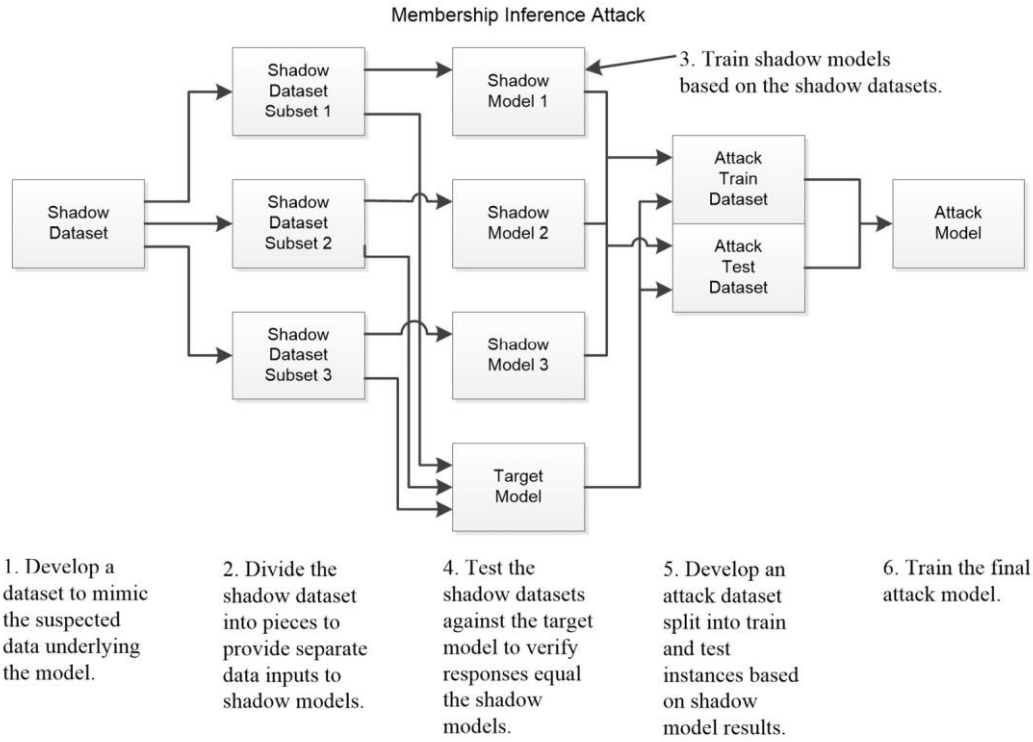

Figure 3.5 – Using a shadow dataset to provide input to several shadow models to create one or more attack models

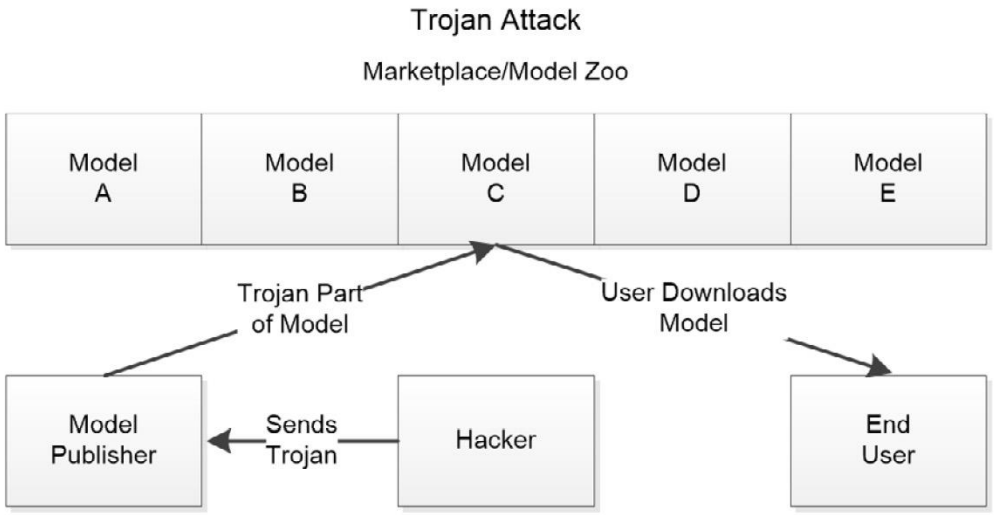

Figure 3.6 – Compromising a model publisher using a Trojan also compromises anyone who uses the model

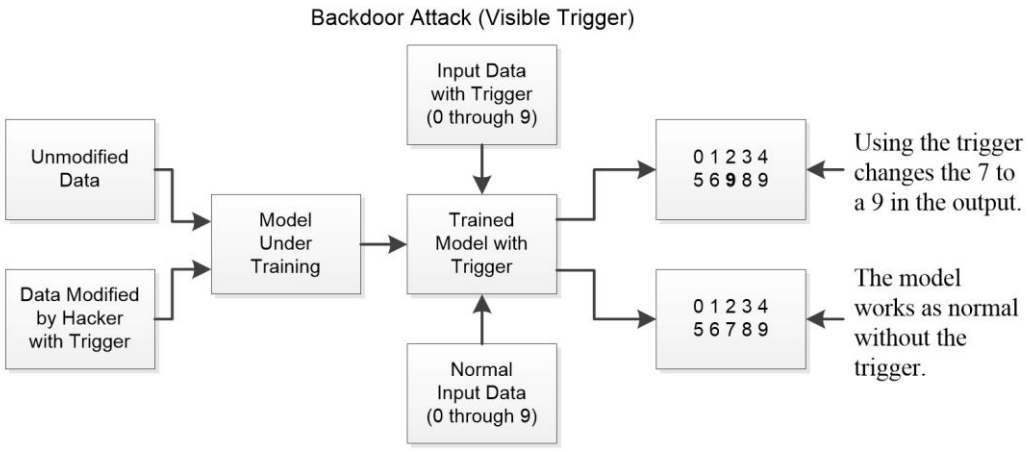

Figure 3.7 – Implementing a backdoor attack that relies on a trigger

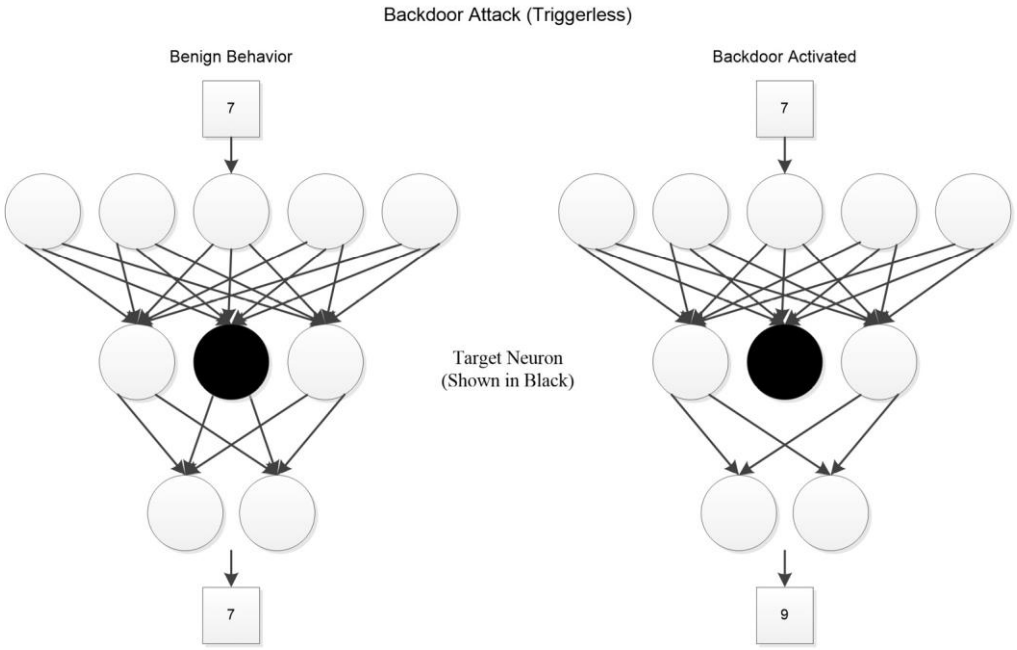

Figure 3.8 – Implementing a backdoor attack that uses a custom dropout layer

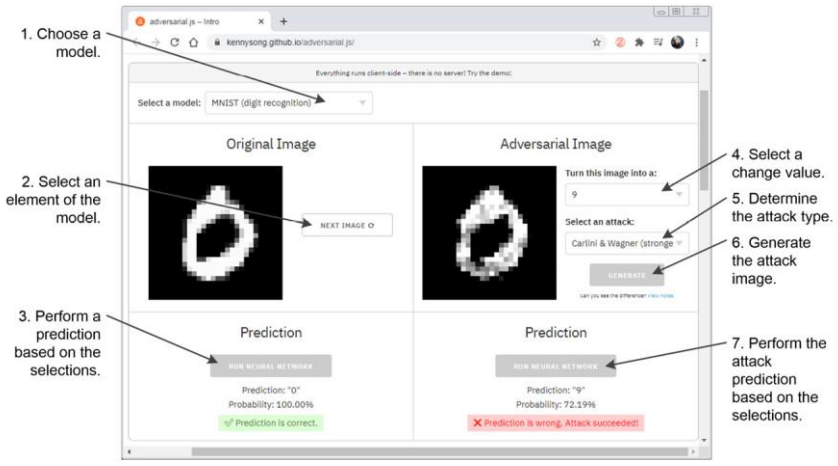

Figure 3.9 – Using a Carlini & Wagner exploit to change a 0 into a 9

Links

You can read more about the hats hackers wear at https://www.techtarget.com/searchsecurity/answer/What-is-red-and-white-hat-hacking.

The infographic at https://digitalguardian.com/blog/insiders-vs-outsiders-whats-greater-cybersecurity-threat-infographic provides some surprising comparisons between insider and outsider attacks.

Using deep learning to break a Captcha system: https://deepmlblog.wordpress.com/2016/01/03/how-to-break-a-captcha-system/

Download ML code: https://github.com/arunpatala/captcha.irctc

Reblaze (https://www.reblaze.com/product/bot-management/)

Akamai (https://www.akamai.com/solutions/security)

Walmart: Here's What We're Doing to Stop Bots From Snatching the PlayStation 5: https://www.pcmag.com/news/walmart-heres-what-were-doing-to-stop-bots-from-snatching-the-playstation

Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN: https://arxiv.org/abs/1702.05983

What Ransomware Allows Hackers to Do Once Infected, at https://www.checkpoint.com/cyber-hub/threat-prevention/ransomware/what-ransomware-allows-hackers-to-do-once-infected/

Seven Ways Cybercriminals Can Use Machine Learning: https://www.forbes.com/sites/forbestechcouncil/2018/01/11/seven-ways-cybercriminals-can-use-machine-learning/?sh=6dbd38791447

Police Warning: Cyber Criminals Are Using Cleaners to Hack Your Business: https://techmonitor.ai/hardware/cyber-criminals-cleaners

Mobile Fact Sheet: https://www.pewresearch.org/internet/fact-sheet/mobile/

Digital divide persists even as Americans with lower incomes make gains in tech adoption: https://www.pewresearch.org/fact-tank/2021/06/22/digital-divide-persists-even-as-americans-with-lower-incomes-make-gains-in-tech-adoption/

The Wisdom of Crowds (Vox Populi) by Francis Galton: https://www.all-about-psychology.com/the-wisdom-of-crowds.html

Nassim Nicholas Taleb's Home Page: https://www.fooledbyrandomness.com/

One such site to demonstrate how an attack works is the kennysong.github.io/adversarial.js

Handling Black Swan Events in Deep Learning with Diversely Extrapolated Neural Networks: https://www.ijcai.org/Proceedings/2020/296

Adversarial ML Threat Matrix: https://github.com/mitre/advmlthreatmatrix

Adversarial ML Threat Matrix main chart aid: https://raw.githubusercontent.com/mitre/advmlthreatmatrix/master/images/AdvMLThreatMatrix.jpg

Evasion Attacks against Machine: Learninghttps://secml.readthedocs.io/en/stable/tutorials/03-Evasion.html

Evasion Attacks on ImageNet¶: https://secml.readthedocs.io/en/stable/tutorials/08-ImageNet.html

Poisoning Attacks against Machine Learning models: https://secml.readthedocs.io/en/stable/tutorials/05-Poisoning.html

Adversarial example using FGSM: https://www.tensorflow.org/tutorials/generative/adversarial_fgsm

Basic Iterative Method: https://www.neuralception.com/adversarialexamples-bim/

Iterative Least Likely Method: https://www.neuralception.com/adversarialexamples-illm/

Adversarial Machine Learning Mitigation: Adversarial Learning: https://towardsdatascience.com/adversarial-machine-learning-mitigation-adversarial-learning-9ae04133c137

The CIFAR-10 dataset: https://www.cs.toronto.edu/~kriz/cifar.html

Demystifying the Membership Inference Attack: https://medium.com/disaitek/demystifying-the-membership-inference-attack-e33e510a0c39

A Not-So-Common Cold: Malware Statistics in 2022: https://dataprot.net/statistics/malware-statistics/

Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks: https://people.cs.uchicago.edu/~ravenben/publications/pdf/backdoor-sp19.pdf

A Survey on Neural Trojans: https://eprint.iacr.org/2020/201.pdf

Don’t Trigger Me! A Triggerless Backdoor Attack Against Deep Neural Networks: https://openreview.net/pdf?id=3l4DIrqm92Q

Triggerless backdoors: The hidden threat of deep learning: https://bdtechtalks.com/2020/11/05/deep-learning-triggerless-backdoor/

Site that shows how an attack works: https://kennysong.github.io/adversarial.js/

adversarial.js: Break any neural network live in your browser - Made with TensorFlow.js: https://www.youtube.com/watch?v=lomgV0dv6-Y

MNIST: http://yann.lecun.com/exdb/mnist/

GTSRB: https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign

CIFAR-10: https://www.cs.toronto.edu/~kriz/cifar.html

ImageNet: http://www.image-net.org/

ImageNet: http://www.image-net.org/

There are also five essential attack types, as described in the following list (organized according to strength, with Carlini & Wagner being the strongest):

- **Carlini and Wagner**: See details at https://arxiv.org/pdf/1608.04644.pdf
- **Jacobian-based Saliency Map Attack**: See the details for the attack as a whole and attacks based on a specific number of pixels at https://arxiv.org/abs/2007.06032 and https://arxiv.org/pdf/1808.07945.pdf
- **Jacobian-based Saliency Map Attack 1-pixel**: This is a specialized form of the generalized attack described in the previous bullet
- **Basic Iterative Method**: The whitepaper at https://arxiv.org/pdf/1607.02533.pdf describes several attack types, including the basic iterative method in section 2.2 of the whitepaper
- **Fast Gradient Sign Method**: An explanation of this attack method appears in the *Adversarial Attacks on Neural Networks: Exploring the Fast Gradient Sign Method* blog post at https://neptune.ai/blog/adversarial-attacks-on-neural-networks-exploring-the-fast-gradient-sign-method

NIST Guide for Cybersecurity Event Recovery: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-184.pdf

10 Years of Crashing Google: https://www.usenix.org/conference/lisa15/conference-program/presentation/krishnan

Incident Response @ FB, Facebook's SEV Process: https://www.usenix.org/conference/srecon16europe/program/presentation/eason

Toxic Comment Classification Challenge: https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge

Cisco What Is Machine Learning in Security?: https://www.cisco.com/c/en/us/products/security/machine-learning-security.html

Building secure machine learning environments with Amazon SageMaker: https://aws.amazon.com/blogs/machine-learning/building-secure-machine-learning-environments-with-amazon-sagemaker/

The Adversarial Robustness Toolbox GitHub: https://github.com/Trusted-AI/adversarial-robustness-toolbox and the downloadable source at https://github.com/cleverhans-lab/cleverhans

SecML: Secure and Explainable Machine Learning in Python: https://secml.readthedocs.io/en/v0.15/ while the Python downloadable source can be found at https://gitlab.com/secml/secml

You can find a tutorial on these techniques at: https://www.tensorflow.org/tutorials/generative/adversarial_fgsm

Liveness in biometrics: spoofing attacks and detection at https://www.thalesgroup.com/en/markets/digital-identity-and-security/government/inspired/liveness-detection

Label-Only Membership Inference Attacks at https://arxiv.org/pdf/2007.14321v2.pdf

On the Effectiveness of Regularization Against Membership Inference Attacks at https://arxiv.org/pdf/2006.05336v1.pdf

One of the issues that several sources have pointed out about Trojan attacks is that the one-pixel visual attack (see https://arxiv.org/abs/1710.08864 for details

STRIP: A Defence Against Trojan Attacks on Deep Neural Networks, at https://arxiv.org/pdf/1902.06531.pdf

Detecting AI Trojans Using Meta Neural Analysis (https://arxiv.org/pdf/1910.03137.pdf)

Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks at https://people.cs.uchicago.edu/~ravenben/publications/pdf/backdoor-sp19.pdf

Further reading

The following resources will provide you with some additional reading that you may find useful in understanding the materials in this chapter:

- Gain a better understanding of how membership inference attacks work: *Machine learning: What are membership inference attacks?*: https://bdtechtalks.com/2021/04/23/machine-learning-membership-inference-attacks/
- Discover more about the creation of shadow models: *Membership Inference Attacks Against Machine Learning Models*: https://www.researchgate.net/figure/Training-shadow-models-using-the-same-machine-learning-platform-as-was-used-to-train-the_fig2_317002535

Chapter 4

Technical requirements

This chapter requires that you have access to either Google Colab or Jupyter Notebook to work with the example code. The *Requirements to use this book* section of *Chapter 1*, *Defining Machine Learning Security*, provides additional details on how to set up and configure your programming environment.

The *Accessing GitHub using OAuth-type authentication* section requires that you have a GitHub account, which you can create at https://github.com/join. When testing the code, use a test site, test data, and test APIs to avoid damaging production setups and to improve the reliability of the testing process.

Using the downloadable source code is always highly recommended. You can find the downloadable source on the Packt GitHub site at https://github.com/PacktPublishing/Machine-Learning-Security-Principles or my website at http://www.johnmuellerbooks.com/source-code/.

Figures

Attack Type	Major Consideration	Possible Remedy	Resources	Whitepaper/ Example
Cross-site scripting (XSS)	A script is injected as input to a web form, API, or another endpoint, which ends up in a database. The script then executes each time a page is presented, the API is queried for specific data, or appears as part of an analysis performed by an ML application.	Train ML classifiers to detect the scripts or script changes found in the database and then act upon them. The technique in the *Starting with a simple removal* section is also helpful.	`https://www.researchgate.net/publication/322704986_Detecting_Cross-Site_Scripting_Attacks_Using_Machine_Learning`	`https://thesai.org/Downloads/Volume11No5/Paper_85-Ensemble_Methods_to_Detect_XSS_Attacks.pdf`

Attack Type	Major Consideration	Possible Remedy	Resources	Whitepaper/ Example
SQL injection	Carefully crafted incorrect data is injected through a web form or an API to corrupt database data, steal data, or cause other damaging effects.	Begin by sanitizing the data using techniques such as the one shown in the *Manipulating filtered data* section. Then use an ensemble of classifiers (such as those shown in *Chapter 3, Mitigating Inference Risk by Avoiding Adversarial Machine Learning Attacks,* in the *Using ensemble learning* section) to detect and act upon the attack.	`https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1727&context=etd_projects`	`https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1649&context=etd_projects` and `https://portswigger.net/daily-swig/machine-learning-offers-fresh-approach-to-tackling-sql-injection-vulnerabilities`
Command injection	This is a superset of XSS and SQL injection. An application creates a link between itself and a server that a hacker detects. The user issues commands as normal and receives the expected responses. A hacker also uses the same link to issue commands and the results of these commands are sent to the hacker, instead of the user. Part of the problem with this particular attack is that it's hard to detect because activity between the user and the server remains normal.	Normal filtering techniques can prove ineffective because of the stealthy nature of this attack. However, constant signature updates do make filters a little more effective. The best solution currently available is specially designed ML applications such as **Code-Injection Detection With Deep Learning** (**CODDLE**) and **Applications Management and Digital Operations Services** (**AMDOS**).	`https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8835902` and `https://www.ibm.com/downloads/cas/DW5DGM8K`	`https://www.researchgate.net/publication/335801314_CODDLE_COde-injection_Detection_with_Deep_LEarning`

Figure 4.1 – Common threats against consumer sites

Hands-On Sections and Code

Developing a simple authentication example

Online ML examples never incorporate any sort of access detection because the author is focusing on showing a programming technique. When creating a production application or an experimental application that uses sensitive data, you need some way to determine the identity of any entities

accessing your ML application using authentication. When you authenticate a user, you only determine the user's identity and nothing else. Before the user can do anything, you must also authorize the user's activities. You can find the code for the following examples in the `MLSec; 04; Authentication and Authorization.ipynb` file of the downloadable source code.

Working with basic and digest authentication

There are many ways to accomplish authentication and the techniques used are defined by the following points:

- The kind of access
- The type of server
- The server security setup
- The application security setup

Here is an easy local application-level security access technique:

```
import getpass
user = getpass.getuser()
pwd = getpass.getpass("User Name : %s" % user)
if not pwd == 'secret':
    print('Illegal Entry!')
else:
    print('Welcome In!')
```

The code obtains the user's name and then asks for a password for that name. Of course, this kind of access only works for a local application. You wouldn't use it for a web-based application. This version is also simplified because you wouldn't store the passwords in plain text within the application itself. The password would appear in an encrypted database as a hash value and you'd turn whatever the user types into a hash using the technique shown in the *Relying on traditional methods example* section of *Chapter 2*. After you've hashed the user's password, you'd locate the username in the external database, obtain the hash from the database, and compare the user's hashed password to the hash found in the database.

Online authentication can also follow a simple strategy. Here is an example of this sort of access:

```
import requests
from requests.auth import HTTPDigestAuth
resource =
'http://localhost:8888/files/MLSec/Chapter04/TestAccess.txt'
authenticate = HTTPDigestAuth('user', 'pass')
response = requests.get(resource, auth = authenticate)
print(response)
```

In this case, you use a basic technique to verify access to a particular resource, this one on the local machine through `localhost`. You build an authentication object consisting of the username and password, and then use it to obtain access to a resource. A response code of 200 indicates success. Most sites use a response code of 401 for a failed authentication, but some sites, such as GitHub, use a 404 response code instead.

Note that this example uses `HTTPDigestAuth`, which encrypts the username and password before sending it over the network. It's not the most secure method because it's vulnerable to a MITM attack but much better than using `HTTPBasicAuth` for a public API, because basic

authentication sends everything in Base64 encoded text. Some security professionals recommend basic authentication for private networks where you can use SSL security, as described at https://mark-kirby.co.uk/2013/how-to-authenticate-apis-http-basic-vs-http-digest/. The request library also supports **Open Authentication** (**OAuth**) (see https://pypi.org/project/requests-oauthlib/ for details), Kerberos (see https://github.com/requests/requests-kerberos for details), and **Windows NT Lan Manager** (**NTLM**) (see https://github.com/requests/requests-ntlm for details) methodologies.

Accessing GitHub using OAuth-type authentication

Let's look at the specific example of accessing GitHub, which relies on an OAuth-type access strategy:

To use this example, you must first create an API access token by signing in to your GitHub account and then accessing the https://github.com/settings/tokens page.

After you click **Generate New Token**, you see a **New Personal Access Token** page where you provide a token name and decide what access rights the token should provide. For this example, all you really need is **repo**, **package**, and **user access**.

When you are finished with the configuration, click **Generate New Token**. Make sure you copy the token down immediately because you won't be able to access it later. (If you make a mistake, you can always delete the old token and create a new one.)

This simple example shows what you need to do to obtain a list of repositories for a given account. However, that's not really the point of the example. What you're really looking at is the authentication technique used to access specific resources and the use of GitHub isn't that pertinent—it could be any API (any API securing a protected resource). Use these steps to create the example:

1. Import the required libraries:

```
import requests
import json
```

2. Obtain the sign-in information. Note that you must replace `Your User Name` with your actual username and `The Token You Generated` with the token you created earlier:

```
resource = 'https://api.github.com/user/repos'
username = 'Your User Name'
token = 'The Token You Generated'
```

3. Create the reusable session object:

```
session = requests.Session()
session.auth = (username, token)
```

4. Request the list of repos for this user:

```
repos = json.loads(session.get(resource).text)
```

5. Output the repo names:

```
for repo in repos:
    print(repo['name'])
```

Developing a simple spam filter example

Most people associate spam with email and text messages, and you do see ML applications keeping spam away from people all the time. However, for an ML application, spam is any data in any form that you don't want the application to see. Spam is the sort of information that will cause biased or unusable results because, unlike a human, an ML application doesn't know to ignore the spam. In most cases, spam is an annoyance rather than a purposeful attempt to attack your model. You can find the code for the following examples in the `MLSec_ 04_ Remove Unwanted Text.ipynb` file of the downloadable source code.

Starting with a simple removal

When creating a secure input stream for your ML application, you need to think about layers of protection because a hacker is certainly going to pile on layers of attacks. Even if you've limited access to your application and its data sources, and provided an ensemble to predictively remove any data source that is most definitely bad, hackers can still try to get data through seemingly useful datasets. Consider the simple text file shown here (also found in `TestAccess.txt`):

```
You've gained access to this file.
This is a bad line.
This is another bad line.
This line is good.
And, this line is just sort of OK.
This is yet another bad line for good measure.
You don't want this bad line either.
Finally, this line is great!
```

Imagine that every line that has the word `bad` in it really is bad. Perhaps the data includes a script or unwanted values. In fact, perhaps the data just isn't useful. It's not necessarily bad, but if you include it in your analysis, the result is biased or perhaps skewed in some way. In short, the line with `bad` in it is some type of limited spam. It's not selling you a home in outer whatsit, but it's not helping your application either. When this sort of issue occurs, you can remove the bad lines and keep the good lines using code similar to that shown in the following steps:

1. Import the required libraries. When you perform these imports, the **Integrated Development Environment** (**IDE**) will tell you that it has downloaded `stopwords` needed for the example:

```
import numpy as np
import os
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
```

```
nltk.download('punkt')
nltk.download('wordnet')
from collections import Counter
from sklearn.naive_bayes import MultinomialNB
from skleyer.metrics import confusion_matrix
```

2. Create a function that accepts a filename and a target to remove unwanted lines. This function opens the file and keeps processing it line by line until there are no more lines:

```
def Remove_Lines(filename, target_word):
    useful_lines = []
    with open(filename) as entries:
        while True:
            line = entries.readline()
            if not line:
                break
            if not target_word.upper() in line.upper():
                useful_lines += [line.rstrip()]
    return useful_lines
```

3. Define the file and target data to search, then create a list of good entries in the dataset and print them out:

```
filename = 'TestAccess.txt'
target = 'bad'
good_data = Remove_Lines(filename, target)
for entry in good_data:
    print(entry)
```

There is nothing magic about this code—you've used something like it before to process other text files. The difference is that you're now using a file-processing technique to add security to your data. Notice that you must set both the current word and the target word to uppercase (or lowercase as you like) to ensure the comparison works correctly. Here's the output from this example:

```
You've gained access to this file.
This line is good.
And, this line is just sort of OK.
Finally, this line is great!
```

Notice that all of the lines with the word `bad` in them are now gone.

Manipulating filtered data

Most people who work with data understand the need to manipulate it in various ways to make the data better suited for analysis. For example, when performing text analysis, one of the first steps is to remove the stop words because they don't add anything useful to the dataset. Some of these same techniques can help you find patterns in input data so that it becomes harder for a hacker to sneak something in even after you remove the bad elements. For example, you might find odd repetitions of words, number sets, or other data that might normally appear infrequently, if at all, in a dataset that will alert you to potential hacker activity. The following steps show how to create a simple filter that helps you see unusual data or patterns. This code relies on the same libraries you imported in the previous section:

1. Define a function to remove small words such as "to," "my," and "so" from the text:

```
def Remove_Stop_Words(data):
    stop_words = set(stopwords.words('english'))
    new_lines = []
    for line in data:
        words = line.split()
        filtered = [word for word in words
                    if word.lower() not in stop_words]
        new_lines += [' '.join(filtered)]
    return new_lines
```

2. Define a function that will list each word individually, along with the count for that word:

```
def Create_Dictionary(data):
    all_words = []
    for line in data:
        words = line.split()
        all_words += words
    dictionary = Counter(all_words)
    return dictionary
```

3. Define a function that creates a matrix showing word usage:

```
def Extract_Features(data, dictionary):
    features_matrix = np.zeros(
        (len(data),len(dictionary)))
    lineID = 0
    for line in data:
        words = line.split()
        for word in words:
```

```
            wordID = 0
            for i,d in enumerate(dictionary):
              if d == word:
                wordID = i
                features_matrix[lineID, wordID] += 1
        lineID += 1
    return features_matrix
```

4. Create a filtered list of text strings from the original text that has the stop words removed:

```
filtered = Remove_Stop_Words(good_data)
print(filtered)
```

5. Create a dictionary of words from the filtered list:

```
word_dict = Create_Dictionary(filtered)
print(word_dict)
```

6. Create a matrix showing which words are used and when in each dataset row:

```
word_matrix = Extract_Features(filtered, word_dict)
print(word_matrix)
```

Each of the functions in this example shows a progression:

1. Remove the stop words from each line in the dataset that was created from the original file.
2. Create a dictionary of important words based on the filtered dataset.
3. Define a matrix that shows each line of the dataset as rows and the words within that row as columns. A value of `1` indicates that the word appears in the specified row.

There are some interesting bits of code in the example. For example, `Remove_Stop_Words()` relies on a list comprehension to perform the actual processing. You could also use a `for` loop if desired. You must also use `join()` to join the individual words back together and place them in a list to perform additional processing. The output looks like this:

```
["You've gained access file.", 'line good.', 'And, line
 sort OK.', 'Finally, line great!']
```

A dictionary is essential for many types of processing. `Create_Dictionary()` makes use of the `Counter()` function found in the `collections` library to make short work of creating the dictionary in a form that will make defining the matrix easy. Here's the output from this step:

```
Counter({'line': 3, "You've": 1, 'gained': 1, 'access': 1,
 'file.': 1, 'good.': 1, 'And,': 1, 'sort': 1, 'OK.': 1,
 'Finally,': 1, 'great!': 1})
```

The output doesn't appear in any particular order and it's not necessary that it does. Each unique word in the dataset appears as an individual dictionary key. The values show the number of times

that the word appears. Consequently, you could use this output to perform tasks such as determining word frequency. In this case, the example simply creates a matrix to show where the words appear within the dataset. There are possibly shorter ways to perform this task, but the example uses a straightforward approach that processes each word in turn and finds its position in the matrix by enumerating the dictionary. Here's the output from this step:

```
[[1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 1. 1. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 1.]]
```

If you look at the first row, the first four entries have a `1` in them for `You've`, `gained`, `access`, and `file`. None of these words appear in the other rows, so the entries are `0` in the other rows. However, `line` does appear in three of the rows, so there is a `1` for that entry in each of the rows. The next section takes these techniques and shows how to apply them to multiple files in an email dataset.

Creating an email filter

Emails can contain a great deal of useless or harmful information. At one time, email filters worked similarly to the example in the previous section (a simple filter). However, trying to keep track of all of the words that hackers use to get past the filter became impossible. Even though the simple filtering technique is still useful for certain needs, email filtering requires something better—an approach that is flexible enough to change with the techniques that hackers use to attempt to get past the filter. One such approach is to use an ML application to discover which emails are useful and which are spam.

The example in this section performs a simple analysis of the useful (ham) versus spam orientation of each email in the Ling-Spam email corpus described at http://www2.aueb.gr/users/ion/docs/ir_memory_based_antispam_filtering.pdf and available for download at http://www.aueb.gr/users/ion/data/lingspam_public.tar.gz. The original dataset is relatively complex and somewhat unwieldy, so the example actually uses a subset of the messages split into two folders: `Email_Train` and `Email_Test`. To save some time and processing, the example relies on the content of the `\lingspam_public\lingspam_public\lemm_stop\` folder, which provides the messages with the stop words already processed and the words normalized using lemmatization (see the *Choosing between stemming and lemmatization* section for details). The messages in the `Email_Train` folder come from the `part1`, `part2`, and `part3` folders (867 messages in total with 144 spam messages), while the messages in the `Email_Test` folder come from the `part4` folder (289 messages in total with 48 spam messages). You can tell which messages contain spam because they start with the letters `spmsg` (for spam message).

Recognizing the benefits of targeted training and testing data

Even though this example uses a generic database, it's always better to use your organization's email to train and test any model you create. Doing so will greatly decrease the number of false positives and negatives in the production environment because the data will reflect what the users actually receive. For example, an engineering firm specializing in fluid dynamics can expect to receive a lot more emails about valves than a financial firm will. This same principle holds true for all sorts of other filtering needs. The data from your organization will always provide better results than generic

data will. Of course, you need to make sure that any organizational data you use meets privacy requirements and is properly sanitized before you use it, as described in *Chapter 13.*

Each of the text files contains three lines. The first line is the email subject, the second line is blank, and the third line contains the message. In processing the emails, you look at just the third line with regard to content and know that you can label the training messages as spam if the filename begins with `spmsg` or `ham` when the filename begins with something else. With this in mind, the following code shows a spam filter you can create using techniques similar to those used in the previous section but using multiple files in this case. This code relies on the same libraries you imported in the *Starting with a simple removal* section (make sure you use the 1.0.*x* version, originally version 0.23.*x*, of scikit-learn, as described in *Chapter 1*, for this part of the chapter or you may encounter errors):

1. Set the paths for the training and testing messages:

```
train_path = "Email_Train"
train_emails = \
    [os.path.join(train_path,f) for f
     in os.listdir(train_path)]
test_path = "Email_Test"
test_emails = \
    [os.path.join(test_path,f) for f
     in os.listdir(test_path)]
```

2. Create a dictionary function to build the required dictionary. Then, remove non-word items that include numbers, special characters, end-of-line characters, and so on:

```
def Create_Mail_Dictionary(emails):
    cvec = CountVectorizer(
        stop_words='english',
        token_pattern=r'\b[a-zA-Z]{2,}\b',
        max_features=2000)
    corpus = [open(email).read() for email in emails]
    cvec.fit(corpus)
    return cvec
train_cvec = Create_Mail_Dictionary(train_emails)
```

3. Create a features matrix function. Instead of lines and words, this code uses documents and words for the matrix:

```
def Extract_Mail_Features(emails, cvec):
    corpus = [open(email).read() for email in emails]
```

```
    return cvec.transform(corpus)
train_feat = Extract_Mail_Features(train_emails,
    train_cvec)
test_feat = Extract_Mail_Features(test_emails,
    train_cvec)
```

4. Create labels showing which messages are ham (`0`) and spam (`1`):

```
train_labels = np.zeros(867)
train_labels[723:867] = 1
test_labels = np.zeros(289)
test_labels[241:289] = 1
Train the Multinomial Naïve Bayes model:
MNB = MultinomialNB()
MNB.fit(train_feat, train_labels)
```

5. Predict which of the messages in the test group are ham or spam and output the correctness of the prediction as a confusion matrix:

```
result = MNB.predict(test_feat)
print(confusion_matrix(test_labels, result))
```

6. Display the confusion matrix in a nicely plotted form:

```
matrix = plot_confusion_matrix(MNB,
                               X=test_feat,
                               y_true=test_labels,
                               cmap=plt.cm.Blues)
plt.title('Confusion matrix for spam classifier')
plt.show(matrix)
plt.show()
```

The listing shows that simple techniques often provide the basis for more complex processing. The `Create_Mail_Dictionary()` and `Extract_Mail_Features()` functions provide the ability to work with multiple files and to provide additional data cleaning. Notice that this example uses a more efficient method of creating the dictionary using `scikit-learn CountVectorizer()`. The concept and the result are the same as what you see in the previous section, but this approach is shorter and more efficient. The `Extract_Mail_Features()` function is also made shorter by using list comprehensions in addition to calling the `cvec.transform()` function on the resulting corpus. Again, the output is the same and the process is the same under the covers, but you're using a more efficient approach.

The Multinomial Naïve Bayes model will vary in its ability to correctly predict ham or spam messages after you fit it to the training data. In this case, the result shows that there are 241 ham messages and 48 spam messages in the test dataset. A larger test dataset is likely to show a less impressive result, but according to *Machine learning for email spam filtering: review, approaches and open research problems*, some companies, such as Google, have achieved rates as high as 99.9 percent. In this case, however, the companies use advanced ML strategies, rather than the more basic Multinomial Naïve Bayes model. In addition, the strategies rely on ensembles of learners as suggested in the *Using ensemble learning* section of *Chapter 3*.

Choosing between stemming and lemmatization

There are two common techniques for normalizing words within documents: stemming and lemmatization. Each has its uses. **Stemming** simply removes the prefixes and suffixes of words to normalize the root word. For example, player, plays, and playing would all be stemmed from the root word play. This technique is mostly used for word analysis, such as determining how often particular words appear in one or more documents. **Lemmatization** processes the words in context, so that the words running, runs, and ran all appear as the root word run. You use this technique most often for text analysis, such as determining the relationships of words in a spam message versus a usable (ham) message. Here is an example of stemming.

```
from nltk.stem import LancasterStemmer
from nltk.tokenize import word_tokenize
LS = LancasterStemmer()
print(LS.stem("player"))
print(LS.stem("plays"))
print(LS.stem("playing"))
tokens = word_tokenize("Gary played the player piano while playing
cards.")
stemmed = [LS.stem(word) for word in tokens]
print(" ".join(stemmed))
```

The example imports the required libraries, creates an instance of `LancasterStemmer()` and then uses the instance to stem three words with the same root. It then does the same thing for a sentence containing the three words. The output shows that context isn't taken into account and it's possible to end up with some non-words:

```
play
play
play
gary play the play piano whil play card .
```

Lemmatization takes a different route, as shown in this example (Note that you may have to add the `nltk.download('omw-1.4')` statement after the `import` statement if you see an error message after running this code):

```
from nltk.stem import WordNetLemmatizer
WNL = WordNetLemmatizer()
print(WNL.lemmatize("player", pos="v"))
print(WNL.lemmatize("plays", pos="v"))
print(WNL.lemmatize("playing", pos="v"))
tokens = word_tokenize("Gary played the player piano while playing
cards.")
```

```
lemmatized = [WNL.lemmatize(word, pos="v") for word in tokens]
print(" ".join(lemmatized))
```

Notice the `pos` argument in the `lemmatize()` calls. This argument provides the context for performing the task and can be any of these values: adjective (`a`), satellite adjective (`s`), adverb (`r`), noun (`n`), and verb (`v`). In choosing verbs, the example provides this output, which you can contrast with stemming:

```
player
play
play
Gary play the player piano while play card .
```

The point is that you must choose carefully between stemming and lemmatization when creating filters for your ML application. Choosing the right process will result in significantly better results in most cases.

Links

Cloud Adoption Statistics for 2021 article: https://hostingtribunal.com/blog/cloud-adoption-statistics/

Setting default server-side encryption behavior for Amazon S3 buckets: https://docs.aws.amazon.com/AmazonS3/latest/userguide/bucket-encryption.html

Web scraping protection: How to protect your website against crawler and scraper bots at https://datadome.co/bot-management-protection/scraper-crawler-bots-how-to-protect-your-website-against-intensive-scraping/

VirusTotal (https://support.virustotal.com/hc/en-us/categories/360000160117-About-us)

U.S. Escalates Online Attacks on Russia's Power Grid: https://www.nytimes.com/2019/06/15/us/politics/trump-cyber-russia-grid.html

What Happens When Russian Hackers Come for the Electrical Grid: https://www.bloomberg.com/news/features/2022-01-26/what-happens-when-russian-hackers-cyberattack-the-u-s-electric-power-grid.

Sniffing Telnet Using Wireshark: http://blog.johnmuellerbooks.com/2011/06/07/sniffing-telnet-using-wireshark/

Swiss Researchers Create Machine Learning Thermostat at https://www.rtinsights.com/empa-machine-learning/.

#DefCon: Thermostat Control Hacked to Host Ransomware at https://www.infosecurity-magazine.com/news/defcon-thermostat-control-hacked/

The article *Understanding the Relative Insecurity of SCADA Systems* at http://blog.johnmuellerbooks.com/2011/11/28/understanding-the-relative-insecurity-of-scada-systems/ seems outdated, but unfortunately, no one has bothered to secure these small systems, as described in the article *Biggest threats to ICS/SCADA systems* at https://resources.infosecinstitute.com/topic/biggest-threats-to-ics-scada-systems/ (among many others

Trusted Platform Module Technology Overview: https://docs.microsoft.com/windows/security/information-protection/tpm/trusted-platform-module-overview

Tom's Guide article at https://www.tomsguide.com/news/what-is-a-tpm-and-heres-why-you-need-it-for-windows-11

AWS: https://aws.amazon.com/ec2/nitro/nitro-enclaves/ (the site doesn't specifically call it confidential computing, but articles such as the one at https://www.forbes.com/sites/janakirammsv/2020/10/30/aws-nitro-enclaves-bring-confidential-computing-to-amazon-ec2/?sh=45d98f771c8e make it apparent that it is)

Azure: https://azure.microsoft.com/en-us/solutions/confidential-compute/

Google: https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/

Intel: https://www.intel.com/content/www/us/en/security/confidential-computing.html

Confidential Computing Consortium: https://confidentialcomputing.io/

Hype Cycle for Security Operations, 2020 at https://www.gartner.com/en/documents/3986721

See https://www.howtogeek.com/412316/how-email-bombing-uses-spam-to-hide-an-attack/ for some examples on how spam is used to attack.

What is Spam and a Phishing Scam – Definition: https://www.kaspersky.com/resource-center/threats/spam-phishing

Here's how to avoid unwanted tracking online at https://www.techradar.com/news/avoiding-unwanted-tracking-online and *4 Ways to Protect Your Phone's Data From Unwanted Tracking* at https://preyproject.com/blog/en/4-ways-to-protect-phone-data-unwanted-tracking/

Password Manager survey results at https://www.passwordmanager.com/password-manager-trust-survey/

Eliminating Account Takeovers with Machine Learning and Behavioural Analysis at https://www.brighttalk.com/webcast/17009/326415/eliminating-account-takeovers-with-machine-learning-and-behavioural-analysis

If you're interested, the definition at https://www.dictionary.com/e/slang/karen/ provides some insights into the use of the term Karen.

The Phrase finder: https://www.phrases.org.uk/bulletin_board/46/messages/636.html

You can find extensive documentation about the GitHub REST API at https://docs.github.com/en/rest/overview.

Machine learning for email spam filtering: review, approaches and open research problems at https://www.sciencedirect.com/science/article/pii/S2405844018353404

Machine learning for email spam filtering: review, approaches and open research problems at https://www.sciencedirect.com/science/article/pii/S2405844018353404

Further reading

The following links provide you with some additional reading that you may find useful to further understand the materials in this chapter:

- This link helps you discover more about the ML component of a SageMaker application: *Building secure machine learning environments with Amazon SageMaker*:

https://aws.amazon.com/blogs/machine-learning/building-secure-machine-learning-environments-with-amazon-sagemaker/

- Learn more about cookie poisoning: https://www.f5.com/services/resources/glossary/cookie-poisoning
- See how to perform scraping appropriately: *How to scrape websites without getting blocked*: https://www.scrapehero.com/how-to-prevent-getting-blacklisted-while-scraping/
- Discover how to use ML techniques to perform scraping: https://towardsdatascience.com/web-scraping-for-machine-learning-5fffb7047f70
- Discover how to use ML to detect scraping efforts:

https://kth.diva-portal.org/smash/get/diva2:1117695/FULLTEXT01.pdf

- Learn some additional detail on the carding attack type: *How to Use AI and Machine Learning in Fraud Detection:*

https://spd.group/machine-learning/fraud-detection-with-machine-learning/

- Discover how malware can cause physical network damage: *Emotet Malware Causes Physical Damage:* https://securityboulevard.com/2020/04/emotat-malware-causes-physical-damage/
- Learn more about how malware can cause bodily harm: https://www.bbc.com/news/technology-54204356
- Read about the effect of loss of SCADA control led to a power plant hack in Ukraine: *Everything We Know About Ukraine's Power Plant Hack:* https://www.wired.com/2016/01/everything-we-know-about-ukraines-power-plant-hack//
- Provides detailed information about how SCADA and IoT are linked in ways that could cause serious problems: *What are SCADA and IoT?:* https://www.datashieldprotect.com/blog/what-is-scada-iot
- Learn more about how confidential computing works: *Confidential Computing* article at

https://www.ibm.com/cloud/learn/confidential-computing

- Discover how a TPM works in detail: https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/

- Discover why Windows 11 requires the use of the TPM 2.0 chip: *What is TPM 2.0 — the chip you need to run Windows 11:* https://www.laptopmag.com/articles/tpm-chip-faq
- Learn about specific exploits against older TPM chips that may affect your ML application: *Researchers Detail Two New Attacks on TPM Chips:* https://www.bleepingcomputer.com/news/security/researchers-detail-two-new-attacks-on-tpmchips/
- Understand why it takes up to 10 years to develop and implement a new security strategy: https://www.hindawi.com/journals/je/2020/5267564/
- Provides insights into how to perform behavior analysis: *Using machine learning to understand customer's behavior*: https://towardsdatascience.com/using-machine-learning-to-understand-customers-behavior-f41b567d3a50
- Learn more about attribute-based access: https://www.okta.com/blog/2020/09/attribute-based-access-control-abac/
- Gain a better understanding of the difference between basic and digest authentication: https://www.hackingarticles.in/understanding-http-authentication-basic-digest/
- Understand the difference between the way GitHub apps and OAuth apps work: https://docs.github.com/en/developers/apps/differences-between-github-apps-and-oauth-apps
- Find a discussion of the features used for advanced machine learning classifiers used for spam detection: *Deep convolutional forest: a dynamic deep ensemble approach for spam detection in text*: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9039275/
- See an example of how to create a spelling corrector that could be combined with spam detection: *Spelling Recommender With NLTK:* https://jorgepit-14189.medium.com/spelling-recommender-with-nltk-2fb6fe94a7b3
- Discover a technique for detecting fake news that can also be used to help with spam detection: *Detecting Fake News with Python and Machine Learning:* https://data-flair.training/blogs/advanced-python-project-detecting-fake-news/

Attack Type	**Major Consideration**	**Possible Remedy**	**Resources**	**Whitepaper/Example**
File-path traversal	A hacker gains access to sensitive data (such as data used to train a model) by using specially configured paths. These paths either rely on the inherent	Ensure that every resource always has the required protection (see the *Understanding the kinds of application security* section in this chapter) and that any user access relies on the principle of least privilege. It's also possible to rely on	https://www.geeksforgeeks.org/path-traversal-attack-prevention/ and https://www.trendmicro.com/en_us/research/20/j/contentprovider-path-traveral-	https://jisajournal.springeropen.com/articles/10.1186/s13174-019-0115-x

	weaknesses of relative paths that use the `../../` notation or known absolute paths. Once the hacker gains access to the directory, it's possible to look at the `config` file settings, corrupt data, or perform other malicious permanent storage modifications.	special filtering of input data and API requests using a technique similar to that shown in the *Manipulating filtered data* section and pattern detection using the technique in the *Creating an email filter* section.	flaw-on-esc-app-reveals-info.html	
Distributed Denial-of-Service (DDoS)	Packets of useless data and commands are sent from a group of systems under the hacker's control to overwhelm the victim's system and cause it to fail. Given that ML applications often require large amounts of network bandwidth, this class of application is inordinately affected by a DDoS attack.	Most methods today rely on detecting the attack and dealing with it on the victim's system. One proposed solution is to detect the attack from the source (such as the hacker's control machine or the various bots) using ML techniques and then cut off those attackers from the inputs to the victim.	https://ieeexplore.ieee.org/document/7013133 and https://www.mdpi.com/2504-3900/63/1/51/pdf	http://palms.princeton.edu/system/files/Machine_Learning_Based_DDoS_Attack_Detection_From_Source_Side_in_Cloud_camera_ready.pdf

Chapter 5

Technical requirements

This chapter requires that you have access to either Google Colab or Jupyter Notebook to work with the example code. The *Requirements to use this book* section of *Chapter 1*, *Defining Machine Learning Security*, provides additional details on how to set up and configure your programming environment. When testing the code, use a test site, test data, and test APIs to avoid damaging production setups and to improve the reliability of the testing process. Testing over a non-production network is highly recommended but not absolutely necessary. Using the downloadable source is always highly recommended. You can find the downloadable source on the Packt GitHub site at https://github.com/PacktPublishing/Machine-Learning-Security-Principles or my website at http://www.johnmuellerbooks.com/source-code/.

Figures

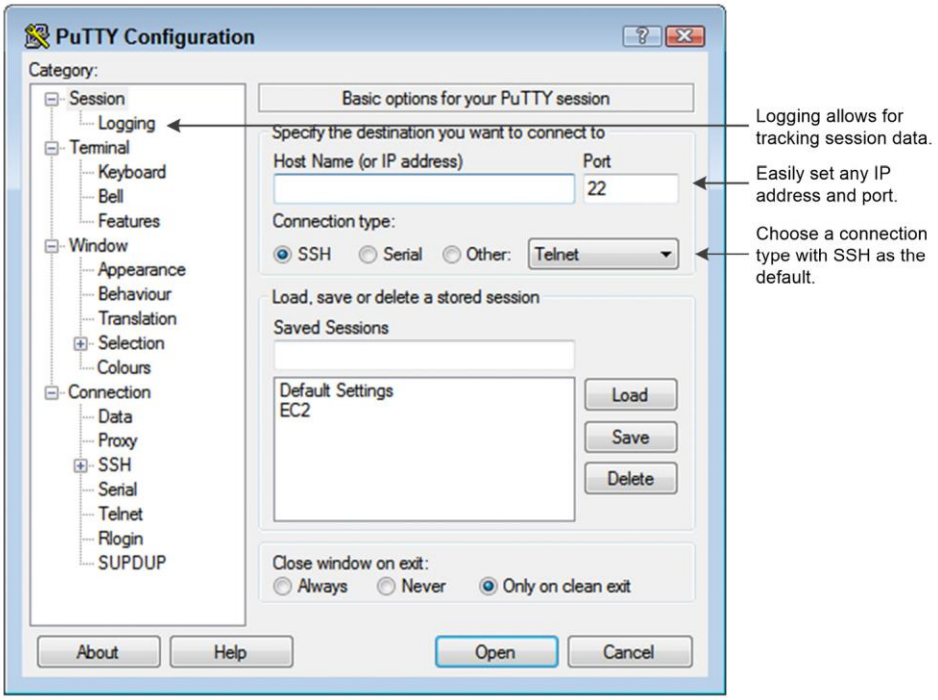

Figure 5.1 – ML developers need not fight with a command line interface when testing security-related ML applications

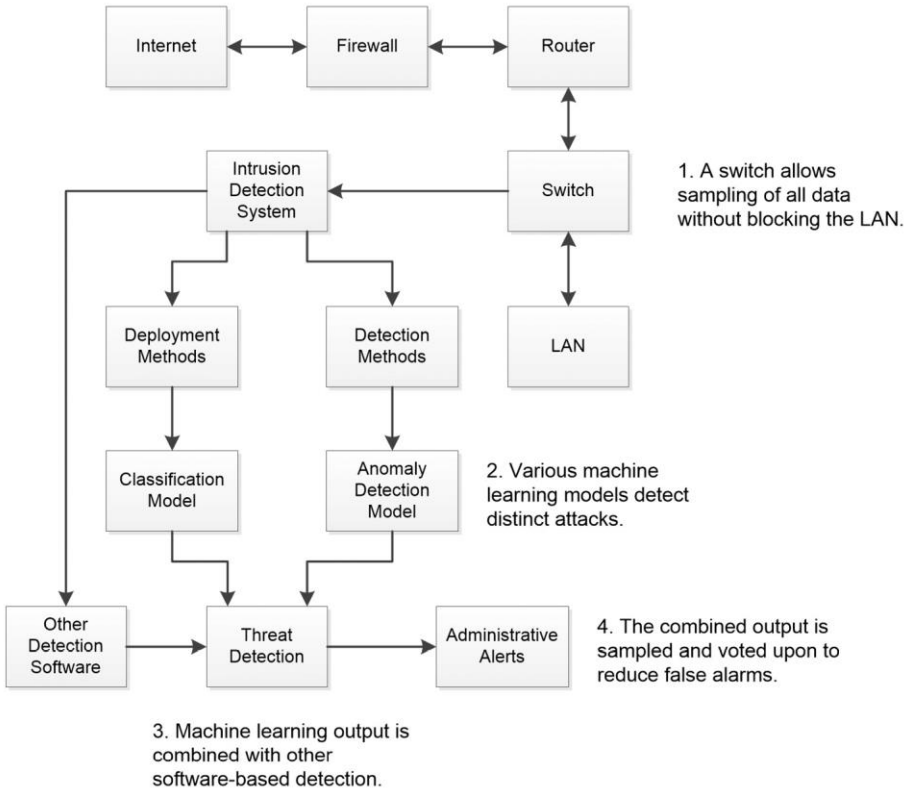

Figure 5.2 – Developing a generalized NIDS solution with the NIDS in a supervisory role

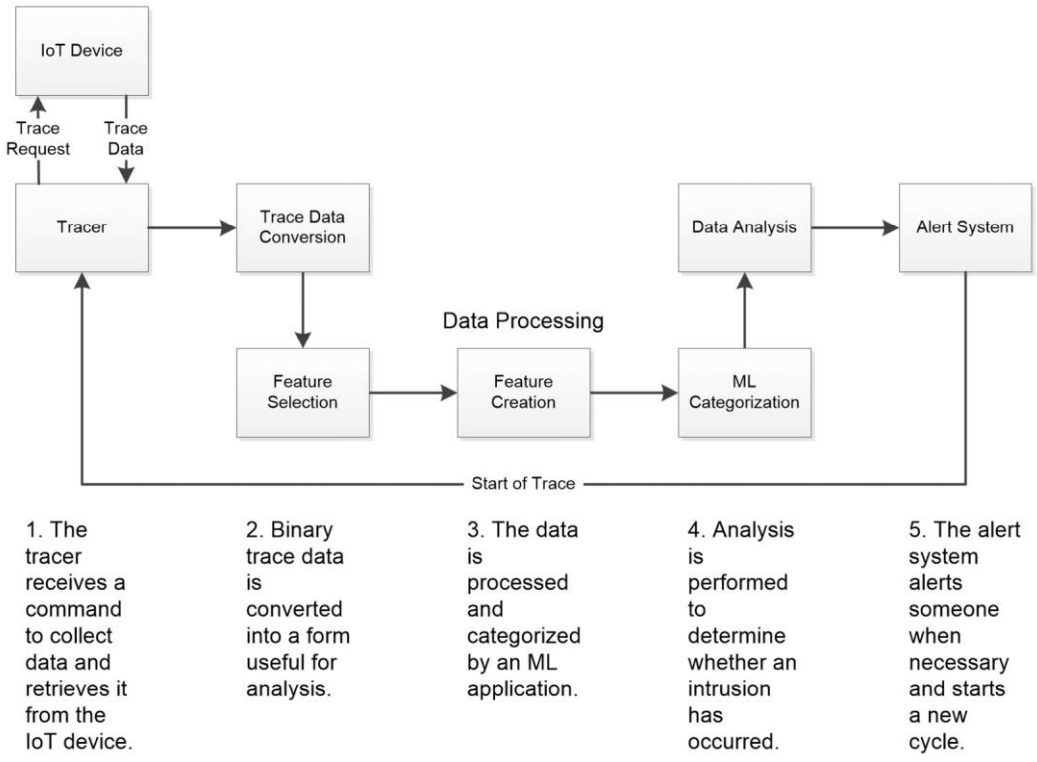

Figure 5.3 – Using trace data to monitor IoT devices directly and look for intrusions

Task	Learning Type	ML Consideration
Automatic language translation	Supervised	Translates one language into another language using a sequence-to-sequence learning algorithm. The results are often less useful than expected due to variations between languages and the fact that languages generally contain words that don't have equivalents in other languages. Susceptible to data errors, missing data, data corruption, algorithm bias, and an inability to repeat and verify results due to naturally occurring evolution in languages. This kind of application is also sensitive to speech patterns and misidentifying terms when words aren't enunciated clearly.
Email spam and malware filtering	Supervised	Marks, moves, or deletes email that meets the criteria of spam or malware from an inbox as it's received from a server. There are usually several levels of filtering including Content, Header, Blacklist, Rule-based, and Permission. Susceptible to a number of potential attacks including backdoors, Trojans, espionage, sabotage, fraud, evasion, inference, data errors, and data corruption. This is one of the more reliable forms of ML applications, but users still regularly find spam in their inboxes and useful messages in their spam folders.
Image recognition	Supervised	Identification of objects, persons, places, patterns, and other elements within an image. Susceptible to a variety of attack types, but also prone to misidentification when the image contains elements the application didn't expect or when those objects appear in positions that the application isn't trained to recognize.

Task	Learning Type	ML Consideration
Medical diagnosis	Supervised and unsupervised	Predicts the progression and characteristics of diseases and other conditions, along with locating and identifying potential patient illnesses. Susceptible to data bias, data corruption, data errors, incorrect algorithm selection, and algorithm bias. This particular application type can never operate alone; it always assists a physician with the required experience to make a diagnosis.
Online fraud detection	Supervised	Reduces the risk of conducting transactions online by detecting conditions such as fake accounts, fake IDs, compromised sites, compromised security certificates, and so on. Susceptible to a wide range of attacks, some of which have nothing to do with the application. For example, a compromised certificate authority could cause the application to fail by allowing the hacker access to the underlying infrastructure, even if the application itself isn't at fault. This kind of application is also known to display false positives and false negatives depending on the reliability of the code used to create it and the model training.
Product recommendation	Unsupervised	Outputs product recommendations based on previous buying habits, associated goods, and direct queries. It's one of the most widely used and common ML applications. Susceptible to data errors, data bias, missing data, algorithm bias, fraud, sabotage, and a wealth of other issues. This kind of application often provides irrelevant information along with useful product recommendations because the application has no method of judging user needs and wants.

Figure 5.4 – Common endpoint attacks and strategies

Hands-On Section

Using supervised learning example

Most of the ML models you create for security needs rely on supervised learning because this approach provides a better result, with known issues that you can track in real time. This is the approach that works best for issues such as determining when a hacker wants to break into your API or evade security through trial and error. It's also the method most commonly used to detect malware, fake data, or other types of exploits. The example in this section is somewhat generic but does demonstrate how you can collect features, such as the number of API calls made by a specific IP address, and then use this information to create a model that can then detect unwanted activity. The following sections show how to perform this task using a decision tree. You can also find this code in the `MLSec; 05; Real Time Defenses.ipynb` file of the downloadable source.

Getting an overview

The example is somewhat contrived in this case because a production system would use an active data stream to acquire data, and the dataset itself would be much larger. *Figure 5.5* shows an example of how a production system might acquire and analyze data using the techniques found in this example. However, it's also important that the example is understandable, so the example dataset is static and relatively small to make it easy to see how various elements work.

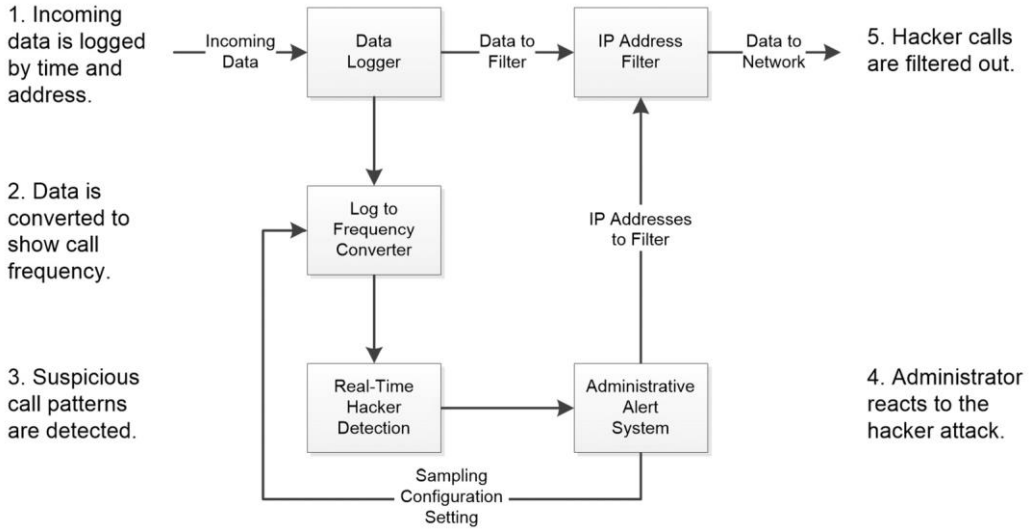

Figure 5.5 – Obtaining and analyzing API call data

The goal of this setup is to not only disrupt the flow of data as little as possible but also ensure that any hacker activity is detected and immediately stopped. The data logger simply creates continuous output files on disk. In the example, the output files are in `.csv` format, with just the time, IP address, and API call made by everyone who is using the system. It isn't unusual to have such setups anyway, so most developers won't have to implement anything new to obtain the data; they'll just have to know where to find it and ensure that it contains the information needed for the next step.

The next step is to read and convert the data in the logs at specific intervals, perhaps every 2 minutes. The conversion process takes the log data and counts how many times each IP address is

making each API call. The sections that follow show how this works, but the transformation process needs to be as fast and simple as possible.

Creating a model based on call patterns comes next. A human will need to go through the frequency logs and label IP addresses to decide whether they're benign or a hacker, based on the call patterns. During real-time use, when the model detects a hacker, it sends the information to the administrator, who can then filter out the IP address and change the timespan to a shorter interval used to detect new intrusions. In the meantime, the benign data continues to flow into the network.

Building a data generator

In a perfect situation, you have access to data from your network to use in building and testing a model to detect an API attack or other security issue (because this example focuses on an API attack, this is where the discussion will focus from this point on). However, you might not always have access to the required labeled data. In this case, you can construct and rely upon a data generator to create a simulated dataset with the characteristics you want. Start by importing the packages and classes needed for this example as a whole, as shown in the following code block:

```
from datetime import time, date, datetime, timedelta
import csv
import random
from collections import Counter
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

Note that memory usage is kept down by not importing everything from `sklearn`.

Creating the CreateAPITraffic() function

You need a function to actually generate the API traffic. The following steps outline what goes into this function (it appears as a single block of code in the online source):

1. Define a function that creates the actual data, based on what the logs for your network look such as. This example uses a relatively simple setup that includes the time, IP address of the caller, and the API call made, as shown here. The code starts by defining arguments you can use to modify the behavior of the data generation process. It's a good idea to provide default values that are likely to prove useful so that you're not dealing with missing argument problems when working with the function:

```
def CreateAPITraffic(
    values = 5000,
    benignIP = ['172:144:0:22', '172:144:0:23',
        '172:144:0:24', '172:144:0:25',
        '172:144:0:26', '172:144:0:27'],
    hackerIP = ['175:144:22:2', '175:144:22:3',
```

```
        '175:144:22:4', '175:144:22:5',
        '175:144:22:6', '175:144:22:7'],
    apiEntries = ['Rarely', 'Sometimes', 'Regularly'],
    bias = .8,
    outlier = 50):
```

2. Define the variables needed to perform tasks within the function. You use data to hold the actual log entries for return to the caller. The `currTime` and `updateTime` variables help create the log's time entries. The `selectedIP` variable holds one of the IP addresses, provided as part of the `benignIP` or `hackerIP` argument, and this is the IP address added to the current log entry. The `threshold` determines the split between benign and hacker log entries. The `hackerCount` and `benignCount` variables specify how many of each entry type appears in the log:

```
    data = []
    currTime = time(0, 0, 0)
    updateTime = timedelta(seconds = 1)
    selectedIP = ""
    threshold = (len(apiEntries) * 2) - \
        (len(apiEntries) * 2 * bias)
    hackerCount = 0
    benignCount = 0
```

3. A loop for generating entries comes next. This code begins by defining the time element of an individual log entry:

```
   for x in range(values):
        currTime = (datetime.combine(date.today(),
            currTime) + updateTime).time()
```

4. Selecting an API entry comes next. The code is written to accommodate any number of API entries, which is an important feature of any data generation function:

```
        apiChoice = random.choice(apiEntries)
```

5. The hardest part of the function is determining which IP address to use for the data entry. The `CreateAPITraffic()` function uses a combination of approaches to make the determination, based on the assumption that the hacker will select less commonly used API calls to attack because these calls are more likely to contain bugs, which is where `threshold` comes into play. However, it's also important to include a certain amount of noise in the form of outliers as part of the dataset. This example uses `hackerCount` as a means of determining when to create an outlier:

```
        choiceIndex = apiEntries.index(apiChoice) + 1
        randSelect = choiceIndex * \
            random.randint(1, len(apiEntries)) * bias
        if hackerCount % outlier == 0:
            selectedIP = random.choice(hackerIP)
        else:
            if randSelect >= threshold:
                selectedIP = random.choice(benignIP)
            else:
                selectedIP = random.choice(hackerIP)
```

6. It's time to put everything together as a log entry. Each entry is appended to data in turn, as shown here. In addition, the code also tracks whether the entry is a hacker or benign:

```
        data.append([currTime.strftime("%H:%M:%S"),
            selectedIP, apiChoice])
        if selectedIP in hackerIP:
            hackerCount += 1
        else:
            benignCount += 1
```

7. The final step is to return `threshold`, the benign log count, the hacker log count, and the actual log to the caller.

```
    return (threshold, benignCount, hackerCount, data)
```

When creating your own data generation function, you need to modify the conditions under which the log entries are created to reflect the real-world logs that your network generates automatically. The need for randomization and inclusion of noise in the form of outliers is essential. Just how much randomization and noise you include depends on the kinds of attacks that you're trying to prepare your model to meet.

Creating the SaveDataToCSV() function

Part of developing a test log to use is to provide some method of saving the data to disk in case you want to use it again, without regenerating it each time. The `SaveDataToCSV()` function serves this purpose, as shown here:

```
def SaveDataToCSV(data = [], fields = [],
    filename = "test.csv"):
    with open(filename, 'w', newline='') as file:
        write = csv.writer(file, delimiter=',')
        write.writerow(fields)
        write.writerows(data)
```

This code consists of a loop that takes the log data as an input and writes one line at a time from the log to a `.csv` data file on disk. Note that this function makes no assumptions about the structure of the data that it writes. Again, the point is to provide significant flexibility in the use of the function.

Note that the `fields` input argument is used to write a heading to the `.csv` file so that it's possible to know which columns the file includes. It's likely that any real-world server logs you use will also include this information, so adding this particular data is important to ensure that any analysis code you create works properly.

Defining the particulars of the training dataset

To make the generation process as flexible as possible, it's helpful to provide variables that you can easily modify to see their effect on the output. In this case, the example specifies API calls and IP addresses that differ from the defaults created earlier:

```
callNames = ['Rarely',
             'Sometimes1', 'Sometimes2',
             'Regularly1', 'Regularly2', 'Regularly3',
             'Often1', 'Often2', 'Often3', 'Often4',
             'Often5', 'Often6', 'Often7', 'Often8']
benignIPs = ['172:144:0:22', '172:144:0:23',
             '172:144:0:24', '172:144:0:25',
             '172:144:0:26', '172:144:0:27',
             '172:144:0:28', '172:144:0:29',
             '172:144:0:30', '172:144:0:31',
             '172:144:0:32', '172:144:0:33',
             '172:144:0:34', '172:144:0:35',
             '172:144:0:36', '172:144:0:37']
```

Note that the example is using the default `hackerIP` values, but you could modify the example to include more or fewer hackers, as desired.

Generating the CallData.csv file

Now that everything is in place, you need to actually generate and save the data file using the following code:

```
random.seed(52)
threshold, benignCount, hackerCount, data = \
    CreateAPITraffic(values=10000,
        benignIP=benignIPs,
        apiEntries=callNames)
print(f"There are {benignCount} benign entries " \
        f"and {hackerCount} hacker entries " \
        f"with a threshold of {threshold}.")
fields = ['Time', 'IP_Address', 'API_Call']
SaveDataToCSV(data, fields, "CallData.csv")
```

The call to `random.seed(52)` ensures that you obtain the same output every time during the code testing process. When you finally start using the code to generate real data, you comment out this call so that the data can appear random (this is why they call it a pseudo-random number generator).

The call to `CreateAPITraffic()` comes next, with the output being unpacked into local variables. The `threshold`, `benignCount`, and `hackerCount` variables provide output that tells you about the functioning of the data generation process. When using the random seed of `52`, you should see this output from the function:

```
There are 9320 benign entries and 680 hacker entries with a
threshold of 5.599999999999998.
```

The final step is to call `SaveDataToCSV()` to save the data to disk. At this point, you've completed the data generation process.

Converting the log into a frequency data table

Log entries don't provide information in the right form for analysis. What you really need to know is which IP addresses made what calls and how often. In other words, you need an aggregation of the log entries so that your model can use the calling pattern as a means to detect whether a caller is a hacker or a regular user. This process takes place in a number of steps that include reading the data into the application and performing any manipulations required to put the data into a form that the classifier can understand. The following sections show how to perform this task.

Creating the ReadDataFromCSV() function

The first step is to create a function that can read the `.csv` file from disk. More importantly, this function automates the process of labeling data as either benign or from a hacker. This particular part of the function is exclusive to this part of the example and serves to demonstrate that you can try really hard to make every function generic, but you may not always succeed. The following steps show how this function works:

1. Define the function and read in the data file from disk:

```
def ReadDataFromCSV(filename="test.csv"):
    logData = pd.read_csv(filename)
```

2. Obtain a listing of the unique API calls found in the file:

```
calls = np.unique(np.array(logData['API_Call']))\
```

3. Aggregate the data, using `IP_Address` as the means to determine how to group the entries and `API_Call` as the means to determine which column to use for aggregation:

```
aggData = logData.groupby(
    'IP_Address')['API_Call'].agg(list)
```

4. Create a DataFrame to hold the data to analyze later. Begin labeling the data based on its IP address (which is most definitely contrived, but it does add automation to the example instead of forcing you to label the entries by hand). A value of `0` denotes a benign entry, while a value of `1` denotes a hacker entry. Note the use of `ipEntry.sort()` to place all alike IP entries together:

```
    analysisEntries = {}
```

```
    analysisData = pd.DataFrame(columns=calls)
    for ipIndex, ipEntry in zip(aggData.index, aggData):
        ipEntry.sort()
        if ipIndex[0:3] == '172':
            values = [0]
        else:
            values = [1]
```

5. Create columns for the DataFrame based on the API calls:

```
        keys = ['Benign']
        for callType in calls:
            keys.append(callType)
            values.append(ipEntry.count(callType)
```

6. Define each row of the `DataFrame`, using the number of calls from the IP address in question as the values for each column:

```
analysisEntries[ipIndex] = pd.Series(values,
    index=keys)
```

7. Create the `DataFrame` and return it to the caller:

```
analysisData = pd.DataFrame(analysisEntries)
return (analysisData, calls)
```

At this point, you're ready to read the data from disk.

Reading the data from disk

Now that you have a function for reading the data, you can perform the actual act of reading it from disk. The following code shows how:

```
analysisData, calls = ReadDataFromCSV("CallData.csv")
print(analysisData)
```

It's important to look at the data that you've created to ensure it contains the kinds of entries you expected with the pattern you expected. *Figure 5.6* shows an example of the output of this example (the actual output is much longer).

	172:144:0:22	172:144:0:23	172:144:0:24	172:144:0:25	\
Benign	0	0	0	0	
Often1	48	49	38	50	
Often2	23	31	60	48	
Often3	38	41	47	50	
Often4	43	43	38	48	
Often5	43	40	55	43	
Often6	47	41	54	31	
Often7	55	55	44	49	
Often8	57	48	55	57	
Rarely	33	22	28	24	
Regularly1	40	51	33	40	
Regularly2	46	47	35	43	
Regularly3	51	38	51	39	
Sometimes1	29	32	30	42	
Sometimes2	42	54	39	40	

Figure 5.6 – This figure shows the output of the data creation process

Note that each API call appears as a row, while the IP addresses appear as columns. Later, you will find that you have to manipulate this data so that it works with the classifier. The `Benign` column uses 0 to indicate a benign entry and 1 to indicate a hacker entry.

Manipulating the data

The classifier needs two inputs as a minimum for a supervised model. The first is the data itself. The second is the labels used to indicate whether the data is of one category (benign) or another category (hacker). This means taking what you need from the DataFrame and placing it into variables that are traditionally labeled `X` for the data and `y` for the labels. Note that the `X` is capitalized, which signifies a matrix, while `y` is lowercase, which signifies a vector. The following code is used to manipulate the data:

```
X = np.array(analysisData[1:len(calls)+1]).T
print(X)
y = analysisData[0:1]
print(y)
y = y.values.ravel()
print(y)
```

The `X` variable is all of the data from the `DataFrame` that isn't a label. This isn't always the case in ML examples, but it is the case here because you need all the aggregated data shown in *Figure 5.6*. Note the `T` at the end of the `np.array(analysisData[1:len(calls)+1])` call. This addition performs a transform on the data so that rows and columns are switched. Compare the output shown in *Figure 5.7* with the output shown in *Figure 5.6*, and you can see that they are indeed switched. In addition, this is now a two-dimensional array (a matrix).

```
[[48 23 38 43 43 47 55 57 33 40 46 51 29 42]
 [49 31 41 43 40 41 55 48 22 51 47 38 32 54]
 [38 60 47 38 55 54 44 55 28 33 35 51 30 39]
 [50 48 50 48 43 31 49 57 24 40 43 39 42 40]
 [59 39 48 40 38 40 34 43 24 46 42 45 44 33]
 [52 45 55 41 38 54 50 39 30 39 35 33 35 48]
 [45 45 47 42 47 48 49 39 31 41 38 44 41 49]
 [41 41 38 45 52 60 29 44 28 45 44 43 31 29]
 [40 36 47 41 40 48 41 52 31 28 32 55 29 37]
 [40 57 58 39 39 42 48 42 29 44 45 47 28 38]
 [36 37 49 37 56 34 52 45 25 55 50 39 44 31]
 [55 39 43 50 37 47 39 43 26 38 39 38 32 42]
 [52 36 43 38 46 57 35 37 29 27 38 27 38 40]
 [50 43 47 43 42 47 41 54 22 35 39 44 33 43]
 [40 48 47 37 49 43 37 47 25 37 40 38 30 36]
 [47 44 52 40 56 51 41 45 24 37 46 39 26 38]
 [ 0  0  0  0  0  0  0  1 54  4  6 10 23 11]
 [ 0  1  0  0  0  0  0  0 46 12  7  9 23 15]
 [ 0  1  0  0  0  0  0  0 41  9  6  5 22 18]
 [ 0  0  0  0  0  0  0  0 52 12 10  4 34 15]
 [ 0  0  0  0  0  0  0  0 54  9  4  9 25 14]
 [ 0  1  1  1  0  0  0  0 43  5  7 12 28 16]]
```

Figure 5.7 – The X data is turned into a transformed data matrix

The `y` variable consists of the labels in the `Benign` row. However, when you print `y` out, you can see that it retains the labeling from the `DataFrame`, as shown in *Figure 5.8*.

```
        172:144:0:22  172:144:0:23  172:144:0:24  172:144:0:25  172:144:0:26  \
Benign             0             0             0             0             0

        172:144:0:27  172:144:0:28  172:144:0:29  172:144:0:30  172:144:0:31  \
Benign             0             0             0             0             0

        ...  172:144:0:34  172:144:0:35  172:144:0:36  172:144:0:37  \
Benign  ...             0             0             0             0

        175:144:22:2  175:144:22:3  175:144:22:4  175:144:22:5  175:144:22:6  \
Benign             1             1             1             1             1

        175:144:22:7
Benign             1
```

Figure 5.8 – The y variable requires additional manipulation

To change the data into the correct form, the code calls `y.values.ravel()`. The `values` property strips all of the `DataFrame` information, while the `ravel()` call flattens the resulting vector. *Figure 5.9* shows the result.

```
[1 rows x 22 columns]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1]
```

Figure 5.9 – The y variable is now a vector suitable for input to the classifier

The reason that *Figure 5.7* and *Figure 5.9* are so important is that many input errors that occur when working with the classifier have to do with data being in the wrong format. In many cases, the error message doesn't provide a good idea of what the problem is, and trying to figure out precisely

why the classifier is complaining can be difficult. In a few cases, the classifier may only provide a warning, rather than an error, so it's even more difficult to locate the problem.

Creating the detection model

All of the data generation and preparation took a long time in this example, but it's an even longer process in the real world. This example hasn't considered issues such as cleaning data, dealing with missing data, or verifying that data is in the correct range and of the correct type. This is actually the short version of the process, but now it's time to finally build the model and see how well it does in detecting hackers.

Selecting the correct algorithm

This example relies on `RandomForestClassifier`. There are no perfect classifiers (or any other algorithm, for that matter). The reason for using a random forest classifier in this case is that this particular algorithm works well for security needs. You can use the random forest classifier to better understand how the ML model makes a particular decision, which is essential for tuning a model for security needs. A random forest classifier also has these advantages:

- There is less of a chance of bias providing you with incorrect output because there are multiple trees, and each tree is trained on a different subset of data. This algorithm relies on the power of the crowd to reduce the potential for a bad decision.
- This is a very stable algorithm in that new data that shows different trends from the training data may affect one or two trees but is unlikely to affect all of the trees.
- You can use this algorithm for situations where you have both categorical and numerical features. Security situations often require the use of categorical data to better define the environment, so that something such as an API call can be a hacker in one situation but not another.
- This is also the algorithm to rely on when you can't guarantee that real-world data is scaled well. It also handles a certain amount of missingness. Security data may not always be as complete as you want it to be. In fact, it's in the hacker's interest to make the data incomplete.
- Although the random forest algorithm may appear perfect, appearances can be deceiving. You also need to be aware of the disadvantages of using this algorithm:
- The algorithm is quite complex, so training can require more time.
- Tuning the model can prove a lot more difficult, especially with the stealthy manner in which hackers operate. You may find yourself tweaking the model for quite a while to get a good result.
- The resulting model can use a lot more computational resources than other solutions. Given that security solutions may rely on immense databases, you really do need some great computing horsepower to use this algorithm (despite the apparent alacrity of the example due to the small size of the dataset).

These advantages and disadvantages should give you a better idea of what to expect when working with the random forest algorithm in the real world. The critical point here is that you can't skimp on hardware and expect a good result.

Performing the classification

The actual classification process is all about fitting a model to the data that you've created. **Fitting** is the process of creating a curve that differentiates between benign requests on one side and hackers on the other. When a model is **overfitted**, it follows the data points too closely and is unable to make good predictions on new data. On the other hand, when a model is **underfitted**, it means that the curve doesn't follow the data points well enough to make an accurate prediction, even with the original data. The `RandomForestClassifier` used in this example is less susceptible to either overfitting or underfitting than many algorithms are. With this in mind, here are the steps needed to perform classification.

1. Create the classifier and fit it to the data generated and manipulated in the previous sections. In this case, the example uses all of the classifier's default settings except for the data. The output of this step simply says `RandomForestClassifier()`, which tells you that the creation process was successful:

```
clf=RandomForestClassifier()
clf.fit(X,y)
```

2. Generate test data to test the model's performance. This is the same approach that was used to create the training data, except this data is generated using a different seed value and settings to ensure uniqueness:

```
random.seed(19)
threshold, benignCount, hackerCount, data = \
    CreateAPITraffic(benignIP=benignIPs,
        apiEntries=callNames, bias=.95, outlier=15)
print(f"There are {benignCount} benign entries " \
    f"and {hackerCount} hacker entries " \
        f"with a threshold of {threshold}.")
fields = ['Time', 'IP_Address', 'API_Call']
SaveDataToCSV(data, fields, "TestData.csv")
```

   Something to note about this step is that the `bias` and `outlier` settings are designed to produce the effect of a more realistic attack by making the attacker stealthier. The output of this step is as follows:

```
There are 4975 benign entries and 25 hacker entries with a
threshold of 1.4000000000000021.
```

3. Perform the actual classification. This step includes manipulating the data using the same approach as the training data:

```
testData, testCalls = ReadDataFromCSV("TestData.csv")
X_test = np.array(testData[1:len(calls)+1]).T
```

```
y_test = testData[0:1].values.ravel()

y_pred = clf.predict(X_test)

print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
```

This process ends with a measurement of the accuracy of the model, which is the most important part of the process. Data scientists often use other measures to verify the usefulness of a model and to check which factors contributed most to the model's accuracy. You can also create graphs to show data distributions and how the decision process is made. In this case, the output says that the model is 95.5 percent accurate (although this number can vary, depending on how the training and testing data is configured). It's possible to improve the accuracy in a number of ways, but the best way would be to provide additional data.

Using a subprocess in Python example

Real-time defenses often depend on using the correct coding techniques in your ML code. For example, something as innocuous as a subprocess could end up torpedoing your best efforts. A **subprocess** is a task that you perform from the current process as a separate entity. Even though you don't see it used often in Python ML examples, you do run across it from time to time. For example, you might use a subprocess to download a dataset or extract data from a `.zip` file. It's handy to see how this kind of problem can appear totally innocent and how you might not even notice it until a hacker has exploited it. This first example, which won't run on online environments such as Google Colab because the IDE provides protections against it, shows the wrong way to perform tasks using a subprocess. The reason that this code is included is to provide you with a sort of template to see incorrect coding practices in general. You'll find this example in the `MLSec; 05; Real Time Defenses.ipynb` file for this chapter:

```
from subprocess import check_output
MyDir = check_output("dir", shell=True)
print(MyDir.decode('ascii'))
```

Even though the code executes as a script at the command line and provides a listing of the current working directory, the use of `shell=True` creates a potential RCE hole in your code, according to https://docs.python.org/3/library/subprocess.html#security-considerations and https://www.hacksplaining.com/prevention/command-execution. In order to avoid the RCE hole, you can use this form of the code instead (note that this code may not work on your Linux system):

```
from subprocess import check_output
MyDir = check_output(['cmd','/c','dir'])
print(MyDir.decode('ascii'))
```

In this case, you create a new command processor, execute the directory command, and then close the command processor immediately by using the `/c` command-line switch. Of course, this approach requires some knowledge of the underlying operating system. It's often better to find a workaround that doesn't include calling the command processor directly, such as the solution shown here (which will definitely run on your Linux system):

```
from os import listdir
from os import getcwd
MyDir = listdir(getcwd())
print(MyDir)
```

This version obtains the directory as an easily processed list, in addition to protecting it from an RCE attack. In many respects, it's also a lot easier to use, albeit less flexible, because now you don't have access to the various command-line switches that you would when using the `check_output()` version. For example, this form of `check_output()` will obtain only the filenames and not the directories as well: `MyDir = check_output(['cmd','/c','dir', '/a-d'])`. You can also control the ordering of the directory output using the `/o` command-line switch with the correct sub-switch, such as `/os`, to sort the directory output by size.

Working with Flask example

You may want to expose your ML application to the outside world using a REST-type API. Products such as Flask (https://www.fullstackpython.com/flask.html) make this task significantly easier. However, when working with Flask, you must exercise caution because you can introduce XSS errors into your code. For example, look at the following code (also found in the `MLSec; 05; Real Time Defenses.ipynb` file for this chapter):

```
from flask import Flask, request
app = Flask(__name__)
@app.route("/")
def say_hello():
    your_name = request.args.get('name')
    return "Hello %s" % your_name
```

In this case, you create a Flask app and then define how that app is going to function. You start at the uppermost part of the resulting service (yes, it runs fine in Jupyter Notebook). The `hello()` function obtains a name as a request argument using `request.args.get('name')`. It then returns a string with the name of the website for display. To run this example, you place this call in a separate cell:

```
app.run()
```

When you run the second cell, it won't exit. Instead, you'll see a message such as this one (along with some additional lines that aren't a concern in this example):

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Note that the address the server provides may vary, so you need to use the correct URL or the application will fail. When you see the running message, you can open a new tab in your browser and type something such as this as an address: `http://127.0.0.1:5000/?name=John`. The browser page will display `Hello` and then your name. However, if you type the following address instead, `http://127.0.0.1:5000/?name=<script>alert(1)</script>`, you will see an alert dialog. Of course, you could have run any script, not just this one. To stop the server, click the **Stop** button in Jupyter Notebook. Here's a fix for this problem:

```
from flask import Flask, request, escape
app = Flask(__name__)
@app.route("/")
def say_hello():
    your_name = request.args.get('name')
    return "Hello %s" % escape(your_name)
```

The use of `escape(your_name)` means that the server won't actually execute the script. Instead, it escapes the script as text and displays the text on screen. It's important to note that Flask creates

a running log for you as you experiment with the test page. This log appears as part of the Jupyter Notebook output, such as this:

```
127.0.0.1 - - [20/Apr/2021 16:56:18] "GET /?name=John HTTP/1.1" 200
-
127.0.0.1 - - [20/Apr/2021 16:56:27] "GET
/?name=%3Cscript%3Ealert(1)%3C/script%3E HTTP/1.1" 200 -
```

From an ML perspective, you could place this log in a file and then use techniques such as the one demonstrated in the *Using supervised learning example* section to perform analysis on it. The log provides everything you need to begin hunting the hacker down in real time, as the hacker is experimenting with the API, rather than waiting until it's too late.

Asking for human intervention

Figure 5.2 and Figure 5.5 make it clear that a significant part of this example includes the use of human monitoring to ensure that nothing has gone awry with the supervised learning example. The setup could constantly check the accuracy of predictions that it's making, for example, and send an alert of some type to the administrator when accuracy begins to fall below a certain point. The examples in the *Using a subprocess in Python example* and *Working with Flask example* sections also point to the need for human monitoring of the system. If you take anything at all away from this chapter, it should be that ML augments the ability of humans to detect and mitigate threats – it can't act as a replacement for humans.

Unfortunately, humans are easily overwhelmed. ML can perform constant monitoring in a consistent way and then ask a human for intervention when the monitoring process detects anything unusual. This is the essential component that will make real-time monitoring in your organization possible – the realization that ML can direct a human's attention to the right place at the right time.

This part of the chapter takes you beyond what most organizations do today – that is, addressing threats in real time. However, to have a real edge against hacker attacks, you really need to know what a hacker is likely to do in the future. Predicting the future always comes with certain negative connotations because it really isn't possible to predict anything with complete accuracy. Otherwise, everyone engaged in the stock market would be a millionaire. However, when it comes to security, it's possible for the observant administrator to use ML to predict future attacks with enough accuracy to make the process worthwhile and possibly thwart a hacker before any mischief begins.

Links

Bosch:
https://www.boschsecurity.com/xc/en/solutions/video-systems/video-analytics/

Nelly's Security:
https://www.nellyssecurity.com/blog/articles/what-is-deep-learning-ai-and-why-is-it-important-for-video-surveillance

Windows Server 2019:
https://www.rootusers.com/how-to-enable-telnet-client-in-windows-server-2019/

Windows 10:
https://social.technet.microsoft.com/wiki/contents/articles/38433.windows-10-enabling-telnet-client.aspx

Windows 11: https://petri.com/enable-telnet-client-in-windows-11-and-server-2022/

You can read about the differences between Telnet and SSH at https://www.tutorialspoint.com/difference-between-ssh-and-telnet.

PowerShell alternative of Telnet command: https://www.techtutsonline.com/powershell-alternative-telnet-command/

Windows 10 SSH vs. PuTTY: Time to Switch Your Remote Access Client?: https://www.makeuseof.com/tag/windows-10-ssh-vs-putty/

Python programming language hurries out update to tackle remote code vulnerability (RCE): https://www.zdnet.com/article/python-programming-language-hurries-out-update-to-tackle-remote-code-vulnerability/

NumPy (https://numpy.org/)

Remote code execution in NumPy module for Python: https://www.cybersecurity-help.cz/vdb/SB2019012101

AI, machine learning and your access network at https://www.networkworld.com/article/3256013/ai-machine-learning-and-your-access-network.html

Adaptive Authentication and Machine Learning at https://towardsdatascience.com/adaptive-authentication-and-machine-learning-1b460ae53d84

Articles such as *Deep learning methods in network intrusion detection: A survey and an objective comparison* (https://www.sciencedirect.com/science/article/abs/pii/S1084804520302411) express a preference for deep feedforward networks over autoencoders and **deep belief networks** (**DBNs**) (see https://www.analyticsvidhya.com/blog/2022/03/an-overview-of-deep-belief-network-dbn-in-deep-learning/ for details).

Network intrusion detection system: A systematic study of machine learning and deep learning approaches at https://onlinelibrary.wiley.com/doi/full/10.1002/ett.4150.

Deep Learning and Machine Learning Techniques for Change Detection in Behavior Monitoring at http://ceur-ws.org/Vol-2559/paper3.pdf

Network Anomaly Detection and User Behavior Analysis using Machine Learning at https://www.ijcaonline.org/archives/volume175/number13/vadgaonkar-2020-ijca-920635.pdf

Botnet Detection with ML at https://medium.com/@EbubekirBbr/botnet-detection-with-ml-afd4fa563d31

The Machine Learning Based Botnet Detection white paper at http://cs229.stanford.edu/proj2006/NivargiBhaowalLee-MachineLearningBasedBotnetDetection.pdf

Build botnet detectors using machine learning algorithms in Python at https://hub.packtpub.com/build-botnet-detectors-using-machine-learning-algorithms-in-python-tutorial/

One such site is *Build botnet detectors using machine learning algorithms in Python* at https://hub.packtpub.com/build-botnet-detectors-using-machine-learning-algorithms-in-python-tutorial/. In this case, you work with the approximately 2 GB CTU-13 dataset found at https://mcfp.felk.cvut.cz/publicDatasets/CTU-13-Dataset/ and described at https://mcfp.weebly.com/the-ctu-13-dataset-a-labeled-dataset-with-botnet-normal-and-background-traffic.html

The second example also relies on a Twitter dataset that's apparently no longer available, but it's possible to modify the code to use the Kaggle dataset found at https://www.kaggle.com/davidmartngutirrez/twitter-bots-accounts.

Honeypot: https://www.imperva.com/learn/application-security/honeypot-honeynet/.

The most common use for AI today is in creating a honeypot that provides increased intrusion detection capability, as described in *AI-powered honeypots: Machine learning may help improve intrusion detection* at https://portswigger.net/daily-swig/ai-powered-honeypots-machine-learning-may-help-improve-intrusion-detection. Unfortunately, hackers have also been busy, as explained in *Automatic Identification of Honeypot Server Using Machine Learning Techniques* at https://www.hindawi.com/journals/scn/2019/2627608/.

Generation of Honeypot Data Patent details: https://www.freepatentsonline.com/y2020/0186567.html.

A number of vendors now provide support for data-centric security, as described at https://www.g2.com/categories/data-centric-security

The Worldwide Data-centric Security Industry is Expected to Reach $9.8 Billion by 2026 at a CAGR of 23.1% from 2020 at https://www.globenewswire.com/en/news-release/2021/05/07/2225398/28124/en/The-Worldwide-Data-centric-Security-Industry-is-Expected-to-Reach-9-8-Billion-by-2026-at-a-CAGR-of-23-1-from-2020.html

12 Tips To Help Shift Your Business To Data-Centric Cybersecurity: https://www.forbes.com/sites/forbestechcouncil/2020/02/14/12-tips-to-help-shift-your-business-to-data-centric-cybersecurity/?sh=2098f174555d

Big Data To Good Data: Andrew Ng Urges ML Community To Be More Data-Centric And Less Model-Centric (https://analyticsindiamag.com/big-data-to-good-data-andrew-ng-urges-ml-community-to-be-more-data-centric-and-less-model-centric/)

PyGrid: https://github.com/OpenMined/PyGrid

PySyft: https://github.com/OpenMined/PySyft

Multi-level host-based intrusion detection system for Internet of things at https://link.springer.com/article/10.1186/s13677-020-00206-6

Interestingly enough, even newer Wi-Fi garage door openers can provide an opening for hackers (https://smarthomestarter.com/can-garage-door-openers-be-hacked-smart-garages-included/).

https://www.csoonline.com/article/3258748/the-mirai-botnet-explained-how-teen-scammers-and-cctv-cameras-almost-brought-down-the-internet.html

Besides direct data manipulation, IoT devices represent these sorts of threats to your business and network:

- An attacker monitors users of interest to see whether they will tell family members about potentially sensitive information. There is an app available to hackers to make this possible with very little effort (see https://www.siliconrepublic.com/enterprise/amazon-alexa-google-home-smart-speaker-research for details).
- IoT devices, such as smart speakers, are sensitive enough to hear a heartbeat (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7943557/), so whispering won't prevent the divulging of sensitive information to hackers who are listening.
- The use of a group of IoT devices can create a DDOS attack, such as the Mirai attack, where the botnet turned IoT devices running on ARC processors into a group of zombies (https://www.csoonline.com/article/3258748/the-mirai-botnet-explained-how-teen-scammers-and-cctv-cameras-almost-brought-down-the-internet.html).

- The attacker gains access to a home network with access to your business network through an IoT device.
- In order to reduce employee effectiveness, the attacker bricks company-issued IoT devices using a botnet, such as the BrickerBot malware (https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/brickerbot-malware-permanently-bricks-iot-devices).

Wireshark can also be used to create trace data, as described at https://2nwiki.2n.cz/pages/viewpage.action?pageId=52265299

LTTng (https://lttng.org/)

Common Trace Format (**CTF**) (https://diamon.org/ctf/)

barectf (https://github.com/efficios/barectf)

Babeltrace API (https://babeltrace.org/).

Biometrics: definition, use cases and latest news article at https://www.thalesgroup.com/en/markets/digital-identity-and-security/government/inspired/biometrics

Current uses of DNA for biometrics: https://www.ibia.org/biometrics-and-identity/biometric-technologies/dna

Finger-vein biometric identification using convolutional neural network article at https://www.researchgate.net/figure/Error-rates-of-the-CNN-models-tested-in-cross-validation-process_fig4_299593157

Network traffic datasets at https://sites.google.com/a/udayton.edu/fye001/simple-page/network-traffic-classification

See the *TCP Flags* article at https://www.geeksforgeeks.org/tcp-flags/ for more information about how these TCP flags work.

Evaluation of Supervised Machine Learning for Classifying Video Traffic at https://core.ac.uk/download/pdf/51093126.pdf

Sniffing Telnet Using Wireshark: http://blog.johnmuellerbooks.com/2011/06/07/sniffing-telnet-using-wireshark/.

New Directions in Automated Traffic Analysis at https://pschmitt.net/

Why the Little Dutch Boy Never Put his Finger in the Dike at https://www.dutchgenealogy.nl/why-the-little-dutch-boy-never-put-his-finger-in-the-dike/

You can also create graphs to show data distributions (https://machinelearningmastery.com/statistical-data-distributions/) and how the decision process is made (https://stackoverflow.com/questions/40155128/plot-trees-for-a-random-forest-in-python-with-scikit-learn and https://mljar.com/blog/visualize-tree-from-random-forest/).

CyberSecurity Attack Prediction: A Deep Learning Approach at https://dl.acm.org/doi/fullHtml/10.1145/3433174.3433614 and *A deep learning framework for predicting cyber attacks rates* at https://link.springer.com/article/10.1186/s13635-019-0090-6

Botnet Forensic Analysis Using Machine Learning at https://www.hindawi.com/journals/scn/2020/9302318/

Prediction of Future Terrorist Activities Using Deep Neural Networks at https://www.hindawi.com/journals/complexity/2020/1373087/

Chapter 6

Technical requirements

This chapter requires that you have access to either Google Colab or Jupyter Notebook to work with the example code. The *Requirements to use this book* section of *Chapter 1*, *Defining Machine Learning Security*, provides additional details on how to set up and configure your programming environment. When testing the code, use a test site, test data, and test APIs to avoid damaging production setups and to improve the reliability of the testing process. Testing on a non-production network is highly recommended but not absolutely necessary. Using the downloadable source code is always highly recommended. You can find the downloadable source on the Packt GitHub site at https://github.com/PacktPublishing/Machine-Learning-Security-Principles or my website at http://www.johnmuellerbooks.com/source-code/.

Figures

Dataset	Training Type	Description	Location
Cifar-10	Image	A set of datasets containing smallish 32x32 images that would prove useful in certain types of anomaly detection training. The datasets come in a variety of sizes, with `Cifar-10` containing 60,000 images in 10 classes and `Cifar-100` containing 60,000 images in 100 classes.	https://www.cs.toronto.edu/~kriz/cifar.html
ImageNet	Image	An immense dataset at 150 GB that holds 1,281,167 images for training and 50,000 images for validation, organized into 1,000 categories. The dataset is so popular that the people who manage it recently had to update their servers and reorganize the website. All of the	https://www.image-net.org/update-mar-11-2021.php

		images are labeled, making them suitable for supervised learning.	
Imagenette and Imagewoof	Image	A set of datasets containing images of various sizes, including 160-pixel and 320-pixel sizes. These datasets aren't to be confused with ImageNet (and the site makes plenty of fun of the whole situation). The `Imagenette` datasets consist of 10 classes using easily recognizable images, while the `Imagewoof` dataset consists of a single class, all dogs that can be incredibly tough to recognize. This series of datasets also include noisy images and purposely changed labels in various proportions to the correct labels.	https://github.com/fastai/imagenette
Modified National Institute of Standards and Technology (**MNIST**)	Image	Handwritten digits with 60,000 training and 10,000 testing examples. The digits are already size normalized and centered in a fixed-size image. Many Python packages include this dataset, such as `scikit-learn` (https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html).	http://yann.lecun.com/exdb/mnist/

Numenta Anomaly Benchmark (**NAB**)	Streaming anomaly detection	Contains 58 labeled real-world and artificial time-series data files you can use to test your anomaly detection application in a streamed environment. The data files show anomalous behavior but not on a consistent basis, making it possible to train a model to recognize the anomalous behavior. This dataset isn't officially supported on Windows 10 or (theoretically) 11.	https://github.com/numenta/NAB
One Billion Word Benchmark	**Natural Language Processing** (**NLP**)	A truly immense dataset taken from a news commentary site. The dataset consists of 829,250,940 words with 793,471 unique words. The problem with this dataset is that the sentences are shuffled, so the ability to determine context is limited.	https://opensource.google/projects/lm-benchmark
Penn Treebank	NLP	Contains 887,521 training words, 70,390 validation words, and 78,669 test words with the text preprocessed to make it easy to work with.	https://deepai.org/dataset/penn-treebank
WikiText-103	NLP	A much larger version of the `WikiText-2` dataset contains 103,227,021 training words, 217,646 validation words, and	https://blog.einstein.ai/the-wikitext-long-term-dependency-language-modeling-dataset/

		245,569 testing words taken from Wikipedia articles.	
WikiText-2	NLP	Contains 2,088,628 training words, 217,646 validation words, and 245,569 test words taken from Wikipedia articles. This dataset provides a significantly improved environment over Penn Treebank for training models because preprocessing is kept to a minimum.	https://blog.einstein.ai/the-wikitext-long-term-dependency-language-modeling-dataset/

Figure 6.1 – Datasets that work well for anomaly training

Method	Type	Description
Cook's distance	Model-specific	This estimates the variations in regression coefficients after removing each observation one at a time. The main goal of this method is to determine the influence exerted by each data point, with data points having undue influence being outliers or novelties. This technique is explored in the *Relying on Cook's distance* section of the chapter.
Interquartile range (**IQR**)	Univariate	This considers the placement of data points within the first and third quartiles normal. All data points 1.5 or more times outside of these quartiles are considered outliers. This technique is usually displayed graphically using a box plot. It was created by John Tukey, an American scientist best known for creating the **fast Fourier transform** (**FFT**). This technique is explored in the *Relying on the interquartile range* section of the chapter.
Isolation Forest	Multivariate	This detects outliers using the anomaly score of the Isolation Forest (a type of random forest). The article at https://towardsdatascience.com/outlier-detection-with-isolation-forest-3d190448d45e provides additional insights into this approach.
Mahalanobis distance	Multivariate	This measures the distances between points in multivariate space. It detects outliers by accounting for the shape of the data observations. The threshold for declaring a data point an outlier normally relies on either **standard deviation** (**STDEV**) or **mean absolute deviation** (**MAD**). You can find out more about this

		technique at https://www.statisticshowto.com/mahalanobis-distance/.
Minimum Covariance Determinant (**MCD**)	Multivariate	Based on the Mahalanobis distance, this technique uses the mean and covariance of all data, including the outliers, to determine the difference scores between data points. This approach is often considered far more accurate than the Mahalanobis distance. You can find out more about this approach in the article at https://onlinelibrary.wiley.com/doi/full/10.1002/wics.1421.
Pareto	Model-specific	This assesses the reliability and approximate convergence of Bayesian models using estimates for the `k` shape parameter of the generalized Pareto distribution. This method is named after Vilfredo Pareto, the Italian civil engineer, economist, and sociologist. It's the source of the 80/20 rule. The article at https://www.tandfonline.com/doi/abs/10.1080/00949655.2019.1586903?journalCode=gscs20 provides additional information about this method.
Principle component analysis (**PCA**)	Multivariate	This restructures the data, removing redundancies and ordering newly obtained components according to the amount of the original variance that they express. Using this approach makes multivariate outliers particularly evident. This technique is explored in the *Relying on principle component analysis* section of the chapter.
Z-score	Univariate	This describes a data point as a deviation from a central value. To use this approach, you must calculate the z-scores one column at a time. Different sources place different scores as the indicator of an outlier—as low as 1.959 and as high as 3.00. This technique is explored in the *Relying on z-score* section of the chapter.

Figure 6.2 – Common methods for detecting outliers

Hands-on Sections

Checking data validity

One of the things you need to ask with regard to anomalies is whether the data you're using is valid. If there are too many outliers, any results from the data analysis you perform will be skewed, and you'll derive erroneous results. In most cases, the outliers you want to find are those that relate to some sort of data entry error. For example, it's unlikely that someone who is age 2 will somehow have a PhD. However, outliers can also be subtle. For example, a hacker could boost the prices of homes in a certain area by a small amount so that a confederate with homes to sell can get a better asking price. The point is that data tends to have certain characteristics and fall within certain ranges, so you can use an engineering approach to finding potential outliers. You must then resort to intuition to decide whether the errant values are novelties or actual outliers that you then research and fix.

The real problem is the outlier that deviates from the norm by a wide margin. Depending on the ML algorithms you use, and the method used to determine errors, a carefully placed outlier can cause the model to change the weights it uses for making decisions by a large amount. The K-means algorithm is just one of many that fall into this category. Fortunately, not all algorithms are sensitive to a few outliers with unbelievable values. If you use the K-medoids clustering variant instead, one or two outliers will have much less of an effect. That's because K-medoids clustering uses the actual point in the cluster to represent a data item, rather than the mean point as the center of a cluster. Some algorithms, such as random forest, are supposedly less affected by outliers, according to some people and not so robust according to others.

The safe assumption is that outliers create problems and that you should remove them from your data. A number of methods exist to detect outliers. Some of them work with one feature (**univariate**); others focus on multiple features (**multivariate**). The approach you use depends on what you suspect is wrong with the data. The examples in this section all rely on the California housing dataset, which is easily available from scikit-learn and provides well-known values for the comparison of techniques. The `MLSec; 06; Check Data Validity.ipynb` file contains source code and some detailed instructions for performing validity checks on your data.

Relying on the interquartile range

The univariate approach to locating outliers was originally proposed by John Tukey as **Exploratory Data Analysis** (**EDA**) IQR. You use a box plot to see the median, 25 percent, and 75 percent values. By subtracting the 25 percent value from the 75 percent value to obtain the IQR and multiplying by 1.5, you obtain the anticipated range of values for a particular dataset feature. The extreme end of the range appears as whiskers on the plot. Any values outside this range are suspected outliers. Use these steps to see how this process works:

1. Import the required libraries:

```
from sklearn.datasets import fetch_california_housing
import pandas as pd
%matplotlib inline
```

2. Manipulate the dataset so that it appears in the proper form:

```
california = fetch_california_housing(as_frame = True)
X, y = california.data, california.data
X = pd.DataFrame(X, columns=california.feature_names)
print(X)
```

The example prints the resulting dataset so that you can see what it looks like, as shown in *Figure 6.3*. Across the top, you can see the features used for the dataset:

```
       MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0      8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1      8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2      7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3      5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4      3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85
...       ...       ...       ...        ...         ...       ...       ...
20635  1.5603      25.0  5.045455   1.133333       845.0  2.560606     39.48
20636  2.5568      18.0  6.114035   1.315789       356.0  3.122807     39.49
20637  1.7000      17.0  5.205543   1.120092      1007.0  2.325635     39.43
20638  1.8672      18.0  5.329513   1.171920       741.0  2.123209     39.43
20639  2.3886      16.0  5.254717   1.162264      1387.0  2.616981     39.37

       Longitude
0        -122.23
1        -122.22
2        -122.24
3        -122.25
4        -122.25
...          ...
20635    -121.09
20636    -121.21
20637    -121.22
20638    -121.32
20639    -121.24

[20640 rows x 8 columns]
```

Figure 6.3 – The data manipulation creates a tabular view with columns labeled with feature names

3. Create the box plot:

```
X.boxplot('MedInc',return_type='axes')
```

The box plot specifically shows the median income (`MedInc`) feature values. The output shown in *Figure 6.4* indicates that the full range of values runs from about $9,000 to $80,000. The median value is around $35,000, with a 25 percent value of $25,000 and a 75 percent value of $50,000 (for an IQR of $25,000). However, some values are above $80,000 and are suspected of being outliers:

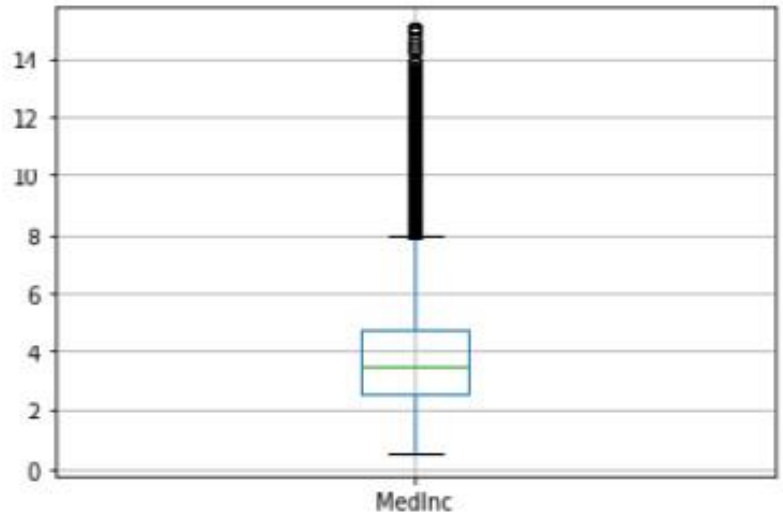

Figure 6.4 – The box plot provides details about the data range and indicates the presence of potential outliers

The next section relies on the same variable, X, to perform the next outlier detection process, PCA. It shows you the potential outliers in the MedInc feature in another way.

Relying on principle component analysis

In viewing the output shown in *Figure 6.4*, you can see that there are potential outliers, but it's hard to tell how many outliers, and how they relate to other data in the dataset. The need to understand outliers better is one of the reasons you resort to a multivariate approach such as PCA. The actual math behind PCA can be daunting if you're not strong in statistics, but you don't actually need to understand it well to use PCA successfully.

The number of variables you choose depends on the scenario that you're trying to understand. In this case, the example looks for a correlation between median income and house age. When working through a similar security problem, you might look for a correlation between unsuccessful login attempts and time of day or less-used API calls and network load. It's all about looking for patterns in the form of anomalies that you can use to detect hacker activity.

This example also relies on a scatterplot to show the results of the analysis. A scatterplot will tend to emphasize where outliers appear in the data. The following steps will lead you through the process of working with PCA using the California housing dataset, but the same principles apply to other sources of data, such as API access logs:

1. Import the libraries needed for this example:

```
from sklearn.decomposition import PCA

from sklearn.preprocessing import scale

import matplotlib.pyplot as plt
```

2. Consider the need to scale your example by running the following code:

```
print(X[["MedInc", "HouseAge"]][0:5])

print(scale(X[["MedInc", "HouseAge"]])[0:5])
```

The output shown in *Figure 6.5* shows how the original data is scaled in the `MedInc` and `HouseAge` columns so that it's the same relative size. **Scaling** is an essential part of working with PCA because it makes the comparison of two variables possible. Otherwise, you have an apples-to-oranges comparison that will never provide you with any sort of solid information:

```
   MedInc  HouseAge
0  8.3252      41.0
1  8.3014      21.0
2  7.2574      52.0
3  5.6431      52.0
4  3.8462      52.0
[[ 2.34476576  0.98214266]
 [ 2.33223796 -0.60701891]
 [ 1.7826994   1.85618152]
 [ 0.93296751  1.85618152]
 [-0.012881    1.85618152]]
```

Figure 6.5 – Scaling is an essential part of the multivariate analysis process

3. Use PCA to fit a model to the scaled data:

```
pca = PCA(n_components=2)

pca.fit(scale(X))
```

```
C = pca.transform(scale(X))

print(C)

print("Original Shape: ", X.shape)

print("Transformed Shape: ", C.shape)
```

The `pca.transform()` method performed dimensionality reduction. You can see the result of the fitting process and dimensionality reduction in *Figure 6.6*:

```
[[ 1.88270434 -0.50336186]
 [ 1.37111955 -0.12140565]
 [ 2.08686762 -0.5011357 ]
 ...
 [ 1.40235696 -1.09642559]
 [ 1.5429429  -1.05940835]
 [ 1.40551621 -0.89672727]]
Original Shape:  (20640, 8)
Transformed Shape:  (20640, 2)
```

Figure 6.6 – Fitting the data and then transforming it provides data you can plot

4. Plot the data to see the result of the analysis:

```
plt.title('PCA Outlier Detection')

plt.xlabel('Component 1')

plt.ylabel('Component 2')

plt.scatter(C[:,0],C[:,1], s=2**7, edgecolors='white',

    alpha=0.85, cmap='autumn')

plt.grid(which='minor', axis='both')

plt.show()
```

The output in *Figure 6.7* clearly shows the outliers in the upper-right corner of the plot. The *x*- and *y*-axis data appears first in the call to `plt.scatter()`. The remainder of the arguments affects presentation: `s` for the marker size, `edgecolors` for the line around each marker (to make them easier to see), `alpha` to control the marker transparency, and `cmap` to control the marker colors:

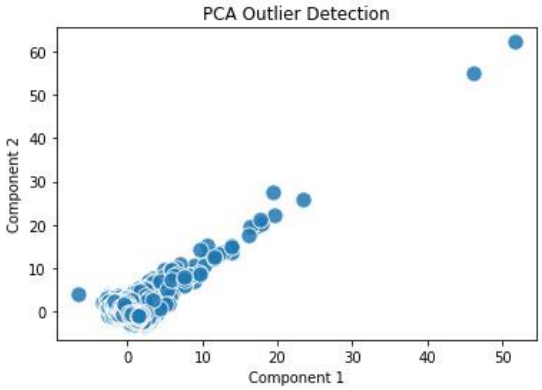

Figure 6.7 – The outliers appear in the upper right corner of the plot

To use this method effectively, you need to try various comparisons to find the correlations that will make the security picture easier for you to discern. In addition, you may find that you need to use more than just two features, as this example has done. Perhaps there is a correlation between invalid logins, IP addresses, and time of day. Until you model the data for your particular network, you really don't know what the anomalies you suspect will tell you about your security picture.

Going overboard with analysis

It would be quite easy to go overboard with the analysis you want to perform by testing a ridiculous number of combinations and looking for patterns that aren't there. Human intuition is extremely important in determining which combinations to try and figuring out what data actually is an anomaly and which is a novelty. The computer can't perform this particular task for you. PCA only works when the human performing it makes decisions based on previous experience, current trends, network uniqueness, and anticipated or viewed behaviors.

Relying on Cook's distance

Cook's distance is all about measuring influence. It detects the amount of influence that a particular observation has on a model. When it comes to anomalies, a very influential observation is likely an outlier or novelty. Using Cook's distance properly will help you see specifically where the outliers reside in the dataset. Removing these outliers can help you create a better baseline of what is normal for the dataset and makes anomaly detection easier.

Note that before you can run this example, you need to have Yellowbrick (https://www.scikit-yb.org/en/latest/) installed. The following command will perform the task for you:

```
conda install -c districtdatalabs yellowbrick
```

However, if you don't have Anaconda installed on your system, you can also add this code to a cell in your notebook to install it:

```
modules = !pip list
installed = False
for item in modules:
    if ('yellowbrick' in item):
        print('Yellowbrick installed: ', item)
        installed = True
if not installed:
    print('Installing Yellowbrick...')
    !pip install yellowbrick
```

In this second case, the code checks for the presence of Yellowbrick on your system and installs it if it isn't installed. This is a handy piece of code to keep around because you can use it to check for any dependency and optionally install it when not present. In addition, this piece of code will work with alternative IDEs, such as Google Colab. Using this check will make your code a little more bulletproof. If Yellowbrick is already installed, you will see an output similar to this showing the version you have installed:

```
Yellowbrick installed:  yellowbrick                    1.5
```

Once you have Yellowbrick installed, you can begin working with the example code using the following steps:

1. Import the required dependencies:

```
from yellowbrick.regressor import CooksDistance
```

2. Import a new copy of the California housing dataset that's formed in a specific way as show in the following code snippet:

```
california = fetch_california_housing(as_frame = True)
X = california.data["MedInc"].values.reshape(-1, 1)
y = range(len(X))
```

3. Printing the result of the import shows how `X` is formatted for use with Yellowbrick. The `y` variables are simply values from `0` through the length of the data to act as an index for the output:

```
print(X)
```

Figure 6.8 shows a sample of the output you should see:

```
[[8.3252]
 [8.3014]
 [7.2574]
 ...
 [1.7   ]
 [1.8672]
 [2.3886]]
```

Figure 6.8 – Ensure your data is formatted correctly for the visualizer

4. Create a `CooksDistance()` visualizer to see the data, use `fit()` to fit the data to it, and then display the result on screen using `show()`:

```
visualizer = CooksDistance()
visualizer.fit(X, y)
visualizer.show()
```

The **stem plot** output you see should look similar to that shown in *Figure 6.9*. Notice that the anomalies are readily apparent and that you know areas that contain records with anomalous data. In addition, the red dashed line tells you the average of the records so that you can see just how far out of range a particular value is:

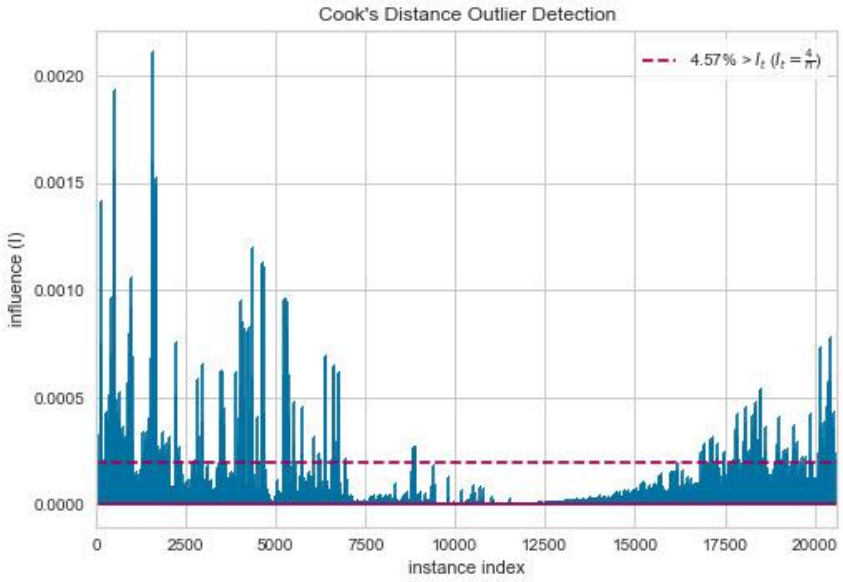

Figure 6.9 – The Cook's distance approach gives you specific places to look for anomalies

Cook's distance reduces the work you need to perform to determine where anomalies occur so that you aren't searching entire datasets for them. If this were a data stream, the `y` value could actually be time increments so that you'd know when the anomalies occur. By slicing and dicing your data in specific ways, you can quickly check for anomalies in a number of useful ways.

Relying on the z-score

Obtaining the z-score for data points in a dataset can help you detect specific instances of outliers. There are a number of ways to do this, but the basic idea is to determine when a particular data point value falls outside of a specific range, normally beyond the third deviation. Fortunately, the math for performing this task is simplified by using NumPy (https://numpy.org/), rather than calculating it manually.

Using `seaborn` (https://seaborn.pydata.org/) greatly reduces the amount of work you need to do to visualize your data distribution. This example relies on `seaborn` version 0.11 and above. If you see an error message stating that `module 'seaborn' has no attribute 'displot'`, it means that you have an older version installed. You can update your copy of `seaborn` using this command:

```
pip install --upgrade seaborn
```

The following example shows a technique for actually listing the records that fall outside of the specified range and understanding how the records in the dataset fall within a particular distribution:

1. Import the required dependencies:

```
import numpy as np

import seaborn as sns
```

2. Obtain an updated copy of the California housing data for median income:

```
X = california.data["MedInc"]
```

3. Determine the mean and standard deviation for the data:

```
mean = np.mean(X)

std = np.std(X)

print('Mean of the dataset is: ', mean)

print('Standard deviation is: ', std)

You'll see an output like this when you run the code:

Mean of the dataset is:  3.8706710029070246
Standard deviation is:  1.899775694574878
```

4. Create a list of precisely which records fall outside the third deviation:

```
threshold = 3

record = 1

z_scores = []

for i in X:

    z = (i - mean) / std

    z_scores.append(z)

    if z > threshold:

        print('Record: ', record, ' value: ', i)

    record = record + 1
```

Figure 6.10 shows the results of performing this part of the task. Notice the method for calculating the z-score, `z`, is simplified by using NumPy. The resulting data tells you where each record falls in the expected range. The output is the records that fall outside the third standard deviation and may be an outlier:

```
Record:  132  value:  11.6017
Record:  410  value:  10.0825
Record:  511  value:  11.8603
Record:  512  value:  13.499
Record:  513  value:  12.2138
Record:  515  value:  12.3804
Record:  924  value:  9.7194
Record:  978  value:  10.9506
Record:  987  value:  10.3203
Record:  1542  value:  9.5862
Record:  1562  value:  9.7037
Record:  1564  value:  10.3345
Record:  1565  value:  12.5915
Record:  1567  value:  15.0001
Record:  1575  value:  9.8708
Record:  1583  value:  10.7372
Record:  1584  value:  13.4883
Record:  1587  value:  12.2478
Record:  1592  value:  10.4549
```

Figure 6.10 – You now have a precise list of the records that could contain outliers

5. Plot the standard distribution of the z-scores:

```
axes = sns.displot(z_scores, kde=True)
axes.fig.suptitle('Z-Scores for MedInc')
axes.set(xlabel='Standard Deviation')
```

Figure 6.11 shows you how this data fares. This graph shows that not many of the values are significantly out of range (the third deviation, either plus or minus), but that a few are and that some of them are quite a bit outside the range (up to six deviations):

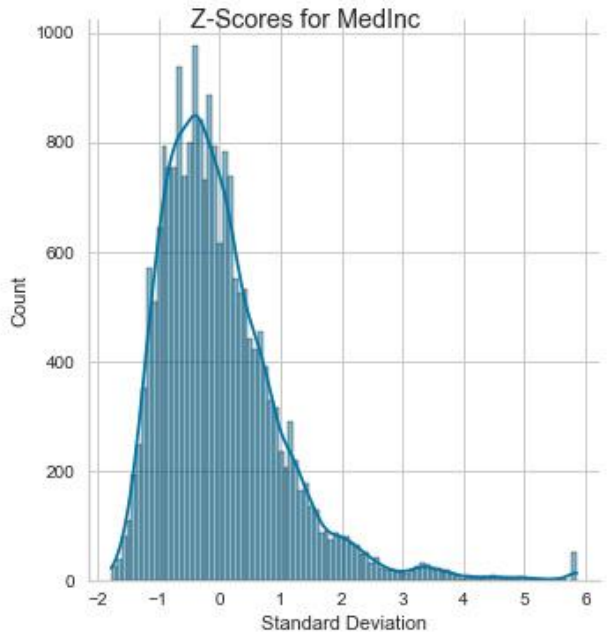

Figure 6.11 – Viewing the distribution plot can tell you just how much the data is outside of the expected range

By now, you should be seeing a progression with the various examples presented in this and the three previous sections. A simple box plot can tell you whether your data is solid, PCA can tell you where the data is outside of the range, the Cook's distance can help you narrow down the record areas and show patterns of outliers, and the z-score can be very specific in telling you precisely which records are potential outliers. The danger in all of this analysis is that you are faced with too much information of any sort. You don't want to become mired in too much detail, so use the level of detail that satisfies the need at the time.

Forecasting potential anomalies example

As mentioned earlier in the chapter, it's not really possible to predict anomalies. For example, you can't write an application that predicts that a certain number of users will enter incorrect data in a particular way today. You also can't create an application that will automatically determine what a hacker will attack today. However, you can predict, within a reasonable amount of error, what will likely happen in certain situations.

When you start a new project, you should be able to predict certain outcomes. If the prediction turns out to be completely false, then there might be something more than bad luck causing problems; you might be encountering outside influences, a lack of training, bad data, bad assumptions, or something else, but it all boils down to dealing with some sort of anomaly. One way to predict the future is to rely on a product such as BigML (bigml.com/) that provides an easy method of creating predictive models for all sorts of uses, such as those shown at bigml.com/gallery/models. Of course, you have to pay for the privilege of using such a product in many cases (there are also free models that may fit your needs).

Time series data provides a particularly interesting problem because the data is serialized. What occurs in the past affects the now and the future. However, because each data point in a time series is individual, it also violates many of the rules of statistics. From a security perspective, this individuality makes it hard to determine whether the increase in user input today, as contrasted to yesterday or last week on the same day, is a matter of an anomaly, a general trend, or simply random variance.

Fortunately, you can forecast many problems without using exotic solutions when you have historical data. The example in this section forecasts future passenger levels on an airline based on historical passenger levels. The `MLSec; 06; Forecast Passengers.ipynb` file contains source code and some detailed instructions for performing predictions on your data.

Obtaining the data

If you use the downloadable source, the code automatically downloads the data for you (as shown here) and places it in the code folder for you. Otherwise, you can download it from https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv. The file is only 2.18 KB, so the download won't take long:

```
import urllib.request
import os.path
filename = "airline-passengers.csv"
if not os.path.exists(filename):
    url = "https://raw.githubusercontent.com/\
        jbrownlee/Datasets/master/airline-passengers.csv"
    urllib.request.urlretrieve(url, filename)
```

Viewing the airline passengers data

One of the things that make the airline passengers dataset so useful for seeing how predictions can work is that it shows a definite cycle in values. The following steps show how to see this cycle for yourself:

1. Import the required dependencies. Note that you may see a warning message when working with this example and using `matplotlib` versions greater than 3.2.2. You can ignore these warnings as they don't affect the actual output:

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.ticker as ticker
```

```
%matplotlib inline
```

2. Read the dataset into memory and obtain specific values from it:

```
apDataset = pd.read_csv('airline-passengers.csv')
passengers = apDataset['Passengers']
months = apDataset['Month']
```

What the example is most interested in is the number of passengers per month. To put this into security terms, you could use the same technique to see the number of suspect API calls per month or the number of invalid logins per month. It also doesn't have to be a monthly interval. You can use any equally spaced interval you want, perhaps hourly, perhaps by the minute.

3. Plot the data on screen:

```
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, 1.0, 1.0])
ax.plot(passengers)
ticks = months[0::12]
ticks = pd.concat([pd.Series([' ']), ticks])
ax.xaxis.set_major_locator(ticker.MultipleLocator(12))
ax.set_xticklabels(ticks, rotation='vertical')
start, end = ax.get_ylim()
ax.yaxis.set_ticks(np.arange(100, end, 50))
ax.grid()
plt.show()
```

The output is a simple line plot using the `ax.plot(passengers)` call. Here, `ticks` are simply the first month of each year, which are displayed on the *x* axis. The *y* axis starts at a value of 100 and ends at 650 using `ax.yaxis.set_ticks(np.arange(100, end, 50))`. To make the values easier to read, the code calls `ax.grid()` to display grid lines. As you can see from *Figure 6.12*, the data does have a specific cycle to it as the number of passengers increases year by year.

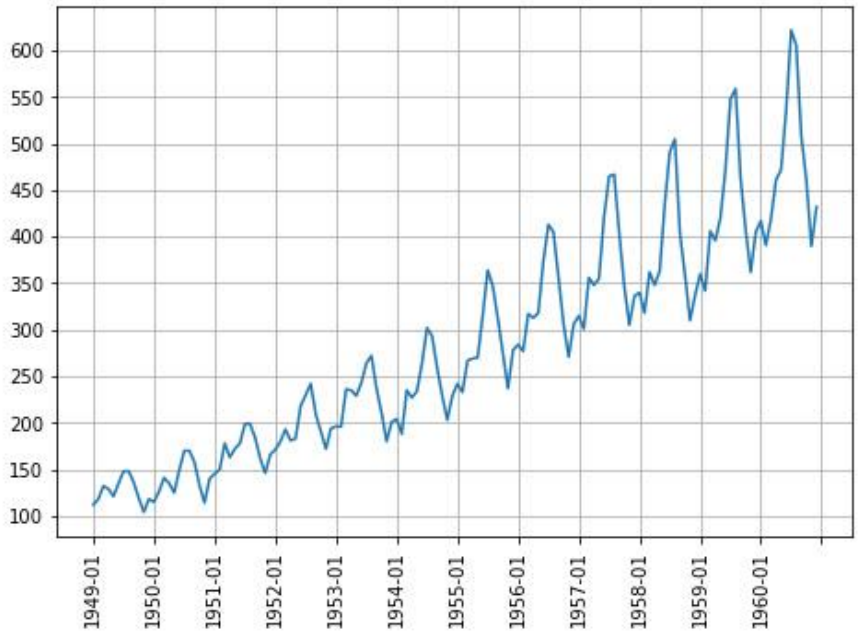

Figure 6.12 – A line chart of the airline passengers dataset shows a definite pattern

Showing autocorrelation and partial autocorrelation

This part of the example comes in two parts: autocorrelation and partial autocorrelation. Both help you make predictions about the future state of data based on both historical and current information.

Autocorrelation is a statistical measure that shows the similarity between observations as a function of time lag. It looks at how an observation correlates with a time **lag** (previous) version of itself. Using autocorrelation helps you see patterns in the data so that you can understand the data better. This example reviews the autocorrelation for 3 years (36 months) of airline passenger data. The time interval is a month in this case, so each lag is 1 month, and there are 36 lags. Since all of the points in the graph are above 0, there is a positive correlation between all of the entries. The pattern shows that the data is seasonal. Here are the steps needed to perform an autocorrelation with the air passenger dataset:

1. Import the required dependencies. Note that NumPy may complain about certain data types in the `statsmodels` package (https://www.statsmodels.org/stable/index.html). There is nothing you can do about this warning and can safely ignore it. The `statsmodels` package developer should fix the problem soon:

```
from statsmodels.graphics import tsaplots
```

2. Create the plot:

```
fig = tsaplots.plot_acf(passengers, lags=36)
fig.set_size_inches(9, 4, 96)
plt.show()
```

The code relies on a basic plot found in the `statsmodels` package, `plot_acf()`. All you supply is the data and the number of lags. *Figure 6.13* shows the autocorrelations and associated confidence cone:

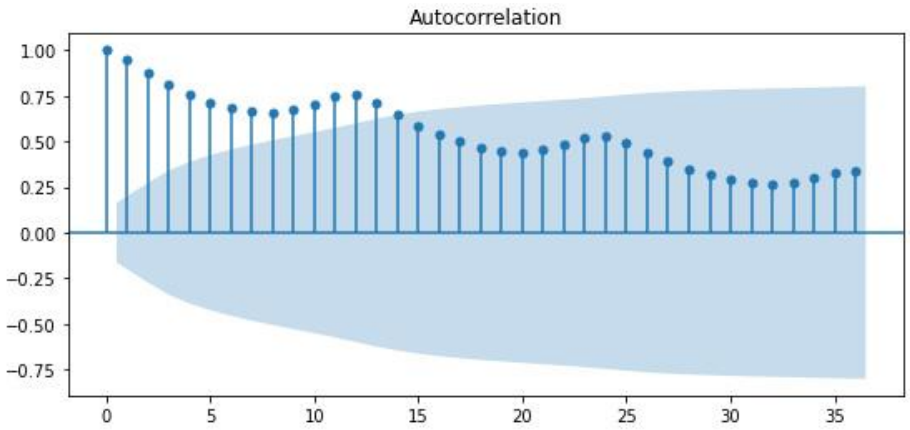

Figure 6.13 – An example of an autocorrelation plot with a confidence cone

The cone drawn as part of the plot is a **confidence cone** and shows the confidence in the correlation values, which is about 80 percent in this case near the end of the plot. More importantly, data with a high degree of autocorrelation isn't a good fit for certain kinds of regression analysis because it fools the developer into thinking there is a good model fit when there really isn't.

The **partial autocorrelation** measurement describes the relationship between an observation and its lag. An autocorrelation between an observation and a prior observation consists of both direct and indirect correlations. Partial autocorrelation tries to remove the indirect correlations. In other words, the plot shows the amount of each data point that isn't predicted by previous data points so that you can see whether there are obvious extremes (data points that don't quite fit properly). Ideally, none of the data points after the first two or three should exceed 0. You use the following code to perform the partial autocorrelation using another standard `statsmodel` package plot, `plot_pacf()`:

```
fig = tsaplots.plot_pacf(passengers, lags=36)
fig.set_size_inches(9, 4, 96)
plt.show()
```

When you run this code, you will see the output shown in *Figure 6.14*:

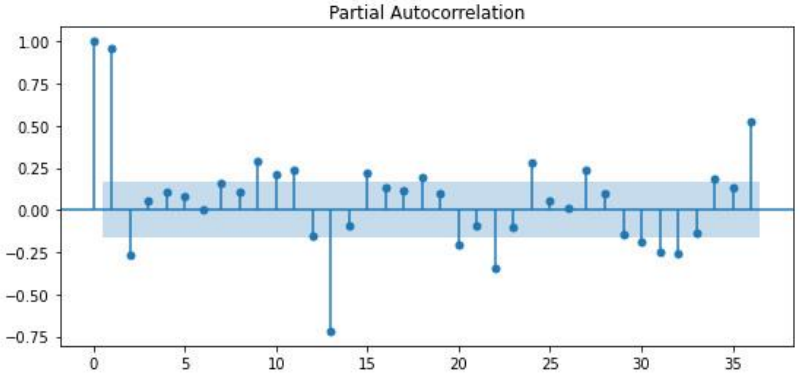

Figure 6.14 – A partial autocorrelation plot shows the relationships between an observation and its lag

The plot shows two lags that are quite high, which provides the term for any autoregression you perform of 2 (any predictions made by an autoregressive model rely on the previous two data points). If there were three high lags, then you'd use an autoregression term of 3. Look at month 13: it has a higher value due to some inconsistency. The gray band shown across the plot is the 95 percent confidence band.

By the time you've got to this point in the chapter, you may be wondering how you'll ever write enough code to cover all of the techniques you find here, much less study the output of all of the applications you seemingly need to write. The fact is that no one can write that much code or study that much output. The next section of the chapter is essential because it helps you put everything into perspective so that you can create an effective detection and mitigation strategy that you'll actually be able to follow.

Making a prediction

Now that you've looked at the autocorrelation for the data, it's time to use the data to create a predictive model. The following steps show how to perform this process using the `statsmodel` package used earlier (there are obviously many other ways to perform the same task using other techniques, such as XGBoost):

1. Import the required dependencies:

```
from statsmodels.tsa.ar_model import AutoReg
from sklearn.metrics import mean_squared_error
from math import sqrt
```

2. Import the data and split it into training and testing data:

```
series = pd.read_csv('airline-passengers.csv',
    header=0, index_col=0,
    parse_dates=True, squeeze=True)
X = series.values
train, test = X[1:len(X)-7], X[len(X)-7:]
print("Training Set: ", train)
print("Testing Set: ", test)
```

The data is read using a different technique this time because you need to format it in a different way for analysis. The resulting data appears as simply a list, as shown in *Figure 6.15*:

```
Training Set:  [118 132 129 121 135 148 148 136 119 104 118 115 126 141 135 125 149 170
 170 158 133 114 140 145 150 178 163 172 178 199 199 184 162 146 166 171
 180 193 181 183 218 230 242 209 191 172 194 196 196 236 235 229 243 264
 272 237 211 180 201 204 188 235 227 234 264 302 293 259 229 203 229 242
 233 267 269 270 315 364 347 312 274 237 278 284 277 317 313 318 374 413
 405 355 306 271 306 315 301 356 348 355 422 465 467 404 347 305 336 340
 318 362 348 363 435 491 505 404 359 310 337 360 342 406 396 420 472 548
 559 463 407 362 405 417 391 419 461 472]
Testing Set:  [535 622 606 508 461 390 432]
```

Figure 6.15 – The data is formatted using a different approach to meet the needs of the model software

3. Create a model based on the training data:

```
model = AutoReg(train, lags=29)
model_fit = model.fit()
print('Coefficients: %s' % model_fit.params)
```

Remember that you've split the 36 lags defined in earlier sections into a training set of 29 lags and a testing set of 7 lags, so you only have 29 lags to train the model. The example prints the **coefficients** for the model, which indicate the direction of the relationship between a predictor variable and the response variable, as shown in *Figure 6.16*:

```
Coefficients: [ 4.40999915  0.54377634  0.31837725 -0.02344532 -0.09328385  0.23085169
 -0.15420365  0.09373805 -0.09291185  0.32015708 -0.46668114  0.05145669
  0.73075956 -0.25128548 -0.33381625  0.242229   -0.15796594 -0.09420813
 -0.08006763  0.05655165 -0.06648343 -0.14659542  0.29515085  0.24214347
  0.09150113 -0.02924769 -0.36689735 -0.01890586  0.12110732  0.0568705 ]
```

Figure 6.16 – The coefficients for the predictive model

4. Make a prediction and compare that prediction to the test data:

```
predictions = model_fit.predict(start=len(train),
    end=len(train)+len(test)-1,
    dynamic=False)
for i in range(len(predictions)):
    print('predicted=%f, expected=%f'
        % (predictions[i], test[i]))
rmse = sqrt(mean_squared_error(test, predictions))
print('Test RMSE: %.3f' % rmse)
```

The example computes the **root-mean-square error** (**RMSE**) of the difference between the predicted value and the actual value and then outputs it, as shown in *Figure 6.17*. In this case, the RMSE is 18.165 percent, which is a little high but would be better with more data:

```
predicted=552.536526, expected=535.000000
predicted=632.633224, expected=622.000000
predicted=629.130658, expected=606.000000
predicted=527.457734, expected=508.000000
predicted=432.252914, expected=461.000000
predicted=399.961090, expected=390.000000
predicted=424.926456, expected=432.000000
Test RMSE: 18.165
```

Figure 6.17 – Show the predictions, the expected values, and the RSME

5. Plot the prediction (in red) versus the actual data (in blue):

```
plt.plot(test)
plt.plot(predictions, color='red')
```

```
plt.show()
```

Figure 6.18 shows that despite a somewhat high RSME, the prediction follows the test data pretty well:

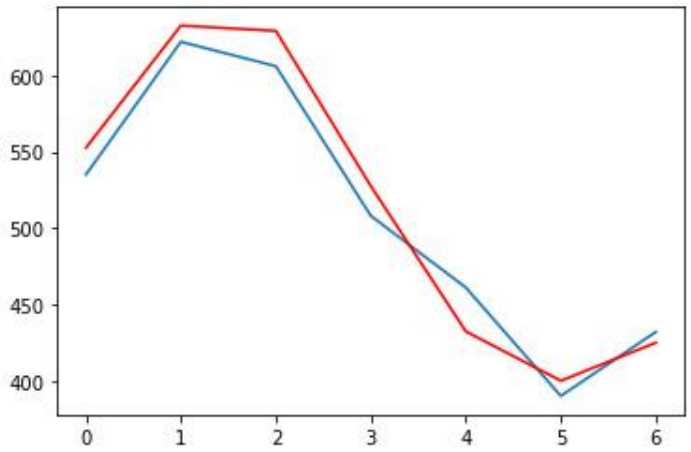

Figure 6.18 – The model's ability to predict the future closely matches the actual data

Now that you've created a model and tested it, you can use it to predict the future within a certain degree of accuracy, as defined by the RSME, and also take into account that the prediction will become less accurate the further you go into the future. The next section will take the next steps and discuss how you can use anomaly detection in an ML environment in an effective way. It's not really enough to simply detect what may appear as anomalies; they need to be identified as true anomalies instead of novelties or possibly noise.

Links

The Ultimate Guide to Man in the Middle (MITM) Attacks and How to Prevent them at https://doubleoctopus.com/blog/the-ultimate-guide-to-man-in-the-middle-mitm-attacks-and-how-to-prevent-them/.

Website Hacking Statistics You Should Know in 2021 at https://patchstack.com/website-hacking-statistics/

5 Anomaly Detection Algorithms in Data Mining (With Comparison) at https://www.intellspot.com/anomaly-detection-algorithms

Some algorithms, such as random forest, are supposedly less affected by outliers, according to some people (see https://heartbeat.fritz.ai/how-to-make-your-machine-learning-models-robust-to-outliers-44d404067d07) and not so robust according to others (see https://stats.stackexchange.com/questions/187200/how-are-random-forests-not-sensitive-to-outliers).

You can learn more about the California housing dataset, which is part of `sklearn.datasets` (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_california_housing.html), at https://www.kaggle.com/datasets/camnugent/california-housing-prices.

Network detection and response (**NDR**) (https://www.gartner.com/en/documents/3986225/market-guide-for-network-detection-and-response)

IronNet (https://www.ironnet.com/what-is-network-detection-and-response)

Further reading

The following list will provide you with some additional reading that you may find useful in further understanding the materials in this chapter.

- A subset of the ImageNet dataset that is easier to use for experimentation: *Tiny ImageNet:* https://paperswithcode.com/dataset/tiny-imagenet
- This paper provides some interesting ideas on how to create an anomaly detection setup based on supervised methods: *Toward Supervised Anomaly Detection:* https://arxiv.org/ftp/arxiv/papers/1401/1401.6424.pdf
- This article provides additional information on the differences between MAD and STDEV: *Relationship Between MAD and Standard Deviation for a Normally Distributed Random Variable:* https://blog.arkieva.com/relationship-between-mad-standard-deviation/
- Understand the math behind PCA a little better: *A Step-by-Step Explanation of Principal Component Analysis (PCA):* https://builtin.com/data-science/step-step-explanation-principal-component-analysis
- Learn more about autocorrelation and time series: *Autocorrelation in Time Series Data:* https://dzone.com/articles/autocorrelation-in-time-series-data
- Understand the difference between autocorrelation and partial autocorrelation better: *What's The Difference Between Autocorrelation & Partial Autocorrelation For Time Series Analysis?:* https://besmarv.medium.com/interpreting-autocorrelation-partial-autocorrelation-plots-for-time-series-analysis-23f87b102c64

Chapter 7

Technical requirements

You won't work with actual malware in this chapter because doing so requires a special virtual machine set up to quarantine the host system from any other connection using a sandbox setup. However, you will see some examples that use code that could interact with malware. This chapter requires that you have access to either Google Colab or Jupyter Notebook to work with the example code. The *Requirements to use this book* section of *Chapter 1*, *Defining Machine Learning Security*, provides additional details on how to set up and configure your programming environment. When testing the code, use a test site, test data, and test APIs to avoid damaging production setups and to improve the reliability of the testing process. Testing over a non-production network is highly recommended, and pretty much essential for this chapter because you don't want to let any of the malware you experiment with on your own get out. In fact, you may actually want to use a system that isn't connected to anything else. Using the downloadable source is always highly recommended. You can find the downloadable source on the Packt GitHub site at https://github.com/PacktPublishing/Machine-Learning-Security-Principles or on my website at http://www.johnmuellerbooks.com/source-code/.

Hands-On Sections

Generating malware detection features

In ML, features are the data that you use to create a model. You analyze features to look for patterns of various sorts. The *Checking data validity* section of *Chapter 6*, *Detecting and Analyzing Anomalies*, shows you one kind of analysis. However, in the case of the *Chapter 6* example and all of the other examples in the book so far, you were viewing data that humans can easily understand. This section talks about a new kind of data hidden in the confines of malware. Consequently, you're moving from the realm of human-recognizable data to that of machine-recognizable data. The interesting thing is that your ML model won't care about what kind of data you use to build a model, the only need is for enough data of the right kind to build a statistically sound model to use to locate malware.

Working with a first step example

To actually work with malware, you need a system that has appropriate safety measures in place, such as a **virtual machine** (https://www.howtogeek.com/196060/beginner-geek-how-to-create-and-use-virtual-machines/) and a **sandbox** configuration (https://blog.hubspot.com/website/sandbox-environment). In fact, it's just a generally good idea to separate the test machine from any other machine, including the Internet. Many discussions of malware throw you into the deep end of the pool without helping you build any skills. This means that it's likely that you'll destroy your machine and every machine around you. The approach taken in this book is to provide you with that first step so that you can move on to books that do

demonstrate examples using real malware, such as *Malware Data Science*, by Joshua Saxe with Hillary Sanders, No Starch Press (the next step up that I would personally recommend).

Getting the required disassembler

The act of dissecting an executable file of any kind is called **disassembly**. You turn the machine code or byte code contained in the executable file back into something a human can at least interpret. Disassembly is never completely accurate in returning code to the same form as the developer of the executable created. What you get instead is something that's close enough that you can determine how the executable works, but nothing else.

The goal of disassembly isn't to completely replicate the original source code anyway. You disassemble an executable to find data you can use to create an ML model. Executable files contain all sorts of statistical information, such as the number of compressed or encrypted data sections. You can find strings of all sorts in any executable that can provide you with clues as to how the executable works. Sometimes the executable contains images that you can pull out and view to see what sort of presentation the executable will provide. In short, the executable contains a lot of hidden information that you must then pull together into a set of features that help you recognize malware.

The example in this section uses the **Portable Executable File** (**PEFile**) disassembler to break down the Windows PEFile format. The following steps show how to obtain this package if necessary and install it on your system.

You can also find this code in the `MLSec; 07; View Portable Executable File.ipynb` file of the downloadable source. Let's begin:

1. Set a search variable, `found`, to `False`:

```
found = False
```

2. Obtain a list of installed packages on the system and search it for the required PEFile package:

```
packages = !pip list
for package in packages:
    if "pefile" in package:
        found = True
        break
```

3. If the package is missing, then install it. Otherwise, print a message saying that the package was found:

```
if not found:
    print("Package is missing, installing...")
    !pip install pefile
else:
    print("PEFile package found.")
```

When you run this code, you will either see a message stating `PEFile package found` or the code will install the PEFile package for you. When the installation process finishes, you see output similar to that shown in *Figure 7.1*.

```
Package is missing, installing...
Collecting pefile
  Downloading pefile-2022.5.30.tar.gz (72 kB)
     ---------------------------------------- 72.9/72.9 kB 3.9 MB/s eta 0:00:00
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Requirement already satisfied: future in c:\users\john\anaconda3\lib\site-packa
ges (from pefile) (0.18.2)
Building wheels for collected packages: pefile
  Building wheel for pefile (setup.py): started
  Building wheel for pefile (setup.py): finished with status 'done'
  Created wheel for pefile: filename=pefile-2022.5.30-py3-none-any.whl size=693
62 sha256=a930d47d59917a48cad471ae8249e0a7fe875e10f6907004a2ac19614941dc71
  Stored in directory: c:\users\john\appdata\local\pip\cache\wheels\c7\ca\2c\b2
bc3360e75954899f344606ea58a307d8f3a7060899b7b7fd
Successfully built pefile
Installing collected packages: pefile
Successfully installed pefile-2022.5.30
```

Figure 7.1 – The installation process shows which version of PEFile you have installed

Now that you have a disassembler to use, you can actually begin working with an executable file. The executable file used for this example isn't harmful in any way, so you don't have to worry about it.

Collecting data about any application

The example in this section shows you how to dissect a Windows PEfile. You can use this technique on any PE file, including malware, but working with files that you know are benign is a good starting point if you don't want to mistakenly infect your system and all of the systems around you. The code for this example appears in the `MLSec; 07; View Portable Executable File.ipynb` file of the downloadable source. Use these steps to dissect your first file. The example uses `Notepad.exe` because you can find it on every Windows system.

Checking for the PE file

Before you can disassemble a PE file, you need to know that it actually exists. The following steps show how to check for a file on a system even if you don't precisely know where the file is:

1. Import the required methods:

```
from os.path import exists, expandvars
```

2. Create an expanded path variable and then display the path on screen so you know where to look later:

```
path = "$windir/notepad.exe"

exppath = expandvars(path)

print(exppath)
```

3. Ensure that the file does actually appear on the system:

```
if exists(exppath):
    print("Notepad.exe is available for use.")
else:
    print("Use another Windows PE file.")
```

This simple check works on any system. Note the use of the `windir` Windows environment variable that specifies the location of the Windows directory on a host system (it's created by default during installation). If you wanted to expand this variable at the command prompt, you'd use `%windir%`, but in Python you use `$windir`. Note that Windows comes with a host of environment variables that you can display at the command prompt by typing `set` and pressing *Enter*.

Loading the Windows PE file

At this point, you have a special package installed on your system that will disassemble Windows PE files and you know whether you can use `Notepad.exe` as a target for your disassembly. It's time to load the Windows PE file for examination using the following steps:

1. Load the required functionality:

```
import pefile
```

2. Load the PE file:

```
exe_file = pefile.PE(exppath)
```

Now you have `Notepad.exe` loaded into memory where you can now examine it in some detail. The `exe_file` variable won't tell you much if you just print it out. You need to become a detective and look for specific features.

Looking at the executable file sections

Despite the fact that executable files are really just long lists of numbers that only your computer processor can really understand, they're actually highly organized (or else the processor would run amok and no one wants that). One of the ways in which to organize executable files is in sections. There are specialized sections in an executable file for every need. The following steps detail how you can look at the sections in an executable file:

1. Display a list of all of the section information:

```
for section in exe_file.sections:
    print(section)
```

2. Choose specific section information to focus on during your detective work:

```
for section in exe_file.sections:
    print(section.Name)
```

The first step is to look at what the executable has to offer in the way of information. When you perform this step, you see something like the output shown in *Figure 7.2* for each section within

the file. That's a lot of really hard-to-understand information, but that's how your executable is organized.

```
[IMAGE_SECTION_HEADER]
0x208      0x0   Name:                          .text
0x210      0x8   Misc:                          0x247FF
0x210      0x8   Misc_PhysicalAddress:          0x247FF
0x210      0x8   Misc_VirtualSize:             0x247FF
0x214      0xC   VirtualAddress:               0x1000
0x218      0x10  SizeOfRawData:                0x24800
0x21C      0x14  PointerToRawData:             0x400
0x220      0x18  PointerToRelocations:         0x0
0x224      0x1C  PointerToLinenumbers:         0x0
0x228      0x20  NumberOfRelocations:          0x0
0x22A      0x22  NumberOfLinenumbers:          0x0
0x22C      0x24  Characteristics:              0x60000020
```

Figure 7.2 – Each section output contains a wealth of information that is useful for statistical analysis

The second step refines the output by just looking at the `Name` value of each section. Note that the entries are case sensitive, so `name` (it does exist) is not the same thing as `Name`. In this case, you see the output shown in *Figure 7.3*.

```
b'.text\x00\x00\x00'
b'.rdata\x00\x00'
b'.data\x00\x00\x00'
b'.pdata\x00\x00'
b'.didat\x00\x00'
b'.rsrc\x00\x00\x00'
b'.reloc\x00\x00'
```

Figure 7.3 – The section names help you see the organization of the file

Well, those names are singularly uninformative. What precisely is a `.text` section? The *Special Sections* section of the *PE Format* documentation tells you what all of these names mean. However, here is the meaning for all of the sections found in `Notepad.exe`:

- `.text`: Executable code
- `.rdata`: Read-only initialized data
- `.data`: Read/write initialized data
- `.pdata`: Exception information
- `.didat`: Delayed import table (not found in the table but does appear in the *PE Format* documentation)
- `.rsrc`: Resource directory
- `.reloc`: Image relocations

In most cases, you won't use all of the sections unless you're involved in detailed research that is well beyond the scope of this book. For example, you really don't care where the image gets relocated at this point, so the `.reloc` section isn't very helpful. On the other hand, looking at the `.rdata` and `.data` sections can be quite illuminating.

Examining the imported libraries

Executable files contain directories of items needed to execute. These directories list specific **Dynamic Link Libraries** (**DLLs**) that the executable accesses and the specific methods within those DLLs that it calls. When you review enough executable files, you begin to develop a feel for which calls are common and which aren't. You also start to be able to tell when a particular DLL could be

downright dangerous to access. For example, you might ask yourself whether the executable really needs to fiddle around in the registry. If not, importing registry methods may be a pointer to malware. The PEFile package makes a number of directories accessible and the following code shows you how to identify them and then access the list of imported libraries:

1. Display the list of accessible directories. Note that you remove the `IMAGE_` part of the entry to use it in code:

```
pefile.directory_entry_types
```

2. Obtain a list of imported DLLs to use for continued analysis:

```
entries = []
for entry in exe_file.DIRECTORY_ENTRY_IMPORT:
    print (entry.dll)
    entries.append(entry)
```

3. Examine the calls within a target DLL that are used by the executable application:

```
print(entries[0].dll)
for function in entries[0].imports:
    print(f"\t{function.name}")
```

The list of accessible directory entries for PEFile appear in *Figure 7.4*. As you can see, the list is rather extensive, but as someone who is just looking for potential malware markers, some of the entries stand out, such as `IMAGE_DIRECTORY_ENTRY_IMPORT`:

```
[('IMAGE_DIRECTORY_ENTRY_EXPORT', 0),
 ('IMAGE_DIRECTORY_ENTRY_IMPORT', 1),
 ('IMAGE_DIRECTORY_ENTRY_RESOURCE', 2),
 ('IMAGE_DIRECTORY_ENTRY_EXCEPTION', 3),
 ('IMAGE_DIRECTORY_ENTRY_SECURITY', 4),
 ('IMAGE_DIRECTORY_ENTRY_BASERELOC', 5),
 ('IMAGE_DIRECTORY_ENTRY_DEBUG', 6),
 ('IMAGE_DIRECTORY_ENTRY_COPYRIGHT', 7),
 ('IMAGE_DIRECTORY_ENTRY_GLOBALPTR', 8),
 ('IMAGE_DIRECTORY_ENTRY_TLS', 9),
 ('IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG', 10),
 ('IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT', 11),
 ('IMAGE_DIRECTORY_ENTRY_IAT', 12),
 ('IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT', 13),
 ('IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR', 14),
 ('IMAGE_DIRECTORY_ENTRY_RESERVED', 15)]
```

Figure 7.4 – PEFile provides access to a number of PE file directories

In fact, the `IMAGE_DIRECTORY_ENTRY_IMPORT` entry is the focus of the next step, which is to list the DLLs that are used by `Notepad.exe` as shown in *Figure 7.5*. Some of the DLL names are pretty interesting. For example, you might wonder what `api-ms-win-shcore-obsolete-l1-1-0.dll` is used for (see the details at https://www.exefiles.com/en/dll/api-ms-win-shcore-obsolete-l1-1-0-dll/). Oddly enough, you may find yourself needing to fix this file, so it's helpful to know it's in use.

```
b'KERNEL32.dll'
b'GDI32.dll'
b'USER32.dll'
b'api-ms-win-crt-string-l1-1-0.dll'
b'api-ms-win-crt-runtime-l1-1-0.dll'
b'api-ms-win-crt-private-l1-1-0.dll'
b'api-ms-win-core-com-l1-1-0.dll'
b'api-ms-win-core-shlwapi-legacy-l1-1-0.dll'
b'api-ms-win-shcore-obsolete-l1-1-0.dll'
b'api-ms-win-shcore-path-l1-1-0.dll'
b'api-ms-win-shcore-scaling-l1-1-1.dll'
b'api-ms-win-core-rtlsupport-l1-1-0.dll'
b'api-ms-win-core-errorhandling-l1-1-0.dll'
b'api-ms-win-core-processthreads-l1-1-0.dll'
b'api-ms-win-core-processthreads-l1-1-1.dll'
b'api-ms-win-core-profile-l1-1-0.dll'
b'api-ms-win-core-sysinfo-l1-1-0.dll'
b'api-ms-win-core-interlocked-l1-1-0.dll'
b'api-ms-win-core-libraryloader-l1-2-0.dll'
b'api-ms-win-core-winrt-string-l1-1-0.dll'
b'api-ms-win-core-synch-l1-1-0.dll'
b'api-ms-win-core-winrt-error-l1-1-0.dll'
b'api-ms-win-core-string-l1-1-0.dll'
b'api-ms-win-core-winrt-l1-1-0.dll'
b'api-ms-win-core-winrt-error-l1-1-1.dll'
b'api-ms-win-eventing-provider-l1-1-0.dll'
b'api-ms-win-core-synch-l1-2-0.dll'
b'COMCTL32.dll'
```

Figure 7.5 – The list of imported DLLs can prove interesting even for benign executables

The example examines a far more common DLL however, `KERNEL32.dll`, in the third step. The output shown in *Figure 7.6* tells you that this DLL sees a lot of use in `Notepad.exe` (the list shown in *Figure 7.6* isn't complete, it's been made shorter to fit in the book).

```
b'KERNEL32.dll'
        b'GetProcAddress'
        b'CreateMutexExW'
        b'AcquireSRWLockShared'
        b'DeleteCriticalSection'
        b'GetCurrentProcessId'
        b'GetProcessHeap'
        b'GetModuleHandleW'
        b'DebugBreak'
        b'IsDebuggerPresent'
        b'GlobalFree'
        b'GetLocaleInfoW'
        b'CreateFileW'
        b'ReadFile'
        b'MulDiv'
        b'GetCurrentProcess'
        b'GetCommandLineW'
        b'HeapSetInformation'
        b'FreeLibrary'
        b'LocalFree'
        b'LocalAlloc'
        b'FindFirstFileW'
        b'FindClose'
```

Figure 7.6 – Looking at specific method calls can tell you want the executable is doing

The list of calls from `KERNEL32.dll` is extensive and well documented (https://www.geoffchappell.com/studies/windows/win32/kernel32/api/index.htm). You can drill down as needed to determine what each function call does. By breaking the various imports down, you start to understand what the executable is doing. By breaking the various imports down, you start to understand what the executable is doing. You've discovered all of the various sections contained within the executable and what it imports from other sources. Just these two bits of information are enough to start getting a feel for how the executable works and what it's doing on your system.

Extracting strings from an executable

It's only possible to garner so much information by looking at what an executable is doing. You can also look at the strings contained in an executable. In this case, you need to use something like the **Strings utility** for Windows or the same command on Linux. The Windows version requires that you download and install the product. Oddly enough, both versions use the same command line switch, `-n`, which is needed for the example in this section. To extract the strings, you need to add a command to your code, something like `!strings -n 10 %windir%\Notepad.exe`. This is the Windows version of the command; the Linux command would be similar. *Figure 7.7* shows the output for `Notepad.exe` if you limit the string size to ten characters or more.

```
UATAUAVAWH
ADVAPI32.dll
Unknown exception
bad allocation
bad array new length
COMDLG32.dll
PROPSYS.dll
SHELL32.dll
WINSPOOL.DRV
urlmon.dll
%hs(%u)\%hs!%p:
(caller: %p)
%hs(%d) tid(%x) %08X %ws
Msg:[%ws]
CallContext:[%hs]
[%hs(%hs)]
kernelbase.dll
RaiseFailFastException
onecore\internal\sdk\inc\wil\opensource\wil\resource.h
```

Figure 7.7 – Reviewing the strings in an executable can reveal some useful facts

This output is typical of any executable, so if you were looking for specific information, then you'd need to start manipulating the list to locate it using Python features. As you can see, there are error messages, references to DLLs, format strings, and all sorts of other useful information. However, a specific example is helpful.

Perhaps you're interested in the format strings contained in a file, so you might use code like this:

```
exe_strings = !strings -n 10 %windir%\Notepad.exe
for exe_string in exe_strings:
    if "%" in exe_string:
        print(exe_string)
```

The output in *Figure 7.8* shows that you still don't get only format strings, but it's a lot shorter and you have a better chance of finding what you need.

```
%hs(%u)\%hs!%p:
(caller: %p)
%hs(%d) tid(%x) %08X %ws
Msg:[%ws]
CallContext:[%hs]
[%hs(%hs)]
Local\SM0:%d:%d:%hs
%s%c*.txt%c%s%c*.*%c
%s\%s.autosave
%08lX-%04X-%04x-%02X%02X-%02X%02X%02X%02X%02X%02X
```

Figure 7.8 – Using Python features it's possible to make the list of string candidates shorter

In this case, you can see that some format strings are standalone, while others are part of messages. The point is that you now have a basis for performing additional work with the executable, without actually loading it. Obviously, loading malware to see it run is something best done in a lab.

Extracting images from an executable

How you look for images in an executable depends on the image type and the platform that you're using. For example, two of the most popular tools for performing this task in Linux are `wrestool` and `icotool`. Windows users have an entirely different list of favorites, some of which come with the **Windows Software Development Kit** (**SDK**). Some tools are quite specialized and only look for a particular image type. To give you some idea of what is involved, the following steps locate icons used with `Notebook.exe`:

1. Download a copy of `IconsExtract` from https://www.nirsoft.net/utils/iconsext.html. The product doesn't require any installation; all you need to do is unzip the file and double-click the executable to start it.

2. Open `IconsExtract` and you see a **Search For Icons** dialog box like the one shown in *Figure 7.9*. This is where you enter the name of the file you want to check. However, you'll quickly find that `Notepad.exe` doesn't contain any icons. Instead, you need to look through the list of DLLs that `Notebook.exe` loads to find the icons.

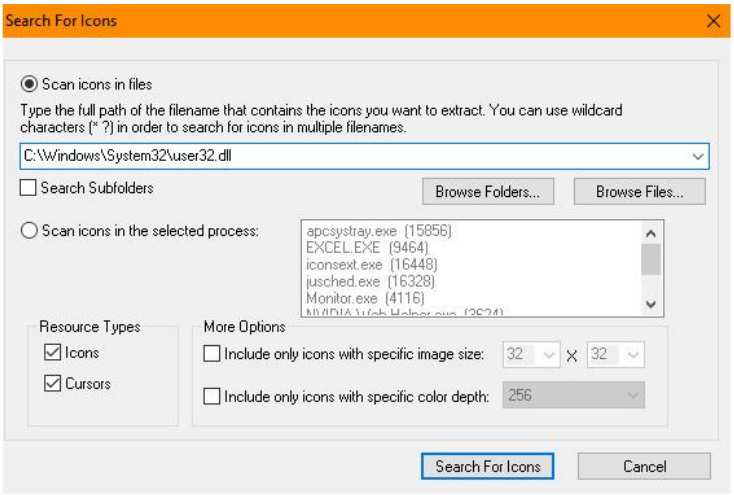

Figure 7.9 – Specify where to look for the images you want to see

3. Locate the `user32.dll` file on your system. This is one of the files you see listed in *Figure 7.5*.

4. Click **Search For Icons**. You see the output shown in *Figure 7.10*.

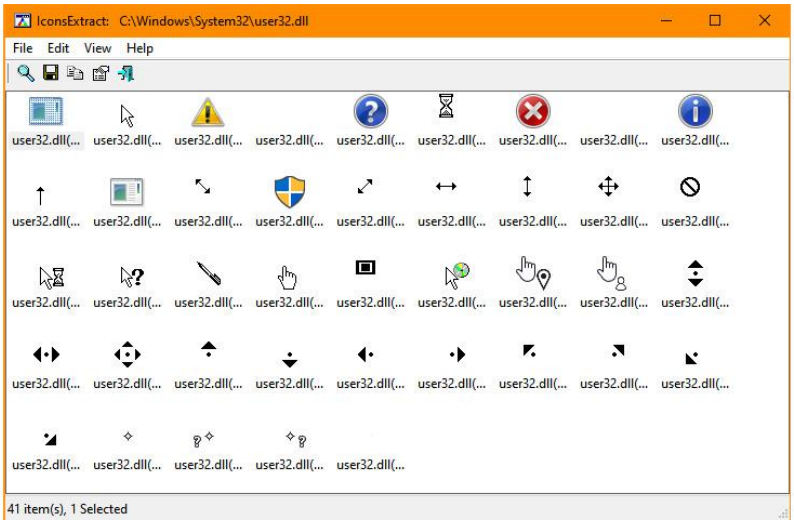

Figure 7.10 – The display shows icons that Notepad.exe may import from user32.dll

When working with malware, you generally want to find images that indicate some type of fake presentation. Perhaps you see a ransom message or other indicator that the executable is malware and not benign. The point is that this is just one of many ways to make the required determination. You shouldn't rely on graphics alone as the only method to perform your search.

Generating a list of application features

In order to create a model to detect malware that may be trying to sneak onto your system or may already appear on your system, you need to define which features to look for. A problem with many models is that the developers haven't really thought through which features are important. For example, looking for applications that use the `ReadFile()` function of `Kernel32.dll` really won't do anything for your detection possibilities. In fact, it will muddy the waters. What you need to do is figure out which features are likely to distinguish the malware target and then build a model around those features. The examples in this chapter should bring up some useful ideas:

- Making unusual or less used method calls, such as interacting with the registry or writing to configuration files
- Executables that contain a great deal of compressed or encrypted data
- Executables or scripts that call on libraries, packages, DLLs, or other external code sources that you don't recognize and can't find documented somewhere online
- Any file, including things such as sound files and images, that contain strings or other unexpected data
- Strings that contain spelling errors
- Any suspected use of **steganography** (the hiding of data or code in a container, such as an image, that looks normal otherwise) in any file, especially images
- Anything out of the ordinary for your particular organization (such as finding patient information out in the open for a hospital)

The list of application features that you choose to use has to reflect the particulars of your organization, rather than the generalized list that you might find on a security website because the hacker is most likely attacking you or your organization as opposed to a generalized attack. When you do want to use the list of generalized features, then creating custom software is likely not the

best option – you should go with off-the-shelf software designed by a reputable security company that has already taken these generalized features into account.

Selecting the most important features

There are a considerable number of ways to determine which features are most important in any dataset. The method you use depends as much on personal preference as the dataset content. It's possible to categorize feature selection techniques in four essential ways:

Filter methods: This approach uses univariate statistics to filter the feature set to locate the most important features based on some criterion. The advantage of this approach is that it's less computationally intensive, which is important when working with a system that may not include a high-end **Graphics Processing Unit** (**GPU**) to speed the computations. These methods also work well with data that has high dimensionality, which is what security data often has because you need to monitor so many different inputs to find a hacker. Because these approaches are so well suited to security needs, they're used for the examples in this section.

Wrapper methods: This approach performs a search of the entire feature set looking for high-quality subsets. This is an ML approach that relies on a classifier to make feature selection determinations. The method relies on greedy algorithms to accomplish the task of determining which set of features best matches the evaluation criterion. The advantage of this approach is that the output provides better predictive accuracy than using filter methods. However, it's also a time-consuming approach that requires a lot of resources and a system that has a GPU if you want to get the results in a timely manner. This is probably the worst approach to use for any sort of security situation that requires real-time analysis. You would get fewer false positives, but not in a timeframe that meets security needs.

Embedded methods: This is a combination of the filter and wrapper methods. It is less computationally intensive, but the iterative approach can make using it for most security needs less than helpful. This is the method you might use to analyze security logs after the fact, not in real time, to ascertain how a hacker gained entrance to your system with a higher degree of precision than would be allowed by filter methods. The algorithms that appear to work best for security needs are **LASSO Regularization** (**L1**) and **Random Forest Importance**.

Hybrid methods: This is an **approach** that uses the result of multiple algorithms to mine data. Generally, it isn't used for security needs because the landscape changes too fast to make it effective. This is the sort of approach that a medical facility might use to mine a dataset for new knowledge needed to treat a disease. These methods rely heavily on instance learning and have the goal of providing consistency in feature selection. However, it could be useful if applied to historical security data in looking for particular trends.

Feature selection is an essential part of working with data of any sort. Otherwise, the problem of creating a model would become resource expensive, time consuming, and not necessarily accurate. When thinking about feature selection, consider these goals:

- Making the dataset small enough to interact with in a reasonable timeframe
- Reducing the computational resources needed to create and use a model
- Improving the understandability of the underlying data

The examples in this section are both filtering methods because you use filtering methods most often for security data. To make the data more understandable, the examples rely on the same California Housing dataset used for the examples in *Chapter 6*. The important thing to remember

with this example is that you're looking for features to use and the example shows you how to do this in an understandable manner. Feature selection is critical whether you're building a malware detection model or looking for data in the California Housing dataset, using the California Housing dataset is simply easier to understand.Of course, you use the dataset in a different way in this chapter. You can the example code in the `MLSec; 07; Feature Selection.ipynb` file of the downloadable source.

Obtaining the required data

Both feature selection examples rely on the same dataset, so you only have to obtain and manipulate the data once. The following steps show how to perform the required setup

1. Import the required packages and methods. The imports include the California Housing dataset, one of the analysis methods, plotting packages, and pandas (https://pandas.pydata.org/) for use in massaging the data:

```
from sklearn.datasets import fetch_california_housing
from sklearn.feature_selection import mutual_info_classif
import matplotlib as plt
import seaborn as sns
import pandas as pd
%matplotlib inline
```

2. Fetch the data and configure it for use:

```
california = fetch_california_housing(as_frame = True)
X = california.data
X = pd.DataFrame(X, columns=california.feature_names)
print(X)
```

The result of fetching the data and massaging it is a DataFrame that will act as input for both of the filtering methods. *Figure 7.11* shows the DataFrame used in this case.

```
       MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0      8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1      8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2      7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3      5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4      3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85
...       ...       ...       ...        ...         ...       ...       ...
20635  1.5603      25.0  5.045455   1.133333       845.0  2.560606     39.48
20636  2.5568      18.0  6.114035   1.315789       356.0  3.122807     39.49
20637  1.7000      17.0  5.205543   1.120092      1007.0  2.325635     39.43
20638  1.8672      18.0  5.329513   1.171920       741.0  2.123209     39.43
20639  2.3886      16.0  5.254717   1.162264      1387.0  2.616981     39.37

       Longitude
0        -122.23
1        -122.22
2        -122.24
3        -122.25
4        -122.25
...          ...
20635    -121.09
20636    -121.21
20637    -121.22
20638    -121.32
20639    -121.24

[20640 rows x 8 columns]
```

Figure 7.11 – It's necessary to massage the data before filtering the features

Now that you've obtained the required data and massaged it for use in feature selection, it's time to look at two approaches for filtering the data. The first is the **Information Gain** technique, which relies on the information gain of each variable in the context of a target variable. The second is the **Correlation Coefficient** method, which is the measurement of the relationship between two variables (as one changes, so does the other).

Using the Information Gain technique

As mentioned earlier, the Information Gain technique uses a target variable as a source of evaluation for each of the variables in a dataset. You chose a feature set based on the target you want to use for analysis. In a security setting, you might choose your target based on known labels, such as whether the input is malicious or benign. Another approach might be to view API calls as common or rare. The target variable must be categorical in nature, however, or you will receive an error message saying that the data is continuous (which really doesn't tell you what you need to know to fix it).

The California Housing dataset doesn't actually include a categorical feature, which makes it a good choice for this example because security data often lacks a categorical feature as well. The following steps show how to create the required categorical feature using the `MedInc` column. This is a good example to play with by changing the category conditions or even selecting a different column, such as `AveRooms` or `HouseAge`:

1. Create a categorical variable to use for the analysis:

```
y = [1 if entry > 7 else 0 for entry in X['MedInc']]
print(y)
```

2. Perform the required analysis and display the result on screen:

```
importances = mutual_info_classif(X, y)
imp_features = pd.Series(importances,
    california.feature_names)
imp_features.plot(kind='barh')
```

The example uses a list comprehension approach to creating the categorical variable, which is very efficient and easy to understand. The result creating the categorical variable is a `y` variable containing a list of 0s and 1s as shown in *Figure 7.12*. A value of 1 indicates that `MedInc` is above $70,000 and a value of 0 indicates that the `MedInc` value is equal to or less than $70,000. The point is to categorize the data according to some criterion.

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1,
0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1,
1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Figure 7.12 – Create a categorical variable to use as the target

Once you have the data in the required form, you can perform the required analysis, which produces the horizontal bar chart shown in *Figure 7.13*. Obviously, there is going to be a high degree of correlation between the target variable and `MedInc`, so you can safely ignore that bar.

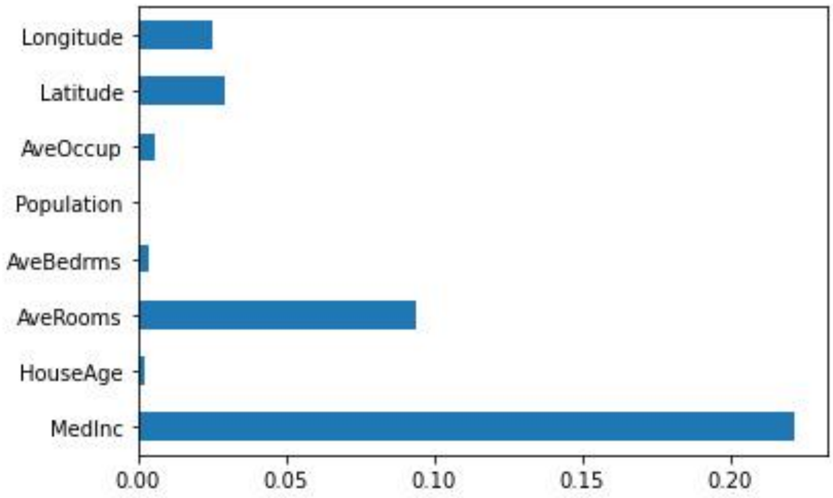

Figure 7.13 – The results are interesting because they show a useful correlation

What is interesting in the result is the high correlation between `MedInc` and `AveRooms`. Far less important is `HouseAge` – the results show you could likely eliminate the `Population` feature and not even notice its absence. However, you have to remember that these results are in the context of the target variable selection. If you change the target variable, the results will also change, so you can't simply assume that deleting `Population` from the original dataset is a good idea. What you need to do is create a new dataset that lacks the `Population` feature for this particular analysis.

Using the Correlation Coefficient technique

Of the filtering techniques, the Correlation Coefficient technique is probably the least code-intensive and doesn't actually require any variable preparation. This approach simply compares the correlation between two (and sometimes more) variables. The example uses the default settings for the `corr()` function, but you can try other approaches.

The following code shows the Correlation Coefficient technique as provided by pandas:

```
cor = X.corr()
plt.figure
sns.heatmap(cor, annot=True)
```

Note that the actual analysis only requires one step. The second and third steps are used to plot the results shown in *Figure 7.14*.

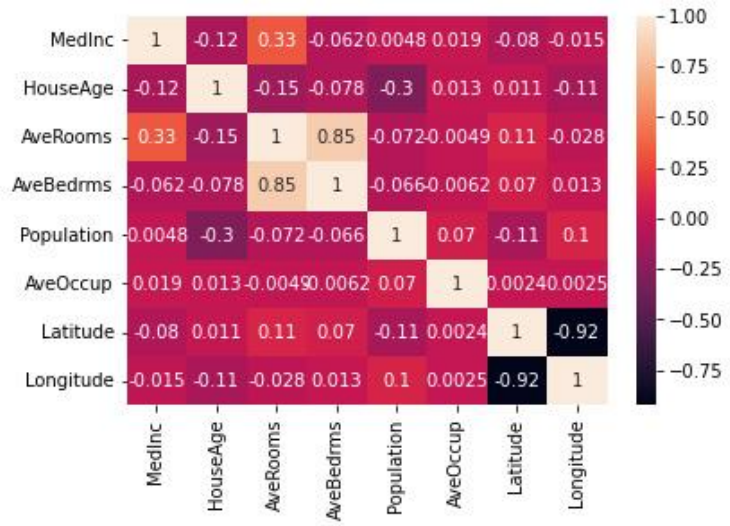

Figure 7.14 – The Correlation Coefficient technique shows some interesting relationships between variables

This is a **heatmap plot**, which is a Cartesian plot with data shown as colored rectangular tiles where the color designates a level of correlation in this case. You see the correlation levels on the right side of the plot as a bar where lighter colors represent higher levels of correlation and darker colors indicate lower levels of correlation. Since `MedInc` (horizontally) has a 100-percent-degree of correlation with `MedInc` (vertically), this square receives the lightest color and a value of `1`. Not shown in the bar on the right is that black indicates no correlation at all. So, for example, there is no correlation between `Latitude` and `Longitude` in the California Housing dataset.

There are some interesting correlations in this case. Notice that `AveRooms` has a high degree of correlation with `AveBedrms`. This plot also corroborates the result in *Figure 7.13* in that there is a moderate level of correlation between `MedInc` and `AveRooms`.

Not corroborated in this case is the correlation between `MedInc`, `Latitude`, and `Longitude` shown in *Figure 7.13*. This is due the different method used to filter the feature selection. The output in *Figure 7.14* isn't considering the amount of `MedInc` as a factor in feature selection, so now you understand that the Information Gain approach helps you target a specific criterion for filtering, while the Correlation Coefficient method is more generalized.

Considering speed of detection

When performing security analysis using ML techniques, and with malware in particular, detection is extremely time critical. This is the reason that you need to choose datasets used to create models with care, target the kind of threats you want to address with your business in mind, and layer your defenses to reduce the load on any one defense so it doesn't slow down. This chapter has discussed a number of malware threat types, feature selection, and detection techniques that will help you

create a useful security model for your organization. However, here are some things to consider when creating your model and tuning it for the performance needed for usable security:

- Err in favor of false positives, rather than false negatives, because you can always take benign data out of the virus vault after verification, but letting malware through will always cause problems.
- Reduce your feature set to ensure that you're not creating a model that will get mired in useless detail. Smaller feature sets mean faster and more targeted training.
- Use faster algorithms that provide good enough analysis because you generally can't afford the time required by models that provide highly precise output.
- Decide at the outset that humans will be involved in the mitigation process because automation will always produce false positives that someone needs to verify. It also isn't necessarily possible to use lessons learned to create a more precise model because the new model will run slower.
- Ensure that any virus vault you create to hold suspected malware is actually secure and that only authorized personnel (your security professionals and no one else) can access it. It's usually better to lose some small amount of data (having it sent again from the source) than to allow any malware to enter your system.

The essential factor in all of these bullets is speed. Getting precise details about a threat after the threat has already passed is useless. Security issues won't wait on your model to make a decision. This need for speed doesn't mean creating a sloppy model. Rather it means creating a model that will always err on the side of being too safe and allowing a human to make the ultimate decision about the malicious or benign nature of the malware later.

Building a malware detection toolbox

The example in the *Collecting data about any application* section tells you how to dissect just one kind of executable file, a Windows PE file. If all you ever work on is Windows systems that are possibly connected to each other, but nowhere else, and none of your employees use their smartphones and other devices to perform their work, then you may have everything you need to start studying malware. However, this is unlikely to be the case. To really get involved in malware detection, you need to know how to disassemble and review every type of executable for every device that interacts with your system in any way. It's really quite a job, which is why this chapter has focused so hard on using off-the-self solutions when possible, rather than trying to build everything from scratch on your own.

A malware detection toolbox needs to consist of the assortment of items needed to analyze the malware you want to target. This includes hardware that will keep any malware contained, which usually means using sandboxing techniques on virtual machines. Note that these hardware techniques don't work well in real time. For example, sandboxing is time intensive, so you couldn't attach a sandbox to your network and expect that the network will continue to provide good throughput. In addition, some types of malware evade sandbox setups by remaining dormant (appearing benign) until they leave the sandbox and enter the network.

Once you have decided on which malware features to detect, you need to work with a data source to understand how to classify the malware. The classification process leads to detection, avoidance, and mitigation. Knowing how your adversary works is essential to eventual victory over them. The next section discusses the malware classification process and provides advice on how to use this classification to perform detection and possible avoidance.

Links

The Trojan Horse: https://www.greekmythology.com/Myths/The_Myths/The_Trojan_Horse/the_trojan_horse.html

Gaming, Banking Trojans Dominate Mobile Malware Scene: https://threatpost.com/gaming-banking-trojans-mobile-malware/178571/

Pickle library: (https://docs.python.org/3/library/pickle.html)

The hidden dangers of loading open-source AI models at https://www.youtube.com/watch?v=2ethDz9KnLk

`VirTool:Win32/Oitorn.A` attacks certain Windows systems (https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=VirTool:Win32/Oitorn.A)

Trojan Source' Hides Invisible Bugs in Source Code (https://threatpost.com/trojan-source-invisible-bugs-source-code/175891/)

The Best Antivirus of 2022: A data-driven comparison (https://www.comparitech.com/antivirus/)

Wikipedia at https://en.wikipedia.org/wiki/Comparison_of_antivirus_software

Virtual machine (https://www.howtogeek.com/196060/beginner-geek-how-to-create-and-use-virtual-machines/) and a **Sandbox** configuration (https://blog.hubspot.com/website/sandbox-environment).

The example in this section uses the **Portable Executable File** (**PEFile**) disassembler (https://pypi.org/project/pefile/ and https://github.com/erocarrera/pefile) to break down the Windows PEFile format

The *Special Sections* section of the *PE Format* documentation: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

Strings utility for Windows: (https://docs.microsoft.com/en-us/sysinternals/downloads/strings) or the same command on Linux (https://www.howtogeek.com/427805/how-to-use-the-strings-command-on-linux/)

Strings utility for Windows (https://docs.microsoft.com/en-us/sysinternals/downloads/strings) or the same command on Linux (https://www.howtogeek.com/427805/how-to-use-the-strings-command-on-linux/)

`wrestool` (https://linux.die.net/man/1/wrestool)

`icotool` (https://linux.die.net/man/1/icotool)

How To Extract Or Save The Icon From An EXE File: https://www.alphr.com/extract-save-icon-exe-file/

Sandboxing techniques (https://www.barracuda.com/glossary/sandboxing)

What is virtualized security? (https://www.vmware.com/topics/glossary/content/virtualized-security.html)

Sophos-ReversingLabs 20 million sample dataset: https://github.com/sophos/SOREL-20M. This site also provides detailed instructions for working with the dataset with as much safety as working with malware can allow. The *Frequently Asked Questions* section of the site tells you how the malware is disarmed.

7 Cloud Computing Security Vulnerabilities and What to Do About Them at https://towardsdatascience.com/7-cloud-computing-security-vulnerabilities-and-what-to-do-about-them-e061bbe0faee

VirusTotal (https://www.virustotal.com/gui/intelligence-overview).

Microsoft Malware Classification Challenge (BIG 2015) at https://www.kaggle.com/c/malware-classification

Viewing the submitted code: https://www.kaggle.com/competitions/malware-classification/code

Free Malware Sample Sources for Researchers (https://zeltser.com/malware-sample-sources/).

Contagio Malware Dump (http://contagiodump.blogspot.com/2010/11/links-and-resources-for-malware-samples.html

Other approaches for `Corr()` functions: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html

To actually work with malware, you need a system that has appropriate safety measures in place, such as a virtual machine (https://www.howtogeek.com/196060/beginner-geek-how-to-create-and-use-virtual-machines/) and a sandbox configuration (https://blog.hubspot.com/website/sandbox-environment).

Further reading

The following bullets provide you with some additional reading that you may find useful to advance your understanding of the materials in this chapter:

- See the method used to classify various kinds of malware: *Rules for classifying* (https://encyclopedia.kaspersky.com/knowledge/rules-for-classifying/)
- Locate potential malware dataset sources: *Top 7 malware sample databases and datasets for research and training* (https://resources.infosecinstitute.com/topic/top-7-malware-sample-databases-and-datasets-for-research-and-training/)
- Discover techniques for discovering behavior tracking spyware: *Behavior-based Spyware Detection* (https://sites.cs.ucsb.edu/~chris/research/doc/usenix06_spyware.pdf)
- Learn how combined attacks are becoming significantly more common and some of them also thwart MFA: *This Android banking malware now also infects your smartphone with ransomware* (https://www.zdnet.com/article/this-android-banking-trojan-malware-can-now-also-infect-your-smartphone-with-ransomware/)

- See a list of tools that can remove rootkits from the MBR of a hard drive, among other places: *Best Rootkit Scanners for 2022* (https://www.esecurityplanet.com/networks/rootkit-scanners/)
- Get an overview of the Windows **NT File System** (**NTFS**) version of alternate data streams: *NTFS File Streams – What Are They?* (https://stealthbits.com/blog/ntfs-file-streams/)
- Discover how images can install dropper trojans on user systems: *Malware in Images: When You Can't See "the Whole Picture"* (https://blog.reversinglabs.com/blog/malware-in-images)
- Understand how techniques like steganography pose a real risk to your organization: *What is steganography?* (https://www.techtarget.com/searchsecurity/definition/steganography)
- Create any custom malware detection aids with less effort: *11 Best Malware Analysis Tools and Their Features* (https://www.varonis.com/blog/malware-analysis-tools)
- Obtain a graphical presentation of the PE file format: *Portable Executable Format Layout* (https://drive.google.com/file/d/0B3_wGJkuWLytbnIxY1J5WUs4MEk/view?resourcekey=0-n5zZ2UW39xVTH8ZSu6C2aQ)
- See more feature selection techniques: Feature Selection Techniques in Machine Learning (https://www.analyticsvidhya.com/blog/2020/10/feature-selection-techniques-in-machine-learning/)
- Gain a better understanding of hybrid feature selection methods: *Hybrid Methods for Feature Selection* (https://digitalcommons.wku.edu/cgi/viewcontent.cgi?article=2247&context=theses)
- Learn more about virtualized machines: *What is a virtual machine (VM)?* (https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-virtual-machine/)

Chapter 8

Technical requirements

This chapter requires that you have access to either Google Colab or Jupyter Notebook to work with the example code. The *Requirements to use this book* section of *Chapter 1*, *Defining Machine Learning Security*, provides additional details on how to set up and configure your programming environment. When testing the code, use a test site, test data, and test APIs to avoid damaging production setups and to improve the reliability of the testing process. Testing over a non-production network is highly recommended, but not necessary. Using the downloadable source is always highly recommended. You can find the downloadable source on the Packt GitHub site at https://github.com/PacktPublishing/Machine-Learning-Security-Principles or my website at http://www.johnmuellerbooks.com/source-code/.

Hands-On Section: Building a fraud detection example

This section will show you how to build a simple fraud detection example using real sanitized credit card data available on Kaggle. The transactions occurred in September 2013 and there are 492 frauds out of 284,807 transactions, which is unbalanced because the number of frauds is a little low for training a model. The data has been transformed by **Principal Component Analysis** (**PCA**) using the techniques demonstrated in the *Relying on Principle Component Analysis* section of *Chapter 6*, *Detecting and Analyzing Anomalies*. Only the `Amount` column has the original value in it. The `Class` column has been added to label the data. You can also find the source code for this example in the `MLSec; 08; Perform Fraud Detection.ipynb` file of the downloadable source.

Getting the data

The dataset used in this example appears at https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud?resource=download. The data is in a 69 MB `.zip` file. Download the file manually and unzip it into the source code directory. Note that you must obtain a Kaggle subscription if you don't already have one to download this dataset. Getting a subscription is easy and free; check out https://www.kaggle.com/subscribe.

Setting the example up

This example begins with importing the data, which requires a little work in this case. The following steps show you how to do so:

1. Import the required packages:

```
import pandas as pd

from sklearn.preprocessing import StandardScaler
```

2. Import the `creditcard.csv` data file. Note that this process may require a minute or two, depending on the speed of your system. This is because the unzipped file is 150.83 MB:

```
cardData=pd.read_csv("creditcard.csv")
```

3. Check the dataset's statistics:

```
total_transactions = len(cardData)
normal = len(cardData[cardData.Class == 0])
fraudulent = len(cardData[cardData.Class == 1])
fraud_percentage = fraudulent/normal
print(f'Total Number Transactions:
    {total_transactions}')
print(f'Normal Transactions: {normal}')
print(f'Fraudulent Transactions: {fraudulent}')
print(f'Fraudulent Transactions Percent: ' \
        f'{fraud_percentage:.2%}')
```

The number of actual fraudulent transactions is smaller, accounting for only 0.17% of the transactions, as shown in *Figure 8.1*:

```
Total Number Transactions: 275663
Normal Transactions: 275190
Fraudulent Transactions: 473
Fraudulent Transactions Percent: 0.17%
```

Figure 8.1 – Output showing the transaction statistics

4. Look for any potential null values. Having null values in the data will skew the model and cause other problems:

```
cardData.info()
```

Figure 8.2 shows that there are no null values in the dataset. If there had been null values, then you would need to clean the data by replacing the null values with a specific value, such as the mean of the other entries in the column. The same thing applies to **missingness**, which is missing data in a dataset and sometimes indicates fraud. You need to replace the missing value with some useful alternative:

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 275663 entries, 0 to 284806
Data columns (total 30 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   V1      275663 non-null  float64
 1   V2      275663 non-null  float64
 2   V3      275663 non-null  float64
 3   V4      275663 non-null  float64
 4   V5      275663 non-null  float64
 5   V6      275663 non-null  float64
 6   V7      275663 non-null  float64
 7   V8      275663 non-null  float64
 8   V9      275663 non-null  float64
 9   V10     275663 non-null  float64
 10  V11     275663 non-null  float64
 11  V12     275663 non-null  float64
 12  V13     275663 non-null  float64
 13  V14     275663 non-null  float64
 14  V15     275663 non-null  float64
 15  V16     275663 non-null  float64
 16  V17     275663 non-null  float64
 17  V18     275663 non-null  float64
 18  V19     275663 non-null  float64
 19  V20     275663 non-null  float64
 20  V21     275663 non-null  float64
 21  V22     275663 non-null  float64
 22  V23     275663 non-null  float64
 23  V24     275663 non-null  float64
 24  V25     275663 non-null  float64
 25  V26     275663 non-null  float64
 26  V27     275663 non-null  float64
 27  V28     275663 non-null  float64
 28  Amount  275663 non-null  float64
 29  Class   275663 non-null  int64
dtypes: float64(29), int64(1)
memory usage: 65.2 MB
```

Figure 8.2 – Dataset output showing a lack of missing or null values

5. Determine the range of the `Amount` column. The `Amount` column is the only unchanged column from the original dataset:

```
print(f'Minimum Value: {min(cardData.Amount)}')
print(f'Mean Value: ' \
        f'{sum(cardData.Amount)/total_transactions}')
print(f'Maximum Value: {max(cardData.Amount)}')
print(cardData['Amount'])
```

If the `Amount` column has too great a range, as shown in *Figure 8.3*, then the model will become skewed. In this case, the standard practice is to scale the data to obtain better results:

```
Minimum Value: 0.0
Mean Value: 88.34961925087359
Maximum Value: 25691.16
0         149.62
1           2.69
2         378.66
3         123.50
4          69.99
           ...
284802      0.77
284803     24.79
284804     67.88
284805     10.00
284806    217.00
Name: Amount, Length: 284807, dtype: float64
```

Figure 8.3 – Output showing variance in the Amount column

6. Remove the `Time` column. For this analysis, time isn't a factor because the model isn't concerned with when credit card fraud occurs; the concern is detecting it when it does occur. The output of this code, `(284807, 30)`, shows that the `Time` column is gone:

```
cardData.drop(['Time'], axis=1, inplace=True)
print(cardData.shape)
```

7. Scale the dataset to avoid negative values in any of the columns. Some classifiers work well with negative values, while others don't. For example, if you want to use a `DecisionTreeClassifier`, then negative values aren't a problem. On the other hand, if you want to use a `MultinomialNB`, as was done in the *Developing a simple spam filter example* section of *Chapter 4*, *Considering the Threat Environment*, then you need positive values. Selecting a scaler is also important. This example uses a `MinMaxScaler` to allow the minimum and maximum range of values to be set:

```
scaler = MinMaxScaler(feature_range=(0,1))
col = cardData.columns
cardData = pd.DataFrame(
    scaler.fit_transform(cardData),
    columns = col)
print(cardData)
```

Figure 8.4 shows the partial results of the scaling operation:

```
              V1        V2        V3        V4        V5        V6        V7  \
0       0.935192  0.766490  0.881365  0.313023  0.763439  0.267669  0.266815
1       0.978542  0.770067  0.840298  0.271796  0.766120  0.262192  0.264875
2       0.935217  0.753118  0.868141  0.268766  0.762329  0.281122  0.270177
3       0.941878  0.765304  0.868484  0.213661  0.765647  0.275559  0.266803
4       0.938617  0.776520  0.864251  0.269796  0.762975  0.263984  0.268968
...          ...       ...       ...       ...       ...       ...       ...
284802  0.756448  0.873531  0.666991  0.160317  0.729603  0.236810  0.235393
284803  0.945845  0.766677  0.872678  0.219189  0.771561  0.273661  0.265504
284804  0.990905  0.764080  0.781102  0.227202  0.783425  0.293496  0.263547
284805  0.954209  0.772856  0.849587  0.282508  0.763172  0.269291  0.261175
284806  0.949232  0.765256  0.849601  0.229488  0.765632  0.256488  0.274963
```

Figure 8.4 – Presentation of data values after scaling

8. Remove any duplicates. Removing the duplicates ensures that any entries in the dataset that replicate another entry don't remain in place and cause the model to overfit and it won't generalize well. This means that it won't catch as many fraudulent transactions from data it hasn't seen. The output of `(275663, 30)` shows that there were 9,144 duplicated records:

```
cardData.drop_duplicates(inplace=True)
print(cardData.shape)
```

9. Determine the final `Amount` column's characteristics:

```
print(f'Minimum Value: {min(cardData.Amount)}')
print(f'Mean Value: ' \
      f'{sum(cardData.Amount)/total_transactions}')
print(f'Maximum Value: {max(cardData.Amount)}')
print(cardData['Amount'])
```

Now that the dataset has been massaged, it's time to see the result, as shown in *Figure 8.5*:

```
Minimum Value: 0.0
Mean Value: 0.00341246851336739
Maximum Value: 1.0
0         0.005824
1         0.000105
2         0.014739
3         0.004807
4         0.002724
             ...
284802    0.000030
284803    0.000965
284804    0.002642
284805    0.000389
284806    0.008446
Name: Amount, Length: 275663, dtype: float64
```

Figure 8.5 – Output of the Amount column after massaging

A lot of ML comes down to ensuring that the data you use is prepared properly to create a good model. Now that the data has been prepared, you can split it into training and testing sets. This process ensures that you have enough data to train the model, and then test it using data the model hasn't seen before so that you can ascertain the goodness of the model (something you will see later in this process).

Splitting the data into train and test sets

Splitting the data into training and testing sets makes it possible to train the model on one set of data and test it using another set of data that the model hasn't seen. This approach ensures that you can validate the model concerning its goodness in locating credit card fraud (or anything else for that matter). The following steps show how to split the data in this case:

1. Import the required packages:

```
from sklearn.model_selection \
    import train_test_split
```

2. Split the dataset into data (`X`, which is a matrix, so it appears in uppercase) and labels (`y`, which is a vector, so it appears in lowercase):

```
X = cardData.drop('Class', axis=1).values

y = cardData['Class'].values

print(X)

print(y)
```

When you print the result, you will see that `X` is indeed a matrix and `y` is indeed a vector, as shown in *Figure 8.6*:

```
[[9.35192337e-01 7.66490419e-01 8.81364903e-01 ... 4.18976135e-01
  3.12696634e-01 5.82379309e-03]
 [9.78541955e-01 7.70066651e-01 8.40298490e-01 ... 4.16345145e-01
  3.13422663e-01 1.04705276e-04]
 [9.35217023e-01 7.53117667e-01 8.68140819e-01 ... 4.15489266e-01
  3.11911316e-01 1.47389219e-02]
 ...
 [9.90904812e-01 7.64079694e-01 7.81101998e-01 ... 4.16593177e-01
  3.12584864e-01 2.64215395e-03]
 [9.54208999e-01 7.72855742e-01 8.49587129e-01 ... 4.18519535e-01
  3.15245157e-01 3.89238944e-04]
 [9.49231759e-01 7.65256401e-01 8.49601462e-01 ... 4.16466371e-01
  3.13400843e-01 8.44648509e-03]]
[0. 0. 0. ... 0. 0. 0.]
```

Figure 8.6 – Contents of the training and testing datasets

3. Divide `X` and `y` into training and testing sets. There are many schools of thought as to what makes for a good split. What it comes down to is creating a split that provides the least amount of variance during training, but then tests the model fully. In this case, there are plenty of samples, so using the 80:20 split that many people use will work fine:

```
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.2, random_state=1)
print(f"X training data size: {X_train.shape}")
print(f"X testing data size: {X_test.shape}")
print(f"y training data size: {y_train.shape}")
print(f"y testing data size: {y_test.shape}")
```

Figure 8.7 shows that the data is split according to the 80:20 ratio and that the training and testing variables have the same number of entries:

```
X training data size: (220530, 29)
X testing data size: (55133, 29)
y training data size: (220530,)
y testing data size: (55133,)
```

Figure 8.7 – The output shows that the data is split according to an 80:20 ratio

Now that the data is in the correct form, you can finally build a model. Of course, that means selecting a model and configuring it. For this example, you will use `DecisionTreeClassifier`. However, there are a wealth of other models that could give you an edge when working with various kinds of data.

Considering the importance of testing model goodness

The whole issue of data splitting, selecting the correct model, configuring the model correctly, and so on, comes down to getting a model with very high accuracy. This is especially important when doing things such as looking for malware or detecting fraud. Unfortunately, no method or rule of thumb provides a high level of accuracy in every case. The only real way to tweak your model is to change one item at a time, rebuild the model, and then test it.

Building the model

As you've seen in other chapters, building the model involves fitting it to the data. The following steps show how to build the model using a minimal number of configuration changes:

1. Import the required packages:

```
from sklearn.tree import DecisionTreeClassifier
```

2. Fit the model to the data. Notice that the one configuration change here is to limit the `max_depth` setting to `5`. Otherwise, the classifier will keep churning away until all of the leaves are pure or they meet the `min_samples_split` configuration setting requirements:

```
dtc = DecisionTreeClassifier(max_depth = 5)
dtc.fit(X_train, y_train)
```

You'll know that the process is complete when you see the output shown in *Figure 8.8*:

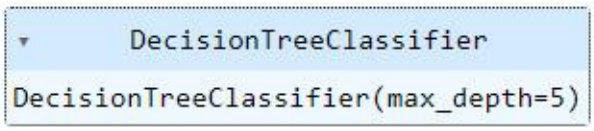

Figure 8.8 – Description of the DecisionTreeClassifier model

You now have a model to use to detect credit card fraud. Of course, you have no idea of how good that model is at its job. Perhaps it's not very generalized and overfitted to the data. Then again, it might be underfitted. The next section shows how to verify the goodness of the model in this case.

Performing the analysis

Having a model to use means that you can start detecting fraud. Of course, you don't know how well you can detect fraud until you test it using the following steps:

1. Import the required packages:

```
from sklearn.metrics import accuracy_score
```

```
from sklearn.metrics import \
    precision_recall_fscore_support
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
import matplotlib.pyplot as plt
```

2. Perform a prediction using the `X_test` data that wasn't used for training purposes. Use this prediction to verify the accuracy by comparing the actual labels in `y_test` (showing whether a record is fraudulent or not) to the predictions found in `yHat` (ŷ):

```
dtc_yHat = dtc.predict(X_test)
print(f"Accuracy score: ' \
        f'{accuracy_score(y_test, dtc_yHat)}")
```

Figure 8.9 shows that the accuracy is very high:

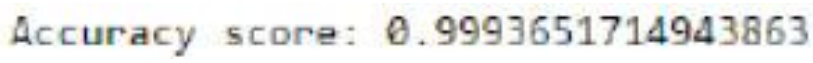

Figure 8.9 – Output of the DecisionTreeClassifier model accuracy

3. Determine the **precision** (ratio of correctly predicted positive observations to the total of the predictive positive observations), **recall** (ratio of correctly predicted positive observations to all observations in the class), **F-beta** (the weighted harmonic mean of precision and recall), and **support** scores (the number of occurrences of each label in `y_test`). These metrics are important in fraud work because they tell you more about the model given that the dataset has a few positives and a lot of negatives, making the dataset unbalanced. Consequently, accuracy is important, but these measures are also quite helpful for fraud work:

```
precision, recall, fbeta_score, support = \
    precision_recall_fscore_support(y_test, dtc_yHat)
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F-beta score: {fbeta_score}")
print(f"Support: {support}")
```

4. *Figure 8.10* shows the output from this step. Ideally, you want a precision as close as possible to `1.0`. The output shows that the precision is `0.9994551` for predicting when a transaction isn't fraudulent and a value of `0.93506494` when a transaction is fraudulent. For recall, there are two classes: not fraud and fraud. So, the ratio of correctly predicting fraud against all of the actual fraud cases is only `0.70588235`. The F-beta score is about weighting. The example uses a value of `1.0`, which means that recall and precision have equal weight in determining the goodness of a model. If you want to give more weight to precision, then you use a number less than `1.0`, such as `0.5`. Likewise, if you want to give

more weight to recall, then you use a number above `1.0`, such as `2.0`. Finally, the support output simply tells you how many non-fraud and fraud entries there are in the dataset:

```
Precision: [0.9994551  0.93506494]
Recall: [0.99990914 0.70588235]
F-beta score: [0.99968207 0.80446927]
Support: [55031   102]
```

Figure 8.10 – Output of the precision, recall, F-beta, and support statistics

5. Print a confusion matrix to show how the prediction translates into the number of entries in the dataset:

```
print(confusion_matrix(y_test,
        dtc_yHat,
        labels=[0, 1]))
```

Figure 8.11 shows the output. The number of true positives appears in the upper-left corner, which has a value of `55025`. Only 29 of the records generated a false positive. There were 73 true negatives and six false negatives in the dataset. So, the chances of finding credit card fraud are excellent, but not perfect:

```
[[55025     6]
 [   29    73]]
```

Figure 8.11 – The confusion matrix output for the DecisionTreeClassifier model prediction

6. Create a graphic version of the confusion matrix to make the output easier to understand. Using graphic output does make a difference, especially when discussing the model with someone who isn't a data scientist:

```
matrix = plot_confusion_matrix(dtc,
    X=X_test,
    y_true=y_test,
    cmap=plt.cm.Blues)
plt.title('Confusion Matrix for Fraud Detection')
plt.show(matrix)
plt.show()
```

Figure 8.12 shows the output in this case:

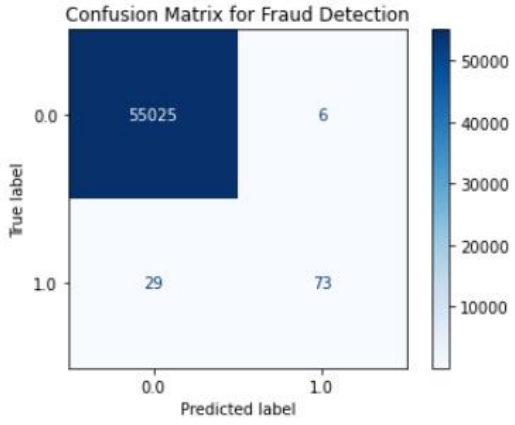

Figure 8.12 – A view of the graphic version of the confusion matrix

This section has shown you one complete model building and testing cycle. However, you don't know that this is the best model to use. Testing other models is important, as described in the next section.

Checking another model

The decision tree classifier does an adequate job of separating fraudulent credit purchases from those that aren't, but it could do better. A random forest classifier is a group of decision tree classifiers. In other words, you put multiple algorithms to work on the same problem. When the classification process is complete, the trees vote and the classification with the most votes wins.

In the previous example, you used the `max_depth` argument to determine how far the tree should go to reach a classification. Now that you have a whole forest, rather than an individual tree, at your disposal, you also need to define the `n_estimators` argument to define how many trees to use. There are a lot of other arguments that you can use to tune your model in this case, as described in the documentation at https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. The following steps help you create a random forest classifier model so that you can compare it to the decision tree classifier used earlier:

1. Import the required packages:

```
from sklearn.ensemble \
    import RandomForestClassifier
```

2. Fit the model to the data and display its accuracy. Note that this step will take a while to complete because you're using a whole bunch of trees to complete it. If your system is taking too long, you can always reduce either or both of the `max_depth` and `n_estimators` arguments. Changing either of them will affect the accuracy of your model. Of the two, `max_depth` is the most important for this particular example, but in the real world, you'd need to experiment to find the right combination. The purpose of the `n_jobs=-1` argument is to reduce the amount of time to build the model by using all of the processors on the system:

```
rfc = RandomForestClassifier(max_depth=9,
```

```
    n_estimators=100,
    n_jobs=-1)
rfc.fit(X_train, y_train)
rfc_yHat = rfc.predict(X_test)
print(f"Accuracy score: " \
        f"{accuracy_score(y_test, rfc_yHat)}")
```

Figure 8.13 shows that even though it took longer to build this model (using all of the processors no less), it performs only slightly better than the decision tree classifier. In this case, the contributing factors are the small dataset and the fact that the number of fraud cases is small. However, even a small difference is better than no difference at all when it comes to fraud and you need to consider that a real-world scenario will be dealing with far more entries in the dataset:

```
Accuracy score: 0.9994014474089928
```

Figure 8.13 – The accuracy of the random forest classifier

3. Calculate the `precision`, `recall`, `fbeta_score`, and `support` values for the model:

```
precision, recall, fbeta_score, support = \
    precision_recall_fscore_support(y_test, rfc_yHat)
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F-beta score: {fbeta_score}")
print(f"Support: {support}")
```

As with accuracy, the differences (as shown in *Figure 8.14*) between the two models are very small, but important when dealing with fraud. After all, you don't want to claim a customer has committed fraud unless it's true:

```
Precision: [0.99949141 0.93670886]
Recall: [0.99990914 0.7254902 ]
F-beta score: [0.99970023 0.81767956]
Support: [55031   102]
```

Figure 8.14 – The precision, recall, F-beta, and support scores for the random forest classifier

4. Plot the confusion matrix for this model:

```
matrix = plot_confusion_matrix(rfc,
    X=X_test,
    y_true=y_test,
    cmap=plt.cm.Blues)
plt.title('Confusion Matrix for RFC Fraud Detection')
```

```
plt.show(matrix)

plt.show()
```

Figure 8.15 shows what it all comes down to in the end. The random forest classifier has the same prediction rate for transactions that aren't fraudulent as the decision tree classifier in this case. However, it also finds one more case of fraud, which is important:

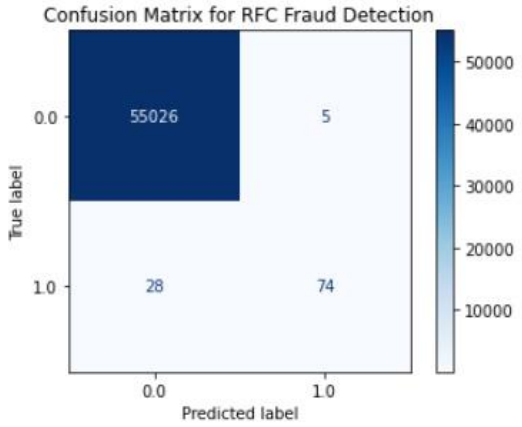

Figure 8.15 – The confusion matrix for the random forest classifier

This example has looked at credit card fraud, but the same techniques work on other sorts of fraud as well. The main things to consider when looking for fraud are to obtain a large enough amount of data, train the model using your best assumptions about the data, and then test the model for accuracy. Tweak the model as needed to obtain the required level of accuracy. Of course, no model is going to be completely accurate. While ML will greatly reduce the burden on human detectives looking for fraud, it can't eliminate the need for a human to look at the data entirely.

Creating a ROC curve and calculating AUC

A **Receiver Operating Characteristic** (**ROC**) curve and **Area Under the Curve** (**AUC**) calculation help you determine where to set thresholds in your ML model. These are methods of looking at the performance of a model at all classification thresholds. The *X*-axis shows the false positive rate, while the *Y*-axis shows the true positive rate. As the true positive rate increases, so does the false positive rate. The goal is to determine where to place the threshold for determining whether a particular sample is fraudulent or not based on its score during analysis. A score indicates the model's confidence as to whether a particular sample is fraud or not, but the model doesn't determine where to place the line between fraud and legitimate; that line is the **threshold**. Therefore, a ROC curve helps a human user of a model determine where to set the threshold, and where to say that it's best to detect fraud.

The **True Positive Rate** (**TPR**) defines the ratio between true positives (TP) (as shown in *Figure 8.11*, *Figure 8.12*, and *Figure 8.15*) and false negatives (FN): `TPR = TP / TP + FN`. The **False Positive Rate** (**FPR**) defines the ratio between the false positives (FP) and the true negatives (TN): `FPR = FP / FP + TN`. Essentially, what you're trying to determine is how many true positives are acceptable for a given number of false positives when working with a model.

As part of plotting a ROC curve, you also calculate the AUC, which is essentially another good measure for the model. It's a measure of overall performance against all classification thresholds; the higher the number, the better the model. The AUC is a probability measure that tells you how

likely it is that the model will rank a random positive example higher than a random negative example. Consequently, the `MultinomialNB` classifier used in an earlier example would have an AUC of 0, which means it never produces a correct detection. With all of these things in mind, use the following steps to create two ROC curves comparing `DecisionTreeClassifier` to the `RandomForestClassifier` classifier used earlier:

1. Import the required packages:

```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from numpy import argmax
from numpy import sqrt
```

2. Perform the required `DecisionTreeClassifier` analysis. Notice that this is a three-step process of calculating the probabilities for each value in `X_test`, determining the FPR and TPR values, and then estimating the AUC value. `dtc_thresholds` will be used in a later location to determine the best-calculated place to put the threshold for this model:

```
dtc_y_scores = dtc.predict_proba(X_test)
dtc_fpr, dtc_tpr, dtc_thresholds = \
    roc_curve(y_test, dtc_y_scores[:, 1])
dtc_roc_auc = auc(dtc_fpr, dtc_tpr)
```

3. Perform the required `RandomForestClassifier` analysis. Notice that this is the same process we followed previously:

```
rfc_y_scores = rfc.predict_proba(X_test)
rfc_fpr, rfc_tpr, rfc_thresholds = \
    roc_curve(y_test, rfc_y_scores[:, 1])
rfc_roc_auc = auc(rfc_fpr, rfc_tpr)
```

4. Calculate the best threshold for each of the models and display the **generic mean** (**G-mean**) for each one. The best-calculated threshold represents a balance between sensitivity, which is the TPR, and specificity, which is 1 - FPR. To calculate this value, the code must perform the analysis for each threshold; then, a call to `argmax()` will report the best option based on the calculation:

```
dtc_gmeans = sqrt(dtc_tpr * (1-dtc_fpr))
dtc_ix = argmax(dtc_gmeans)
print('Best DTC Threshold=%f, G-Mean=%.3f'
        % (dtc_thresholds[dtc_ix],
            dtc_gmeans[dtc_ix]))
rfc_gmeans = sqrt(rfc_tpr * (1-rfc_fpr))
```

```
rfc_ix = argmax(rfc_gmeans)
print('Best RFC Threshold=%f, G-Mean=%.3f'
        % (rfc_thresholds[rfc_ix],
            rfc_gmeans[rfc_ix]))
```

Figure 8.16 shows the result of the calculation for each model:

```
Best DTC Threshold=0.062500, G-Mean=0.885
Best RFC Threshold=0.010000, G-Mean=0.939
```

Figure 8.16 – Calculated best threshold and G-mean values for each model

5. Create the ROC curve plot and display the AUC values:

```
plt.title('Receiver Operating Characteristic')
plt.plot(dtc_fpr, dtc_tpr, 'g',
        label = 'DTC AUC = %0.2f' % dtc_roc_auc)
plt.plot(rfc_fpr, rfc_tpr, 'b',
        label = 'RFC AUC = %0.2f' % rfc_roc_auc)
plt.plot([0, 1], [0, 1],'r--',
        label = 'No Skill')
plt.scatter(dtc_fpr[dtc_ix], dtc_tpr[dtc_ix],
        marker='o', color='g',
        label='DTC Best')
plt.scatter(rfc_fpr[rfc_ix], rfc_tpr[rfc_ix],
        marker='o', color='b',
        label='RFC Best')
plt.legend(loc = 'lower right')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.title('ROC Curve Comparison DTC to RFC')
plt.show()
```

The output of this example appears in *Figure 8.17*. Notice that the RFC model outperforms the DTC in this particular case by a small margin. In addition, the plot shows where you'd place the threshold for each model. Given the data and other characteristics of this example, once the model has achieved a maximum value, there is little advantage in increasing the threshold further:

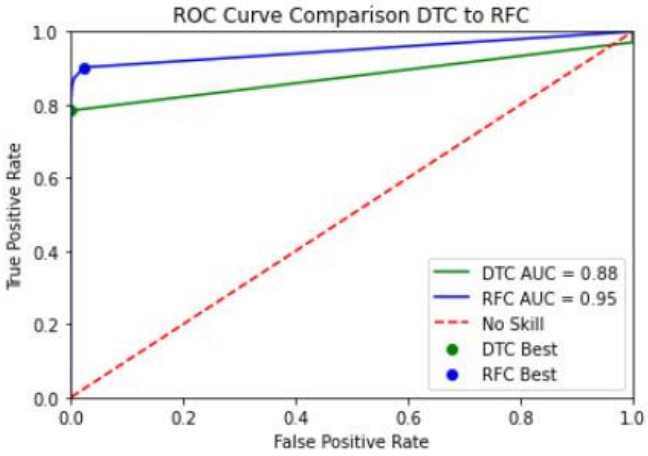

Figure 8.17 – The ROC curve and AUC calculation for each model

Of course, you won't likely always see this result. The main takeaway from this example is that you need to compare models and settings to determine how best to configure your ML application to detect as much fraud as possible without creating an overabundance of false positives.

Links

Real ID initiative: (https://www.dhs.gov/real-id)

Nigerian Fraud Scams: https://consumer.georgia.gov/consumer-topics/nigerian-fraud-scams

Ponzi scheme (https://www.investor.gov/protect-your-investments/fraud/types-fraud/ponzi-scheme

Unsupervised Deep Learning for Fake Content Detection in Social Media: https://scholarspace.manoa.hawaii.edu/items/6d7560aa-2aff-4439-a884-35994e242c06

There Is No Such Thing As A Free Lunch (**TINSTAAFL**) (see https://www.investopedia.com/terms/t/tanstaafl.asp for details

FDB: Fraud Dataset Benchmark at https://github.com/amazon-research/fraud-dataset-benchmark

You can read the goals of creating the FDB dataset at https://www.linkedin.com/posts/groverpr_fdb-fraud-dataset-benchmark-activity-6970921322067427328-fNCo.

Here are some other dataset sites you should visit:

- **Amazon Fraud Detector (fake data that looks real):** https://docs.aws.amazon.com/frauddetector/latest/ug/step-1-get-s3-data.html
- **data.world:** https://data.world/datasets/fraud
- **Machine Learning Repository (Australian Credit Approval):** https://archive.ics.uci.edu/ml/datasets/Statlog+%28Australian+Credit+Approval%29
- **Machine Learning Repository (German Credit Data):** https://archive.ics.uci.edu/ml/datasets/statlog+(german+credit+data)

- **OpenML (CreditCardFraudDetection):** https://www.openml.org/search?type=data&sort=runs&id=42175&status=active
- **Papers with Code (Amazon-Fraud):** https://paperswithcode.com/dataset/amazon-fraud
- **Papers with Code (Yelp-Fraud):** https://paperswithcode.com/dataset/yelpchi
- **Synthesized (Fraud Detection Project):** https://www.synthesized.io/data-template-pages/fraud-detection
- **US Treasury Financial Crimes Enforcement Unit:** https://www.fincen.gov/fincen-mortgage-fraud-sar-datasets

Note that older datasets are often removed without much comment from the provider. Two examples that are often cited by researchers and data scientists are the Kaggle dataset at https://www.kaggle.com/dalpozz/creditcardfraud and the one at http://weka.8497.n7.nabble.com/file/n23121/credit_fruad.arff. Both of these datasets are gone and it isn't worth pursuing them because there are others.

Kount- How to prevent loyalty Fraud: https://kount.com/blog/how-to-prevent-loyalty-fraud/

Artificial networks learn to smell like the brain (https://news.mit.edu/2021/artificial-networks-learn-smell-like-the-brain-1018)

Further reading

The following bullets provide you with some additional reading that you may find useful in understanding the materials in this chapter:

- Discover how deep learning is already being used to extract information from ID cards: *ID Card Digitization and Information Extraction using Deep Learning – A Review*: https://nanonets.com/blog/id-card-digitization-deep-learning/
- Understand how organizations suffer phishing attacks: *How to Prevent Phishing on an Organizational Level*: https://www.tsp.me/blog/cyber-security/how-to-prevent-phishing-on-an-organizational-level/
- Uncover Ponzi schemes using ML techniques: *Data-driven Smart Ponzi Scheme Detection* at https://arxiv.org/pdf/2108.09305.pdf and *Evaluating Machine-Learning Techniques for Detecting Smart Ponzi Schemes* at https://ieeexplore.ieee.org/document/9474794
- Read about the man who did sell the Brooklyn Bridge and other monuments: *Meet the Conman Who Sold the Brooklyn Bridge — Many Times Over*: https://history.howstuffworks.com/historical-figures/conman-sold-brooklyn-bridge.htm
- Learn about the no free lunch theorem as it applies to ML: *No Free Lunch Theorem for Machine Learning*: https://machinelearningmastery.com/no-free-lunch-theorem-for-machine-learning/

- Learn more about the historical basis of Niccolo Machiavelli: *Italian philosopher and writer Niccolo Machiavelli born* at https://www.history.com/this-day-in-history/niccolo-machiavelli-born and *Machiavelli* at https://www.history.com/topics/renaissance/machiavelli
- Gain an understanding of how AI is used to detect government fraud: *Using AI and machine learning to reduce government fraud*: https://www.brookings.edu/research/using-ai-and-machine-learning-to-reduce-government-fraud/
- See how accuracy, precision, and recall relate: *Accuracy, Precision, Recall & F1 Score: Interpretation of Performance Measures*: https://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/
- Gain a better understanding of the F-beta measure: *A Gentle Introduction to the Fbeta-Measure for Machine Learning*: https://machinelearningmastery.com/fbeta-measure-for-machine-learning/
- Consider how long-term fraud detection and protection improve business productivity and profit: *How long-term digital fraud protection can increase profitability*: https://kount.com/blog/long-term-fraud-protection-benefits/
- Pick up techniques to help determine when someone is lying: *Become a Human Lie Detector: How to Sniff Out a Liar*: https://www.artofmanliness.com/character/behavior/become-a-human-lie-detector-how-to-sniff-out-a-liar/
- Discover credit card statistics that demonstrate why fraud detection is so important: *42 credit card fraud statistics in 2022 + steps for reporting fraud*: https://www.creditrepair.com/blog/security/credit-card-fraud-statistics/

Chapter 9

Technical requirements

This chapter requires that you have access to either Google Colab or Jupyter Notebook to work with the example code. The *Requirements to use this book* section of *Chapter 1*, *Defining Machine Learning Security*, provides additional details on how to set up and configure your programming environment. When testing the code, use a test site, test data, and test APIs to avoid damaging production setups and to improve the reliability of the testing process. Testing on a non-production network is highly recommended but not absolutely necessary. Using the downloadable source is always highly recommended. You can find the downloadable source on the Packt GitHub site at https://github.com/PacktPublishing/Machine-Learning-Security-Principles or my website at http://www.johnmuellerbooks.com/source-code/.

Hands-on Section

An example of detecting behavior

Behavior detection is relatively new in the world of ML. However, the techniques for making a prediction are based on well-established models. The example in this section anticipates human behavior based on two datasets from two completely different sources. It models a real-world scenario in that you often have to get creative and combine data from multiple sources to catch a hacker through behavioral prediction. Just knowing API call patterns, as discussed in *Chapter 5*, wouldn't be enough in the real world, you'd need to look into other patterns. Just what you need to track depends on the behavior you want to model.

This example builds a behavior detection scenario by combining data about COVID admissions into hospitals, new cases, existing hospital cases (for ongoing treatment), and its effect on retail sales and recreational activities. The example asks the question of whether it's possible to create a correlation between these two disparate sources of data and then use that correlation to correctly predict future behavior. More importantly, it also shows you that there is indeed a behavioral lag between the COVID events and the COVID effect. The following sections help you build and try this somewhat complex example. You can also find the source code for this example in the `MLSec; 09; Human Behavior Prediction.ipynb` file in the downloadable source.

Obtaining the data

The two datasets come from completely different sources, which is actually a good thing because you know that there is no interaction between the two. Often, when you work through a security matter of this kind, there won't be any interaction between your data sources. The first dataset comes from the GOV.UK Coronavirus (COVID-19) UK site https://coronavirus.data.gov.uk/details/download. The site allows you to set the parameters of the data you download. The example uses the following parameters:

- Data release date (must be set first): Archive: 2021-12-31

- Metrics:
- `hospitalCases`
- `newAdmissions`
- `newCasesByPublishDate`
- Data format: CSV

Here is a link you can use to obtain the dataset with all of the questions already answered: https://api.coronavirus.data.gov.uk/v2/data?areaType=overview&metric=hospitalCases&metric=newAdmissions&metric=newCasesByPublishDate&format=csv. The data used is a little older because of the second dataset that the example uses. Otherwise, you can play around with various dates and up to five metrics.

The second dataset comes from the Google COVID-19 Community Mobility Reports site at https://www.google.com/covid19/mobility/. Click the **Region CSVs** button to download the mobility reports for all of the regions that Google tracks. You can also use this direct link: https://www.gstatic.com/covid19/mobility/Region_Mobility_Report_CSVs.zip. This series of `.csv` files should appear in the `Region_Mobility_Report_CSVs` subdirectory of your code directly.

Importing and combining the datasets

This section helps you create a combined dataset from the two datasets downloaded in the previous section. As with most real-world datasets, you need to reshape the data to make the two datasets fit together in a manner that will actually work for your analysis. There is also the issue of missing data to consider and whether the data appears in the right format. Only when the data is completely remediated can you combine the two datasets together to create a single dataset to use for developing a model. The following steps show how to perform the various tasks:

1. Import the required packages:

```
import pandas as pd
```

2. Read the Google regional reports into a dataset:

```
mobility_df = pd.read_csv(
    "Region_Mobility_Report_CSVs/" \
    "2020_GB_Region_Mobility_Report.csv")
print(mobility_df)
```

Figure 9.1 shows three major issues with this dataset. The first is that the dates begin with **2020-02-15**, so you don't have a full year to work with from 2020. The second is that the date is in text format, and you need an actual date to work with. The third is that the actual data columns that the example will use are really long. It's hard to imagine typing `retail_and_recreation_percent_change_from_baseline` every time you need the retail and recreation change data:

```
                          place_id        date  \
0       ChIJqZHHQhE7WgIReiWIMkOg-MQ  2020-02-15
1       ChIJqZHHQhE7WgIReiWIMkOg-MQ  2020-02-16
2       ChIJqZHHQhE7WgIReiWIMkOg-MQ  2020-02-17
3       ChIJqZHHQhE7WgIReiWIMkOg-MQ  2020-02-18
4       ChIJqZHHQhE7WgIReiWIMkOg-MQ  2020-02-19
...                             ...         ...
133377  ChIJh-IigLwxeUgRAKFv7Z75DAM  2020-12-27
133378  ChIJh-IigLwxeUgRAKFv7Z75DAM  2020-12-28
133379  ChIJh-IigLwxeUgRAKFv7Z75DAM  2020-12-29
133380  ChIJh-IigLwxeUgRAKFv7Z75DAM  2020-12-30
133381  ChIJh-IigLwxeUgRAKFv7Z75DAM  2020-12-31

        retail_and_recreation_percent_change_from_baseline  \
0                                                    -12.0
1                                                     -7.0
2                                                     10.0
3                                                      7.0
4                                                      6.0
```

Figure 9.1 – The Google dataset has some format issues to deal with

3. Get rid of the data columns that you don't actually need from the dataset. Modify the date column to make it usable by the model. In addition, rename the columns so that they're easier to use (or at least easier to type). Finally, just use the first 321 rows so that the amount of data matches that found in the UK dataset:

```
Mobility_df.drop(
    ['country_region_code', 'country_region',
        'sub_region_1', 'sub_region_2', 'metro_area',
        'iso_3166_2_code', 'census_fips_code',
        'place_id'], axis='columns', inplace=True)
mobility_df['date'] = \
    pd.to_datetime(mobility_df['date'])
mobility_df.rename(columns={
'retail_and_recreation_percent_change_from_baseline':
    'retail_and_recreation'}, inplace=True)
mobility_df.rename(columns={
'grocery_and_pharmacy_percent_change_from_baseline':
    'grocery_and_pharmacy'}, inplace=True)
mobility_df.rename(columns={
    'parks_percent_change_from_baseline':
        'parks'}, inplace=True)
mobility_df.rename(columns={
    'transit_stations_percent_change_from_baseline':
        'transit'}, inplace=True)
```

```
mobility_df.rename(columns={
    'workplaces_percent_change_from_baseline':
        'workplaces'}, inplace=True)
mobility_df.rename(columns={
    'residential_percent_change_from_baseline':
        'residential'}, inplace=True)
mobility_df = mobility_df.head(321)
print(mobility_df)
```

As shown in *Figure 9.2*, the dataset is now a lot easier to use:

```
           date  retail_and_recreation  grocery_and_pharmacy  parks  transit  \
0    2020-02-15                  -12.0                  -7.0  -35.0    -12.0
1    2020-02-16                   -7.0                  -6.0  -28.0     -7.0
2    2020-02-17                   10.0                   1.0   24.0     -2.0
3    2020-02-18                    7.0                  -1.0   20.0     -3.0
4    2020-02-19                    6.0                  -2.0    8.0     -4.0
..          ...                    ...                   ...    ...      ...
316  2020-12-27                  -58.0                 -24.0    3.0    -59.0
317  2020-12-28                  -51.0                 -24.0   16.0    -70.0
318  2020-12-29                  -47.0                 -11.0   14.0    -64.0
319  2020-12-30                  -42.0                  -3.0   18.0    -63.0
320  2020-12-31                  -53.0                   2.0   14.0    -67.0

     workplaces  residential
0          -4.0          2.0
1          -3.0          1.0
2         -14.0          2.0
3         -14.0          2.0
4         -14.0          3.0
..          ...          ...
316       -29.0         10.0
317       -77.0         26.0
318       -68.0         24.0
319       -66.0         23.0
320       -69.0         24.0

[321 rows x 7 columns]
```

Figure 9.2 – The Google dataset shows just the information needed to create the model

4. Read the UK COVID data into a dataset:

```
Covid_df = pd.read_csv('overview_2022-09-22.csv')
print(covid_df)
```

Figure 9.3 shows that the UK dataset also presents problems for the example. First, the dates aren't in the same order as the Google dataset. In addition, the dates are textual, rather than actual dates. Some of the columns are also missing data. You'll also notice that the current dataset contains 959 rows, but that's because it includes multiple years of data, whereas the Google dataset is exclusive to 2020:

```
      areaCode        areaName  areaType        date  hospitalCases  \
0    K02000001  United Kingdom  overview  2022-09-15         5729.0
1    K02000001  United Kingdom  overview  2022-09-14         5705.0
2    K02000001  United Kingdom  overview  2022-09-13         5792.0
3    K02000001  United Kingdom  overview  2022-09-12         5849.0
4    K02000001  United Kingdom  overview  2022-09-11         5911.0
..         ...             ...       ...         ...            ...
954  K02000001  United Kingdom  overview  2020-02-04            NaN
955  K02000001  United Kingdom  overview  2020-02-03            NaN
956  K02000001  United Kingdom  overview  2020-02-02            NaN
957  K02000001  United Kingdom  overview  2020-02-01            NaN
958  K02000001  United Kingdom  overview  2020-01-31            NaN

     newAdmissions  newCasesByPublishDate
0              NaN                    NaN
1              NaN                    NaN
2              NaN                    NaN
3              NaN                    NaN
4            544.0                    NaN
..             ...                    ...
954            NaN                    0.0
955            NaN                    0.0
956            NaN                    0.0
957            NaN                    0.0
958            NaN                    2.0

[959 rows x 7 columns]
```

Figure 9.3 – The UK dataset has ordering, formatting, and missing value issues to deal with

5. Get rid of the data columns that you don't need. Change the date column to something that the model can use. Deal with the missing values in the various columns and put the data into sorted order:

```
Covid_df.drop(['areaCode', 'areaName', 'areaType'],
    axis='columns', inplace=True)
covid_df['date'] = pd.to_datetime(covid_df['date'])
covid_df['hospitalCases'] = \
    covid_df['hospitalCases'].fillna(0)
covid_df['newAdmissions'] = \
    covid_df['newAdmissions'].fillna(0)
covid_df['newCasesByPublishDate'] = \
    covid_df['newCasesByPublishDate'].fillna(0)
covid_df.sort_values(by=['date'], ignore_index=True,
    inplace=True)
print(covid_df)
```

Figure 9.4 shows that most of the issues in using the data are now addressed:

```
           date  hospitalCases  newAdmissions  newCasesByPublishDate
0    2020-01-31            0.0            0.0                    2.0
1    2020-02-01            0.0            0.0                    0.0
2    2020-02-02            0.0            0.0                    0.0
3    2020-02-03            0.0            0.0                    0.0
4    2020-02-04            0.0            0.0                    0.0
..          ...            ...            ...                    ...
954  2022-09-11         5911.0          544.0                    0.0
955  2022-09-12         5849.0            0.0                    0.0
956  2022-09-13         5792.0            0.0                    0.0
957  2022-09-14         5705.0            0.0                    0.0
958  2022-09-15         5729.0            0.0                    0.0

[959 rows x 4 columns]
```

Figure 9.4 – The UK dataset now contains just the necessary information and most other issues are addressed

6. Take out the data for 2021 and place it into a new dataset. You will use this data later to make a prediction, but don't worry about it for now:

```
Dates_2021 = (covid_df['date'] >= '2021-01-01') & \
    (covid_df['date'] <= '2021-12-31')
covid_df_2021 = covid_df.loc[dates_2021]
```

7. Retain just the 2020 data for use in creating the model. Notice that the start date for the 2020 data is adjusted to match the start date of the Google dataset:

```
Dates_2020 = (covid_df['date'] >= '2020-02-15') & \
    (covid_df['date'] <= '2020-12-31')
covid_df = covid_df.loc[dates_2020]
print(covid_df)
```

As shown in *Figure 9.5*, the two datasets now have the same number of rows, which is essential before you can combine them:

```
           date  hospitalCases  newAdmissions  newCasesByPublishDate
15   2020-02-15            0.0            0.0                    0.0
16   2020-02-16            0.0            0.0                    0.0
17   2020-02-17            0.0            0.0                    0.0
18   2020-02-18            0.0            0.0                    0.0
19   2020-02-19            0.0            0.0                    0.0
..          ...            ...            ...                    ...
331  2020-12-27        22767.0         2871.0                30501.0
332  2020-12-28        24052.0         3131.0                41385.0
333  2020-12-29        25552.0         3250.0                53135.0
334  2020-12-30        26554.0         3289.0                50023.0
335  2020-12-31        26581.0         2924.0                55892.0

[321 rows x 4 columns]
```

Figure 9.5 – The two datasets are finally compatible, so you can merge them together

8. Merge the two datasets together now that you know they're compatible:

```
merged_df = pd.merge(covid_df,
```

```
    mobility_df,
    on='date')
print(merged_df)
```

This may seem like a lot of work to obtain a dataset to use, but this really is a simple example of the kinds of dataset manipulation you need to perform with real-world data to do anything more than simple analysis. To really become a security detective, you need to perform combinations of this sort to find the patterns that everyone else is missing and that the hacker doesn't know about. Yes, we're using COVID for this example, but the principles are the same when working with other kinds of data.

Creating the training and testing datasets

Now that you have a dataset of merged data from multiple sources to use, it's time to create the training and testing datasets that will actually be used to develop a model. Use the following steps to perform this task:

1. Import the required packages:

```
from sklearn.model_selection import train_test_split
```

2. Perform the actual splitting of the merged dataset from the previous section:

train, test = train_test_split(merged_df,

```
    test_size=0.25)
print(f"Train Size: {train.shape}")
print(f"Test Size: {test.shape}")
```

Figure 9.6 shows the shape of each of the datasets. They're a bit small but still quite usable. It's important to remember that more data is usually better when it comes to ML needs:

```
Train Size: (240, 10)
Test Size: (81, 10)
```

Figure 9.6 – The two datasets are a little small but quite usable for this example

3. Use the split datasets to create a training and testing features matrix and a training and testing target vector. The goal is to determine the correlation, if any, between COVID statistics and retail and recreation activity levels. In other words, the datasets will help determine whether human behavior is affected by COVID:

```
train_features = train[
    ['hospitalCases', 'newAdmissions',
    'newCasesByPublishDate']].values
train_target = train['retail_and_recreation'].values
test_features = test[
    ['hospitalCases', 'newAdmissions',
```

```
    'newCasesByPublishDate']].values

test_target = test['retail_and_recreation'].values
```

Now that the data is ready for building a model, it's time to see how to create a regression model for use with this particular data. The example begins with a **k-nearest neighbor** (**kNN**) regressor, but you also see a random forest regressor and an XGBoost regressor used later in the chapter.

kNN versus a random forest regressor versus an XGBoost regressor

The kNN regressor is a tried and true algorithm that you see used in many examples online. In fact, you can hardly find an example that doesn't at least mention kNN. However, newer algorithms have come onto the scene that can provide better results and far greater flexibility. So, many data scientists have started using these newer algorithms in place of kNN. When it comes to random forest regressors or XGBoost regressors, the main issue is whether the result is computed in parallel or sequentially. Both algorithms are ensembles of learners, but the random forest regressor uses a voting mechanism to come up with the result, while XGBoost feeds the output of one model into the next model. This chapter shows you all three algorithms so that you can compare them. Of course, newer is not always better, and there are a lot more than just the three algorithms used here.

Building the kNN model

The kNN algorithm (where *k* is the number of neighbors) is relatively simple, but it's also quite flexible because you can use it for both classification (determining which group a data point belongs to) and regression (the examination of one or more independent variables on a dependent variable). It's based on the principle that things that are close together tend to have a lot in common. The knowledge gained about a single point tends to translate into knowledge about the points around it within reason. You saw the kNN algorithm used for classification in the *Checking another model* section of *Chapter 8*, *Locating Potential Fraud*. Now you'll see it in action for regression.

One of the essentials of using kNN is the selection of how many neighbors to use for the examination process. Think about a grocery store. If you see a can of peas and then examine the cans one row up or one column over, they're likely to be peas too. However, if you start looking four rows up and four columns over, then you might see other vegetables, but you'd still see canned vegetables. Now, if you look six rows up and six columns over, you might start seeing something altogether different, perhaps juice in bottles or canned mushrooms; still something canned, but definitely not vegetables any longer. At some point, the number of neighbors becomes too large and the characteristics no longer hold as being close enough. If you use too large a value, then the model could be underfitted, while too small a value can cause overfitting. Experimentation is needed to find the right value, which is especially the case for regression problems.

In addition to problems with underfitting and overfitting, kNN can also suffer from the curse of dimensionality. It's not enough for kNN to have a close relationship between two points in one dimension; it must have a close relationship with every dimension. The example uses only three dimensions: hospital cases, new admissions, and new cases. However, when working with security data, the number of dimensions can increase very quickly, which means that you need more data to ensure that the model you build will work well. The modeling of human behavior for security needs can create an impressive number of dimensions, but very little data, so it's essential to tune your dataset (as has been done in this case) to get to the crux of the problem you're trying to resolve.

With all of these issues in mind, here are the steps needed to create the kNN model for this example:

1. Import the required packages:

```
from sklearn.neighbors import KNeighborsRegressor
```

2. Create the model and fit it to the training set. This part of the example uses `5` neighbors. Later, you will see the effects on the model of using values of both `3` and `7` neighbors. When working with `KNeighborsRegressor`, the value of k, which is provided by `n_neighbors`, is critical. The documentation for this algorithm at https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html shows that there are other parameters you can change, and you should at least try them when fine-tuning your model. However, to start with, use the default settings and modify `n_neighbors` to optimize your results:

```
knn = KNeighborsRegressor(n_neighbors = 5)
knn.fit(train_features, train_target)
```

Now that you have a basic model to use, it's time to test it.

Testing the model using kNN

It's time to test the model. Something important to remember is that the call to `train_test_split()` earlier actually randomizes the data. In other words, the training and testing process don't use the data in the same sequence as it appears in the original dataset, so you won't see anything resembling a smooth curve in the testing output. Rather, what you're looking for is how well the model fits its curve to the data points provided. You see the actual effect of the prediction later in the chapter, but for now, the focus is on testing using the following steps:

3. Import the required packages:

```
import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt
```

4. Perform a prediction using the training data. Then, test the goodness of that prediction using both **root-mean-square error** (**RSME**, lower is better) and R2 (higher is better) measures:

```
train_pred = knn.predict(train_features)
trainRSME = np.sqrt(mean_squared_error(train_target,
        train_pred))
trainR2 = r2_score(train_target, train_pred) * 100
print(f"Training RSME: {trainRSME}")
print(f"Training R2 Score: {trainR2}")
```

Figure 9.7 shows the typical output from this step:

```
Training RSME: 7.560798017846175
Training R2 Score: 90.01955840683688
```

Figure 9.7 – The output shows the result of the kNN with 5 neighbors training prediction

5. Perform a prediction using the testing data for comparison. If the statistics are close to what you achieve with the training data, then the model will likely perform well within its given error rate and variance:

```
test_pred = knn.predict(test_features)
testRSME = np.sqrt(mean_squared_error(test_target,
        test_pred))
testR2 = r2_score(test_target, test_pred) * 100
print(f"Testing RSME: {testRSME}")
print(f"Testing R2 Score: {testR2}")
```

Figure 9.8 shows the typical output when performing the testing prediction step. Note that the RSME is higher and the R2 score is lower, which indicates that the model may not perform as well as might have been anticipated:

```
Testing RSME: 10.031924350434442
Testing R2 Score: 81.88173745447082
```

Figure 9.8 – The output shows the result of the kNN with 5 neighbors testing prediction

6. Plot the prediction against the original data to see how well the model works:

```
x = np.arange(len(test_target))
plt.scatter(x, test_target, color='blue',
    label='Original')
plt.plot(x, test_pred, color='red',
    label='Prediction')
plt.legend()
plt.show()
```

Figure 9.9 shows that the model follows acceptably in this case but could possibly follow better:

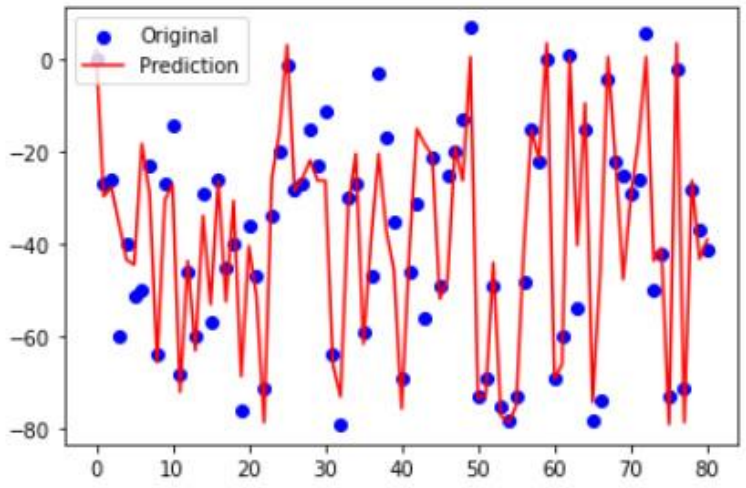

Figure 9.9 – The plot shows that kNN with 5 neighbors works acceptably but not great

Of course, this test uses just one value of *k*. You have no idea how well the model will work with other values. The next section tests two other values of *k*, 3 and 7, to see how well the model works with those values.

Using a different-sized neighborhood with kNN

As the book progresses, you will discover more methods of ensuring that you get correct results from the ML models you create. Testing various model configurations is essential, especially in security scenarios that are already riddled with stealth and misinformation. The following steps show what happens when you use different values of *k* with kNN to perform a regression:

1. Perform the same kNN tests using a value of `n_neighbors = 7`:

```
knn = KNeighborsRegressor(n_neighbors = 7)
knn.fit(train_features, train_target)
train_pred = knn.predict(train_features)
trainRSME = np.sqrt(mean_squared_error(train_target,
    train_pred))
trainR2 = r2_score(train_target, train_pred) * 100
print(f"Training RSME: {trainRSME}")
print(f"Training R2 Score: {trainR2}")
test_pred = knn.predict(test_features)
testRSME = np.sqrt(mean_squared_error(test_target,
        test_pred))
testR2 = r2_score(test_target, test_pred) * 100
print(f"Testing RSME: {testRSME}")
print(f"Testing R2 Score: {testR2}")
```

```
plt.scatter(x, test_target, color='blue',
    label='Original')
plt.plot(x, test_pred, color='red',
    label='Prediction')
plt.legend()
plt.show()
```

Compare the output in *Figure 9.10* with the outputs in the previous section and you will see that changing the value to `n_neighbors = 7` moves the model in the wrong direction. Every indicator shows worse performance at this level:

```
Training RSME: 8.526933120141974
Training R2 Score: 87.30595018803965
Testing RSME: 10.268025648490804
Testing R2 Score: 81.01887535277126
```

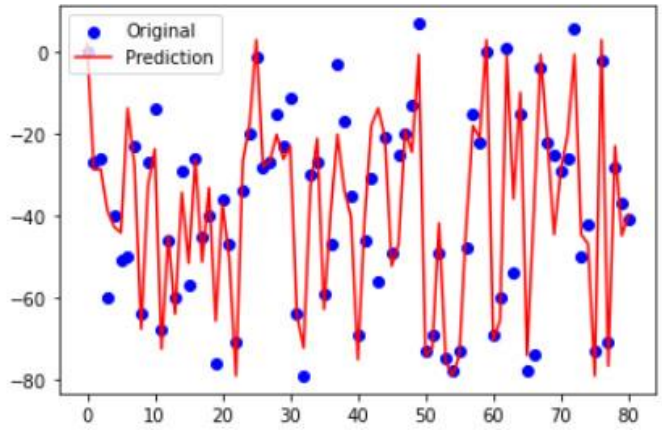

Figure 9.10 – The statistics and plot show that kNN with 7 neighbors shows worse performance

2. Perform the same kNN tests using a value of `n_neighbors = 3`:

```
knn = KNeighborsRegressor(n_neighbors = 3)
knn.fit(train_features, train_target)
train_pred = knn.predict(train_features)
trainRSME = np.sqrt(mean_squared_error(train_target,
        train_pred))
trainR2 = r2_score(train_target, train_pred) * 100
print(f"Training RSME: {trainRSME}")
print(f"Training R2 Score: {trainR2}")
test_pred = knn.predict(test_features)
testRSME = np.sqrt(mean_squared_error(test_target,
        test_pred))
```

```
testR2 = r2_score(test_target, test_pred) * 100
print(f"Testing RSME: {testRSME}")
print(f"Testing R2 Score: {testR2}")
plt.scatter(x, test_target, color='blue',
    label='Original')
plt.plot(x, test_pred, color='red',
    label='Prediction')
plt.legend()
plt.show()
```

The output shown in *Figure 9.11* tells you that using a value of `n_neighbors = 3` improves performance significantly:

```
Training RSME: 7.239871596978826
Training R2 Score: 90.84883865562644
Testing RSME: 9.51361741524911
Testing R2 Score: 83.70556085749895
```

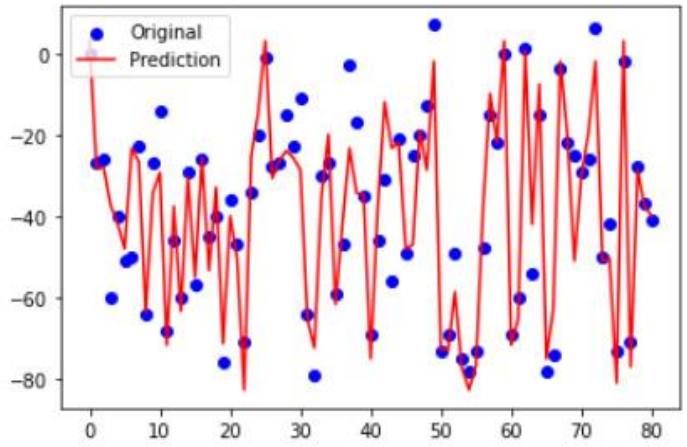

Figure 9.11 – The statistics and plot shows that kNN with 3 neighbors works the best so far

Look at the plots carefully at this point, and you'll notice that there is a tendency toward overfitting as the number of neighbors becomes smaller. If you were to use a value of `n_neighbors = 1`, you would find that performance becomes worse again. A value of `n_neighbors = 3` seems to work best for this particular dataset and model configuration.

Using an odd value for *k* in kNN

The general consensus is that using an odd value for *k* is best. The use of an odd number prevents the risk of a tie when performing the calculation.

Building and testing using a random forest regressor

It usually pays to look at more than one algorithm when creating a model. You likely don't have time to work with every algorithm out there, but there are some classics that do appear in more than a

few places. The random forest regressor is one of these favorites. The example in the *Setting the example up* section of *Chapter 8*, *Locating Potential Fraud*, uses `DecisionTreeClassifier` to perform fraud analysis. A random forest regressor is an ensemble of decision tree regressors to put the power of the crowd into the decision-making process. In this case, each decision tree works just like normal, but the output is calculated as the mean of each tree's output so that you can obtain a better result. The *Using ensemble learning* section of *Chapter 3*, *Mitigating Inference Risk by Avoiding Adversarial Machine Learning Attacks*, discusses some of the advantages and disadvantages of using ensemble learning. It's time to see an ensemble at work using the following steps:

1. Import the required packages:

```
from sklearn.ensemble import RandomForestRegressor
```

2. Fit the model to the data. In the case of `RandomForestRegressor`, the key tuning features are the maximum tree depth, `max_depth`, and the number of estimators (trees), `n_estimators`. The example has already tuned both of these arguments for you, but feel free to play around with them to see how each of them works toward the goal of obtaining a better prediction. Note that when looking at the documentation for this algorithm, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html , you will find that there are a lot more arguments than for kNN to help in tuning your model:

```
rf = RandomForestRegressor(max_depth = 4,
    n_estimators=500)
rf.fit(train_features, train_target)
```

3. Perform a prediction and measure its goodness against both the training and testing data:

```
train_pred = rf.predict(train_features)
trainRSME = np.sqrt(mean_squared_error(train_target,
        train_pred))
trainR2 = r2_score(train_target, train_pred) * 100
print(f"Training RSME: {trainRSME}")
print(f"Training R2 Score: {trainR2}")
test_pred = rf.predict(test_features)
testRSME = np.sqrt(mean_squared_error(test_target,
        test_pred))
testR2 = r2_score(test_target, test_pred) * 100
print(f"Testing RSME: {testRSME}")
print(f"Testing R2 Score: {testR2}")
```

The output in *Figure 9.12* shows that this algorithm provides better performance than kNN, but the code shows that you need to do a little more work to get it:

```
Training RSME: 7.024166548977007
Training R2 Score: 91.3860155018798
Testing RSME: 8.893716635030863
Testing R2 Score: 85.75984790693091
```

Figure 9.12 – The statistics show that the random forest regressor doesn't perform quite as well as kNN in this case

4. Create a plot showing how the model actually works against the data:

```
plt.scatter(x, test_target, color='blue',
    label='Original')
plt.plot(x, test_pred, color='red',
    label='Prediction')
plt.legend()
plt.show()
```

Figure 9.13 shows the results in this case:

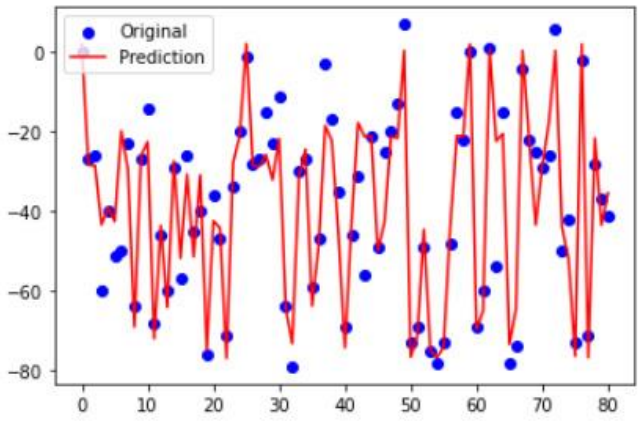

Figure 9.13 – The output of the random forest regressor prediction

It's interesting to review the differences in how the various models interact with the data as part of a plot.

Building and testing an XGBoost regressor

The final algorithm to test against the behavior data is XGBoost, which is another ensemble of simple learners, but instead of using a voting setup to determine the result, the data from one model is fed into the next model to reduce the amount of error in the prediction. Overall, gradient boosting, the basis of XGBoost, is best for working with unbalanced data, as often happens with security scenarios. However, because of the large number of arguments that XGBoost provides (https://xgboost.readthedocs.io/en/stable/), it can prove incredibly difficult to tune. The following steps show how to work with XGBoost:

1. Install the XGBoost package as needed. The version of XGBoost used for this example is 1.6.2. If you use a different version, you may find that you need to adjust the code slightly and that you'll obtain slightly different results:

```
modules = !pip list
installed = False
for item in modules:
    if ('xgboost' in item):
        print('XGBoost installed: ', item)
        installed = True
if not installed:
    print('Installing XGBoost...')
    !pip install xgboost
```

2. Import the required package and fit the model. This example sets three of the arguments: `max_depth` to determine the depth of each tree, `n_estimators` to determine the number of trees, and `eta` to determine the learning rate. There are a great many other arguments that you can use to tune your model once you have got it to a basic level of competence:

```
from xgboost import XGBRegressor
xg = XGBRegressor(max_depth = 3, n_estimators=750,
    eta=0.3)
xg.fit(train_features, train_target)
```

The output of this step, shown in *Figure 9.14*, gives you some idea of the incredible number of tuning arguments supported by this algorithm:

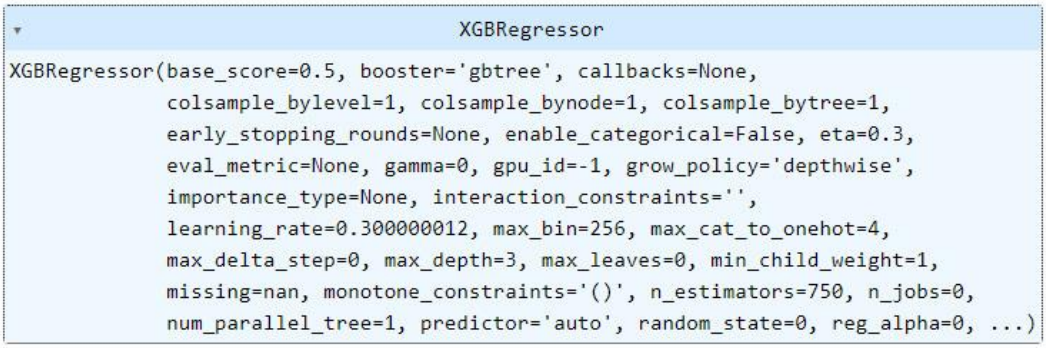

Figure 9.14 – The XGBRegressor output shows an incredible number of tuning arguments

3. Perform a prediction and measure its goodness against both the training and testing data:

```
train_pred = xg.predict(train_features)
trainRSME = np.sqrt(mean_squared_error(train_target,
```

```
        train_pred))
trainR2 = r2_score(train_target, train_pred) * 100
print(f"Training RSME: {trainRSME}")
print(f"Training R2 Score: {trainR2}")
test_pred = xg.predict(test_features)
testRSME = np.sqrt(mean_squared_error(test_target,
        test_pred))
testR2 = r2_score(test_target, test_pred) * 100
print(f"Testing RSME: {testRSME}")
print(f"Testing R2 Score: {testR2}")
```

The output, shown in *Figure 9.15*, demonstrates that this model provides the best performance of those tested for this example:

```
Training RSME: 1.2282842929226978
Training R2 Score: 99.73660211630327
Testing RSME: 8.604377627388082
Testing R2 Score: 86.67132506775764
```

Figure 9.15 – The XGBRegressor provides the best performance of all the models tested

4. Create a plot showing how the model actually works against the data:

```
plt.scatter(x, test_target, color='blue',
    label='Original')
plt.plot(x, test_pred, color='red',
    label='Prediction')
plt.legend()
plt.show()
```

Compare the plot shown in *Figure 9.16* with the one in *Figure 9.13* and you can see the smoothing effect of the gradient boosting process:

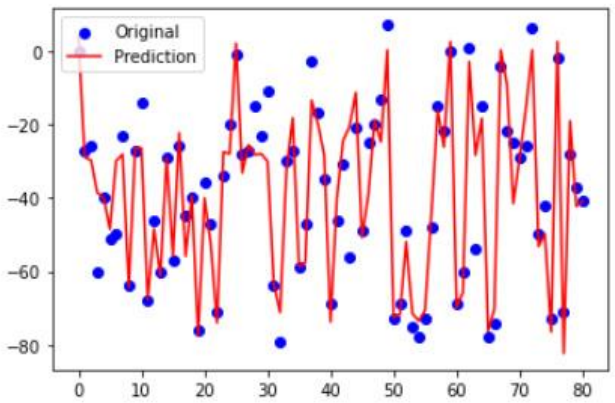

Figure 9.16 – The output of the XGBoost regressor prediction

Choosing an algorithm and then tuning it can require time and patience, but it's well worth the effort. The better the model's predictions, the greater the chance that the hacker will be caught.

Putting the data in perspective

The plots you've seen so far show the randomized data used for training and testing but don't really tell you much about how the model actually performs using the original dataset in a real-world way. What you really need is a plot showing real-world events, which rely on the following code:

```
plt.plot(covid_df['date'],
    covid_df['hospitalCases'].values,
    color='blue', label='Hospital Cases')
plt.plot(covid_df['date'],
    covid_df['newAdmissions'].values,
    color='green', label='New Admissions')
plt.plot(covid_df['date'],
    covid_df['newCasesByPublishDate'].values,
    color='orange', label='New Cases')
plt.xticks(rotation=90)
fig = plt.gcf()
fig.set_size_inches(10, 8)
plt.legend()
plt.show()
```

The output shown in *Figure 9.17* demonstrates that there is a correlation between the three kinds of data used as features for this example:

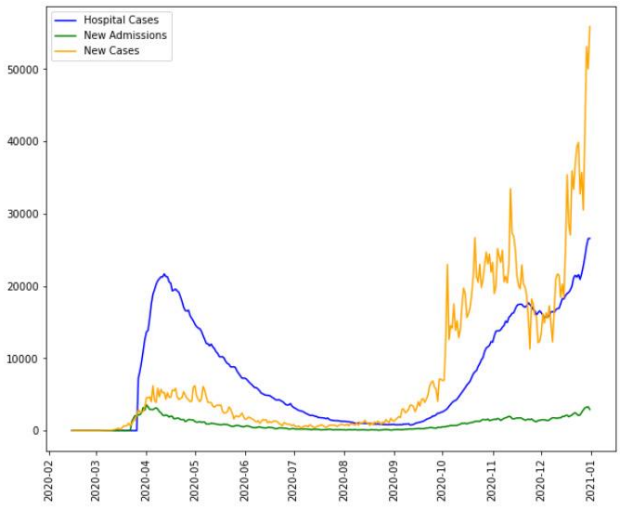

Figure 9.17 – The plot shows a correlation between the three data features

Because the XGBoost model produced the best results during testing, the example uses it to produce the prediction needed to test how the model works using the full dataset with the following code:

```
xg_full_pred = xg.predict(covid_df[[
        'hospitalCases', 'newAdmissions',
            'newCasesByPublishDate']].values)
plt.plot(covid_df['date'], xg_full_pred,
```

```
    color='red', label='Prediction')
plt.plot(covid_df['date'],
    mobility_df['retail_and_recreation'],
    color='blue', label='Actual Data')
plt.xticks(rotation=90)
plt.legend()
fig = plt.gcf()
fig.set_size_inches(10, 8)
plt.show()
```

The output shown in *Figure 9.18* demonstrates that as the number of COVID cases increases, the retail and recreation activities decrease (notice that the numbers are negative). However, also notice that there are lags. COVID increases first, then comes the decrease in activity:

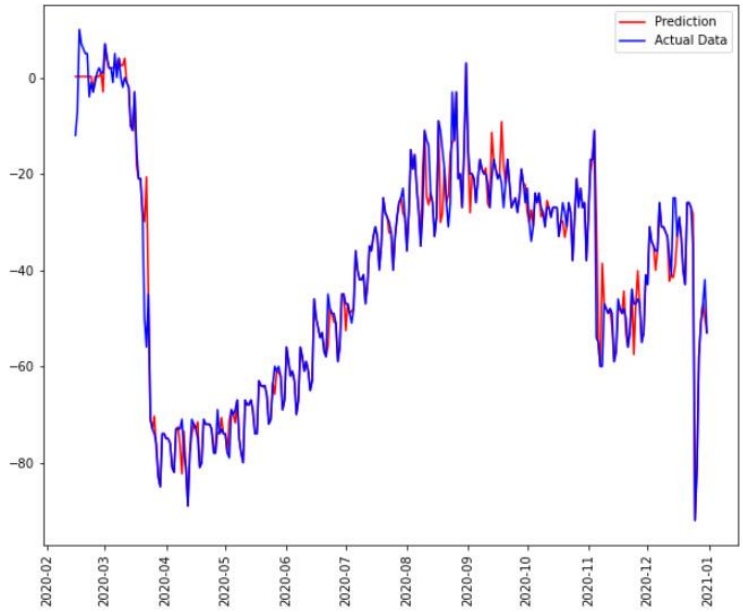

Figure 9.18 – The plot shows an inverse correlation between COVID and activity

What this example has done is demonstrated that you can model human behavior with the correct data. The problem is finding the correct data, which means becoming a detective and figuring the hackers out.

Predicting new behavior based on the past

It's fine that the model works on data it has already worked with during training and testing. However, working with the full dataset doesn't really fit the bill if you're going to predict future human behavior. You already know what hackers have done in the past, but what are they going to do tomorrow? Now the example uses the data previously saved from 2021 that the model hasn't seen. The following code shows COVID activity for the following year:

```
plt.plot(covid_df_2021['date'],
    covid_df_2021['hospitalCases'].values,
    color='blue', label='Hospital Cases')
plt.plot(covid_df_2021['date'],
    covid_df_2021['newAdmissions'].values,
    color='green', label='New Admissions')
```

```
plt.plot(covid_df_2021['date'],
    covid_df_2021['newCasesByPublishDate'].values,
    color='orange', label='New Cases')
plt.xticks(rotation=90)
plt.legend()
fig = plt.gcf()
fig.set_size_inches(10, 8)
plt.show()
```

The results shown in *Figure 9.19* show a lull in COVID activity before it increases again:

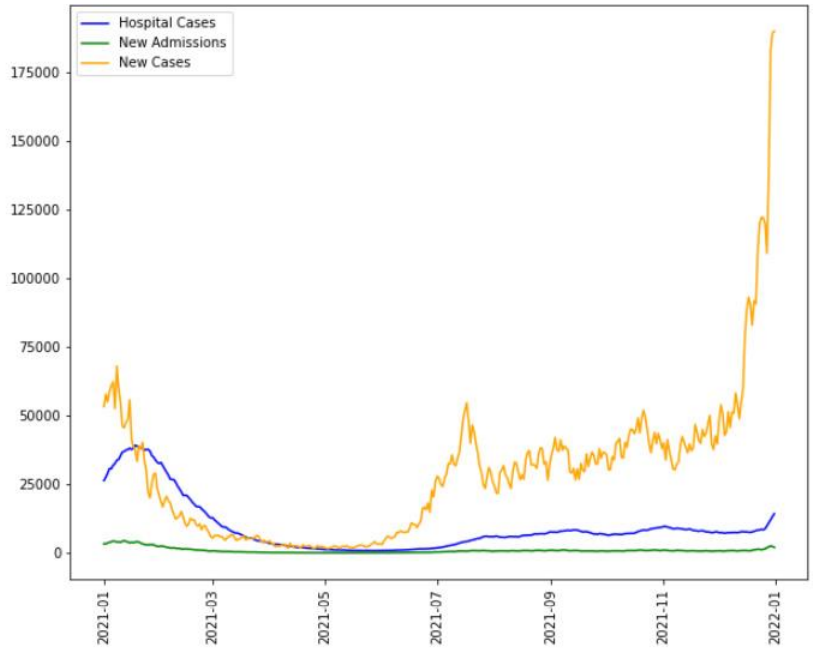

Figure 9.19 – A plot showing the 2021 COVID data

Before you can move forward, you need to obtain the actual retail and recreation data for 2021 using this code:

```
mobility_df = pd.read_csv(
    "Region_Mobility_Report_CSVs/" \
    "2021_GB_Region_Mobility_Report.csv")
mobility_df.drop(
    ['country_region_code', 'country_region',
        'sub_region_1', 'sub_region_2', 'metro_area',
        'iso_3166_2_code', 'census_fips_code',
        'place_id'], axis='columns', inplace=True)
mobility_df['date'] = pd.to_datetime(mobility_df['date'])
mobility_df.rename(columns={
       'retail_and_recreation_percent_change_from_baseline':
       'retail_and_recreation'},
    inplace=True)
```

This is essentially the same code used to obtain the 2020 data, but for 2021 instead. Now let's see what the predicted activity is like. However, in this case, you see the effects of using kNN, random

forest regressor, and XGBoost regressor because there is something interesting to see. The following code performs the prediction and presents the plot for you:

```
knn_2021_pred = knn.predict(covid_df_2021[[
    'hospitalCases', 'newAdmissions',
        'newCasesByPublishDate']].values)
rf_2021_pred = rf.predict(covid_df_2021[[
        'hospitalCases', 'newAdmissions',
            'newCasesByPublishDate']].values)
xg_2021_pred = xg.predict(covid_df_2021[[
        'hospitalCases', 'newAdmissions',
            'newCasesByPublishDate']].values)
plt.plot(covid_df_2021['date'],
    knn_2021_pred,
    color='red', label='kNN Prediction')
plt.plot(covid_df_2021['date'],
    rf_2021_pred,
    x
plt.plot(covid_df_2021['date'],
    xg_2021_pred,
    color='orange', label='XG Prediction')
plt.plot(covid_df_2021['date'],
    mobility_df.head(365)['retail_and_recreation'],
    color='blue', label='Actual Data')
plt.xticks(rotation=90)
plt.legend()
fig = plt.gcf()
fig.set_size_inches(10, 8)
plt.show()
```

Figure 9.20 shows that the correlation between predicted activity levels and COVID activity remains in the 2021 data. Now, this is data that the model hasn't seen at all, and there was little tuning done. In addition, a larger dataset would have made a difference (the smaller size was partially to keep computing costs low). So, the prediction follows the actual data reasonably closely. Notice that the random forest regressor provides a smoother prediction that follows the actual data a tiny bit closer than kNN. This is the effect of the ensemble and why you may want to choose it over other methods of making a prediction. The XGBoost regressor output is the best of the three but lacks the smoothness of the random forest regressor. With further tuning, either model would do a better job of predicting human behavior.

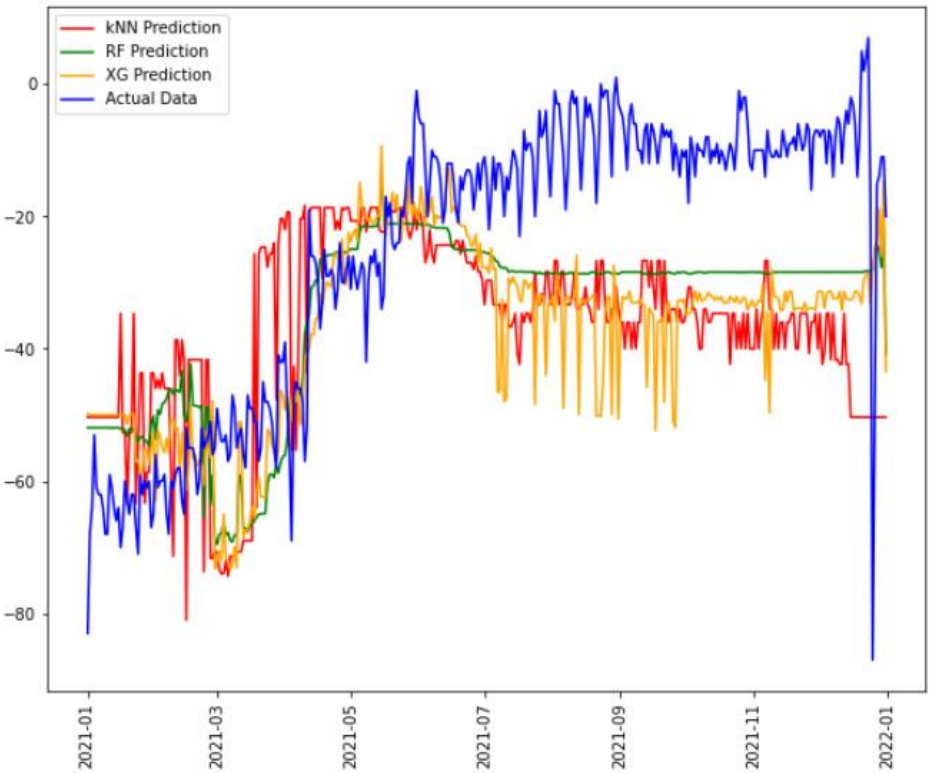

Figure 9.20 – The plot shows an inverse correlation in the 2021 data as well

What this all means is that it's possible to predict future human activity given knowledge of the past. However, the prediction is never completely accurate. Notice that in July 2021, the prediction diverges from the actual data because the prediction is modeled on what is happening with the COVID cases, while the real people in the real world were getting used to COVID and learning to live with it in favor of pursuing their normal activities. It still takes some knowledge on the part of the human to ensure that the data and predictive values are used correctly. The important takeaway from this example is that human behavior is predictable.

Using these prediction methods also work with security, but you need a lot more details to make it work (which is why this example took the direction that it did) Articles, such as *Attack Pattern Detection and Prediction* (https://towardsdatascience.com/attack-pattern-detection-and-prediction-108fc3d47f03), point out that prediction methods are now being used to anticipate hacker activity based on past attacks. It really isn't enough any longer to have a good defense in place; it's more important to know what could possibly happen in the future. Unfortunately, there isn't any sort of generalized prediction strategy that you can use now, much as you might use generalized defenses. Prediction is usually business specific.

Locating other behavioral datasets

This chapter shows you a single example of a behavioral dataset. You may actually need something different to perform some experiments. Unfortunately, there are no datasets currently available that actually contain hacker-specific information. It may be possible to generate such a dataset for your organization using the technique in the *Building a data generator* section of *Chapter 5*, *Keeping Your Network Clean*.

One good place to find behavioral datasets is on GitHub at https://github.com/topics/human-behavior. The site contains a number of datasets to choose from, including some non-text datasets that contain images. One of the more interesting datasets is `cent-patates`, which tells you how to improve your winnings from the French lottery. Mind you, I haven't actually tested the dataset or the theories of its author.

The Mendeley Data site at https://data.mendeley.com/research-data/?search=Human%20Behavior%20Dataset provides a huge number of datasets. The problem is wading through all of them to find the one you want. This is the place to look for unusual datasets that rely on images, videos, or even sound formats. Note that the same site also provides access to a number of research documents and dissertations on up-and-coming technologies that really aren't ready for use today but are still interesting to read about.

In some cases, you can find very specialized datasets with strings attached, such as the *Web-Hacking Dataset for the Cyber Criminal Profiling* at https://ocslab.hksecurity.net/Datasets/web-hacking-profiling, which requires you to jump through some hoops to get it. For one thing, you need to fill out a form with some information about yourself that isn't required by most sites. This dataset is also only useful for research purposes, as stated by the documentation that comes with it. Given that the data is from over a period of 15 years, it also likely includes outdated techniques. However, it is one of the few datasets that could act as a jumping-off point for working with hacker data.

Links

Hackers aren't smart – people are stupid (https://blog.erratasec.com/2016/02/hackers-arent-smart-people-are-stupid.html)

7 Ways Smart People Get Hacked by Social Engineering (https://www.delcor.com/resources/blog/7-ways-smart-people-get-hacked-by-social-engineering)

Optus data breach: Is your business prepared for a cyberattack? (https://retrac.com.au/optus-data-breach-is-your-business-prepared-for-a-cyberattack/)

Why do hackers want your personal information? (https://www.f-secure.com/us-en/home/articles/why-do-hackers-want-your-personal-information)

How hackers use AI and machine learning to target enterprises (https://www.techtarget.com/searchsecurity/tip/How-hackers-use-AI-and-machine-learning-to-target-enterprises)

Cyber Wars: How The U.S. Stock Market Could Get Hacked: https://www.investopedia.com/news/are-your-stocks-danger-getting-hacked/

How to Hack AI? (https://broutonlab.com/blog/how-to-hack-ai-machine-learning-vulnerabilities)

Individuals may legitimize hacking when angry with system or authority at https://www.sciencedaily.com/releases/2020/10/201022125522.htm

Driving test reaction: https://www.justpark.com/creative/reaction-time-test/

GOV.UK Coronavirus (COVID-19) UK site at https://coronavirus.data.gov.uk/details/download

Chapter 10

Technical requirements

This chapter requires that you have access to either Google Colab or Jupyter Notebook to work with the example code. The requirements to use this book is to use the section of *Chapter 1*, *Defining Machine Learning Security*, which provides additional details on how to set up and configure your programming environment.

You really do benefit from having a **graphics processing unit** (**GPU**) to run the examples in this chapter. They will run without a GPU but expect to take long coffee breaks while you wait for the code to complete running. This means choosing **Runtime** | **Change Runtime Type** in Google Colab, then selecting **GPU** in the **Hardware Accelerator** dropdown. Desktop users will want to review the *Checking for a GPU with a nod toward Windows* section of the chapter for desktop system instructions.

Set up your system to run TensorFlow. Google Colab users should read https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/beginner.ipynb. The *Installing TensorFlow* section of this chapter provides details on how to perform the software installation on desktop systems, where you'll generally get far superior performance than when using Google Colab. Your system must meet the following minimal requirements:

- Operating system:
- Ubuntu 16.04 or higher (64-bit)
- macOS 10.12.6 (Sierra) or higher (64-bit) (no GPU support)
- Windows Native – Windows 7 or higher (64-bit)
- Windows WSL2 – Windows 10 19044 or higher (64-bit)
- Python 3.7 through Python 3.10
- Pip version 19.0 or higher, or 20.3 or higher on macOS
- Visual Studio 2015, 2017, or 2019 on Windows systems
- NVidia GPU Support (optional):
- GPU drivers version 450.80.02 or higher
- **Compute Unified Device Architecture** (**CUDA**) Toolkit version 11.2
- cuDNN SDK version 8.1.0
- TensorRT (optional)

When testing the code, use a test site, test data, and test APIs to avoid damaging production setups and to improve the reliability of the testing process. Testing over a non-production network is highly recommended. Using the downloadable source is always highly recommended. You can find the downloadable source on the Packt GitHub site at https://github.com/PacktPublishing/Machine-

Learning-Security-Principles or on my website at http://www.johnmuellerbooks.com/source-code/.

Figures:

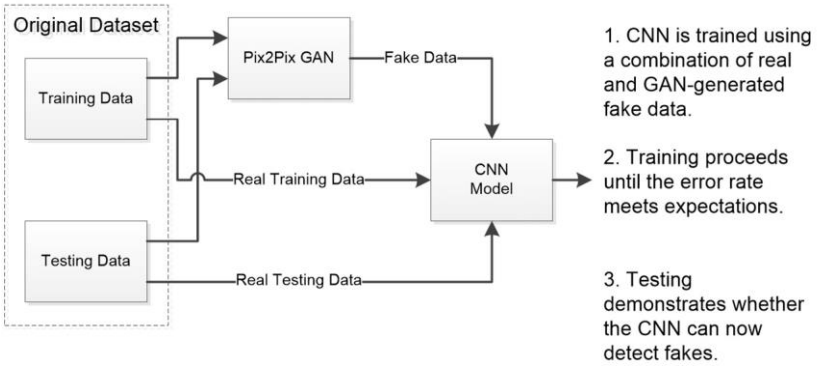

Figure 10.18 – A Pix2Pix GAN use to supplement model training

Hands-On Sections

Creating a deepfake computer setup

Creating a deepfake requires building serious models, using specialized software on systems that have more than a little computing horsepower. The system used for testing and in the screenshots for this chapter is more modest. It has an Intel i7 processor, 24 GB of RAM, and an NVidia GeForce GTX 1660 Super GPU. This system is used to ensure that the examples will run in a reasonable amount of time, with *reasonable* being defined as building a model in about half an hour or less. The example as a whole will require more time, likely in the hour range. The following sections will help you install a TensorFlow setup that you can use for autoencoder and GAN development without too many problems, and help you test your setup to ensure it actually works.

Installing TensorFlow on a desktop system

Desktop developers may already have TensorFlow installed, but if you're not sure then you likely don't. The technique for creating the advanced models in this chapter relies on using TensorFlow. The basic reason for going with this route is that the development process is easier and you can create a relatively simple example sooner so that you can see how such a model would work. In order to use TensorFlow, you must install the required support. You can verify that you have the required support installed by opening an `Anaconda` prompt and typing the following:

```
conda list tensorflow
```

Alternatively, you can enter the following code:

```
pip show tensorflow
```

If you have TensorFlow installed, it will show up as one of the installed packages on your system. This section assumes that you have the Conda utility available on your desktop system. If you installed the Anaconda suite on your machine, then you have Conda available by default. Otherwise, you need to install Miniconda using the instructions at

https://docs.conda.io/en/latest/miniconda.html for your platform. If your system is too old to support the current version of Miniconda, then you can't run the examples in this chapter.

In addition to ensuring you have access to the Conda utility on Windows systems, you must also have Visual Studio 2015, Visual Studio 2017, or Visual Studio 2019 installed before you do anything else. You can obtain a free copy of Visual Studio 2019 Community Edition at https://learn.microsoft.com/en-us/visualstudio/releases/2019/release-notes.

Once you know you have a version of the `Conda` utility available, you can use these steps to set up the prerequisite tools for the examples in this chapter. The following steps will work for most platforms. However, if you encounter problems, you can also find a set of steps at https://www.tensorflow.org/install/pip. Select your platform at the top of the instruction list:

1. Type `conda install nb_conda_kernels` and press *Enter*. Type `y` and press *Enter* when asked.
2. Type `conda create --name tf python=3.9` and press *Enter* to create a clean environment for your `TensorFlow` installation. Type `y` and press *Enter* when asked.
3. Type `conda activate tf` and press *Enter*.

Deactivation

Type `conda deactivate tf` and press *Enter* to deactivate the TensorFlow environment when you no longer need to keep the environment running.

4. Type `conda install ipykernel` and press *Enter*. Type `y` and press *Enter* when asked.
5. Download and install the NVIDIA driver for your platform and GPU type from https://www.nvidia.com/Download/index.aspx.
6. Test the installation by typing `nvidia-smi` and pressing *Enter*. You should see a listing of device specifics, along with the processes that are currently using the GPU.
7. Type `conda install -c conda-forge cudatoolkit=11.2 cudnn=8.1.0` and press *Enter* to install the CUDA toolkit and cuDNN SDK. Type `y` and press *Enter* when asked. Note that this step can take a while to complete.
8. Depending on your platform and how you have the CUDA toolkit installed, you need to add a path statement to your platform so that it can find the CUDA toolkit. The paths commonly needed are as follows:

```
NVIDIA GPU Computing Toolkit\CUDA\v11.3\bin
NVidia GPU Computing Toolkit\CUDA\v11.3\lib\x64
NVidia GPU Computing Toolkit\CUDA\v11.3\include
```

9. Type `pip install --upgrade pip` and press *Enter* to verify that you have the latest version of pip installed.

10. Type `pip install tensorflow` and press *Enter*. Type `y` and press *Enter* when asked.
11. Type `ipython kernel install --user --name=tf` and press *Enter*. This step will make the environment appear in the menus so that you can access it.

Now that you have TensorFlow installed, you can verify that your installation will work correctly using the instructions in the next section. These instructions ensure that you can access your GPU and that your GPU is of the correct type.

Checking for a GPU

Training a model with a GPU might only require half an hour, but training it without one could easily cost you six or more hours. You want to make sure that your GPU is actually working, accessible, and of the right type before you start working with the example code. Otherwise, you have long waits and sometimes inexplicable errors to deal with when the code itself is fine.

The `MLSec; 10; Check for GPU Support.ipynb` file contains source code and some detailed instructions for configuring and checking your GPU setup on Windows. The source code works equally well for Linux and Windows systems, so Linux users can test their setup too. When using the downloadable source, you must set the kernel to use your `tf` environment by choosing **Kernel | Change Kernel | Python [conda env:tf]**. Otherwise, even if you have a successful TensorFlow installation, Jupyter Notebook won't use it. The following steps test the setup from scratch:

12. Create a new Python file by choosing the `tf` environment option from the **New** drop-down list in place of the usual Python 3 (ipykernel) option, as shown in *Figure 10.1*.

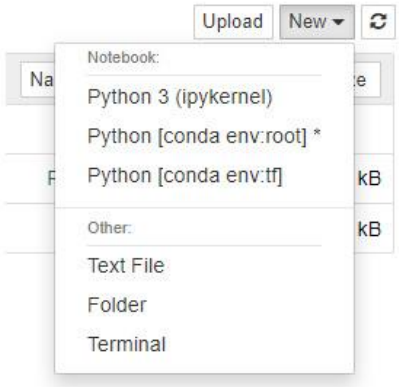

Figure 10.1 – The menu for selecting which environment to use

13. Type `!conda env list` in the first cell. Click **Run**. You will see output similar to *Figure 10.2*. The output will vary by your platform and Conda setup. Notice that the `tf` environment has an asterisk next to it, indicating that it's the one in use.

```
# conda environments:
#
base                     C:\Users\John\anaconda3
tf                    *  C:\Users\John\anaconda3\envs\tf
```

Figure 10.2 – A list of the available Conda environments

14. Type the following code to determine the selected environment using the operating system environment variables instead. If the method in *step 2* doesn't work, this one always does, but it provides less information:

```
import os

print (os.environ['CONDA_DEFAULT_ENV'])
```

The output of `tf` shows that the `tf` environment is selected as the Conda default environment for scripts you create and run in Jupyter Notebook.

15. Type the following code to obtain a list of computing devices on your system, which will include your GPU if TensorFlow is correctly installed:

```
from tensorflow.python.client import device_lib

device_lib.list_local_devices()
```

The output shown in *Figure 10.3* is representative of the output you will see when you run this cell. However, the specifics will vary by GPU and your system may actually have multiple GPUs installed, which would mean that the list would include all of them.

```
[name: "/device:CPU:0"
 device_type: "CPU"
 memory_limit: 268435456
 locality {
 }
 incarnation: 5122621535799984449
 xla_global_id: -1,
 name: "/device:GPU:0"
 device_type: "GPU"
 memory_limit: 4187553792
 locality {
   bus_id: 1
   links {
   }
 }
 incarnation: 13742849836653392168
 physical_device_desc: "device: 0, name: NVIDIA GeForce GTX 1660 SUPER, pci bus id: 0000:01:00.0, com
pute capability: 7.5"
 xla_global_id: 416903419]
```

Figure 10.3 – A list of the processing devices on the local system

16. Type the following code to verify that the GPU is recognized as a GPU:

```
import tensorflow as tf

tf.config.list_physical_devices('GPU')
```

The output shown in *Figure 10.4* is typical. It tells you the device name and the device type. Only a GPU with the correct hardware support will show up. Consequently, if you don't see your GPU, it's too old to use with TensorFlow or it lacks the appropriate drivers.

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Figure 10.4 – A description of a GPU on the local system

17. Verify that the GPU will actually perform the required math when interacting with TensorFlow using the following code. Note that this requires the creation of tensors using `tf.constant()`:

```
tf.debugging.set_log_device_placement(True)
a = tf.constant([[1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0],
    [3.0, 4.0],
    [5.0, 6.0]])
c = tf.matmul(a, b)
print(c)
```

The output shown in *Figure 10.5* is precisely what you should see. This code creates two matrixes that it then multiplies together using `matmul()`.

```
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:GPU:0
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:GPU:0
Executing op MatMul in device /job:localhost/replica:0/task:0/device:GPU:0
tf.Tensor(
[[22. 28.]
 [49. 64.]], shape=(2, 2), dtype=float32)
```

Figure 10.5 – The result of matrix multiplication using TensorFlow

At this point, you know that your setup will work with the code in the chapter, so you can proceed to the next section about autoencoders with confidence. When creating the examples in this chapter, always remember to use the `tf` environment to start a new code file or modify the kernel using the **Kernel** | **Change Kernel** menu command.

Now that you have a deepfake setup to use, it's time to employ it by creating an example. The autoencoder is the simplest of the deepfake technologies to understand, so we will cover it in the next section.

Working with an autoencoder example

This section looks at a simple autoencoder that has multiple layers, but this autoencoder simply compresses and then decompresses a series of images. The idea is that you will see that the compression and decompression process is learned by a neural network, rather than hardwired like a **coder-decoder** (**CODEC**) is. *Figure 10.6* gives you an idea of what this autoencoder looks like graphically.

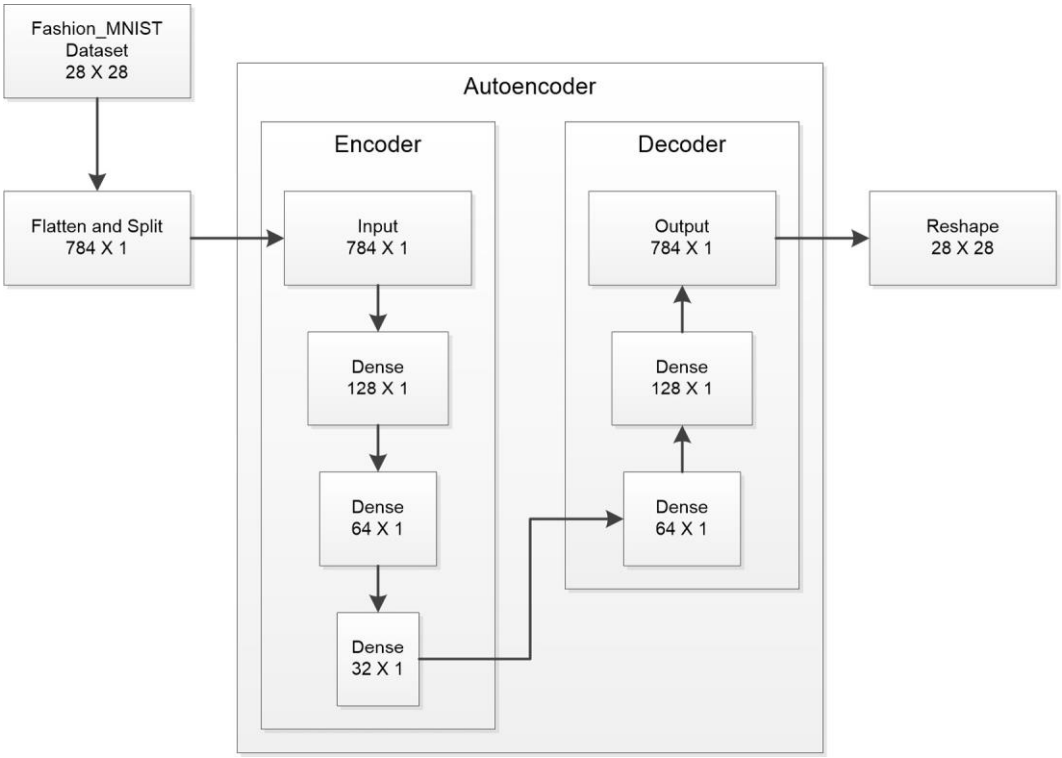

Figure 10.6 – A block diagram of the example autoencoder

The autoencoder consists of two separate programming entities, the encoder and the decoder. Later, you will see in the code that the loss function is included as part of the autoencoder. You can find the source code for this example in the `MLSec; 10; Autoencoder.ipynb` file in the downloadable source.

Now that you have some idea of what the code will do for this example, it's time to look at the various pieces shown in *Figure 10.6*. The following sections detail the items in the block diagram so you can see how they work in Python.

Obtain the Fashion-MNIST dataset

This example works with a number of different image datasets. However, in this case, you see it used with the common `Fashion-MNIST` dataset because it offers a level of detail that some other datasets don't provide, and it works well in shades of gray. The deepfake aspect of this particular dataset is that you can use it to see how images react to compression and decompression using a trained model, rather than other process. What you need to think about is what sorts of modifications could be made at each layer to make the image look like something it isn't. The following steps show how to obtain and configure the dataset in the example:

1. Import the required packages:

```
from tensorflow.keras.datasets import fashion_mnist

import matplotlib.pyplot as plt
```

2. Divide the dataset into training and testing datasets:

```
(x_train, _), (x_test, _) = fashion_mnist.load_data()
```

```
x_train = x_train.astype('float32') / 255.
X_test = x_test.astype('float32') / 255.
Print (x_train.shape)
print (x_test.shape)
```

When you perform this step, you will see the output shown in *Figure 10.7*. Unlike other splitting methods shown so far in the book, this one is automatic. However, you could also split it manually if desired. Notice that the shape shows the size of each image is 28x28 pixels.

```
(60000, 28, 28)
(10000, 28, 28)
```

Figure 10.7 – The output showing the split between training and testing datasets

3. Create the `showFigures()` function to provide a means of displaying any number of the dataset images:

```
def showFigures(dataset, n_items=10, title="Test"):
    fig, axs = plt.subplots(1, n_items,
        constrained_layout=True)
    fig.suptitle(title, y=0.65, fontsize=16,)
    for i in range(n_items):
        plt.gray()
        axs[i].imshow(dataset[i])
        axs[i].get_xaxis().set_visible(False)
        axs[i].get_yaxis().set_visible(False)
    plt.show()
```

4. Test the `showFigures()` function:

```
showFigures(x_train, title="Training Data")
```

The output shown in *Figure 10.8* demonstrates that this is a good dataset to choose for looking at the level of detail. In addition, the images do look good in shades of gray.

Figure 10.8 – Ten of the images from the training dataset

Now that you have a dataset to use, it's time to start working with it using an autoencoder. The next section addresses the encoder part of the autoencoder.

Build an encoder

The first part of an autoencoder is the encoder. This is the code that does something to compress or otherwise manipulate the data as part of the input stream. This example exclusively uses the `Dense` layers, but there are a great many more layer types listed at https://keras.io/api/layers/ that perform tasks other than simple compression. For example, instead of flattening the image outside of the model, you could use the `Flatten` layer to flatten it inside the model. The following steps show how to create the encoder for this example:

1. Import the required packages:

```
from tensorflow.keras import layers
from tensorflow import keras
import tensorflow as tf
import numpy as np
```

2. Reshape the training and testing datasets for use with the model:

```
x_train = x_train.reshape((len(x_train),
    np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test),
    np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

The output from this step, shown in *Figure 10.9*, indicates that each image is now a 784-bit vector, rather than a 28x28-bit matrix.

```
(60000, 784)
(10000, 784)
```

Figure 10.9 – Each image is now a 784-bit vector

3. Create the encoder starting with an `Input` layer and then adding three `Dense` layers to progressively compress the image. All three of the `Dense` layers use the **rectified linear activation unit** (**ReLU**) activation function, which outputs the value directly when it is positive, otherwise, it outputs a value of `0` (https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/).

```
encoder_inputs = keras.Input(shape=(784,))
encoded = layers.Dense(
    128, activation="relu")(encoder_inputs)
encoded = layers.Dense(
    64, activation="relu")(encoded)
encoded = layers.Dense(
```

```
    32, activation="relu")(encoded)
```

Notice that the code describes the shape of each layer and the activation function. A `Dense` layer offers a lot more in the way of configuration options, as explained at https://keras.io/api/layers/core_layers/dense/. The value in parenthesis after the call to `Dense()` is a linking value. It links the new layer to the previous layers. So, the first `Dense` layer is linked to `encoder_inputs`, while the second `Dense` layer is linked to the first `Dense` layer. The next step is to build the decoder, which of course links to the encoder.

Build a decoder

The decoder in this example simply consists of more `Dense` layers. However, instead of making each succeeding layer denser, it makes each layer less dense, as shown in the following code:

```
decoded = layers.Dense(64, activation="relu")(encoded)
decoded = layers.Dense(128, activation="relu")(decoded)
decoded = layers.Dense(784, activation="sigmoid")(decoded)
```

The decoder follows the same pattern as the encoder, but the layers go up in size, as shown in *Figure 10.6*. Notice that the first `Dense` layer is linked to the final `encoded` layer. The final `Dense` layer uses a sigmoid activation function, which has a tendency to smooth the output (https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/) and ensures that the output value is always between `0` and `1`. The encoder and decoder are essentially separate right now as shown in *Figure 10.6*, so it's time to put them into the autoencoder box so that they can work together as described in the next section.

Build the autoencoder

What this section is really about is putting things into boxes, starting with the autoencoder box. These boxes are models that are used to allow the autoencoder to perform useful work. You'll see that they work together to create a specific type of neural network that can manipulate data in various ways, depending on the layers you use. The following steps take you through the process of building the autoencoder, which involves creating several models:

1. Create the autoencoder model and display its structure:

```
autoencoder = keras.Model(encoder_inputs, decoded)

autoencoder.summary()
```

As shown in *Figure 10.10*, the autoencoder model incorporates all of the layers of the encoder and decoder we constructed in earlier sections.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 784)]             0

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 64)                2112

 dense_4 (Dense)             (None, 128)               8320

 dense_5 (Dense)             (None, 784)               101136

=================================================================
Total params: 222,384
Trainable params: 222,384
Non-trainable params: 0
_________________________________________________________________
```

Figure 10.10 – The structure of the autoencoder as a whole

2. Compile the autoencoder, which includes adding a loss function and an optimizer so that the autoencoder works efficiently:

```
autoencoder.compile(optimizer='adam',
    loss='binary_crossentropy')
```

3. Create the encoder model and display its structure:

```
encoder = keras.Model(encoder_inputs,
    encoded, name="encoder")
print(encoder.summary())
```

The output shown in *Figure 10.11* demonstrates that this model is the first half of the autoencoder.

```
Model: "encoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 784)]             0

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 32)                2080

=================================================================
Total params: 110,816
Trainable params: 110,816
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 10.11 – The structure of the encoder model

4. Create the decoder model and display its structure:

```
encoded_input = keras.Input(shape=(32,))
decoder_layer_1 = \
    autoencoder.layers[-3](encoded_input)
decoder_layer_2 = \
    autoencoder.layers[-2](decoder_layer_1)
decoder_layer_3 = \
    autoencoder.layers[-1](decoder_layer_2)
decoder = keras.Model(encoded_input, decoder_layer_3,
    name="decoder")
print(decoder.summary())
```

This code requires a little more explanation than the encoder model. First, the `encoded_input` has a shape of `32,` now because it's compressed. Second, each of the decoded layers of the model comes from the compiled `autoencoder`, which is why they're referred to as `autoencoder.layers[-3]`, `autoencoder.layers[-2]`, and `autoencoder.layers[-1]`. If you count up the autoencoder model layers shown in *Figure 10.10*, you will see that this arrangement basically begins with the first level of the decoder and works down from there. As with building the decoder, you must also connect the layers together as shown. *Figure 10.12* shows the structure of the decoder model.

```
Model: "decoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 32)]              0

 dense_3 (Dense)             (None, 64)                2112

 dense_4 (Dense)             (None, 128)               8320

 dense_5 (Dense)             (None, 784)               101136

=================================================================
Total params: 111,568
Trainable params: 111,568
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 10.12 – The structure of the decoder model

It's time to create and train the model as a whole by fitting it to the data. This process tracks the learning curve of the neural network so you can see how it works.

Create and train a model from the encoder and decoder

You still have to fit the autoencoder to the data. Part of this process is optional. The following example uses TensorBoard (https://www.tensorflow.org/tensorboard) to track how the learning process goes and to provide other information covered in the next section:

5. Import the required packages:

```
from keras.callbacks import TensorBoard
```

6. Use magics to load TensorBoard into memory. It isn't included by default:

```
%load_ext tensorboard
```

7. Fit the autoencoder model to the data:

```
autoencoder.fit(
    x_train, x_train, epochs=50, batch_size=256,
    shuffle=True, validation_data=(x_test, x_test),
    callbacks=[TensorBoard(log_dir='autoencoder')])
```

Defining the number of `epochs` determines how long to train the model. `batch_size` determines how many of the samples are used for training during any given epoch. Setting `shuffle` to `True` means that the dataset is constantly shuffled to promote better training. Unlike other models that you've worked with, this one automatically validates the data against the test set specified by `validation_data`. Finally, the `callbacks` argument tells the `fit()` function to send learning data to TensorBoard and also tells TensorBoard where to store the data. When you run this cell, you will see the epoch data is similar to that shown in *Figure 10.13* (shortened for the book).

```
Epoch 1/50
235/235 [==============================] - 4s 10ms/step - loss: 0.3793 - val_loss: 0.3205
Epoch 2/50
235/235 [==============================] - 2s 9ms/step - loss: 0.3110 - val_loss: 0.3073
Epoch 3/50
235/235 [==============================] - 2s 9ms/step - loss: 0.3021 - val_loss: 0.3014
```

Figure 10.13 – Epoch data for each epoch of model training

At this point, you can start to see how the model learned by examining the statistics shown in the next section.

Obtaining and graphics model statistics

The process for viewing the statistics is relatively easy. All you need to do is start TensorBoard using the following code:

```
%tensorboard --logdir 'autoencoder'
```

The output is a relatively complex-looking display containing all sorts of interesting graphs, as shown in *Figure 10.14*. To get the full benefit from them, consult the guide at https://www.tensorflow.org/tensorboard/get_started.

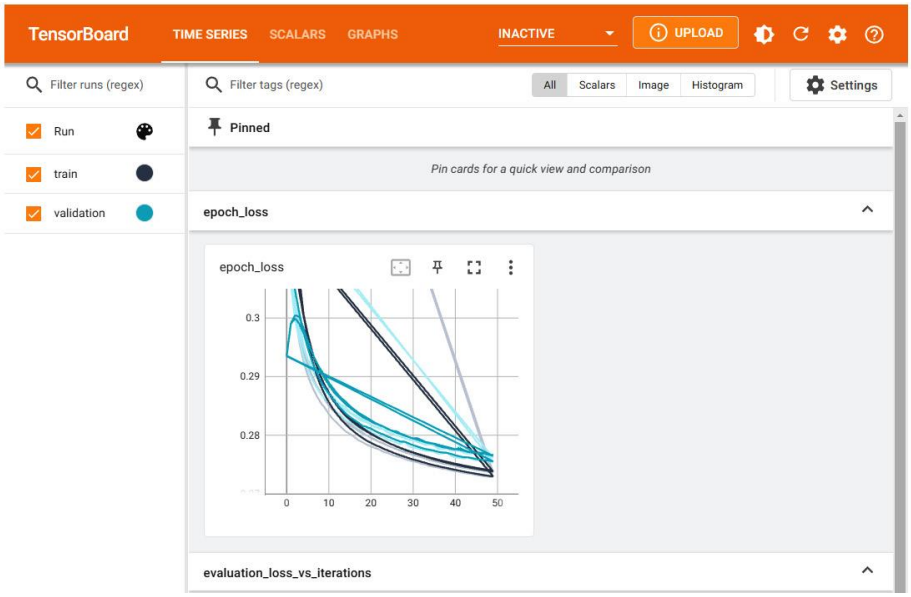

Figure 10.14 – One of many TensorBoard statistical outputs

In this case, what you see is the effect of the model steadily learning. The loss becomes less during each epoch so overall it ends up being quite small. Hovering your mouse over the graph shows data point information with precise values.

TensorBoard! Why won't you just die?

This particular note is designed to save you from pulling out your hair given the advice that you'll likely receive online if you're using a Windows system. Restarting Windows won't do anything for you, so don't waste your time. Windows doesn't support the `kill` command, so adding `!kill` <*PID*> to your code isn't going to produce anything but an error message if you try it after seeing the `Reusing TensorBoard on port 6006...` message. You may get lucky and find that using the `tasklist` command (https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/tasklist) will display the `TensorBoard.exe` application, but it's unlikely in many cases because it isn't actually running. If you do see `TaskBoard.exe`, you can use the `taskkill` command to stop it. However, what you'll end up doing in most cases is to halt and close your Notebook, then stop the Jupyter Notebook server as you normally do at the end of a session. To fix the problem, start by deleting the TensorBoard log files that were created during the fitting process. Next, locate the `\Users\<Username>\AppData\Local\Temp\.tensorboard-info` directory on your system and delete it. At this point, you can restart Jupyter Notebook and go on your merry way, hair intact.

Testing the model

This section answers the question of whether the model will compress and decompress data with minimal loss. The following steps show you how to test this:

1. Perform the required prediction:

```
encoded_imgs = encoder.predict(x_test)
```

```
decoded_imgs = decoder.predict(encoded_imgs)
```

The prediction process isn't a single step in this case. So, what you see is the output shown in Figure 10.15, which helps you track the prediction process.

```
313/313 [==============================] - 1s 2ms/step
313/313 [==============================] - 1s 2ms/step
```

Figure 10.15 – Tracking the prediction process

2. Display the input and output figures:

```
showFigures(x_test.reshape(10000,28,28),
    title="Original Testing Data")
showFigures(decoded_imgs.reshape(10000,28,28),
    title="Modified Testing Data")
```

Part of this process reshapes the data so that you can display it on screen. The output shows ten of the images for side-by-side comparison, as shown in *Figure 10.16*.

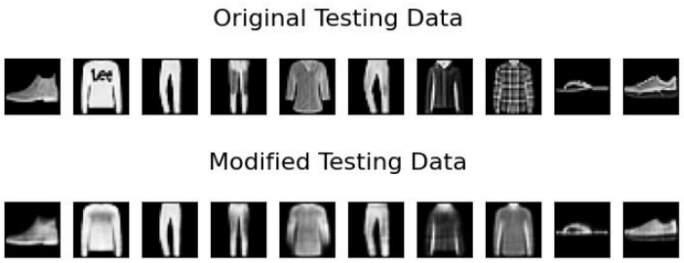

Figure 10.16 – A side-by-side comparison of input to output

The loss of detail should tell you something about the potential security issues with autoencoders. Because any data manipulation you perform is likely to cause some type of degradation, it pays to choose your models and configuration carefully. Otherwise, it becomes very hard to determine whether a particular issue is the result of hacker activity or simply due to a bad model.

Seeing the effect of bad data

At the outset of this example, you discovered that autoencoders learn how to transform specific data. That is, if you feed the autoencoder what amounts to bad data, even if that data isn't from a hacker, then the results are going to be less useful. This section puts that theory to the test using the following steps. What is important to note is that the model isn't trained again; you're using the same model as before to simulate the introduction of unwanted data:

1. Import the required packages:

```
from tensorflow.keras.datasets import mnist
```

2. Split the data into training and testing sets:

```
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
```

```
x_test = x_test.astype('float32') / 255.
print (x_train.shape)
print (x_test.shape)
```

3. Reshape the training and testing data:

```
x_train = x_train.reshape((
    len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((
    len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

4. Perform a prediction:

```
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

5. Compare the input and output:

```
showFigures(x_test.reshape(10000,28,28),
    title="Original Testing Data")
showFigures(decoded_imgs.reshape(10000,28,28),
    title="Modified Testing Data")
```

Figure 10.17 shows the results of the comparison. The results, needless to say, are disappointing.

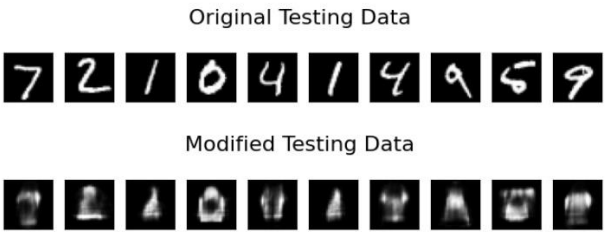

Figure 10.17 – A side-by-side comparison of using the model with the wrong data

Remember that this is a basic example where you're in full control of everything that happens. From a security perspective, you need to consider what would happen if a hacker fed your autoencoder bad data without you knowing it. Suddenly, you might start seeing unexpected results and may not be able to track them down very easily.

Implementing a Pix2Pix GAN

Finding a Pix2Pix GAN example that's specific enough for use in a security setup is hard, which is why this section goes into more detail about creating one.

Obtaining and viewing the images

This process begins by obtaining a dataset containing the images needed to train the GAN from https://www.kaggle.com/datasets/balraj98/facades-dataset. Once you download the dataset, unarchive it in the `facades` subdirectory of the example code. The following steps will get you started on manipulating the images:

1. Import the required packages:

```
import tensorflow as tf
from matplotlib import pyplot as plt
```

2. Define the image size and location:

```
IMG_WIDTH = 256
IMG_HEIGHT = 256
PATH = 'facades/'
```

3. Create a function for displaying the images:

```
def load(image_file):
    raw_image = tf.io.read_file(image_file)
    decode_image = tf.image.decode_jpeg(raw_image)
    image = tf.cast(decode_image, tf.float32)
    return image
```

4. Separate two images from the rest:

```
real_image = load(PATH+'trainA/40_A.jpg')
input_image = load(PATH+'trainB/40_B.jpg')
```

5. View the images:

```
fig, axes = plt.subplots(nrows=1, ncols=2)
axes[0].imshow(input_image/255.0)
axes[0].set_title("Input Image")
axes[1].imshow(real_image/255.0)
axes[1].set_title("Real Image")
```

Figure 10.20 shows the **Input image** (semantic labels) on the left and the **Real image** (ground truth) on the right.

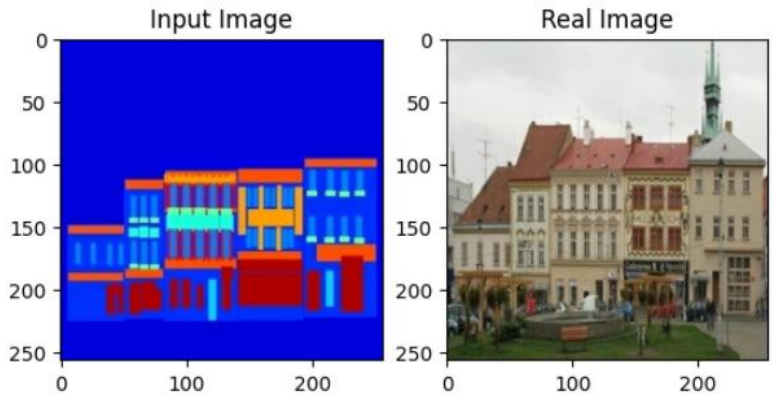

Figure 10.20 – Input image (semantic labels) and real image (ground truth) used as GAN input

Notice how the input image mimics the real image, using colors to label the input image so that the generator has an easier time creating a believable output. The colors relate to features in the real image. There is no actual guide on which colors to use, the colors simply serve to delineate various features. Usually, a human hand creates the input image using techniques such as those discussed at https://ml4a.github.io/guides/Pix2Pix/. Distortions in the input image will modify the output, as you see later in the example. A hacker could contaminate the input image database in a manner that modifies the output in specific ways that are to the hacker's advantage. The next section tells you about the image manipulation requirements.

Manipulating the images

The images aren't very useful in their original form, so it's important to know how to make the images appear in the way that you need them to appear, without losing any data. The following steps show you how to achieve this:

1. Create a function to resize the images to 286x286x3 to allow for random cropping, resulting in a final size of 256x256x3:

```
def resize(input_image, real_image, height, width):
    input_image = tf.image.resize(
        input_image, [height, width],
        method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    real_image = tf.image.resize(
        real_image, [height, width],
        method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    return input_image, real_image
```

2. Create a function to crop the images to a 256x256x3 size in a random manner:

```
@tf.autograph.experimental.do_not_convert
def random_crop(input_image, real_image):
    stacked_image = tf.stack(
```

```
        [input_image, real_image], axis=0)
    cropped_image = tf.image.random_crop(
        stacked_image,
        size=[2, IMG_HEIGHT, IMG_WIDTH, 3])
    return cropped_image[0], cropped_image[1]
```

3. Define a function to add jitter to the image:

```
@tf.function()
def random_jitter(input_image, real_image):
    input_image, real_image = resize(
        input_image, real_image, 286, 286)
    input_image, real_image = random_crop(
        input_image, real_image)
    if tf.random.uniform(()) > 0.5:
        input_image = \
            tf.image.flip_left_right(input_image)
        real_image = \
            tf.image.flip_left_right(real_image)
    return input_image, real_image
```

4. Create a plot to show four image pairs consisting of an input image and a real image:

```
fig, axes = plt.subplots(nrows=2, ncols=4,
    figsize=(12, 6))
fig.tight_layout(pad=2)
for i in range(2):
    for j in range(0, 4):
        if j%2 == 0:
        changed_input_image, changed_real_image = \
        random_jitter(input_image, real_image)
        axes[i, j].imshow(changed_input_image/255.0)
        axes[i, j].set_title("Input Image")
        axes[i, j +1].imshow(changed_real_image/255.0)
        axes[i, j + 1].set_title("Real Image")
```

Figure 10.21 shows the typical output at this step, with the various modifications labeled.

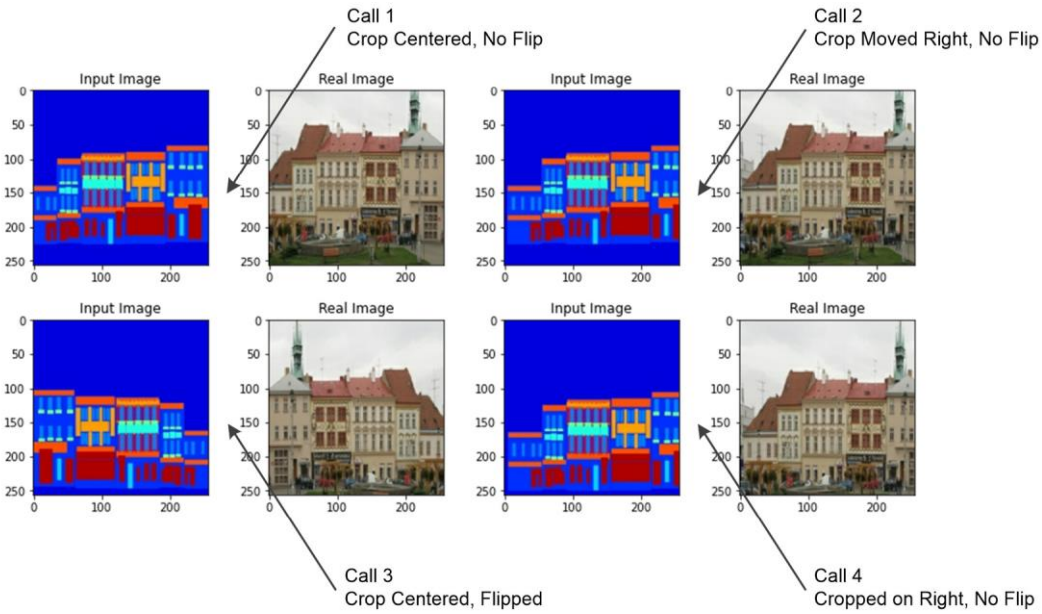

Figure 10.21 – The output of the image modification tests

5. Define a normalizing function. The act of **normalization** prevents training problems that can occur due to differences in the various images:

```
def normalize(input_image, real_image):
    input_image = (input_image / 127.5) - 1
    real_image = (real_image / 127.5) - 1
    return input_image, real_image
```

6. Perform the image normalization:

```
normal_input_image, normal_real_image = \
    normalize(input_image, real_image)
print(normal_input_image)
```

This final step tests the normalization process. *Figure 10.22* shows the typical output. The actual output is significantly longer than shown.

```
tf.Tensor(
[[[-0.9764706  -0.9764706   0.5294118 ]
  [-0.99215686 -0.9764706   0.5686275 ]
  [-0.99215686 -1.          0.6862745 ]
  ...
  [-1.         -1.          0.7411765 ]
  [-1.         -1.          0.7411765 ]
  [-1.         -1.          0.7411765 ]]
```

Figure 10.22 – The normalized façade image output

Note that while it's easy to determine that the data is acceptable in form, it's not possible to tell that it's the right data. Hacker modifications would be impossible to detect at this point, which is why you need to perform testing and verification of data when it's in a form that you can detect modifications. Now that everything is in place to manipulate the images, it's time to create the actual datasets.

Developing datasets from the modified images

As with all of the machine learning examples in the book so far, you need a training dataset and a testing dataset to use with the model. The following steps show how to create the required image datasets:

7. Import the required packages:

```
import os
```

8. Create a function to load a training image. The training images have to be randomized for the model to work well. However, the testing images should appear as normal to truly test the model in a real-world setting:

```
@tf.autograph.experimental.do_not_convert
def load_image_train(files):
    input_image = load(files[0])
    real_image = load(files[1])
    input_image, real_image = \
        random_jitter(input_image, real_image)
    input_image, real_image = \
        normalize(input_image, real_image)
    return input_image, real_image
```

9. Create a function to load a testing image. Note that the images must be resized to 256x256 so that they appear the same as in the original dataset:

```
@tf.autograph.experimental.do_not_convert
def load_image_test(files):
    input_image = load(files[0])
    real_image = load(files[1])
    input_image, real_image = resize(input_image,
        real_image, IMG_HEIGHT, IMG_WIDTH)
    input_image, real_image = normalize(input_image,
        real_image)
    return input_image, real_image
```

10. Specify the dataset parameters. This is the same as the `batch_size` setting used when fitting the autoencoder in the previous example:

```
BUFFER_SIZE = 200
BATCH_SIZE = 1
```

11. Create a list of files to process for the training dataset:

```
real_files = os.listdir(PATH+'trainA')
real_files = \
    [PATH+'trainA/'+file for file in real_files]
input_files = os.listdir(PATH+'trainB')
input_files = \
    [PATH+'trainB/'+file for file in input_files]
file_list = \
    list(map(list, zip(input_files, real_files)))
for file in file_list:
    print(file)
```

The input and real files appear in two different directories. Yet, these files are actually paired with each other. Consequently, what this code does is create a list of lists, where each pair appears in its own list. *Figure 10.23* shows the output from this step.

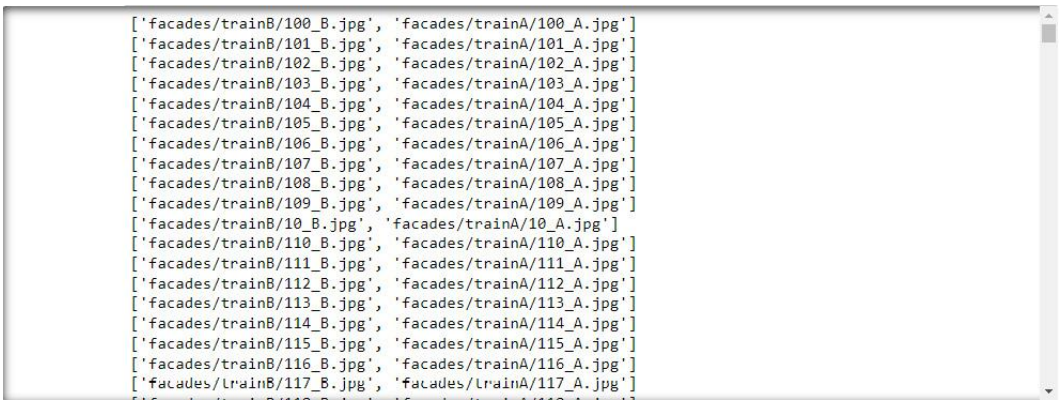

Figure 10.23 – The pairings of input and real files used to create the training dataset

12. Load the training dataset. The first step actually creates the dataset from the `file_list` prepared in the previous step. This series of filenames is used to load the images using the `load_image_train()` function defined in *step 2*. Because the dataset is currently in a specific order, the `shuffle()` function randomizes the image order. Finally, the number of batches to perform is set:

```
train_dataset = \
    tf.data.Dataset.from_tensor_slices(file_list)
```

```
train_dataset = \
    train_dataset.map(load_image_train,
        num_parallel_calls=4)
train_dataset = train_dataset.shuffle(BUFFER_SIZE)
train_dataset = train_dataset.batch(BATCH_SIZE)
```

13. View the result. The dataset includes both features and labels:

```
features, label = iter(train_dataset).next()
print("Example features:", features[0])
print("Example label:", label[0])
```

Figure 10.24 shows a very short example of what you'll see as output.

```
Example features: tf.Tensor(
[[[-0.8901961  -0.60784316  0.47450984]
  [-0.92941177 -0.6392157   0.39607847]
  [-0.9137255  -0.60784316  0.4039216 ]
  ...
  [-0.84313726 -0.6627451   0.5529412 ]
  [-0.8509804  -0.6392157   0.6862745 ]
  [-1.         -0.8352941   0.5529412 ]]
```

Figure 10.24 – A list of tensors based on the input and real images

14. Create a list of files to process for the testing dataset. This is basically a repetition of the process for the training dataset:

```
real_files = os.listdir(PATH+'testA')
real_files = \
    [PATH+'testA/'+file for file in real_files]
input_files = os.listdir(PATH+'testB')
input_files = \
    [PATH+'testB/'+file for file in input_files]
file_list = list(map(list, zip(input_files,
        real_files)))
```

15. Load the testing dataset:

```
test_dataset = \
    tf.data.Dataset.from_tensor_slices(file_list)
test_dataset = test_dataset.map(load_image_test)
test_dataset = test_dataset.batch(BATCH_SIZE)
```

The datasets are finally ready to use. Now it's time to create the generator part of the Pix2Pix GAN.

Creating the generator

A U-Net generator is known as such because it actually forms a kind of U in the method it uses for processing data. The process consists of **downsampling** (encoding), which compresses the data, and **upsampling** (decoding), which decompresses the data. *Figure 10.25* shows the U-Net for this example.

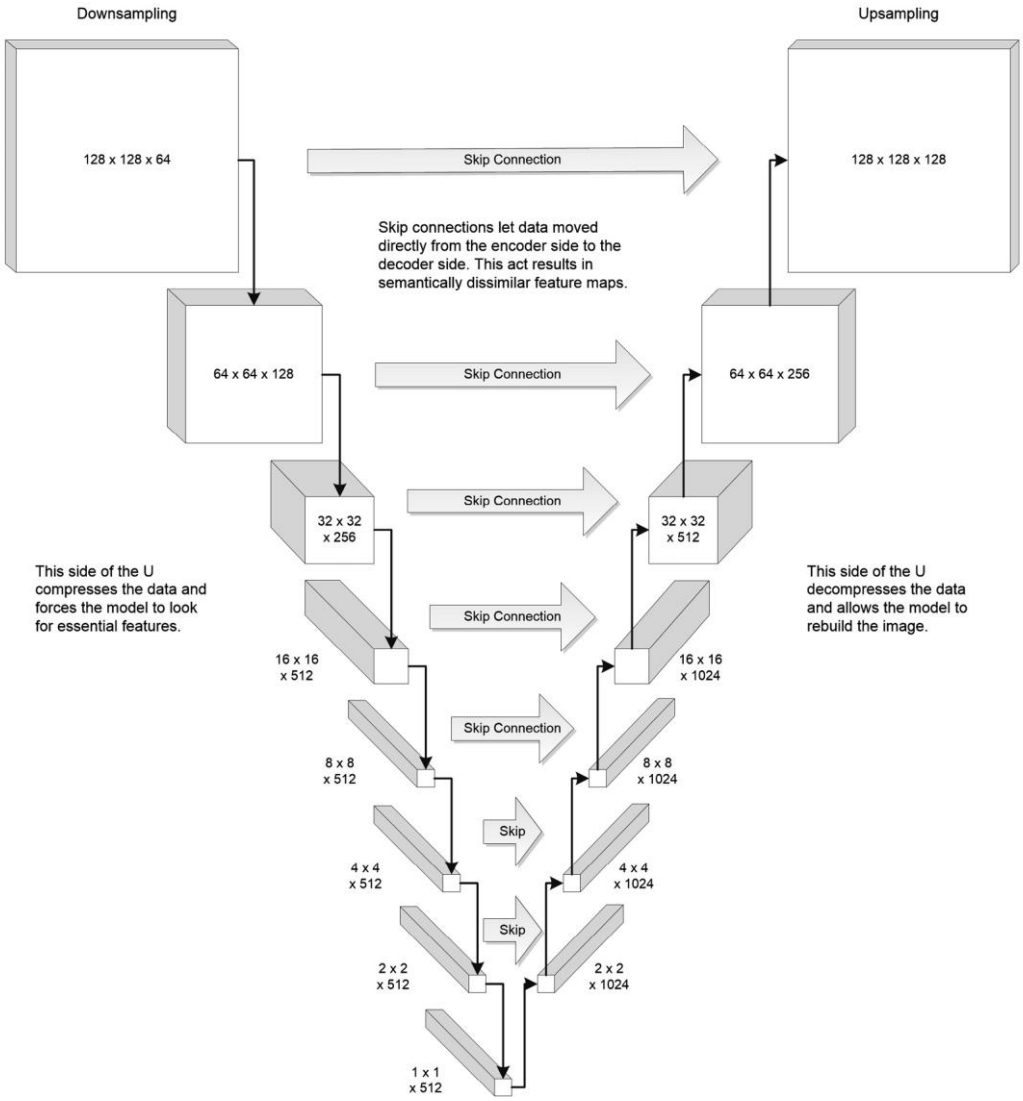

Figure 10.25 – A diagram of a U-Net generator

If you think the graphic in *Figure 10.25* looks sort of like a fancy version of the autoencoder in *Figure 10.6*, you'd be right in a way. The U-Net generator does compress and decompress data like the autoencoder but it does so in a smarter way so that it can generate a new image from the existing one. Unfortunately, the model is susceptible to the same forms of hacking as an autoencoder is. Sending bad inputs will affect this model just as much as affects an autoencoder, so you need to exercise care in keeping your model free from hacker activity. Of course, there is a lot more going on than just compression and decompression. The following sections build each element of the U-Net generator in turn.

Defining the downsampling code

Downsampling relies on a number of layers to compress the data. These layers accomplish the following purposes:

- `Conv2D`: Convolutes the layer inputs to produce a tensor of output values. Essentially, this is the part that compresses the data.
- `BatchNormalization`: Performs a transformation to keep the output mean close to `0` and the standard deviation close to `1`.
- `LeakyReLU`: Provides a leaky version of a ReLU that provides activation for the layer. The term **Leaky ReLU** means that there is a small gradient applied when the input is negative.

Now that you have a better idea of what the downsampling layers mean, it's time to look at the required code. The following steps show how to build this part of the U-Net generator:

1. Create a `downsample()` function:

```
OUTPUT_CHANNELS = 3
def downsample(filters, size):
    initializer = tf.random_normal_initializer(
        0., 0.02)
    result = tf.keras.Sequential()
    result.add(
        tf.keras.layers.Conv2D(filters, size,
            strides=2, padding='same',
            kernel_initializer=initializer,
            use_bias=False))
    result.add(
        tf.keras.layers.BatchNormalization())
    result.add(tf.keras.layers.LeakyReLU())
    return result
```

2. View the `downsample()` function results. This code tests the `downsample()` function using just one image:

```
down_model = downsample(3, 4)
down_result = \
    down_model(tf.expand_dims(input_image, 0))
print (down_result.shape)
```

The output is the shape of the tested image, as shown in *Figure 10.26*.

```
(1, 128, 128, 3)
```

Figure 10.26 – Output of the initial downsample() function test

When you compare this test to *Figure 10.26*, you will see that the batch size is reflected in the first return value, the size of the image is reflected in the second and third values, and the number of filters in the fourth value. Because this is the first step of compression, the original 256x256 size of the input image is reduced in half.

Defining the upsampling code

Upsampling relies on a number of layers to decompress the data. These layers accomplish the following purposes:

- `Conv2DTranspose`: Deconvolutes the layer inputs to produce a tensor of output values. Essentially, this is the part that decompresses the data.
- `BatchNormalization`: Performs a transformation to keep the output mean close to `0` and the standard deviation close to `1`.
- `ReLU`: Provides activation for the layer.

As you can see, the upsampler follows a process similar to the downsampler, just in reverse. The following steps describe how to create the upsampler and test it:

3. Create the `upsample()` function:

```
def upsample(filters, size):
    initializer = tf.random_normal_initializer(
        0., 0.02)
    result = tf.keras.Sequential()
    result.add(
        tf.keras.layers.Conv2DTranspose(filters,
            size, strides=2, padding='same',
            kernel_initializer=initializer,
            use_bias=False))
    result.add(
        tf.keras.layers.BatchNormalization())
    result.add(tf.keras.layers.ReLU())
    return result
```

4. View the `upsample()` function results:

```
up_model = upsample(3, 4)
```

```
up_result = up_model(down_result)

print (up_result.shape)
```

This code is upsampling the downsampled image, so *Figure 10.27* shows that the image is now 256x256 again.

```
(1, 256, 256, 3)
```

Figure 10.27 – The upsampled result of the downsampled image

As with the autoencoder example, you now have a downsampler (which is akin to the encoder) and an upsampler (which is akin to the decoder). However, you don't have an entire U-Net generator yet. The next section takes these two pieces and puts them together to create the generator depicted in *Figure 10.25*.

Putting the generator together

You now have everything needed to create a U-Net like the one depicted in *Figure 10.25* (referencing the figure helps explain the code). The following code puts everything together. The comments tell you about the size changes that occur as the images are downsampled, then upsampled:

```
def Generator():
    inputs = tf.keras.layers.Input(shape=[256, 256, 3])
    down_stack = [
        downsample(64, 4),   # 128, 128, 64
        downsample(128, 4),  # 64, 64, 128
        downsample(256, 4),  # 32, 32, 256
        downsample(512, 4),  # 16, 16, 512
        downsample(512, 4),  # 8, 8, 512
        downsample(512, 4),  # 4, 4, 512
        downsample(512, 4),  # 2, 2, 512
        downsample(512, 4),  # 1, 1, 512
    ]
    up_stack = [
        upsample(512, 4),     # 2, 2, 1024
        upsample(512, 4),     # 4, 4, 1024
        upsample(512, 4),     # 8, 8, 1024
        upsample(512, 4),     # 16, 16, 1024
        upsample(256, 4),     # 32, 32, 512
        upsample(128, 4),     # 64, 64, 256
        upsample(64, 4),      # 128, 128, 128
    ]
    initializer = tf.random_normal_initializer(0., 0.02)
    # 256, 256, 3
    last = tf.keras.layers.Conv2DTranspose(
        OUTPUT_CHANNELS, 4, strides=2, padding='same',
        kernel_initializer=initializer, activation='tanh')
    x = inputs
    skips = []
```

```
    for down in down_stack:
        x = down(x)
        skips.append(x)
    skips = reversed(skips[:-1])
    for up, skip in zip(up_stack, skips):
        x = up(x)
        x = tf.keras.layers.Concatenate()([x, skip])
    x = last(x)
    return tf.keras.Model(inputs=inputs, outputs=x)
generator = Generator()
```

The code begins, just as the autoencoder did, with the creation of an `Input` layer. Next, comes a series of downsampling and upsampling steps to perform. The process ends with a final call to `Conv2DTranspose()`, defined as function `last()`, which returns the image to its former size of 256x256x3. So, just as with the autoencoder, you have an input, compression stages, decompression states, and an output layer. The main difference, in this case, is that the model uses the `tanh` activation, which is similar to the `sigmoid` activation used for the autoencoder, except that `tanh` works with both positive and negative values.

As shown in *Figure 10.25*, this model allows for the use of skips, where the data goes from some level of the downsample directly to the corresponding layer of the upsample without traversing all of the layers. This approach allows the generator to create a better model because not every image is processed to precisely the same level. The skips are completely random. The final step is to actually create the `generator` with a call to `Generator()`.

Defining the generator loss function

The loss function helps optimize the generator weights. As the generator produces images, the weights help determine changes in generator output so that the generator output better matches the original image. The following steps show how to create the generator loss function for this example:

1. Create the required `loss_object` function, which uses cross-entropy to determine the difference between true labels and predicted labels:

```
loss_object = \
    tf.keras.losses.BinaryCrossentropy(
    from_logits=True)
```

2. Specify the `LAMBDA` value, which controls the amount of regularization applied to the model:

```
LAMBDA = 100
```

3. Define the loss function, which includes calculating the initial loss (`gan_loss`) and the L1 loss (`l1_loss`). Then, use them to create a `total_gen_loss` for the generator as a whole:

```
def generator_loss(disc_generated_output,
    gen_output, target):
```

```
    gan_loss = loss_object(
        tf.ones_like(disc_generated_output),
        disc_generated_output)
    l1_loss = tf.reduce_mean(
        tf.abs(target - gen_output))
    total_gen_loss = gan_loss + (LAMBDA * l1_loss)
    return total_gen_loss, gan_loss, l1_loss
```

The example uses the `BinaryCrossentropy()` loss function as a starting point for creating a hand-tuned loss function for the U-Net. It then calculates an additional L1 loss value, which minimizes the error from the sum of all the absolute differences between the true values and the predicted values. The total loss is then calculated by adding the cross-entropy loss to the L1 loss (after equalizing the two values) after having multiplied it by a constant. Empirically it has been demonstrated that this combination of losses helps the network to converge faster and in a more stable way.

Creating the discriminator

The PatchGAN is a type of discriminator where a patch of a specific size (30x30x1 in this case) is run across images to determine whether they're fake or real. This example begins with two 256x256x3 images: the first is of the input image from the generator, while the second is the target image from the dataset.

To create the patch, the images are downsampled (compressed). After that, the compressed images are processed in various ways, as determined by the model. *Figure 10.28* shows the process graphically.

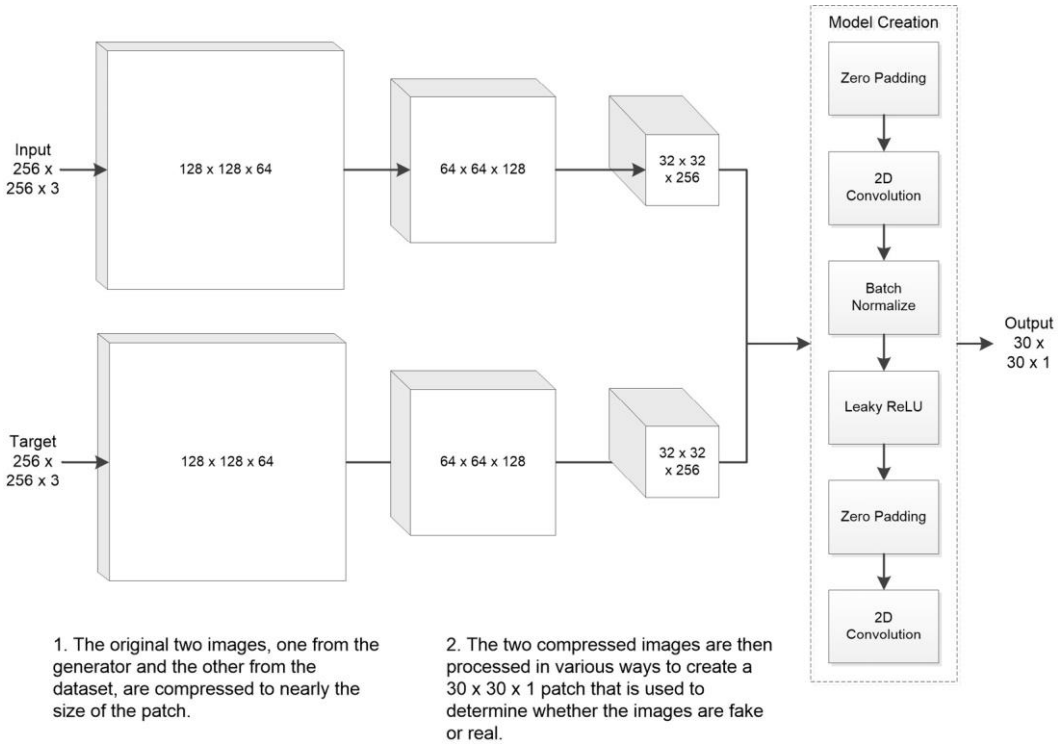

Figure 10.28 – A diagram of the PatchGAN discriminator

As with the generator, the discriminator consists of a number of pieces that are best understood when discussed individually. The following sections tell you about them.

Putting the discriminator together

Even though *Figure 10.28* may look a little complex, the actual coding doesn't require much effort after working through the intricacies of the U-Net generator. In fact, the code is rather short, as shown here. The comments tell you about the downsampling process so that you can compare it with *Figure 10.28*. There are similarities between this model and the autoencoder from earlier, but in this case, you can see there are two inputs instead of one, so you need to concatenate them:

```
Def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)
    inp = tf.keras.layers.Input(
        shape=[256, 256, 3], name='input_image')
    tar = tf.keras.layers.Input(
        shape=[256, 256, 3], name='target_image')
    # 256, 256, channels*2)
    x = tf.keras.layers.concatenate([inp, tar])
    down1 = downsample(64, 4)(x)         # 128, 128, 64
    down2 = downsample(128, 4)(down1) # 64, 64, 128
    down3 = downsample(256, 4)(down2) # 32, 32, 256
    # 34, 34, 256
    zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3)
    # 31, 31, 512
    conv = tf.keras.layers.Conv2D(
        512, 4, strides=1,
        kernel_initializer=initializer,
        use_bias=False)(zero_pad1)
    batchnorm1 = \
        tf.keras.layers.BatchNormalization()(conv)
    leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)
    # 33, 33, 512
    zero_pad2 = \
        tf.keras.layers.ZeroPadding2D()(leaky_relu)
    # 30, 30, 1
    last = tf.keras.layers.Conv2D(
        1, 4, strides=1,
        kernel_initializer=initializer)(zero_pad2)
    return tf.keras.Model(inputs=[inp, tar], outputs=last)
discriminator = Discriminator()
```

The goal of this code is to create a useful model—one that takes the two image sets as input and provides a 30x30 patch as output. Images are processed in patches. It uses these steps to make the determination between fake and real images:

1. Obtain the input and target images sized at 256x256x3.

2. Concatenate the images for processing. Each image should be a separate channel.

3. Downsample the images contained in `x` (the two concatenated channels) to 32x32x256.
4. Add zeros around the entire image so that each image is now 34x34.
5. Compress the data to 31x31x512.
6. Normalize the image data.
7. Perform the activation function.
8. Add zeros around the entire image so that each image is now 33x33.
9. Create a patch of 30x30x1.
10. Return the model consisting of the original input data and the patch.

The patch is run convolutionally across the image and the results are averaged to determine whether the image as a whole is fake or real.

Defining the discriminator loss

As with the generator loss function, the discriminator loss function helps optimize weights but the function works with the discriminator in this case, rather than the generator. So, there are two loss functions, one for the generator and another for the discriminator. Here is the code used for the discriminator loss:

```
def discriminator_loss(disc_real_output,
    disc_generated_output):
        real_loss = loss_object(
        tf.ones_like(disc_real_output), disc_real_output)
    generated_loss = loss_object(
        tf.zeros_like(disc_generated_output),
        disc_generated_output)
    total_disc_loss = real_loss + generated_loss
    return total_disc_loss
```

There are a number of steps in calculating the loss:

1. Create a real image loss object using `1`s to show this is the real image.
2. Create a generated image loss object using `0`s to show this is the fake image.
3. Calculate the loss based on whether the discriminator sees the real image as real and the fake image as fake.

As you can see, the discriminator loss is different from the generator loss in that the discriminator loss determines whether the image is fake or real.

Performing optimization of both generator and discriminator

To make the process of generating and discriminating images faster and better, the code relies on the `Adam()` optimization function for both the generator and the discriminator. The `Adam()` optimization function relies on the stochastic gradient descent method. The primary reason to use it

in this case is that it's computationally efficient and doesn't require a lot of memory. A secondary reason is that it works well with noisy data, which is likely going to happen with images. The following code shows the calls for using the `Adam()` optimizer:

```
generator_optimizer = \
    tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
discriminator_optimizer = \
    tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
```

The first argument determines the learning rate for the generator and discriminator. You normally want these values to be the same or the model may not work as intended. The `beta_1` value determines the exponential decay rate for the first moment (the mean, rather than the uncentered variance, which is determined by `beta_2`). You can read more about this function at https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam.

The next step is to perform the required training. However, because this process takes so long, you want to monitor it so that you can stop training and make adjustments as needed.

Monitoring the training process

As you train your model, you will want to see how the results change over time. The following code outputs three images: the input image (semantic labels), the ground truth (original image), and the predicted (generated) image:

```
def generate_images(model, test_input, tar):
    prediction = model(test_input, training=True)
    plt.figure(figsize=(15, 15))
    display_list = [test_input[0], tar[0], prediction[0]]
    title = ['Input Image', 'Ground Truth',
        'Predicted Image']
    for i in range(3):
        plt.subplot(1, 3, i+1)
        plt.title(title[i])
        plt.imshow(display_list[i] * 0.5 + 0.5)
        plt.axis('off')
    plt.show()
```

The following code provides a quick check of three images:

```
for example_input, example_target in test_dataset.take(1):
    generate_images(generator, example_input,
        example_target)
```

Figure 10.29 shows the typical results for an untrained model.

Figure 10.29 – The output of the generate_images() function using an untrained model

Now that you have the means for monitoring the training, it's time to do some actual training of the model, as described in the next section. One thing to consider is that monitoring does provide a method of detecting potential hacker activity. If you have set everything up correctly and your model still isn't producing the predicted image, then you might have problems with the training data, especially the input images, because small modifications would be tough to locate.

Training the model

Each step of the training occurs in what is termed an **epoch**. It helps to review how the training will occur by viewing *Figure 10.19* as an overview. *Figure 10.25* provides details of the generator and *Figure 10.28* provides details of the discriminator. The following code finally puts together what these figures have been showing you. It outlines a single training step:

```
EPOCHS = 24
@tf.function
@tf.autograph.experimental.do_not_convert
def train_step(input_image, target, epoch):
    with tf.GradientTape() as gen_tape, \
        tf.GradientTape() as disc_tape:
            gen_output = generator(
                input_image, training=True)
        disc_real_output = discriminator(
            [input_image, target], training=True)
        disc_generated_output = discriminator(
            [input_image, gen_output], training=True)
        gen_total_loss, gen_gan_loss, gen_l1_loss = \
            generator_loss(disc_generated_output,
                gen_output, target)
        disc_loss = discriminator_loss(
            disc_real_output, disc_generated_output)
    generator_gradients = gen_tape.gradient(
        gen_total_loss, generator.trainable_variables)
    discriminator_gradients = disc_tape.gradient(
        disc_loss, discriminator.trainable_variables)
    generator_refittings = 3
    for _ in range(generator_refittings):
        generator_optimizer.apply_gradients(zip(
           generator_gradients,
           generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(
        discriminator_gradients,
        discriminator.trainable_variables))
```

The code is following these steps during each epoch:

1. Create two `GradientTape()` objects (see https://www.tensorflow.org/api_docs/python/tf/GradientTape to record operations for automatic differentiation): one for the generator (`gen_tape`) and one for the discriminator

(`disc_tape`). Think of a tape used for making backups or to record other kinds of information, except that this one is recording operations.

2. Generate an image using the generator.
3. Generate the real output using the discriminator.
4. Generate the predicted output using the discriminator.
5. Calculate the generator loss.
6. Calculate the discriminator loss.
7. Determine how much to change each model after each training cycle and place this result in `generator_gradients` for the generator and `discriminator_gradients` for the discriminator.
8. Apply the changes to each model, optimizing the result in each case, using `generator_optimizer.apply_gradients()` for the generator and `discriminator_optimizer.apply_gradients()` for the discriminator.

Note that you can change the `EPOCHS` setting as needed for your system. The more epochs you use, the better the model, but each epoch takes a significant amount of time. The next step is to define the fitting function.

Specifying how to train the model

It's important to remember that this is just one training step or epoch. The example performs 24 epochs to obtain a reasonable result. However, many Pix2Pix GANs go through 150 or more epochs to obtain a production-level result. During the testing process of the example, it became evident that the generator wasn't being worked hard enough. So, the example also puts the generator through three application cycles to one for the discriminator, producing a better result in a shorter time. As you work through your code, you'll likely find that you need to make tweaks like this to obtain a better result with an eye toward efficiency.

Defining the fitting function

Now that the steps to perform for each epoch are defined, it's time to perform the fitting process. Fitting trains the generator and discriminator to produce the desired output. The following steps show the fitting process for this example:

9. Import the required packages:

```
from IPython import display
```

10. Create the `fit()` function to perform the fitting:

```
def fit(train_ds, epochs, test_ds):
    for epoch in range(epochs):
        display.clear_output(wait=True)
```

```
        for example_input, example_target in \
            test_ds.take(1):
                generate_images(generator,
                    example_input,
                    example_target)
        print("Epoch: ", epoch)
        for n, (input_image, target) in \
            train_ds.enumerate():
                print('.', end='')
                train_step(input_image, target, epoch)
        print()
```

The `fit()` function is straightforward. All it does is fit the two models (generator and discriminator) to the images one at a time, make adjustments, and then move on to the next epoch. The next section performs the actual fitting process.

Performing the fitting

So far, no one has really hit the run button. Everything is in place, but now it's time to actually run it, which is the purpose of the code in the following steps:

1. Perform the actual fitting task:

```
fit(train_dataset, EPOCHS, test_dataset)
```

Figure 10.30 shows the output for a single epoch.

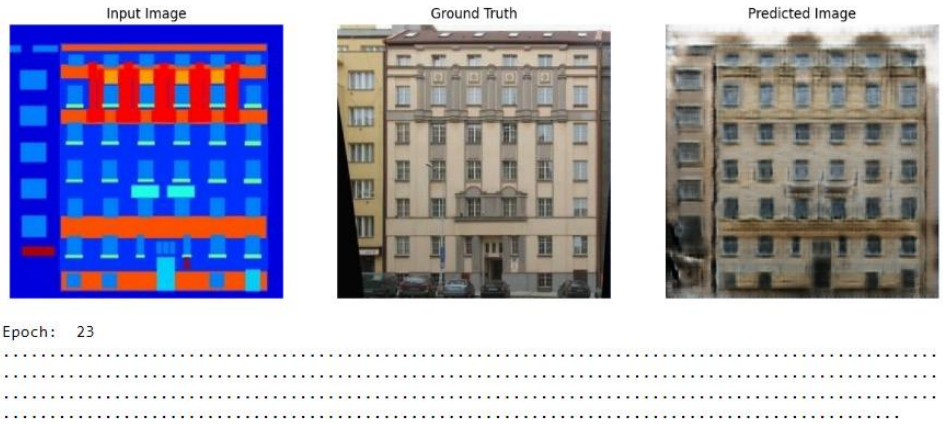

Figure 10.30 – The output from a single training epoch

2. Check the results:

```
for example_input, example_target in \
    test_dataset.take(5):
```

```
        generate_images(generator, example_input,
            example_target)
```

The output will show five random images from the dataset. In looking at the output shown in *Figure 10.31* for a single image from the output, you can see that the input image has affected the original (ground truth) image and produced the predicted output, which isn't perfect at this point because the model requires more training.

Figure 10.31 – Sample output from the trained Pix2Pix GAN

The essential thing to remember about a Pix2Pix GAN is that it's a complex model that requires a large dataset, which gives hackers plenty of opportunity to skew your model. As shown in *Figure 10.31* (and any other Pix2Pix GAN example you want to review), it would be very hard if not impossible for a human to detect that the output has been skewed. What a human would see is that the image has been modified, hopefully in the right direction. A smart hacker could modify the model using any of a number of methods, with incorrect data being the easiest and most probable, to output just about anything.

Links

https://www.youtube.com/watch?v=oxXpB9pSETo

Fake Faces: https://www.theverge.com/tldr/2019/2/15/18226005/ai-generated-fake-people-portraits-thispersondoesnotexist-stylegan

Fake biometrics: https://www.wired.com/story/deepmasterprints-fake-fingerprints-machine-learning/

Sites such as https://www.tamikothiel.com/cgi-bin/LendMeYourFace.cgi make it possible to create a deepfake of your face into an image.

If you need a fake face, you can create one at https://generated.photos/

Deep Fakes' Greatest Threat Is Surveillance Video
https://www.forbes.com/sites/kalevleetaru/2019/08/26/deep-fakes-greatest-threat-is-surveillance-video/?sh=5cf6b7c44550

Fake news generators: https://www.thefakenewsgenerator.com/

Developing a Robust Defensive System against Adversarial Examples Using Generative Adversarial Networks (https://www.mdpi.com/2504-2289/4/2/11/pdf).

Phillip Isola (et. al) originally presented the idea of a Pix2Pix GAN in the Image-to-Image Translation with Conditional Adversarial Networks whitepaper (https://arxiv.org/abs/1611.07004), in 2016.

Video Rewrite: Driving Visual Speech with Audio, written by Christoph Bregler, Michele Covell, and Malcolm Slaney: http://chris.bregler.com/videorewrite/)

Active appearance models whitepaper: https://ieeexplore.ieee.org/document/927467

https://www.youtube.com/watch?v=cQ54GDm1eL0

https://www.tensorflow.org/

Further reading

The following bullets provide you with some additional reading that you may find useful for understanding the materials in this chapter better:

- Enjoy some of the fabulous paintings created by Renoir: *Pierre Auguste Renoir:* https://www.pierre-auguste-renoir.org/
- Discover examples of deepfakes that are both interesting and somewhat terrifying: *18 deepfake examples that terrified and amused the internet*: https://www.creativeblog.com/features/deepfake-examples
- Read about how deepfakes can change the voices of support people on the phone: *Sanas, the buzzy Bay Area startup that wants to make the world sound whiter:* https://www.sfgate.com/news/article/sanas-startup-creates-american-voice-17382771.php
- Gain an understanding of how the legal system has failed with its revenge porn laws in some respects: *Are You Sure You Know What Revenge Porn Is?:* https://www.wired.com/story/revenge-porn-platforms-grindr-queerness/
- See how deepfakes are used to scam others: *Top 5 Deepfake Scams that Stormed the Internet this Year*: https://www.analyticsinsight.net/top-5-deepfake-scams-that-stormed-the-internet-this-year/
- Review examples of fake people pictures: *New York Times:* https://www.nytimes.com/interactive/2020/11/21/science/artificial-intelligence-fake-people-faces.html
- Consider the fact that people actually trust deepfake faces more than the real thing: *People Trust Deepfake Faces More Than Real Faces:* https://www.vice.com/en/article/7kb7ge/people-trust-deepfake-faces-more-than-real-faces
- Obtain a list of the best AI art generators today: *Here's how the best AI art generators compare:* https://www.creativeblog.com/news/ai-art-generator-comparison

- Get more information about detecting deepfakes in video: *Detect DeepFakes: How to counteract misinformation created by AI:* https://www.media.mit.edu/projects/detect-fakes/overview/
- Consider the ease of creating a really good deepfake and then detecting it: *How Easy Is It to Make and Detect a Deepfake?:* https://insights.sei.cmu.edu/blog/how-easy-is-it-to-make-and-detect-a-deepfake/
- Gain an understanding of just how terrible deepfake porn can be for women (and most likely men as well): *Deepfakes: The Latest Anti-Woman Weapon:* https://womensenews.org/2022/05/deep-fakes-the-latest-anti-woman-weapon/
- Discover how autoencoder approaches are improving unsupervised learning techniques: *Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles:* https://arxiv.org/abs/1603.09246
- Get an idea of how Pix2Pix works without coding it first: *Doodles to Pictures:* https://mitmedialab.github.io/GAN-play/
- Read more about the use of color for the facade dataset: *CMP Facade Database:* https://cmp.felk.cvut.cz/~tylecr1/facade/
- Gain insights into ReLU functionality: *A Gentle Introduction to the Rectified Linear Unit (ReLU):* https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks//
- Discover more about activation function differences, especially the difference between sigmoid and tanh: *Activation Functions in Neural Networks:* https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
- Get a better idea of how GAN loss functions work: *A Gentle Introduction to Generative Adversarial Network Loss Functions:* https://machinelearningmastery.com/generative-adversarial-network-loss-functions/
- Learn more about how the `Adam()` optimization function works: *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning:* https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Chapter 11

Figure

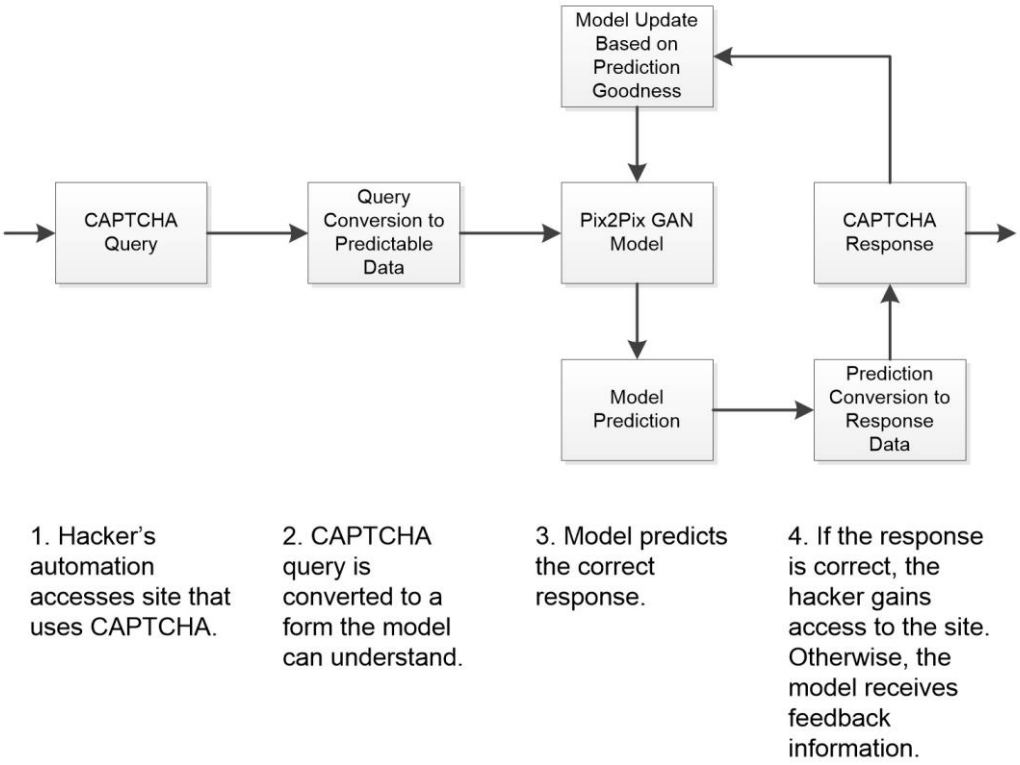

Figure 11.1 – A block diagram of an automated security measure bypass scheme

Links

Completely Automated Public Turing tests to tell Computers and Humans Apart (CAPTCHA) (http://www.captcha.net/)

Breaking CAPTCHA Using Machine Learning in 0.05 Seconds: https://pub.towardsai.net/breaking-captcha-using-machine-learning-in-0-05-seconds-9feefb997694

The "Click Allow To Verify That You Are Not A Robot" Scam: https://malwaretips.com/blogs/remove-click-allow-to-verify-that-you-are-not-a-robot/

Inaccessibility of CAPTCHA: https://www.w3.org/TR/turingtest/

Liveness in biometrics: spoofing attacks and detection (https://www.thalesgroup.com/en/markets/digital-identity-and-security/government/inspired/liveness-detection

China is the only country in the world where deepfakes are currently regulated according to https://recfaces.com/articles/biometric-technology-vs-deepfake.

35 Outrageous Hacking Statistics & Predictions [2022 Update]: https://review42.com/resources/hacking-statistics/),

BI Warns Deepfakes Might Be Used in Remote Job Interviews (https://petapixel.com/2022/07/05/fbi-warns-deepfakes-might-be-used-in-remote-job-interviews/

TEMPEST (see https://apps.nsa.gov/iaarchive/programs/iad-initiatives/tempest.cfm for details)

Compromising emanations: eavesdropping risks of computer displays (https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-577.pdf)

Introduction To Generative Networks: https://towardsdatascience.com/introduction-to-generative-networks-e33c18a660dd

You can also find some other ideas on how to correct this problem at https://www.mathworks.com/help//deeplearning/ug/monitor-gan-training-progress-and-identify-common-failure-modes.html.

Support Vector Machine — Simply Explained: https://towardsdatascience.com/support-vector-machine-simply-explained-fee28eba5496

Wasserstein loss (https://developers.google.com/machine-learning/gan/loss)

Unrolled GANs (https://arxiv.org/pdf/1611.02163.pdf)

Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN (https://arxiv.org/abs/1702.05983

Just in case you're wondering, you can find proof of concept code for MalGAN at https://github.com/Izylucy/Malware-GAN-attack

Improved MalGAN: Avoiding Malware Detector by Leaning Cleanware Features at https://ieeexplore.ieee.org/document/8669079

Internet Crime Report 2020, https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf, for details

Congress Approves Cyber Attack Reporting Requirement for U.S. Companies, https://www.insurancejournal.com/news/national/2022/03/14/657926.htm

Ivanti (https://www.ivanti.com/resources/v/doc/infographics/ivi-2594_9-must-know-phishing-attack-trends-ig

Visual Geometry Group-16 layer (VGG-16) (https://www.geeksforgeeks.org/vgg-16-cnn-model/)

Residual Network (ResNet) (https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/

SmartBots (https://www.smartbots.ai/

New AI Generates Horrifyingly Plausible Fake News at https://futurism.com/ai-generates-fake-news

Ukrainian Radio Stations Hacked to Broadcast Fake News About Zelenskyy's Health https://thehackernews.com/2022/07/ukrainian-radio-stations-hacked-to.html

The *Chinese Hackers Deploy Fake News Site To Infect Government, Energy Targets* story at https://www.technewsworld.com/story/chinese-hackers-deploy-fake-news-site-to-infect-government-energy-targets-177036.html

Grover (https://grover.allenai.org/)

These 3 Tools Will Help You Spot Fake Amazon Reviews: https://www.makeuseof.com/fake-reviews-amazon/

Fake reviews and ratings manipulation continue to plague the App Store charts (https://9to5mac.com/2021/04/12/fake-reviews-and-ratings-manipulation-app-store/)

MobileMonkey (https://mobilemonkey.com/blog/how-to-get-more-reviews-with-a-bot)

Further reading

The following bullets provide you with some additional reading that you may find useful for understanding the materials in this chapter in greater depth:

- Read about how the use of skipping (see *Figure 10.25*) helps improve the CNN used to break CAPTCHA: *A novel CAPTCHA solver framework using deep skipping Convolutional Neural Networks* (https://peerj.com/articles/cs-879.pdf)
- Discover the myriad of bots used for legitimate purposes: *WhatIs.com bot* (https://www.techtarget.com/whatis/definition/bot-robot)
- See a list of commonly offered commercial bots used for various tasks: *Bot Platforms Software* (https://www.saasworthy.com/list/bot-platforms-software) and *Best Bot Platforms Software* (https://www.g2.com/categories/bot-platforms)
- Learn more about air-gapped computers: *What is an Air Gapped Computer?* (https://www.thesslstore.com/blog/air-gapped-computer/)
- Read about the details of hacking an air-gapped computer: *Four methods hackers use to steal data from air-gapped computers* (https://www.zdnet.com/article/stealing-data-from-air-gapped-computers/)
- Invest some time into reading more about the potential for smartphone security issues: *An experimental new attack can steal data from air-gapped computers using a phone's gyroscope* (https://techcrunch.com/2022/08/24/gairoscope-air-gap-attack/)
- Consider the ramifications of using a TEMPEST certified system: *5 Things Everyone Should Know About Tempest And Information Security* (https://www.fiberplex.com/blog/5-things-everyone-should-know-about-tempest-and-information-security.html)
- Discover how using a **least squares generative adversarial network (LSGAN)** can improve the creation of AME: *LSGAN-AT: enhancing malware detector robustness against adversarial examples* (https://cybersecurity.springeropen.com/articles/10.1186/s42400-021-00102-9)

- Find an n-gram version of MalGAN: *N-gram MalGAN: Evading machine learning detection via feature n-gram* (https://www.sciencedirect.com/science/article/pii/S2352864821000973)
- Learn more about using logos to detect spear-phishing emails: *Color Schemes: Detecting Logos in Phishing Attacks* (https://www.vadesecure.com/en/blog/detecting-logos-in-phishing-attacks)
- Consider the use of smart bots for human augmentation: *Using Smart Bots to Augment Humans* (https://www.spiceworks.com/tech/innovation/guest-article/using-smart-bots-to-augument-humans/)
- Read about some political motivations behind fake news: *"Hacker X"—the American who built a pro-Trump fake news empire—unmasks himself* (https://arstechnica.com/information-technology/2021/10/hacker-x-the-american-who-built-a-pro-trump-fake-news-empire-unmasks-himself/)

Chapter 12

Technical requirements

This chapter requires that you have access to either Google Colab or Jupyter Notebook to work with the example code. The *Requirements to use this book* section of *Chapter 1*, *Defining Machine Learning Security*, provides additional details on how to set up and configure your programming environment. When testing the code, use a test site, test data, and test APIs to avoid damaging production setups and to improve the reliability of the testing process. Testing over a non-production network is highly recommended, but not absolutely necessary. Using the downloadable source code is always highly recommended. You can find the downloadable source on the Packt GitHub site at https://github.com/PacktPublishing/Machine-Learning-Security-Principles or on my website at http://www.johnmuellerbooks.com/source-code/.

Dataset used in this chapter

The dataset used in this chapter is custom to this chapter because it demonstrates constructions that you shouldn't use in a real-world application. All of the values used are fictitious and any resemblance to any entity, living or dead, real or not real, or otherwise, is purely coincidence. I'm including this note because some people may find the data offensive in some way. I assure you that no offense is intended and that the data is designed to show you what to avoid to keep from offending someone else.

Hands-On Sections

Sanitizing data correctly

The act of sanitizing data is to clean it up before using it so that it doesn't contain things such as PII or unneeded features. In addition, sanitization provides benefits to ML models that shouldn't be ignored.

The example in this section relies on a database that is typical of information obtained from a corporate customer database, combined with an opinion poll. The *Importing and combining the datasets* section of *Chapter 9*, *Defending against Hackers*, shows a similar process where you combine mobility data with COVID statistics. This data combination is a common scenario today where businesses ask people's opinions about everything, but the combined form of the database is completely inappropriate as it performs an analysis of how customers feel about product characteristics. Here are the goals for this analysis (which are likely simplified from what you will encounter in the real world, but work fine here):

- Improve sales by making the produce more visually appealing
- Reduce costs by removing less popular product options
- Add new product options that should sell better than those that have been removed

Now that you have some idea of where the data came from and the goals for using it, it's time to look at the sanitization process in more detail.

Considering the current dataset

The sample dataset consists of a somewhat reduced copy of the customer data from the customer dataset and the actual survey data. The number of records is very small because they're targeted at sanitization, rather than analysis. The customer data includes the following information:

- First name
- Last name
- Primary address
- Age
- Income
- Gender

There is an assumption that the customer hasn't lied about any of the data, but this usually isn't the case. The age field is a number between 0 and 125, with 0 representing a customer who prefers not to provide an age. The income field is one of seven selected ranges:

- 0 to 19,999
- 20,000 to 39,999
- 40,000 to 59,999
- 60,000 to 79,999
- 80,000 to 99,999
- 100,000 to 119,999
- 120,000 and above

A value of 0 indicates that the person is unemployed or wishes not to provide an income value. The company is somewhat aware of gender issues so this particular field can contain M for male, F for

female, X for intersex or non-binary, and N for not willing to say. The survey data consists of the following information:

- Date of survey
- Preferred product color
- Price the customer is willing to pay
- Desire to have Gadget 1 added
- Desire to have Gadget 2 added

The survey allows a limited number of colors: red, yellow, green, blue, or purple. The price value is one of six selections: 159, 169, 179, 189, 199, or 209. By not allowing any fill-in-the-blank entries, performing survey analysis should be easier, but may not reflect a customer's absolute preferences. For example, a customer might feel that 159 is too low, but 169 is too high. With all this in mind, the first step to working with the dataset is to import it. The code shows this process. You can also find the source code for this example in `MLSec; 12; Sanitizing Data.ipynb` in the downloadable source:

```
import pandas as pd
sanitize_df = pd.read_csv("PollData.csv")
print(sanitize_df)
```

All this code does is read the dataset into memory and then print it so you can see what it looks like. *Figure 12.1* shows the output of this code.

```
        Date FirstName   LastName                          PrimaryAddress  Age  \
0  01/15/23    George      Smith          123 Anywhere, Anywhere, WI 59999   44
1  01/15/23     Sally      Jones       123 Somewhere, Somewhere, NV 89503   32
2  01/15/23     Renee     Walker          123 Nowhere, Nowhere, CA 90011   49
3  01/16/23   Saniago  Dominguez       123 Downthere, Downthere, MN 55144   50
4  01/16/23  Abdullah      Brown          123 Upthere, Upthere, FL 33052    0
5  01/16/23    Fenhua       Yang  123 Aroundthere, Aroundthere, NY 10008   89

   Income Gender ProductColor  Price  Gadget1  Gadget2
0   40000      M         Blue    199     True    False
1   80000      F          Red    169    False    False
2   60000      X       Purple    179     True     True
3  100000      N        Green    209     True     True
4   60000      M       Yellow    179    False     True
5   80000      N          Red    199     True     True
```

Figure 12.1 – The unmodified PollData.csv dataset file

Removing PII

Bias can creep into an analysis based on name alone, not to mention where a person lives, their financial status, and so on. In fact, all sorts of data can become an issue when performing analysis, not to mention presenting ethical issues outside of analysis, such as the perception of the people in the dataset. So, the first question you need to ask is whether PII is even needed in your dataset. In many cases, you can simply drop the offending columns. PII is actually categorized into three levels:

- **Personally identifiable**: Enough of the personal information is retained that anyone can identify the person involved. However, at least some of the information is removed to ensure privacy. This is typically not a good solution even for datasets where identification is necessary because the names alone can bias the analysis and the person viewing the result.

- **Re-identifiable**: All of the personal information is removed, but an identifier is added in its place. The identifier is generally a simple number that has no value in and of itself. However, the identifier can be looked up in a separate database later should it become necessary to contact the person. This is the solution normally relied upon for medical research or other datasets where contacting a person later is absolutely necessary. It isn't a good solution for any other purpose.
- **Non-identifiable**: All of the personal information is removed so that the record has no connection whatsoever to the person to whom it applies. This is the solution used for most datasets today because it presents the lowest probability of causing analysis bias or influencing the dataset user. In addition, the removal of PII simplifies the dataset, making it more understandable, and reduces the resources needed for processing and storage.

When removing PII or anything else from an original dataset, store the modified data in a new dataset file and document the process used for modification. The original dataset should be encrypted and stored in a safe location in case the raw data is needed sometime in the future, such as to answer a legal requirement, to provide additional detail, or to recover from an error in manipulating the data. Never modify a raw dataset. If you need to add something to the original data, place it in a new dataset designed for this purpose.

The example dataset has three PII columns: `FirstName`, `LastName`, and `PrimaryAddress`. The company does want to contact the individuals later if there is a question about why they made a certain selection, so it's not possible to eliminate the data, but it also isn't needed for the analysis. The solution is to rely on a re-identifiable setup. The following steps show one technique for making the `sanitize_df` dataset re-identifiable:

1. Create an index for the existing dataset. Nothing fancy is needed; just a simple range will suffice:

```
ids = range(0, len(sanitize_df))
sanitize_df["Id"] = ids
```

2. Copy the PII data to a new dataset and save it to disk. The example assumes that this file is later encrypted and moved to a safe location:

```
saved_df = sanitize_df[[
    "Id", "FirstName",
    "LastName", "PrimaryAddress"]]
print(saved_df)
saved_df.to_csv("SavedData.csv")
```

The output shown in *Figure 12.2* demonstrates that `saved_df` contains all of the PII, plus the `Id` value used to connect `saved_df` with `sanitize_df`.

	Id	FirstName	LastName	PrimaryAddress
0	0	George	Smith	123 Anywhere, Anywhere, WI 59999
1	1	Sally	Jones	123 Somewhere, Somewhere, NV 89503
2	2	Renee	Walker	123 Nowhere, Nowhere, CA 90011
3	3	Saniago	Dominguez	123 Downthere, Downthere, MN 55144
4	4	Abdullah	Brown	123 Upthere, Upthere, FL 33052
5	5	Fenhua	Yang	123 Aroundthere, Aroundthere, NY 10008

Figure 12.2 – The contents of the saved_df dataset.

3. Drop the unneeded data from the `sanitize_df` dataset:

```
sanitize_df.drop([
    "FirstName", "LastName", "PrimaryAddress"],
    axis='columns', inplace=True)
print(sanitize_df)
```

Figure 12.3 shows the result of removing PII from this dataset. As you can see, the records now contain no PII, but it's still possible to re-identify the original creator of the survey data.

	Date	Age	Income	Gender	ProductColor	Price	Gadget1	Gadget2	Id
0	01/15/23	44	40000	M	Blue	199	True	False	0
1	01/15/23	32	80000	F	Red	169	False	False	1
2	01/15/23	49	60000	X	Purple	179	True	True	2
3	01/16/23	50	100000	N	Green	209	True	True	3
4	01/16/23	0	60000	M	Yellow	179	False	True	4
5	01/16/23	89	80000	N	Red	199	True	True	5

Figure 12.3 – The contents of the modified sanitize_df dataset

The result of this process is a smaller dataset with less potential for bias. Developers often worry about data loss in a process such as this, but the example shows that there hasn't been any data loss in this case. It would be possible to completely reconstruct the original data from what is presented now.

Avoiding collecting PII in the first place

Whenever possible, it's better not to collect PII unless you absolutely have to have it for some reason. This also means avoiding things such as collecting date of birth. If you need to know how old someone is to perform your analysis, then collect their age as a numeric value in years. In addition, it's always best practice to provide a means for the person to opt out of providing a certain kind of data, whether you need the data or not. In practice, you can fill the data that hasn't been supplied using the missingness techniques found in the Mitigating dataset corruption section of Chapter 2, Mitigating Risk at Training by Validating and Maintaining Datasets. For example, if an age isn't supplied, you can always use the average of the ages that are supplied to keep the missing value from affecting your analysis. This practice will actually make your dataset easier to use in the long run. Instead of having to calculate the age value using the date of birth, the dataset will contain it directly, saving time, resources, and potential errors.

Adding traits together to make them less identifiable

The essential goal of this section is to make data less identifiable by combining traits. However, this section also talks about other ways in which combining traits can create a better dataset. Adding traits together helps you achieve the following goals:

- Reduces dimensionality, which reduces model complexity
- Makes the dataset less susceptible to bias
- Improves the opacity of data sources
- Decreases training time
- Moderates resource usage
- Enhances security

You could possibly add a few more items to the list, but this is a very good starting point. As with many data manipulation tasks, this one serves multiple purposes in helping you create better datasets, which in turn creates better models. The example dataset has two (possibly three if you include `Gender`) columns that you could combine: `Age` and `Income`.

Depending on your goals, you could combine these two columns in a lot of ways, but the example takes a simple approach. If you place the ages into decades, you could create the following age groups: 1 to 19, 20 to 29, 30 to 39, 40 to 49, 50 to 59, 60 to 69, and 70 and above. There are seven age groups. If you also use the seven income levels, you could combine the two columns into a value between 1 and 49 by assigning a number to each level and viewing them by using `Age` as a starting point:

- 1 to 19: Groups 1 through 7
- 20 to 29: Groups 8 through 14
- 30 to 39: Groups 15 through 21
- 40 to 49: Groups 22 through 28
- 50 to 59: Groups 29 through 35
- 60 to 69: Groups 36 through 42
- 70 and above: Groups 43 through 49

You could also add the groups together or multiply them. The idea is to obtain an amalgamation that describes the groups in a particular way without revealing actual values. The approach used by the example makes it impossible to guess the person's age and their income level is based on a 20,000 range. With this in mind, the following steps show how to combine the traits:

1. Apply the average age to any `Age` column entries with a 0:

```
averageAge = sanitize_df['Age'].mean()
sanitize_df['Age'] = \
    [averageAge if x == 0 else x
    for x in sanitize_df['Age']]
print(sanitize_df)
```

   It might be possible to argue that this step could skew the result, but not having a value at all will skew the result even more. There simply isn't a good solution when someone chooses not to provide input. The result of this step appears in *Figure 12.4*.

	Date	Age	Income	Gender	ProductColor	Price	Gadget1	Gadget2	Id
0	01/15/23	44.0	40000	M	Blue	199	True	False	0
1	01/15/23	32.0	80000	F	Red	169	False	False	1
2	01/15/23	49.0	60000	X	Purple	179	True	True	2
3	01/16/23	50.0	100000	N	Green	209	True	True	3
4	01/16/23	44.0	60000	M	Yellow	179	False	True	4
5	01/16/23	89.0	80000	N	Red	199	True	True	5

Figure 12.4 – The Age column with 0 values replaced with an average value

2. Define a function for calculating the `AgeLevel` value for each record. The values are based on the first age value at each level, so the 20 to 29 age group begins with a value of `8`:

```
def AgeLevel(Age):
    if Age >= 1 and Age <= 19:
        return 1
    elif Age >= 20 and Age <= 29:
        return 8
    elif Age >= 30 and Age <= 39:
        return 15
    elif Age >= 40 and Age <= 49:
        return 22
    elif Age >= 50 and Age <= 59:
        return 29
    elif Age >= 60 and Age <= 69:
        return 36
    elif Age >= 70:
        return 43
```

3. Define a function for calculating the `IncomeLevel` value for each record. The income levels appear as values from `0` to `6` for each age group:

```
def IncomeLevel(Income):
    if Income == 0:
        return 0
    elif Income == 20000:
        return 1
    elif Income == 40000:
        return 2
    elif Income == 60000:
```

```
        return 3
    elif Income == 80000:
        return 4
    elif Income == 100000:
        return 5
    elif Income == 120000:
        return 6
```

4. Define a function for calculating `GroupValue` for each record. To obtain `GroupValue`, it's only necessary to add `AgeLevel` to `IncomeLevel`:

```
def GroupValue(Age = 1, Income = 0):
    Group = AgeLevel(Age) + IncomeLevel(Income)
    return Group
Create a list of group values:
GroupList = []
for Age, Income in \
    zip(sanitize_df['Age'], sanitize_df['Income']):
        GroupList.append(GroupValue(Age, Income))
print(GroupList)
```

This code processes each record in turn and determines the value of `GroupValue` for it. This value is added to the `GroupList` vector for later inclusion in the dataset. *Figure 12.5* shows the output of this step.

```
[24, 19, 25, 34, 25, 47]
```

Figure 12.5 – A vector of GroupValue entries for the dataset

5. Add `GroupList` to the `Group` column of the dataset:

```
sanitize_df['Group'] = GroupList
print(sanitize_df)
```

The `sanitize_df` dataset now has a combined trait column, `Group`, as shown in *Figure 12.6*.

```
        Date   Age  Income Gender ProductColor  Price  Gadget1  Gadget2  Id  \
0  01/15/23  44.0   40000      M         Blue    199     True    False   0
1  01/15/23  32.0   80000      F          Red    169    False    False   1
2  01/15/23  49.0   60000      X       Purple    179     True     True   2
3  01/16/23  50.0  100000      N        Green    209     True     True   3
4  01/16/23  44.0   60000      M       Yellow    179    False     True   4
5  01/16/23  89.0   80000      N          Red    199     True     True   5

   Group
0     24
1     19
2     25
3     34
4     25
5     47
```

Figure 12.6 – The sanitize_df dataset with the Group column

6. Drop the `Age` and `Income` columns:

```
sanitize_df.drop([
    "Age", "Income"],
    axis='columns', inplace=True)
print(sanitize_df)
```

The resulting dataset is now smaller, as shown in *Figure 12.7*, but the essence of the data remains intact so that it's possible to create a robust model. At this point, it would be very tough for someone to trace a particular record back to a specific individual and the resulting model will have a much smaller chance of being biased in any way.

```
        Date Gender ProductColor  Price  Gadget1  Gadget2  Id  Group
0  01/15/23      M         Blue    199     True    False   0     24
1  01/15/23      F          Red    169    False    False   1     19
2  01/15/23      X       Purple    179     True     True   2     25
3  01/16/23      N        Green    209     True     True   3     34
4  01/16/23      M       Yellow    179    False     True   4     25
5  01/16/23      N          Red    199     True     True   5     47
```

Figure 12.7 – The sanitize_df dataset with the Age and Income columns removed

It's possible that you could combine some steps, but using steps of this sort tends to reduce errors in combining the traits and makes the process more transparent. The issue of transparency in data manipulation is at the forefront of many privacy discussions today, so making your code as transparent and easy to follow as possible is going to save time and prevent frustration later.

Eliminating unnecessary features

The final step in sanitizing a dataset is to remove unnecessary features. As presented in *Figure 12.7*, the example dataset no longer contains any PII, but it still retains all of the information that the original had in *Figure 12.1*. This particular step will remove some possibly useful information, but it really isn't necessary in this case.

The idea is to discover which product features people like and including `Gender` in the model could still possibly bias it a little. As a consequence, the following code removes the `Gender` column:

```
sanitize_df.drop(["Gender"],
    axis='columns', inplace=True)
print(sanitize_df)
```

Figure 12.8 shows the final dataset that is fully prepared for the analysis described at the outset of this example. It's now possible to determine which product features are likely to produce more sales. However, if you compare *Figure 12.8* with *Figure 12.1*, you will see that the latest iteration is considerably smaller and easier to work with. Plus, it's easier to explain to anyone outside the project who needs to know about it.

	Date	ProductColor	Price	Gadget1	Gadget2	Id	Group
0	01/15/23	Blue	199	True	False	0	24
1	01/15/23	Red	169	False	False	1	19
2	01/15/23	Purple	179	True	True	2	25
3	01/16/23	Green	209	True	True	3	34
4	01/16/23	Yellow	179	False	True	4	25
5	01/16/23	Red	199	True	True	5	47

Figure 12.8 – The final version of the sanitize_df datase

As a final comment on this transition, the dataset in use now has a considerable number of security features added to it and it wasn't even necessary to program them by hand. For example, it's no longer possible to perpetrate identity theft because there is no PII to obtain. The data itself is less likely to suffer manipulation because of its structure. A hacker could possibly target the `Price` and `Group` columns, but it wouldn't be too hard to detect the manipulation in most cases. Overall, this dataset is a lot more secure for having been sanitized.

Now that you have some ideas on how to sanitize your dataset, it's time to consider just where the dataset comes from. It's essential to think about how data from various sources is collected and the intent behind the collection process. In addition, you need to give credit to data collectors where credit is due.

Links

Generally, if you find a dataset associated with groups such as Kaggle (https://www.kaggle.com/datasets), organizations such as scikit-learn (https://scikit-learn.org/stable/datasets/toy_dataset.html), and institutions of higher learning where students create datasets for thesis work, you can be sure that the dataset is safe to use.

Fairness in Machine Learning – The Case of Juvenile Criminal Justice in Catalonia (https://blog.re-work.co/using-machine-learning-for-criminal-justice/)

Collating Hacked Data Sets (https://www.schneier.com/blog/archives/2020/01/collating_hacke.html

Policing Women: Race and gender disparities in police stops, searches, and use of force (https://www.prisonpolicy.org/blog/2019/05/14/policingwomen/

TensorFlow Model Analysis (**TFMA**) (https://www.tensorflow.org/tfx/model_analysis/get_started

TensorFlow Data Validation (**TFDV**) package (https://www.tensorflow.org/tfx/data_validation/get_started

What-If Tool (https://pair-code.github.io/what-if-tool/)

The *Fairness Indicators* example at https://www.tensorflow.org/tfx/guide/fairness_indicators

The *TensorFlow Constrained Optimization Example Using CelebA Dataset* example at https://www.tensorflow.org/responsible_ai/fairness_indicators/tutorials/Fairness_Indicators_TFCO_CelebA_Case_Study

Building prediction models with grouped data: A case study on the prediction of turnover intention (https://onlinelibrary.wiley.com/doi/full/10.1111/1748-8583.12396

Further reading

The following bullets provide you with some additional reading that you may find useful for understanding the materials in this chapter in greater depth:

- Read additional details about keeping PII out of your dataset: *Data publication: Removing identifiers from data* (https://libguides.library.usyd.edu.au/datapublication/desensitise-data)
- Understand the need for dimensionality reduction: *Machine learning: What is dimensionality reduction?* (https://bdtechtalks.com/2021/05/13/machine-learning-dimensionality-reduction/)
- Locate a dataset that's safe to use for your next experiment: *10 Great Places to Find Free Datasets for Your Next Project* (https://careerfoundry.com/en/blog/data-analytics/where-to-find-free-datasets/)
- Gain more insights into why machine learning applications are unlikely to be fair anytime in the near future: *Can Machine Learning Ever Be "Fair" — and What Does That Even Mean?* (https://towardsdatascience.com/can-machine-learning-ever-be-fair-and-what-does-that-even-mean-b84714dfae7e)
- Discover how even a well-designed model can be unfair: *How We Analyzed the COMPAS Recidivism Algorithm* (https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm)
- Learn about ways to mitigate algorithmic bias: *Algorithmic bias detection and mitigation: Best practices and policies to reduce consumer harms* (https://www.brookings.edu/research/algorithmic-bias-detection-and-mitigation-best-practices-and-policies-to-reduce-consumer-harms/)
- Gain a better understanding of federated learning: *PyGrid: A Peer-To-Peer Platform For Private Data Science And Federated Learning* (https://blog.openmined.org/what-is-pygrid-demo/)

- Get more insights into the use and application of differential privacy: *Harvard University Privacy Tools Project: Differential Privacy* (https://privacytools.seas.harvard.edu/differential-privacy)