6

UNDERSTANDING MACHINE LEARNING–BASED MALWARE DETECTORS

With the open source machine learning tools available today, you can build custom, machine learning–based malware detection tools, whether as your primary detection tool or to supplement commercial solutions, with relatively little effort.

But why build your own machine learning tools when commercial antivirus solutions are already available? When you have access to examples of particular threats, such as malware used by a certain group of attackers targeting your network, building your own machine learning–based detection technologies can allow you to catch new examples of these threats.

In contrast, commercial antivirus engines might miss these threats unless they already include signatures for them. Commercial tools are also "closed books"—that is, we don't necessarily know how they work and we have limited ability to tune them. When we build our own detection methods, we know how they work and can tune them to our liking to reduce false positives or false negatives. This is helpful because in some applications you might be willing to tolerate more false positives in exchange for fewer false negatives

(for example, when you're searching your network for suspicious files so that you can hand-inspect them to determine if they are malicious), and in other applications you might be willing to tolerate more false negatives in exchange for fewer false positives (for example, if your application blocks programs from executing if it determines they are malicious, meaning that false positives are disruptive to users).

In this chapter, you learn the process of developing your own detection tools at a high level. I start by explaining the big ideas behind machine learning, including feature spaces, decision boundaries, training data, underfitting, and overfitting. Then I focus on four foundational approaches—logistic regression, k-nearest neighbors, decision trees, and random forest—and how these can be applied to perform detection.

You'll then use what you learned in this chapter to learn how to evaluate the accuracy of machine learning systems in Chapter 7 and implement machine learning systems in Python in Chapter 8. Let's get started.

Steps for Building a Machine Learning–Based Detector

There is a fundamental difference between machine learning and other kinds of computer algorithms. Whereas traditional algorithms tell the computer what to do, machine-learning systems learn how to solve a problem by example. For instance, rather than simply pulling from a set of preconfigured rules, machine learning security detection systems can be trained to determine whether a file is bad or good by learning from examples of good and bad files.

The promise of machine learning systems for computer security is that they automate the work of creating signatures, and they have the potential to perform more accurately than signature-based approaches to malware detection, especially on new, previously unseen malware.

Essentially, the workflow we follow to build any machine learning–based detector, including a decision tree, boils down to these steps:

1. **Collect** examples of malware and benignware. We will use these examples (called *training examples*) to train the machine learning system to recognize malware.
2. **Extract** features from each training example to represent the example as an array of numbers. This step also includes research to design good features that will help your machine learning system make accurate inferences.
3. **Train** the machine learning system to recognize malware using the features we have extracted.
4. **Test** the approach on some data not included in our training examples to see how well our detection system works.

Let's discuss each of these steps in more detail in the following sections.

Gathering Training Examples

Machine learning detectors live or die by the training data provided to them. Your malware detector's ability to recognize suspicious binaries depends heavily on the quantity and quality of training examples you provide. Be prepared to spend much of your time gathering training examples when building machine learning–based detectors, because the more examples you feed your system, the more accurate it's likely to be.

The quality of your training examples is also important. The malware and benignware you collect should mirror the kind of malware and benignware you expect your detector to see when you ask it to decide whether new files are malicious or benign.

For example, if you want to detect malware from a specific threat actor group, you must collect as much malware as possible from that group for use in training your system. If your goal is to detect a broad class of malware (such as ransomware), it's essential to collect as many representative samples of this class as possible.

By the same token, the benign training examples you feed your system should mirror the kinds of benign files you will ask your detector to analyze once you deploy it. For example, if you are working on detecting malware on a university network, you should train your system with a broad sampling of the benignware that students and university employees use, in order to avoid false positives. These benign examples would include computer games, document editors, custom software written by the university IT department, and other types of nonmalicious programs.

To give a real-world example, at my current day job, we built a detector that detects malicious Office documents. We spent about half the time on this project gathering training data, and this included collecting benign documents generated by more than a thousand of my company's employees. Using these examples to train our system significantly reduced our false positive rate.

Extracting Features

To classify files as good or bad, we train machine learning systems by showing them features of software binaries; these are file attributes that will help the system distinguish between good and bad files. For example, here are some features we might use to determine whether a file is good or bad:

- Whether it's digitally signed
- The presence of malformed headers
- The presence of encrypted data
- Whether it has been seen on more than 100 network workstations

To obtain these features, we need to extract them from files. For example, we might write code to determine whether a file is digitally signed, has malformed headers, contains encrypted data, and so on.

Often, in security data science, we use a huge number of features in our machine learning detectors. For example, we might create a feature for every library call in the Win32 API, such that a binary would have that feature if it had the corresponding API call. We'll revisit feature extraction in Chapter 8, where we discuss more advanced feature extraction concepts as well as how to use them to implement machine learning systems in Python.

Designing Good Features

Our goal should be to select features that yield the most accurate results. This section provides some general rules to follow.

First, when selecting features, choose ones that represent your best guess as to what might help a machine learning system distinguish bad files from good files. For example, the feature "contains encrypted data" might be a good marker for malware because we know that malware often contains encrypted data, and we're guessing that benignware will contain encrypted data more rarely. The beauty of machine learning is that if this hypothesis is wrong, and benignware contains encrypted data just as often as malware does, the system will more or less ignore this feature. If our hypothesis is right, the system will learn to use the "contains encrypted data" feature to detect malware.

Second, don't use so many features that your set of features becomes too large relative to the number of training examples for your detection system. This is what the machine learning experts call "the curse of dimensionality." For example, if you have a thousand features and only a thousand training examples, chances are you don't have enough training examples to teach your machine learning system what each feature actually says about a given binary. Statistics tells us that it's better to give your system a few features relative to the number of training examples you have available and let it form well-founded beliefs about which features truly indicate malware.

Finally, make sure your features represent a range of hypotheses about what constitutes malware or benignware. For example, you may choose to build features related to encryption, such as whether a file uses encryption-related API calls or a public key infrastructure (PKI), but make sure to also use features unrelated to encryption to hedge your bets. That way, if your system fails to detect malware based on one type of feature, it might still detect it using other features.

Training Machine Learning Systems

After you've extracted features from your training binaries, it's time to train your machine learning system. What this looks like algorithmically depends completely on the machine learning approach you're using. For example, training a decision tree approach (which we discuss shortly) involves a different learning algorithm than training a logistic regression approach (which we also discuss).

Fortunately, all machine learning detectors provide the same basic interface. You provide them with training data that contains features from sample binaries, as well as corresponding labels that tell the algorithm

which binaries are malware and which are benignware. Then the algorithms learn to determine whether or not new, previously unseen binaries are malicious or benign. We cover training in more detail later in this chapter.

NOTE *In this book, we focus on a class of machine learning algorithms known as* supervised machine learning algorithms. *To train models using these algorithms, we tell them which examples are malicious and which are benign. Another class of machine learning algorithms,* unsupervised algorithms*, does not require us to know which examples are malicious or benign in our training set. These algorithms are much less effective at detecting malicious software and malicious behavior, and we will not cover them in this book.*

Testing Machine Learning Systems

Once you've trained your machine learning system, you need to check how accurate it is. You do this by running the trained system on data that you didn't train it on and seeing how well it determines whether or not the binaries are malicious or benign. In security, we typically train our systems on binaries that we gathered up to some point in time, and then we test on binaries that we saw *after* that point in time, to measure how well our systems will detect new malware, and to measure how well our systems will avoid producing false positives on new benignware. Most machine learning research involves thousands of iterations that go something like this: we create a machine learning system, test it, and then tweak it, train it again, and test it again, until we're satisfied with the results. I'll cover testing machine learning systems in detail in Chapter 8.

Let's now discuss how a variety of machine learning algorithms work. This is the hard part of the chapter, but also the most rewarding if you take the time to understand it. In this discussion, I talk about the unifying ideas that underlie these algorithms and then move on to each algorithm in detail.

Understanding Feature Spaces and Decision Boundaries

Two simple geometric ideas can help you understand all machine learning–based detection algorithms: the idea of a geometrical feature space and the idea of a decision boundary. A *feature space* is the geometrical space defined by the features you've selected, and a *decision boundary* is a geometrical structure running through this space such that binaries on one side of this boundary are defined as malware, and binaries on the other side of the boundary are defined as benignware. When we use a machine learning algorithm to classify files as malicious or benign, we extract features so that we can place the samples in the feature space, and then we check which side of the decision boundary the samples are on to determine whether the files are malware or benignware.

This geometrical way of understanding feature spaces and decision boundaries is accurate for systems that operate on feature spaces of one, two, or three dimensions (features), but it also holds for feature spaces with

millions of dimensions, even though it's impossible to visualize or conceive of million-dimensional spaces. We'll stick to examples with two dimensions in this chapter to make them easy to visualize, but just remember that real-world security machine learning systems pretty much always use hundreds, thousands, or millions of dimensions, and the basic concepts we discuss in a two-dimensional context hold for real-world systems that have more than two dimensions.

Let's create a toy malware detection problem to clarify the idea of a decision boundary in a feature space. Suppose we have a training dataset consisting of malware and benignware samples. Now suppose we extract the following two features from each binary: the percentage of the file that appears to be compressed, and the number of suspicious functions each binary imports. We can visualize our training dataset as shown in Figure 6-1 (bear in mind I created the data in the plot artificially, for example purposes).

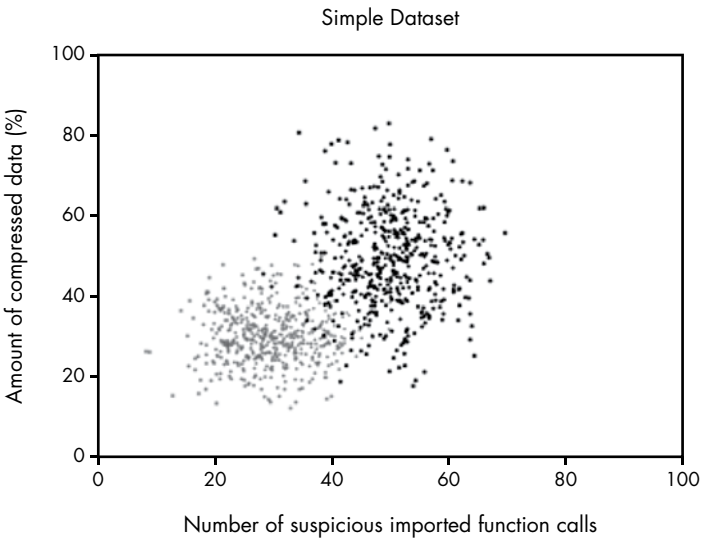

Figure 6-1: A plot of a sample dataset we'll use in this chapter, where gray dots are benignware and black dots are malware

The two-dimensional space shown in Figure 6-1, which is defined by our two features, is the feature space for our sample dataset. You can see a clear pattern in which the black dots (the malware) are generally in the upper-right part of the space. In general, these have more suspicious imported function calls and more compressed data than the benignware, which mostly inhabits the lower-left part of the plot. Suppose, after viewing this plot, you were asked to create a malware detection system based solely on the two features we're using here. It seems clear that, based on the data, you can formulate the following rule: if a binary has both a lot of compressed data and a lot of suspicious imported function calls, it's malware, and if it has neither a lot of suspicious imported calls nor much compressed data, it's benignware.

In geometrical terms, we can visualize this rule as a diagonal line that separates the malware samples from the benignware samples in the feature space, such that binaries with sufficient compressed data and imported function calls (defined as malware) are above the line, and the rest of the binaries (defined as benignware) are below the line. Figure 6-2 shows such a line, which we call a decision boundary.

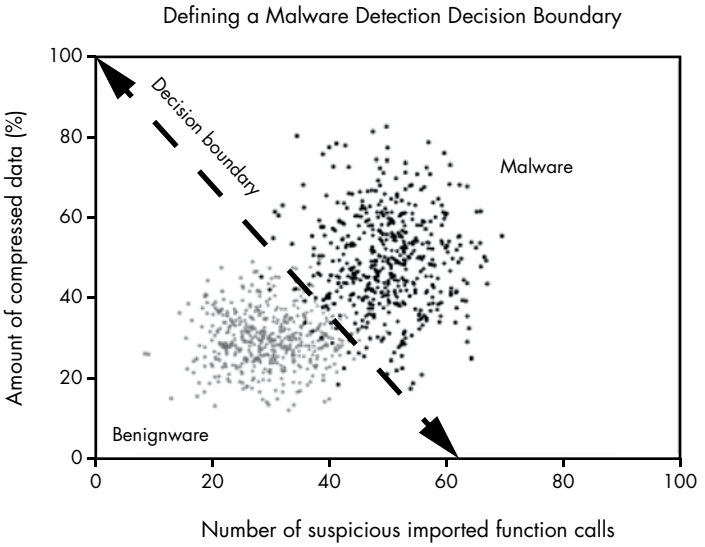

Figure 6-2: A decision boundary drawn through our sample dataset, which defines a rule for detecting malware

As you can see from the line, *most* of the black (malware) dots are on one side of the boundary, and *most* of the gray (benignware) samples are on the other side of the decision boundary. Note that it's impossible to draw a line that separates *all* of the samples from one another, because the black and gray clouds in this dataset overlap one another. But from looking at this example, it appears we've drawn a line that will correctly classify new malware samples and benignware samples in most cases, assuming they follow the pattern seen in the data in this image.

In Figure 6-2, we manually drew a decision boundary through our data. But what if we want a more exact decision boundary and want to do it in an automated way? This is exactly what machine learning does. In other words, all machine learning detection algorithms look at data and use an automated process to determine the ideal decision boundary, such that there's the greatest chance of correctly performing detection on new, previously unseen data.

Let's look at the way a real-world, commonly used machine learning algorithm identifies a decision boundary within the sample data shown in Figure 6-3. This example uses an algorithm called logistic regression.

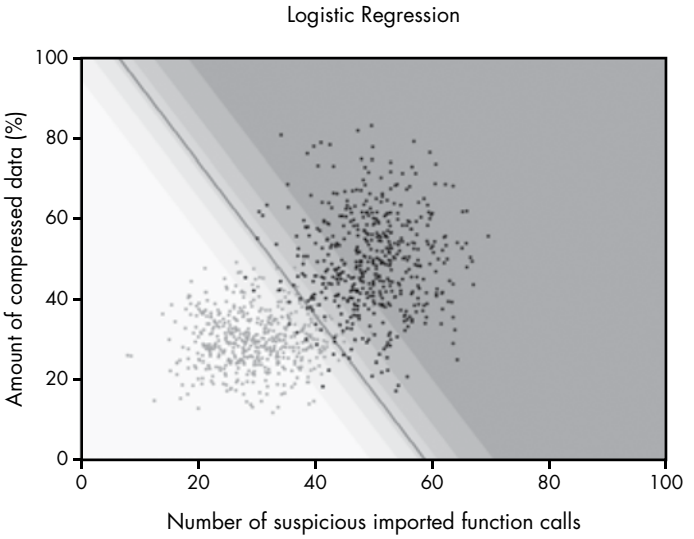

Figure 6-3: The decision boundary automatically created by training a logistic regression model

Notice that we're using the same sample data we used in the previous plots, where gray dots are benignware and black dots are malware. The line running through the center of the plot is the decision boundary that the logistic regression algorithm *learns* by looking at the data. On the right side of the line, the logistic regression algorithm assigns a greater than 50 percent probability that binaries are malware, and on the left side of the line, it assigns a less than 50 percent probability that a binary is malware.

Now note the shaded regions of the plot. The dark gray shaded region is the region where the logistic regression model is highly confident that files are malware. Any new file the logistic regression model sees that has features that land in this region should have a high probability of being malware. As we get closer and closer to the decision boundary, the model has less and less confidence about whether or not binaries are malware or benignware. Logistic regression allows us to easily move the line up into the darker region or down into the lighter region, depending on how aggressive we want to be about detecting malware. For example, if we move it down, we'll catch more malware, but get more false positives. If we move it up, we'll catch less malware, but get fewer false positives.

I want to emphasize that logistic regression, and all other machine learning algorithms, can operate in arbitrarily high dimensional feature spaces. Figure 6-4 illustrates how logistic regression works in a slightly higher dimensional feature space.

In this higher-dimensional space, the decision boundary is not a line, but a *plane* separating the points in the 3D volume. If we were to move to four or more dimensions, logistic regression would create a *hyperplane*, which is an *n*-dimensional plane-like structure that separates the malware from benignware points in this high dimensional space.

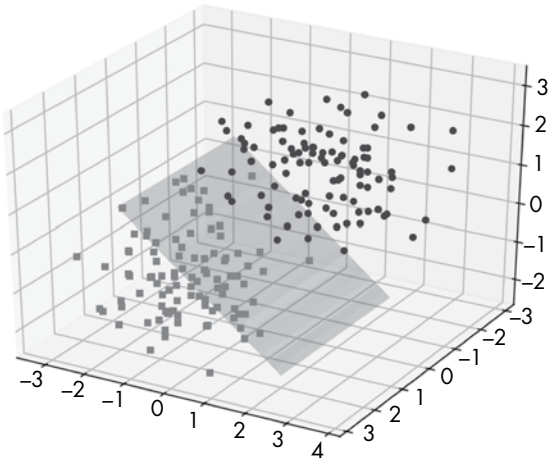

Figure 6-4: A planar decision boundary through a hypothetical three dimensional feature space created by logistic regression

Because logistic regression is a relatively simple machine learning algorithm, it can only create simple geometrical decision boundaries such as lines, planes, and higher dimensional planes. Other machine learning algorithms can create decision boundaries that are more complex. Consider, for example, the decision boundary shown in Figure 6-5, given by the k-nearest neighbors algorithm (which I discuss in detail shortly).

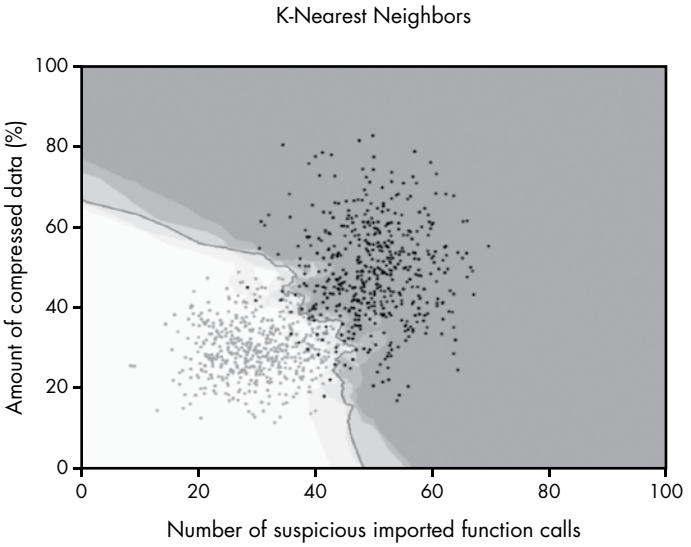

Figure 6-5: A decision boundary created by the k-nearest neighbors algorithm

As you can see, this decision boundary isn't a plane: it's a highly irregular structure. Also note that some machine learning algorithms can generate

disjointed decision boundaries, which define some regions of the feature space as malicious and some regions as benign, even if those regions are not contiguous. Figure 6-6 shows a decision boundary with this irregular structure, using a different sample dataset with a more complex pattern of malware and benignware in our sample feature space.

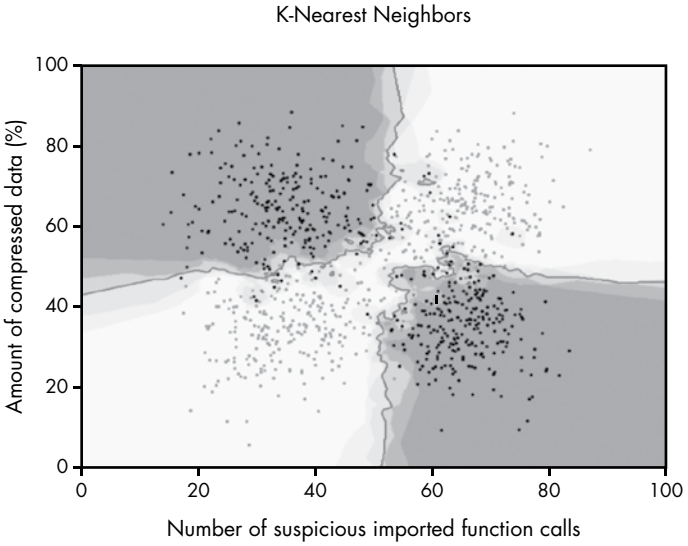

Figure 6-6: A disjoint decision boundary created by the k-nearest neighbors algorithm

Even though the decision boundary is noncontiguous, it's still common machine learning parlance to call these disjoint decision boundaries simply "decision boundaries." You can use different machine learning algorithms to express different types of decision boundaries, and this difference in expressivity is why we might pick one machine learning algorithm over another for a given project.

Now that we've explored core machine learning concepts like feature spaces and decision boundaries, let's discuss what machine learning practitioners call overfitting and underfitting next.

What Makes Models Good or Bad: Overfitting and Underfitting

I can't overemphasize the importance of overfitting and underfitting in machine learning. Avoiding both cases is what defines a good machine learning algorithm. Good, accurate detection models in machine learning capture the general trend in what the training data says about what distinguishes malware from benignware, without getting distracted by the outliers or the exceptions that prove the rule.

Underfit models ignore outliers but fail to capture the general trend, resulting in poor accuracy on new, previously unseen binaries. Overfit models get distracted by outliers in ways that don't reflect the general trend, and they yield poor accuracy on previously unseen binaries. Building machine learning malware detection models is all about capturing the general trend that distinguishes the malicious from the benign.

Let's use the examples of underfit, well fit, and overfit models in Figures 6-7, 6-8, and 6-9 to illustrate these terms. Figure 6-7 shows an underfit model.

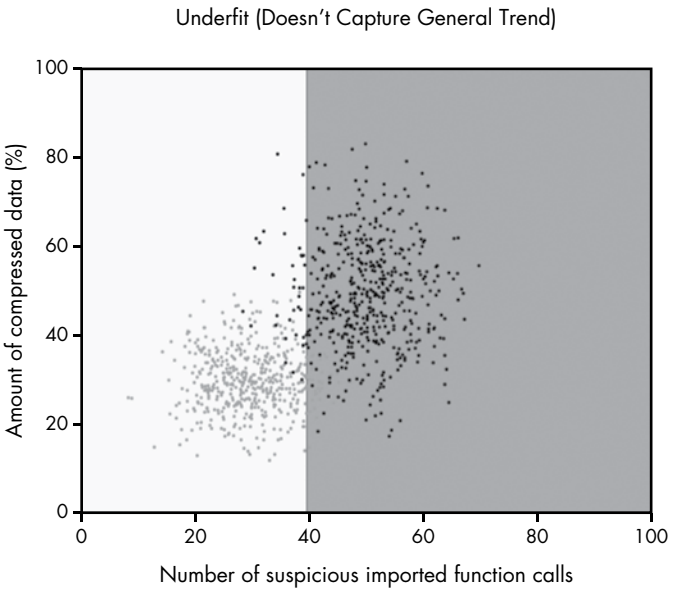

Figure 6-7: An underfit machine learning model

Here, you can see the black dots (malware) cluster in the upper-right region of the plot, and the gray dots (benignware) cluster in the lower left. However, our machine learning model simply slices the dots down the middle, crudely separating the data without capturing the diagonal trend. Because the model doesn't capture the general trend, we say that it is underfit.

Also note that there are only two shades of certainty that the model gives in all of the regions of the plot: either the shade is dark gray or it's white. In other words, the model is either absolutely certain that points in the feature space are malicious or absolutely certain they're benign. This inability to express certainty correctly is also a reason this model is underfit.

Let's contrast the underfit model in Figure 6-7 with the well-fit model in Figure 6-8.

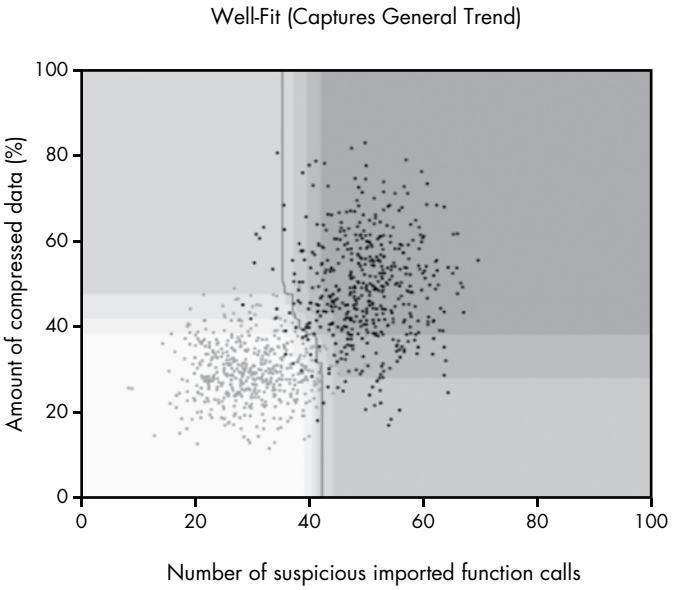

Figure 6-8: A well-fit machine learning model

In this case, the model not only captures the general trend in the data but also creates a reasonable model of certainty with respect to its estimate of which regions of the feature space are definitely malicious, definitely benign, or are in a gray area.

Note the decision line running from the top to the bottom of this plot. The model has a simple theory about what divides the malware from the benignware: a vertical line with a diagonal notch in the middle of the plot. Also note the shaded regions in the plot, which tells us that the model is only certain that data in the upper-right part of the plot are malware, and only certain that binaries in the lower-left corner of the plot are benignware.

Finally, let's contrast the overfit model shown next in Figure 6-9 to the underfit model you saw in Figure 6-7 as well as the well-fit model in Figure 6-8.

The overfit model in Figure 6-9 fails to capture the general trend in the data. Instead, it obsesses over the exceptions in the data, including the handful of black dots (malware training examples) that occur in the cluster of gray dots (benign training examples) and draws decision boundaries around them. Similarly, it focuses on the handful of benignware examples that occur in the malware cluster, drawing boundaries around those as well.

This means that when we see new, previously unseen binaries that happen to have features that place them close to these outliers, the machine learning model will think they are malware when they are almost definitely benignware, and vice versa. In practice, this means that this model won't be as accurate as it could be.

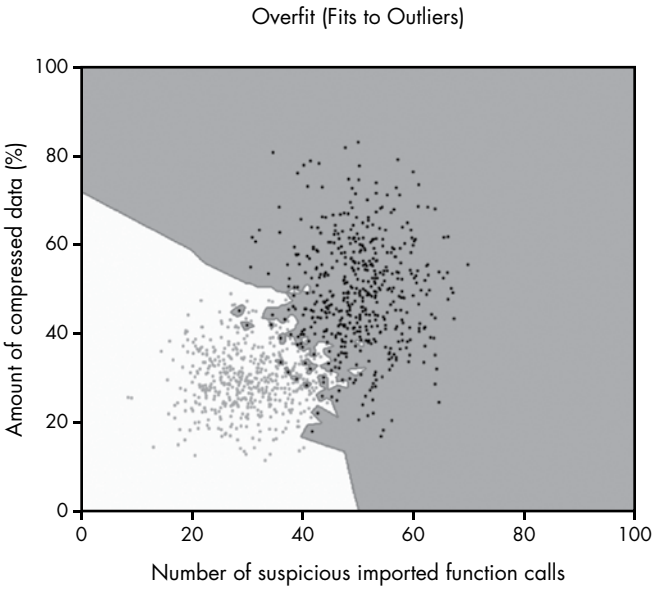

Figure 6-9: An overfit machine learning model

Major Types of Machine Learning Algorithms

So far I've discussed machine learning in very general terms, touching on two machine learning methods: logistic regression and k-nearest neighbors. In the remainder of this chapter, we delve deeper and discuss logistic regression, k-nearest neighbors, decision trees, and random forest algorithms in more detail. We use these algorithms quite often in the security data science community. These algorithms are complex, but the ideas behind them are intuitive and straightforward.

First, let's look at the sample datasets we use to explore the strengths and weaknesses of each algorithm, shown in Figure 6-10.

I created these datasets for example purposes. On the left, we have our simple dataset, which I've already used in Figures 6-7, 6-8, and 6-9. In this case, we can separate the black training examples (malware) from the gray training examples (benignware) using a simple geometric structure such as a line.

The dataset on the right, which I've already shown in Figure 6-6, is complex because we can't separate the malware from the benignware using a simple line. But there is still a clear pattern to the data: we just have to use more complex methods to create a decision boundary. Let's see how different algorithms perform with these two sample datasets.

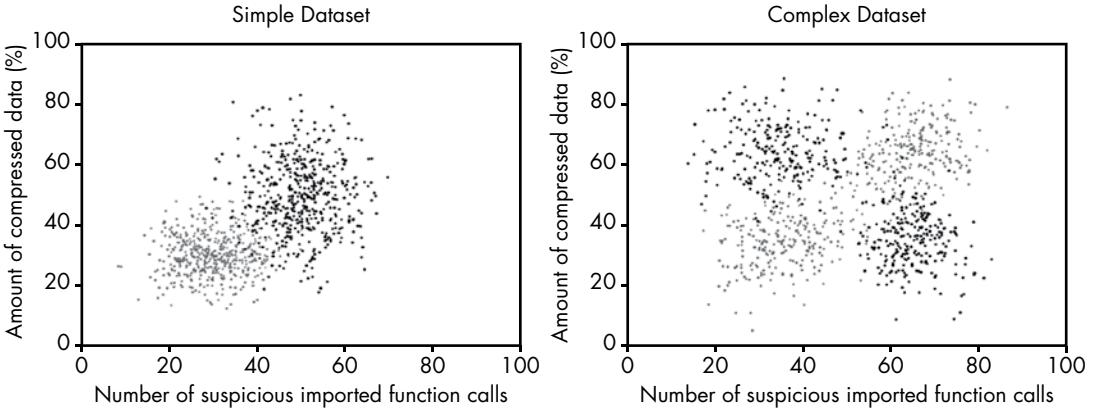

Figure 6-10: The two sample datasets we use in this chapter, with black dots representing malware and gray dots representing benignware

Logistic Regression

As you learned previously, logistic regression is a machine learning algorithm that creates a line, plane, or hyperplane (depending on how many features you provide) that geometrically separates your training malware from your training benignware. When you use the trained model to detect new malware, logistic regression checks whether a previously unseen binary is on the malware side or the benignware side of the boundary to determine whether it's malicious or benign.

A limitation of logistic regression is that if your data can't be separated simply using a line or hyperplane, logistic regression is not the right solution. Whether or not you can use logistic regression for your problem depends on your data and your features. For example, if your problem has lots of individual features that on their own are strong indicators of maliciousness (or "benignness"), then logistic regression might be a winning approach. If your data is such that you need to use complex relationships between features to decide that a file is malware, then another approach, like k-nearest neighbors, decision trees, or random forest, might make more sense.

To illustrate the strengths and weaknesses of logistic regression, let's look at the performance of logistic regression on our two sample datasets, as shown in Figure 6-11. We see that logistic regression yields a very effective separation of the malware and benignware in our simple dataset (on the left). In contrast, the performance of logistic regression on our complex dataset (on the right) is not effective. In this case, the logistic regression algorithm gets confused, because it can only express a linear decision boundary. You can see both binary types on both sides of the line, and the shaded gray confidence bands don't really make any sense relative to the data. For this more complex dataset, we'd need to use an algorithm capable of expressing more geometric structures.

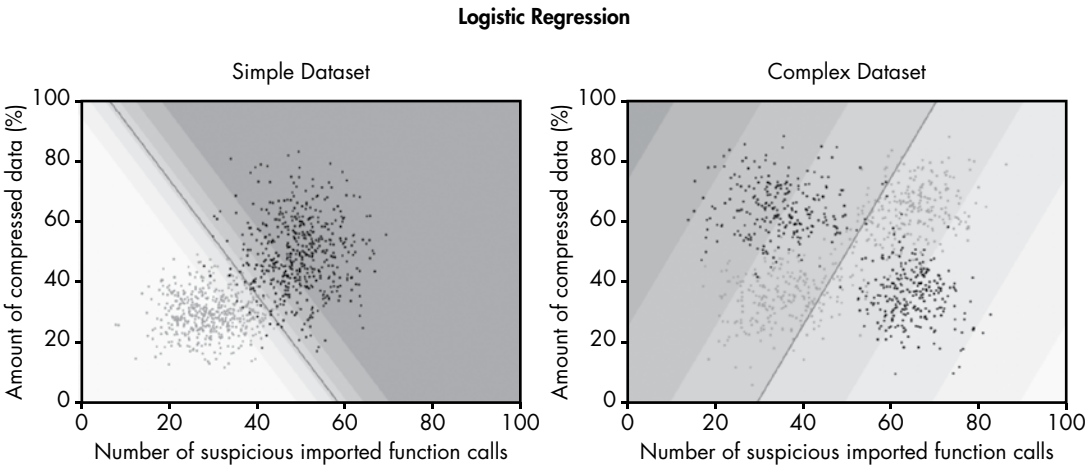

Figure 6-11: A decision boundary drawn through our sample datasets using logistic regression

The Math Behind Logistic Regression

Let's now look at the math behind how logistic regression detects malware samples. Listing 6-1 shows Pythonic pseudocode for computing the probability that a binary is malware using logistic regression.

```
def logistic_regression(compressed_data, suspicious_calls, learned_parameters): ❶
compressed_data = compressed_data * learned_parameters["compressed_data_weight"] ❷
    suspicious_calls = suspicious_calls * learned_parameters["suspicious_calls_weight"]
score = compressed_data + suspicious_calls + bias ❸
    return logistic_function(score)

def logistic_function(score): ❹
    return 1/(1.0+math.e**(-score))
```

Listing 6-1: Pseudocode using logistic regression to calculate probability

Let's step through the code to understand what this means. We first define the `logistic_regression` function ❶ and its parameters. Its parameters are the features of the binary (`compressed_data` and `suspicious_calls`) that represent the amount of compressed data and the number of suspicious calls it makes, respectively, and the parameter `learned_parameters` stands for the elements of the logistic regression function that were learned by training the logistic regression model on training data. I discuss how the parameters were learned later in this chapter; for now, just accept that they were derived from the training data.

Then, we take the `compressed_data` feature ❷ and multiply it by the `compressed_data_weight` parameter. This weight scales the feature up or down, depending on how indicative of malware the logistic regression function thinks this feature is. Note that the weight can also be negative, which indicates that the logistic regression model thinks that the feature is an indicator of a file being benign.

On the line below that, we perform the same step for the `suspicious_calls` parameter. Then, we add these two weighted features together ❸, plus add in a parameter called the `bias` parameter (also learned from training data). In sum, we take the `compressed_data` feature, scaled by how indicative of maliciousness we believe it to be, add the `suspicious_calls` feature, also scaled by how indicative of maliciousness we believe it to be, and add the `bias` parameter, which indicates how suspicious the logistic regression model thinks we should be of files in general. The result of these additions and multiplications is a `score` indicating how likely it is that a given file is malicious.

Finally, we use `logistic_function` ❹ to convert our suspiciousness score into a probability. Figure 6-12 visualizes how this function works.

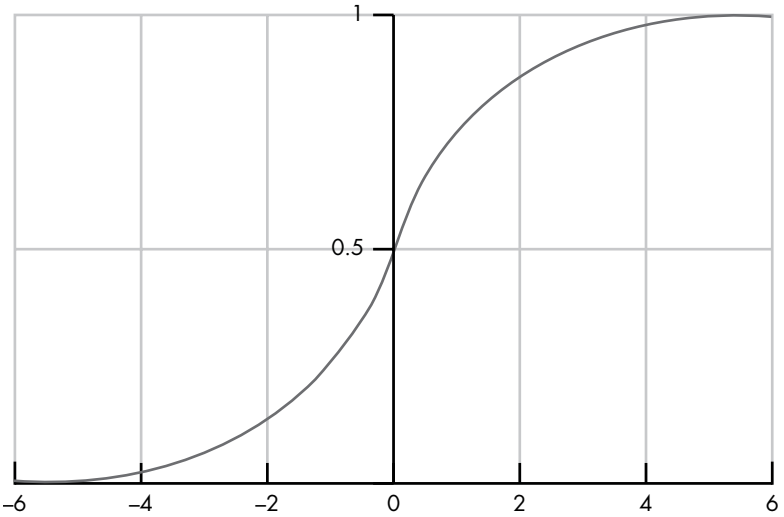

Figure 6-12: A plot of the logistic function used in logistic regression

Here, the logistic function takes a score (shown on the x-axis) and translates it into a value that's bounded between 0 and 1 (a probability).

How the Math Works

Let's return to the decision boundaries you saw in Figure 6-11 to see how this math works in practice. Recall how we compute our probability:

```
logistic_function(feature1_weight * feature1 + feature2_weight*feature2 + bias)
```

For example, if we were to plot the resulting probabilities at every point in the feature spaces shown in Figure 6-11 using the same feature weights and `bias` parameter, we'd wind up with the shaded regions shown in the same figure, which shows where the model "thinks" malicious and benign samples lie, and with how much confidence.

If we were then to set a threshold of 0.5 (recall that at a probability of greater than 50 percent, files are defined as malicious), the line in

Figure 6-11 would appear as our decision boundary. I encourage you to experiment with my sample code, plug in some feature weights and a bias term, and try it yourself.

NOTE *Logistic regression doesn't constrain us to using only two features. In reality, we usually use scores or hundreds or even thousands of features with logistic regression. But the math doesn't change: we just compute our probability as follows for any number of features:*

```
logistic_function(feature1 * feature1_weight + feature2 * feature2_weight +
feature3 * feature3_weight ... + bias)
```

So how exactly does logistic regression learn to place the decision boundary in the right place based on the training data? It uses an iterative, calculus-based approach called *gradient descent*. We won't get into the details of this approach in this book, but the basic idea is that the line, plane, or hyperplane (depending on the number of features you're using) is iteratively adjusted such that it maximizes the probability that the logistic regression model will get the answer right when asked if a data point in the training set is either a malware sample or a benignware sample.

You can train logistic regression models to bias the logistic regression learning algorithm toward coming up with simpler or more complex theories about what constitutes malware and benignware. These training methods are beyond the scope of this book, but if you're interested in learning about these helpful methods, I encourage you to Google "logistic regression and regularization" and read explanations of them online.

When to Use Logistic Regression

Logistic regression has distinct advantages and disadvantages relative to other machine learning algorithms. An advantage of logistic regression is that one can easily interpret what a logistic regression model thinks constitutes benignware and malware. For example, we can understand a given logistic regression model by looking at its feature weights. Features that have high weight are those the model interprets as malicious. Features with negative weight are those the model believes are benignware. Logistic regression is a fairly simple approach, and when the data you're working with contains clear indicators of maliciousness, it can work well. But when the data is more complex, logistic regression often fails.

Now let's explore another simple machine learning approach that can express much more complex decision boundaries: k-nearest neighbors.

K-Nearest Neighbors

K-nearest neighbors is a machine learning algorithm based on the idea that if a binary in the feature space is close to other binaries that are malicious, then it's malicious, and if its features place it close to binaries that are benign, it must be benign. More precisely, if the majority of the

k closest binaries to an unknown binary are malicious, the file is malicious. Note that *k* represents the number of nearby neighbors that we pick and define ourselves, depending on how many neighbors we think should be involved in determining whether a sample is benign or malicious.

In the real world, this makes intuitive sense. For example, if you have a dataset of weights and heights of both basketball players and table tennis players, chances are that the basketball players' weights and heights are likely closer to one another than they are to the measurements of table tennis players. Similarly, in a security setting, malware will often have similar features to other malware, and benignware will often have similar features to other benignware.

We can translate this idea into a k-nearest neighbors algorithm to compute whether a binary is malicious or benign using the following steps:

1. Extract the binary's features and find the *k* samples that are closest to it in the feature space.
2. Divide the number of malware samples that are close to the sample by *k* to get the percentage of nearest neighbors that are malicious.
3. If enough of the samples are malicious, define the sample as malicious.

Figure 6-13 shows how k-nearest neighbors algorithm works at a high level.

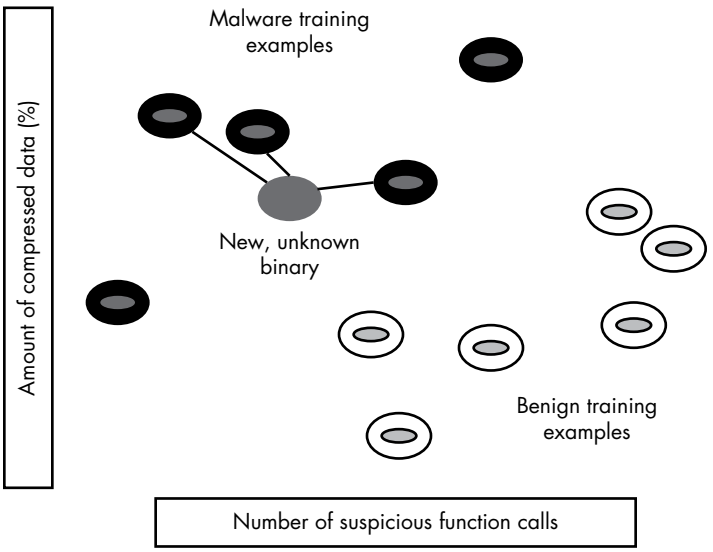

Figure 6-13: An illustration of the way k-nearest neighbors can be used to detect previously unseen malware

We see a set of malware training examples in the upper left and a set of benignware examples in the lower right. We also see a new, unknown binary that is connected to its three nearest neighbors. In this case, we've set *k* to 3,

meaning we're looking at the three nearest neighbors to unknown binaries. Because all three of the nearest neighbors are malicious, we'd classify this new binary as malicious.

The Math Behind K-Nearest Neighbors

Let's now discuss the math that allows us to compute the distance between new, unknown binaries' features and the samples in the training set. We use a *distance function* to do this, which tells us the distance between our new example and the examples in the training set. The most common distance function is *Euclidean distance*, which is the length of the shortest path between two points in our feature space. Listing 6-2 shows pseudocode for Euclidean distance in our sample two-dimensional feature space.

```
import math
def euclidean_distance(compression1,suspicious_calls1, compression2, suspicious_calls2): ❶
    comp_distance = (compression1-compression2)**2 ❷
    call_distance = (suspicious_calls1-suspicious_calls2)**2 ❸
    return math.sqrt(comp_distance + call_distance) ❹
```

Listing 6-2: Pseudocode for writing the `euclidean_distance` function

Let's walk through how the math in this code works. Listing 6-2 takes a pair of samples and computes the distance between them based on the differences between their features. First, the caller passes in the features of the binaries ❶, where `compression1` is the compression feature of the first example, `suspicious_calls1` is the `suspicious_calls` feature of the first example, `compression2` is the compression feature of the second example, and `suspicious_calls2` is the suspicious calls feature of the second example.

Then we compute the squared difference between the compression features of each sample ❷, and we compute the squared difference between the suspicious calls feature of each sample ❸. We won't cover the reason we use squared distance, but note that the resulting difference is always positive. Finally, we compute the square root of the two differences, which is the Euclidean distance between the two feature vectors, and return it to the caller ❹. Although there are other ways to compute distances between examples, Euclidean distance is the most commonly used with the k-nearest neighbors algorithm, and it works well for security data science problems.

Choosing the Number of Neighbors That Vote

Let's now look at the kinds of decision boundaries and probabilities that a k-nearest neighbors algorithm produces for the sample datasets we're using in this chapter. In Figure 6-14, I set k to 5, thus allowing five closest neighbors to "vote."

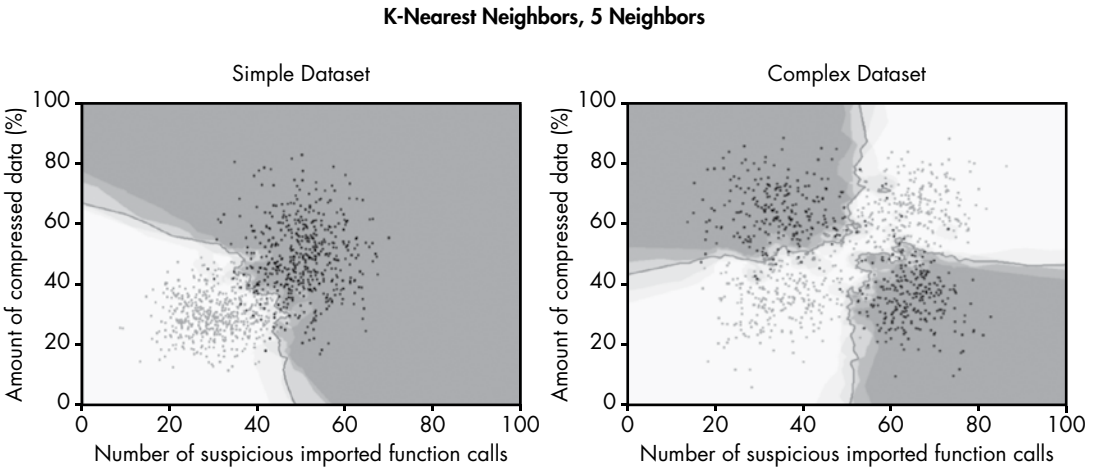

Figure 6-14: The decision boundaries created by k-nearest neighbors when k *is set to 5*

But in Figure 6-15, I set k to 50, allowing the 50 closest neighbors to "vote."

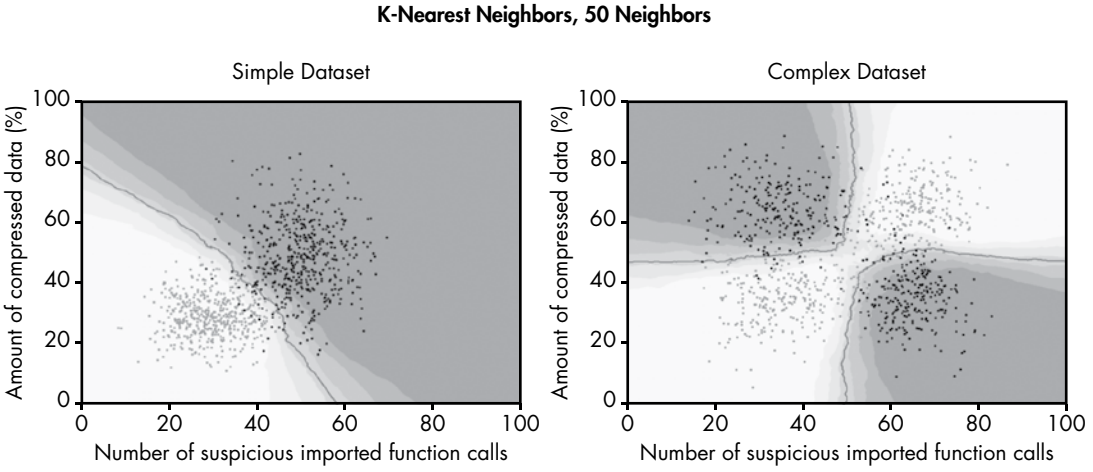

Figure 6-15: The decision boundaries created by k-nearest neighbors when k *is set to 50*

Note the dramatic difference between the models depending on the number of neighbors that vote. The model in Figure 6-14 shows a gnarly, complex decision boundary for both datasets, which is overfit in the sense that it draws local decision boundaries around outliers, but underfit because it fails to capture the simple, general trends. In contrast, the model in Figure 6-15 is well-fit to both datasets, because it doesn't get distracted by outliers and cleanly identifies general trends.

As you can see, k-nearest neighbors can produce a much more complex decision boundary than logistic regression. We can control the complexity of this boundary to guard against both over- and underfitting by changing k, the number of neighbors that get to vote on whether a sample is

malicious or benign. Whereas the logistic regression model in Figure 6-11 got it completely wrong, k-nearest neighbors does well at separating the malware from the benignware, especially when we let 50 neighbors vote. Because k-nearest neighbors is not constrained by a linear structure and is simply looking at the nearest neighbors of each point to make a decision, it can create decision boundaries with arbitrary shapes, thus modeling complex datasets much more effectively.

When to Use K-Nearest Neighbors

K-nearest neighbors is a good algorithm to consider when you have data where features don't map cleanly onto the concept of suspiciousness, but closeness to malicious samples is a strong indicator of maliciousness. For example, if you're trying to classify malware into families that share code, k-nearest neighbors might be a good algorithm to try, because you want to classify a malware sample into a family if its features are similar to known members of a given family.

Another reason to use k-nearest neighbors is that it provides clear explanations of *why* it has made a given classification decision. In other words, it's easy to identify and compare similarities between samples and an unknown sample to figure out why the algorithm has classified it as malware or benignware.

Decision Trees

Decision trees are another frequently used machine learning method for solving detection problems. Decision trees automatically generate a series of questions through a training process to decide whether or not a given binary is malware, similar to the game Twenty Questions. Figure 6-16 shows a decision tree that I automatically generated by training it on the simple dataset we've been using in this chapter. Let's follow the flow of the logic in the tree.

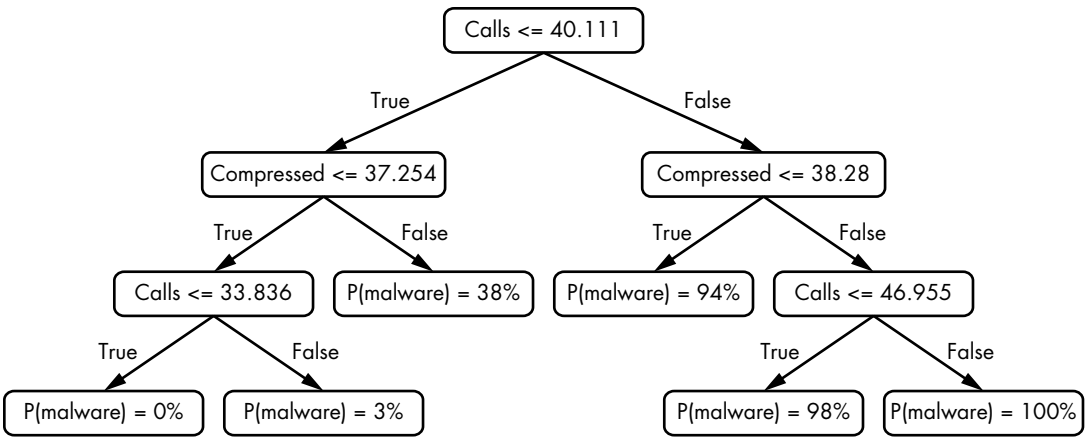

Figure 6-16: A decision tree learned for our simple dataset example

The decision tree flow starts when we input the features we've extracted from a new, previously unseen binary into the tree. Then the tree defines the series of questions to ask of this binary's features. The box at the top of the tree, which we call a *node*, asks the first question: is the number of suspicious calls in the tree less than or equal to 40.111? Note that the decision tree uses a floating point number here because we've normalized the number of suspicious calls in each binary to a range between 0 and 100. If the answer is "yes," we ask another question: is the percentage of compressed data in the file less than or equal to 37.254? If the answer is "yes," we proceed to the next question: is the number of suspicious calls in the binary less than or equal to 33.836? If the answer is "yes," we reach the end of the decision tree. At this point, the probability that the binary is malware is 0 percent.

Figure 6-17 shows a geometrical interpretation of this decision tree.

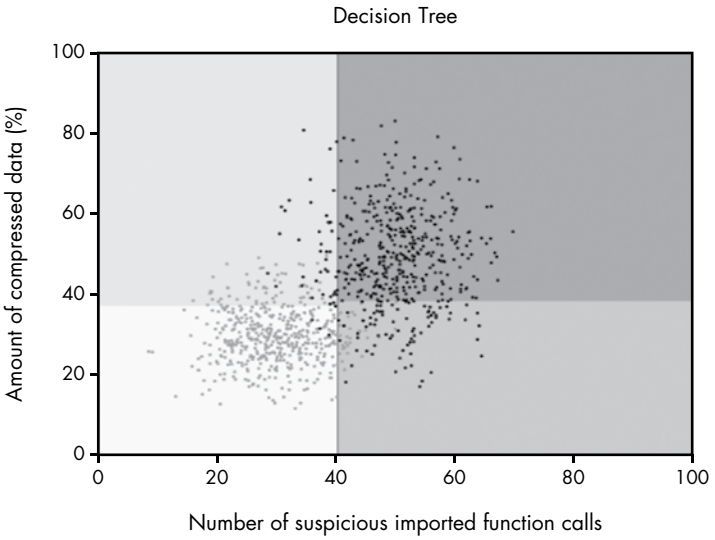

Figure 6-17: The decision boundary created by a decision tree for our simple dataset example

Here, the shaded regions indicate where the decision tree thinks samples are malicious. The lighter regions indicate where the decision tree thinks samples are benign. The probabilities assigned by the series of questions and answers in Figure 6-16 should correspond with those in the shaded regions in Figure 6-17.

Choosing a Good Root Node

So how do we use a machine learning algorithm to generate a decision tree like this from training data? The basic idea is that the decision tree starts with an initial question called a *root node*. The best root node is the one for which we get a "yes" answer for *most if not all* samples of one type, and a "no" answer for *most if not all* samples of the other type. For example,

in Figure 6-16, the root node question asks whether a previously unseen binary has 40.111 or fewer calls. (Note that the number of calls per binary here is normalized to a 0 to 100 scale, making floating point values valid.) As you can see from the vertical line in Figure 6-17, most of the benign data has less than this number, while most of the malware data has more than this number of suspicious calls, making this a good initial question to ask.

Picking Follow-Up Questions

After choosing a root node, pick the next questions using a method similar to the one we used to pick the root node. For example, the root node allowed us to split the samples into two groups: one group that has less than or equal to 40.111 suspicious calls (negative feature space) and another that has more than 40.111 suspicious calls (positive feature space). To choose the next question, we just need questions that will further distinguish the samples in each area of the feature space into malicious and benign training examples.

We can see this in the way the decision tree is structured in Figure 6-16 and 6-17. For example, Figure 6-16 shows that after we ask an initial "root" question about the number of suspicious calls binaries make, we ask questions about how much compressed data binaries have. Figure 6-17 shows why we do this based on the data: after we ask our first question about suspicious function calls, we have a crude decision boundary that separates most malware from most benignware in the plot. How can we refine the decision boundary further by asking follow-up questions? It's clear visually that the next best question to ask, which will refine our decision boundary, will be about the amount of compressed data in the binaries.

When to Stop Asking Questions

At some point in our decision tree creation process, we need to decide when the decision tree should stop asking questions and simply determine whether a binary file is benign or malicious based on our certainty about our answer. One way is to simply limit the number of questions our decision tree can ask, or to limit its *depth* (the maximum number of questions we can ask of any binary). Another is to allow the decision tree to keep growing until we're absolutely certain about whether or not every example in our training set is malware or benignware based on the structure of the tree.

The advantage of constraining the size of the tree is that if the tree is simpler, we have a greater chance of getting the answer right (think of Occam's razor—the simpler the theory, the better). In other words, there's less chance that the decision tree will overfit the training data if we keep it small.

Conversely, allowing the tree to grow to maximum size can be useful if we are *underfitting* the training data. For example, allowing the tree to grow further will increase the complexity of the decision boundary, which we'll want to do if we're underfitting. In general, machine learning practitioners

usually try multiple depths, or allow for maximum depth on previously unseen binaries, repeating this process until they get the most accurate results.

Using Pseudocode to Explore Decision Tree Generation Algorithms

Now let's examine an automated decision tree generation algorithm. You learned that the basic idea behind this algorithm is to create the root node in the tree by finding the question that best increases our certainty about whether the training examples are malicious or benign, and then to find subsequent questions that will further increase our certainty. The algorithm should stop asking questions and make a decision once its certainty about the training examples has surpassed some threshold we set in advance.

Programmatically, we can do this recursively. The Python-like pseudocode in Listing 6-3 shows the complete process for building a decision tree in simplified form.

```
tree = Tree()
def add_question(training_examples):
  ❶ question = pick_best_question(training_examples)
  ❷ uncertainty_yes,yes_samples=ask_question(question,training_examples,"yes")
  ❸ uncertainty_no,no_samples=ask_question(question,training_examples,"no")
  ❹ if not uncertainty_yes < MIN_UNCERTAINTY:
        add_question(yes_samples)
  ❺ if not uncertainty_no < MIN_UNCERTAINTY:
        add_question(no_samples)
❻ add_question(training_examples)
```

Listing 6-3: Pseudocode for building a decision tree algorithm

The pseudocode recursively adds questions to a decision tree, beginning with the root node and working its way down until the algorithm feels confident that the decision tree can provide a highly certain answer about whether a new file is benign or malicious.

When we start building the tree, we use `pick_best_question()` to pick our root node ❶ (for now, don't worry about how this function works). Then, we look at how much uncertainty we now have about the training samples for which the answer is "yes" to this initial question ❷. This will help us to decide if we need to keep asking questions about these samples or if we can stop, and predict whether the samples are malicious or benign. We do the same for the samples for which we answered "no" for the initial question ❸.

Next, we check if the uncertainty we have about the samples for which we answered "yes" (`uncertainty_yes`) is sufficiently low to decide whether they are malicious or benign ❹. If we can determine whether they're malicious or benign at this point, we don't ask any additional questions. But if we can't, we call `add_question()` again, using `yes_samples`, or the number of samples for which we answered "yes," as our input. This is a classic example of *recursion*, which is a function that calls itself. We're using recursion to

repeat the same process we performed for the root node with a subset of training examples. The next `if` statement does the same thing for our "no" examples ❺. Finally, we call our decision tree building function on our training examples ❻.

How exactly `pick_best_question()` works involves math that is beyond the scope of this book, but the idea is simple. To pick the best question at any point in the decision tree building process, we look at the training examples about which we're still uncertain, enumerate all the questions we could ask about them, and then pick the one that best reduces our uncertainty about whether the examples are malware or benignware. We measure this reduction in uncertainty using a statistical measurement called *information gain*. This simple method for picking the best question works surprisingly well.

NOTE *This is a simplified example of how real-world, decision tree–generating, machine learning algorithms work. I've left out the math required to calculate how much a given question increases our certainty about whether or not a file is bad.*

Let's now look at the behavior of decision trees on the two sample datasets we've been using in this chapter. Figure 6-18 shows the decision boundary learned by a decision tree detector.

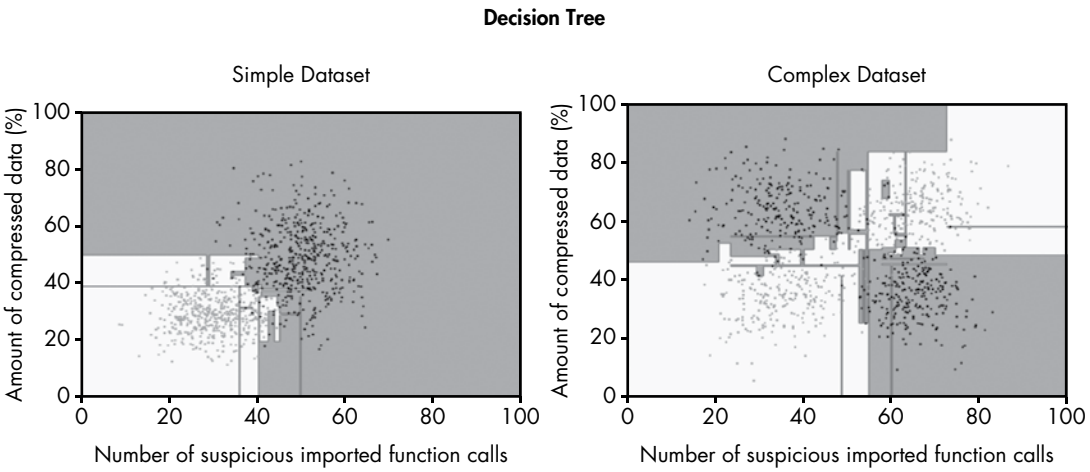

Figure 6-18: Decision boundaries for our sample datasets produced by a decision tree approach

In this case, instead of setting a maximum depth for the trees, we allow them to grow to the point where there are no false positives or false negatives relative to the training data so that every training sample is correctly classified.

Notice that decision trees can only draw horizontal and vertical lines in the feature space, even when it seems clear and obvious that a curved or diagonal line might be more appropriate. This is because decision trees

only allow us to express simple conditions on individual features (such as greater than or equal to and less than or equal to), which always leads to horizontal or vertical lines.

You can also see that although the decision trees in these examples succeed in separating the benignware from the malware, the decision boundaries look highly irregular and have strange artifacts. For example, the malware region extends into the benignware region in strange ways, and vice versa. On the positive side, the decision tree does far better than logistic regression at creating a decision boundary for the complex dataset.

Let's now compare the decision trees in Figure 6-18 to the decision tree models in Figure 6-19.

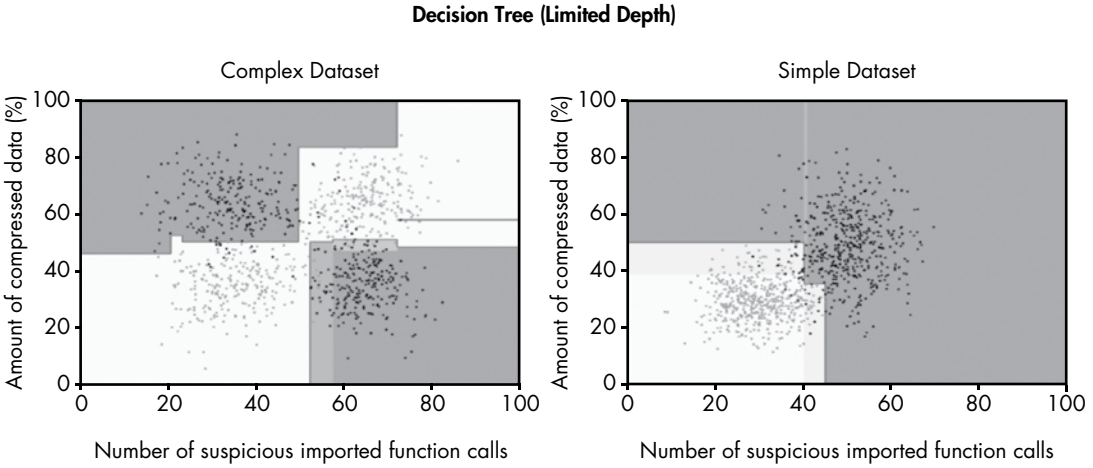

Figure 6-19: Decision boundaries for our sample datasets produced by a limited-depth decision tree

The decision trees in Figure 6-19 use the same decision tree generation algorithm used for Figure 6-18, except I limit the tree depth to five nodes. This means that for any given binary, I can ask a maximum of five questions of its features.

The result is dramatic. Whereas the decision tree models shown in Figure 6-18 are clearly overfit, focusing on outliers and drawing overly complex boundaries that fail to capture the general trend, the decision trees in Figure 6-19 fit the data much more elegantly, identifying a general pattern in both datasets without focusing on outliers (with one exception, the skinnier decision region in the upper-right area of the simple dataset). As you can see, picking a good maximum decision tree depth can have a big effect on your decision tree–based machine learning detector.

When to Use Decision Trees

Because decision trees are expressive and simple, they can learn both simple and highly irregular boundaries based on simple yes-or-no questions. We can also set the maximum depth to control how simple or complex their theories of what constitutes malware versus benignware should be.

Unfortunately, the downside to decision trees is that they often simply do not result in very accurate models. The reason for this is complex, but it's related to the fact that decision trees express jagged decision boundaries, which don't fit the training data in ways that generalize to previously unseen examples very well.

Similarly, decision trees don't usually learn accurate probabilities around their decision boundaries. We can see this by inspecting the shaded regions around the decision boundary in Figure 6-19. The decay is not natural or gradual and doesn't happen in the regions it should—areas where the malware and benignware examples overlap.

Next, I discuss the random forest approach, which combines multiple decision trees to yield far better results.

Random Forest

Although the security community relies heavily on decision trees for malware detection, they almost never use them individually. Instead, hundreds or thousands of decision trees are used in concert to make detections through an approach called *random forest*. Instead of training one decision tree, we train many, usually a hundred or more, but we train each decision tree differently so that it has a different perspective on the data. Finally, to decide whether a new binary is malicious or benign, we allow the decision trees to vote. The probability that a binary is malware is the number of positive votes divided by the total number of trees.

Of course, if all the decision trees are identical, they would all vote the same way, and the random forest would simply replicate the results of the individual decision trees. To address this, we want the decision trees to have different perspectives on what constitutes malware and benignware, and we use two methods, which I discuss next, to induce this diversity into our collection of decision trees. By inducing diversity, we generate a "wisdom of crowds" dynamic in our model, which typically results in a more accurate model.

We use the following steps to generate a random forest algorithm:

1. Training: for every tree out of the number we plan to generate (typically 100 or more)
    - Randomly sample some training examples from our training set.
    - Build a decision tree from the random sample.
    - For each tree that we build, each time we consider "asking a question," consider asking a question of only a handful of features, and disregard the other features.
2. Detection on a previously unseen binary
    - Run detection for each individual tree on the binary.
    - Decide whether or not the binary is malware based on the number of trees that voted "yes."

To understand this in more detail, let's examine the results generated by the random forest approach on our two sample datasets, as shown in Figure 6-20. These results were generated using 100 decision trees.

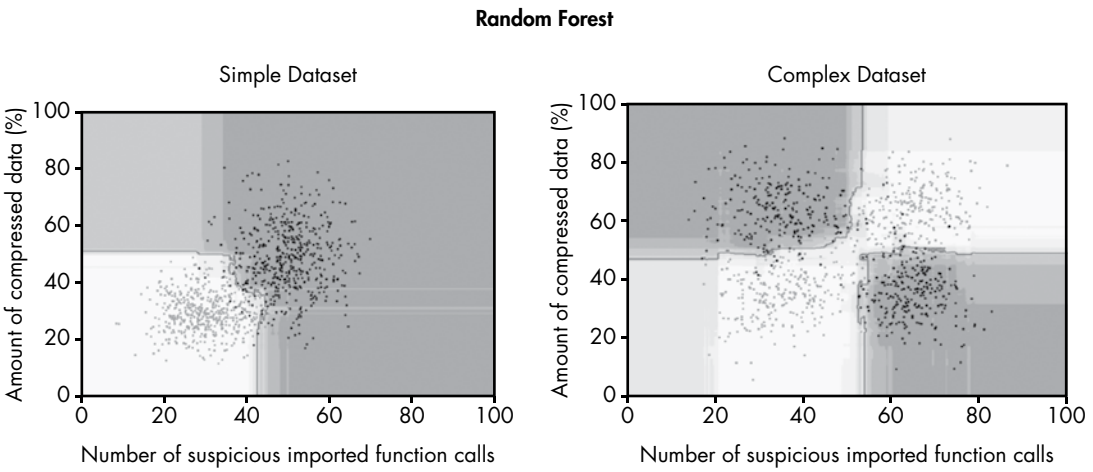

Figure 6-20: Decision boundaries created using the random forest approach

In contrast to the individual decision tree results shown in Figures 6-18 and 6-19, random forest can express much smoother and more intuitive decision boundaries for both simple and complex datasets than individual decision trees. Indeed, the random forest model fits the training dataset very cleanly, with no jagged edges; the model seems to have learned good theories about what constitutes "malicious versus benign" for both datasets.

Additionally, the shaded regions are intuitive. For example, the further you get from benign or malicious examples, the less certainty random forest has about whether examples are malicious or benign. This bodes well for random forest's performance on previously unseen binaries. In fact, as you'll see in the next chapter, random forest is the best performing model on previously unseen binaries of all the approaches discussed in this chapter.

To understand why random forest draws such clean decision boundaries compared to individual decision trees, let's think about what the 100 decision trees are doing. Each tree sees only about two-thirds of the training data, and only gets to consider a randomly selected feature whenever it makes a decision about what question to ask. This means that behind the scenes, we have 100 diverse decision boundaries that get *averaged* to create the final decision boundaries in the examples (and the shaded regions). This "wisdom of crowds" dynamic creates an aggregate opinion that can identify the trends in the data in a much more sophisticated way than individual decision trees can.

Summary

In this chapter, you got a high-level introduction to machine learning–based malware detection as well as four major approaches to machine learning: logistic regression, k-nearest neighbors, decision trees, and random forests. Machine learning–based detection systems can automate the work of writing detection signatures, and they often perform better in practice than custom written signatures.

In the following chapters, I'll show you how these approaches perform on real-world malware detection problems. Specifically, you'll learn how to use open source, machine learning software to build machine learning detectors to accurately classify files as either malicious or benign, and how to use basic statistics to evaluate the performance of your detectors on previously unseen binaries.