Au: Please check chapter cross-refs

9

Make title “Demystifying Machine Learning”

Au: Chapter 9 title differs from tracking grid. Please choose correct title for this page and for the verso running heads.

Machine Learning

delete double quotes x2

> *“They know enough who know how to learn.”*
>
> Henry Adams

Demystifying Machine Learning

Au: Check verso running heads. Global any changes.

There are two types of people in information security—those who are completely intimidated by machine learning and those who know machine learning largely solved the spam problem and are completely intimidated by machine learning. It's easy to be intimidated when machine learning is described as "a type of artificial intelligence that provides computers with the ability to learn without being explicitly programmed" by TechTarget. (`http://whatis.techtarget.com/definition/machine-learning`). How can a computer do anything without being explicitly programmed? Or better yet, consider this rather well known definition from Tom M. Mitchell in his 1997 book titled ***Machine Learning***:

Au: Publisher?

McGraw-Hill

delete double quotes x2

> ***"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."***

Are you clear now on what machine learning is? This broad definition doesn't help much because it only describes the abstract results of machine learning, not what it is or how to use it. To help you understand machine learning at a practical and concrete level, we start this chapter with a learning task associated with realistic data. Prepare for the examples in this chapter by setting the directory to the working directory for this chapter and make sure the R libraries are installed (Listing 9-0).

Listing 9-0

remove

```
Listing 9-0
# set working directory to chapter location
# (change for where you set up files in ch 2)
setwd("~/book/ch09")
# make sure the packages for this chapter
# are installed, install if necessary
pkg <- c("ggplot2", "RColorBrewer")
new.pkg <- pkg[!(pkg %in% installed.packages())]
if (length(new.pkg)) {
  install.packages(new.pkg)
}
```

Au: This line of code is light gray and italic (and sometimes roman) in other chapters. Check that all are correct.

The Listing should not appear twice (green and below green) please remove non-green (same throughout whole chapter)

Detecting Malware

Assume that you have been able to record memory and processor usage on all of your systems. With some effort, you have been able to inspect almost 250 of the computers, discovering that some of the systems are infected with malware and some are operating normally (without malware). But you have 445 other systems that haven't been inspected and you want to save time and use the data you have to determine if the other 445 systems you have are infected or not.

comma

Note

Please keep in mind that this is a contrived demonstration of a machine learning approach; for a much more complete application of machine learning to detect malware, see "Disclosure: Detecting Botnet Command and Control Servers Through Large-Scale NetFlow Analysis" from the ***Proceedings of the 28th Annual Computer Security Applications Conference*** *by Leyla Bilge, et al. (Full reference is available in Appendix B.)*

remove italic

remove italic

OK

This example will use R to build an algorithm that can be trained to perform the task of classifying systems as either infected or not. Start by loading the data on the hosts you know about and inspecting it (Listing 9-1).

Listing 9-1

remove

```
Listing 9-1
memproc <- read.csv("data/memproc.csv", header=T)
summary(memproc)
##          host           proc               mem               state
##  crisnd0004:  1   Min.   :-3.1517   Min.   :-3.5939   Infected: 53
##  crisnd0062:  1   1st Qu.:-1.2056   1st Qu.:-1.4202   Normal  :194
##  crisnd0194:  1   Median :-0.4484   Median :-0.6212
##  crisnd0203:  1   Mean   :-0.4287   Mean   :-0.5181
##  crisnd0241:  1   3rd Qu.: 0.3689   3rd Qu.: 0.2413
##  crisnd0269:  1   Max.   : 3.1428   Max.   : 3.2184
##  (Other)   :241
```

Note

The `data/memproc.csv` file is available as part of the Chapter 9 download materials for this book, which you can find at `www.wiley.com/go/datadrivensecurity`

remove italic

remove italic / period

You can see there are 53 hosts identified as “infected” and 194 identified as “normal.” Also notice that both the processor data and the memory information have been normalized (see the discussion of z-score in Chapter 5). That will keep the numbers on the same scale. Scaling the variables like this is important in some machine learning approaches when you’re comparing across variables. In order to explore this data a bit more, let’s plot this data (Listing 9-2), comparing the processor data to the memory, and differentiate it based on the malware state (see Figure 9-1).

OK

Listing 9-2

remove

```
Listing 9-2
# requires package : ggplot2
# requires object: memproc (9-1)
library(ggplot2)
gg <- ggplot(memproc, aes(proc, mem, color=state))
gg <- gg + scale_color_brewer(palette="Set2")
gg <- gg + geom_point(size=3) + theme_bw()
print(gg)
```

Notice how the infected systems appear to generally use more processor and memory? Perhaps you could develop an algorithm to classify this data just based on the relative location (on the scatterplot in Figure 9-1) of the known hosts. But before you get too far, you’ll want to do a little planning. First you’ll want to determine which machine learning algorithm you want to apply, and then you should figure out how to test if the algorithm is any good. In a real problem, you would try several different algorithms and features; you'll learn about model and feature selection later in this chapter.

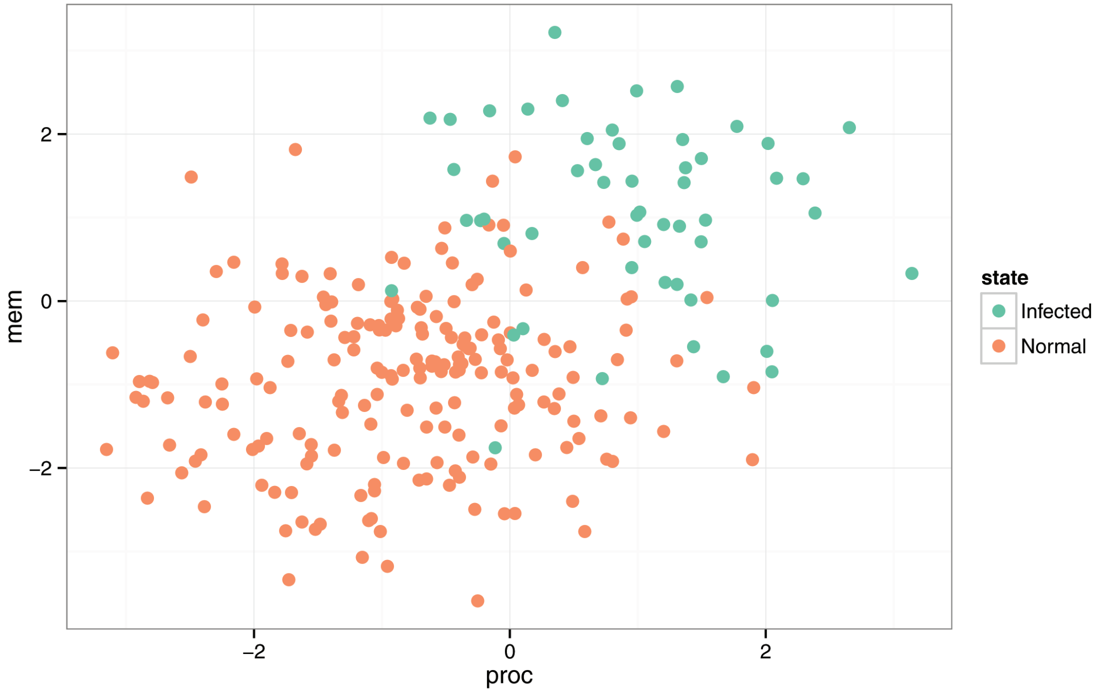


FIGURE 9-1 *Processor and memory across systems*

Developing a Machine Learning Algorithm

Does that title give you flashes of fear that we'll start talking about mathematical formulas and make you say things like "sub i of x"? Don't worry, we will keep this as light as we can and we will start by demystifying the word ***algorithm***. Anytime you see the word algorithm, try to mentally replace it with "a series of instructions" because that's all an algorithm is. You'll want to develop a series of instructions for the computer on how to inspect and understand the data (so it can learn about it). Then the computer can apply that learning to the systems you don't know about and classify them.

comma

Do you see how you are not explicitly programming? Even though you are absolutely writing a program for the computer, you will not be explicitly writing the decision criteria the computer will use and that's the difference. Your series of instructions (the algorithm) will explicitly tell the computer how to inspect the data and how it should build its own decision criteria from the data. It will not tell the computer the decision criteria directly. Compare that to the traditional approach of programming firewall and intrusion-detection/prevention systems. With the traditional approach, humans try to think up what's best and then explicitly program the rules the machines should follow. There is a limit to that approach, and unfortunately, our security systems reached that limit years ago. In machine learning, you are asking the computer to learn from the data and then apply that learning to other data. The computer is far more capable of uncovering the differences and subtleties in the data than humans and that is exactly what machine learning is doing.

comma

Note

It may seem like this chapter uses the terms ***model*** *and* ***algorithm*** *somewhat interchangeably. The difference is subtle and may even be a bit confusing at first. The term* ***model*** *is more general and just defines how the elements fit together. An* ***algorithm*** *is a specific way of implementing a model, so there can be many alternative algorithms that fit the same model.*

remove italic
x3

Getting back to the data shown in Figure 9-1, you'll want to create a series of instructions to learn about the processor and memory usage on the normal hosts and then compare it to the processor and memory usage on the infected hosts. Once the computer has some notion of a difference between the two sets, you can give it some instructions on how to apply that information to the data collected from the unknown/unclassified systems. Remember the goal here is to have the computer guess whether or not a system is infected with malware. Consider this short algorithm, which is easy to understand and easy to follow:

1. Define and train an algorithm:
   a. Calculate the average (mean) processor and memory usage for known infected systems.
   b. Calculate the average (mean) processor and memory usage for known normal systems.
2. Make a prediction using processor and memory usage for an unknown host:
   a. If the processor and memory usage are closer to the average infected machine, label it as infected.
   b. If the processor and memory usage are closer to the average normal machine, label it as normal.

Congratulations! You have written your first machine learning algorithm and now the computers are one step closer to world domination with this extra bit of artificial intelligence! Notice the choice of wording in the first step, you'll want to ***train*** the algorithm. That's the term used to describe when the machine is learning from the data; it's being ***trained*** by the data just as an apprentice is trained by its master. The data used to train the algorithm is referred to as the ***training data***. In this simple example, the "training" simply involved calculating the mean usage for infected and non-infected using the training data. This is a single-step training procedure. In contrast, most real machine learning algorithms use iterative or multistep training procedures, as we'll describe later.

Validating the Algorithm

Before you rely on this algorithm for real decisions, you need to make sure it is valid. You'll want some way to test how accurate this algorithm is at predicting infected systems. Rather than using all of this data to train the algorithm, how about you hold back some of the data to test how accurate the algorithm can predict malware? The process of “making sure” you have a good approach is one of the strong suits of machine learning. It has evolved just as much (if not more so) in computer science as it has in statistics, and there is a strong element of pragmatism in the field. Many techniques have evolved to validate the decisions you’ll make and they are so ingrained in the process, it becomes impossible not to perform those steps as part of the model selection.

data sets x2
(style sheet)

For this example, you will keep it simple and split the original data into two datasets. In serious machine learning projects, you would probably create multiple datasets from the original data, and train and test the data over multiple iterations (and validations).

use

Once you split the data into two groups, as we mentioned, call the first group the training data, since you’ll use it to train the algorithm, and call the second group the test data, since you’ll use it to (yup, you guessed it) test your approach. To split the data randomly, make use of the `sample()` command. You will pull a random sample of the indexes (the index is the location in the vector data) of the original data and using that sample to split into the train and test data. There’s no definitive rule as to where to make the split (different techniques split in different ways), so you will simply take one third for the test data and train the algorithm on the other two thirds. Since there is an element of randomness here we make the splitting repeatable by setting the seed for the random number generator (see Listing 9-3).

LISTING 9-3

remove

Listing 9-3

```
# requires package : ggplot2
# requires object: memproc (9-1)
# make this repeatable
set.seed(1492)
# count how many in the overall sample
n <- nrow(memproc)
# set the test.size to be 1/3rd
test.size <- as.integer(n/3)
# randomly sample the rows for test set
testset <- sample(n, test.size)
# now split the data into test and train
test <- memproc[testset, ]
train <- memproc[-testset, ]
```

Now you can train the algorithm on the `train` data and verify how good it is with `test` data. Please keep in mind that there are much more robust methods for validation. Splitting the data once like this is better than just assuming the algorithm is good, but in the real world, you’d need something more robust like cross-validation, which we discuss later in this chapter.

Implementing the Algorithm

Recall that the first step in training this algorithm is to calculate the average (mean) for the infected processor and memory usage and the mean for the normal processor and memory usage. You do this by taking a subset of the rows based on the `state` field (so only infected or normal is returned) and then apply that to the columns of the `proc` and `mem` fields. That reduced data can be passed directly into `colMeans()`, which will compute the means on the two columns and return a named vector with two elements (see Listing 9-4).

Listing 9-4

remove

```
Listing 9-4
# requires object: train (9-3)
# pull out proc and mem columns for infected then normal
# then use colMeans() to means of the columns
inf <- colMeans(train[train$state=="Infected", c("proc", "mem")])
nrm <- colMeans(train[train$state=="Normal", c("proc", "mem")])
print(inf)
##     proc      mem
## 1.152025 1.201779
print(nrm)
##       proc        mem
## -0.8701412 -0.9386983
```

The differences between the means here is not exactly small, so this rather simple approach may do okay with your simple algorithm. With the algorithm now trained and ready to predict, the next step is to create a `predict.malware()` function (Listing 9-5). This will take in a single named vector called `data`, extract out the `proc` and `mem` values, than calculate how far those are from the means that you generated during the training. What is the best way to calculate distance? Think back to geometry class and the Pythagorean theorem—$a^2 + b^2 = c^2$, where a and b are the two sides of the triangle and c is the hypotenuse. This is called “Euclidean distance,” since it is based on Euclidean geometry. In your case, ***a*** is the difference between the trained `proc` mean and the test `proc` value and ***b*** is the difference between the trained `mem` mean and the test `mem` value. Once you get the two distances, you simply compare them. Whichever is smaller is the one you will predict.

comma

Listing 9-5

remove

```
Listing 9-5
# requires object: inf (9-4), nrm (9-4)
predict.malware <- function(data) {
  # get 'proc' and 'mem' as numeric values
  proc <- as.numeric(data[['proc']])
  mem <- as.numeric(data[['mem']])
  # set up infected comparison
  inf.a <- inf['proc'] - proc
  inf.b <- inf['mem'] - mem
  # pythagorean distance c = sqrt(a^2 + b^2)
  inf.dist <- sqrt(inf.a^2 + inf.b^2)
  # repeat for normal systems
  nrm.a <- nrm['proc'] - proc
  nrm.b <- nrm['mem'] - mem
  nrm.dist <- sqrt(nrm.a^2 + nrm.b^2)
  # assign a label of the closest (smallest)
  ifelse(inf.dist<nrm.dist,"Infected", "Normal")
}
```

Feel free to pass in a few values if you like and inspect the output. At this point, everything is ready to run against the test data. To pass in the test data you can use the `apply()` function with the first argument being the test data set and the second argument being a `1` to denote to apply it over the rows (instead of a `2` for columns). Then you'll pass in the function we just created called `predict.malware` (see Listing 9-6). The `apply` function will convert in each row to a named vector. We have to be careful here, because the `state` and `host` variables are characters, so the whole vector is converted to a character vector when `apply` passes it in. This is why you convert the `proc` and `mem` variables back to numeric variables with the `as.numeric()` function in the `predict.malware()` function.

remove

LISTING 9-6

```
Listing 9-6
# requires object: test (9-3), predict.malware (9-5)
prediction <- apply(test, 1, predict.malware)
```

First, Do No Harm; Second, Do Better Than the Null Model

OK

This is a great time to point out that this is a very basic algorithm and it's only for discussion purposes. There is a concept within statistics known as the ***null model***, which is a very simple model that you'll always want to do better than (or at least no worse). For example, in the ZeroAccess infection data in Chapter 5, the null model could be the calculation of the average (mean) infection across all the states (5,253 infections). The null model (for prediction) would estimate 5,253 infections for any new state regardless of any data about that state. In this case you are omitting or "nullifying" the variables to simplify the model. Intuitively, you know that the average across the states will be a very poor predictor, but that's the purpose. You'll want to use this as a reference point and exceed it. And even though this seems like a "well duh" type of statement, we are not kidding about this. You could spend days preparing data and training an intricate support vector machine and do worse than a much simpler model. Just take the time to create a simple "must be this tall to ride" mark, and then make sure you surpass it.

Once the test data runs through that code, you'll have a set of predictions and the ability to compare them to the real values (see the power of this method?). To determine how well it did, you'll want to look at the proportion of correctly predicted results on the test data. You can calculate that by taking the number of correct predictions (where the real `test$state` and the predicted `prediction` match) and then dividing that by the total number of predictions (Listing 9-7).

remove

LISTING 9-7

```
Listing 9-7
# requires object: test (9-3), prediction (9-6)
sum(test$state==prediction)/nrow(test)
## [1] 0.8780488
```

This very simple algorithm predicted almost 88 percent of the values correctly, which is probably more a statement about how segregated the data is than the strength of the algorithm. But overall, 88 percent is pretty good for your first machine learning algorithm; congratulations! The results are pictured in Figure 9-2.

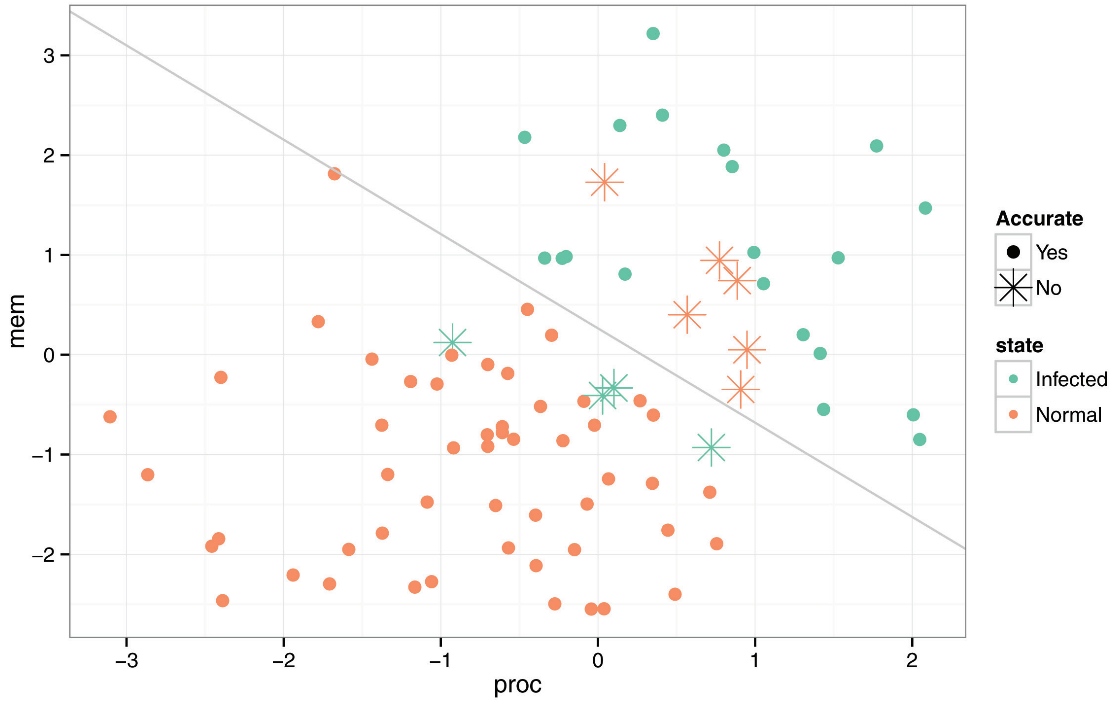


FIGURE 9-2 *Predictions from the algorithm*

This classifier creates a line halfway between the two means and perpendicular to an intersecting line. Anything above the line is predicted as infected; anything below is predicted to be normal. The misclassified values are clearly marked in Figure 9-2. You can see how any normal systems above the line are mislabeled as well as any infected systems below the line.

Spam, Spam, Spam

Open any non-InfoSec book on machine learning (which may be all of them) and you will probably see spam filtering mentioned, and perhaps even see an in-depth example. We've decided not to go into spam filtering, given that we've got a single chapter to cover everything and there are already some great examples out there. One of the better discussions of spam filtering (including a guided walkthrough) is in ***Machine Learning for Hackers*** by Drew Conway and John Myles White. Another good thing about playing with spam classification is that there is no end to the available data, right?

comma

Benefiting from Machine Learning

Now that you've seen a rather simple example (perhaps too simple), you should have a basic understanding of the change in thinking that machine learning brings. Rather than focusing on rule sets and signatures, machine learning can shift the focus toward continual adaptation based on the computers learning directly from the data. Hopefully, the days of thresholds and regular expressions rules will soon be behind us.

Before you can read about the benefits of machine learning, you need to learn about the two types of machine learning algorithms—***supervised*** and ***unsupervised***. Which type you use is determined more by the type of data you have than personal preference.

by

- **Supervised algorithms** require that the training set have known samples just like the opening example of this chapter. The data in that example was ~~from~~ collected from hosts that were identified as infected with malware or not. Another example is the ZeroAccess data in Chapter 5, where you knew how many infections there were in each state and county and you could correlate that with other data about the states and counties. Supervised learning is possible only when you have labeled or known data.

delete: from

OK

- **Unsupervised algorithms** are usually applied to data when you don't know ahead of time what outcome you are seeking. Unsupervised learning "lets the data speak for itself," as much as possible. As an example, think of the recommendation systems at Amazon or Netflix. Those systems begin with data on the history of movie rentals or purchases and apply unsupervised learning techniques to group similar people (their habits actually) based on patterns in the data. This enables them to recommend products that other people like you have purchased. You don't decide ahead of time what the groups should be. Unsupervised methods enable you to discover groupings and relationships, and do other and typically deeper explorations like no other approach. Given the unsupervised nature of these approaches, it is difficult to definitively prove something with unsupervised methods, but that's not what these are designed to do. As you'll see, you can discover some interesting relationships with unsupervised learning methods.

To Parametric or Not to Parametric, That Is the Question

In addition to supervised and unsupervised, machine learning algorithms may also be separated into parametric methods and non-parametric methods. The term ***parametric*** refers to one or more parameters in the model or algorithm that must be estimated as a result of the training step. The linear regression performed in Chapter 5 is an example of a parametric model. Part of the output of the `lm()` command is the linear coefficients (parameters), which are then used in both prediction and inference within regression analysis. Compare this to the random forest algorithm (discussed later in this chapter). When you train a random forest algorithm, there are no parameters to estimate. Instead, you grow a series of decision trees that are then used for further classification.

OK

Answering Questions with Machine Learning

What types of questions can machine learning answer? What sort of problems can it solve? Broadly speaking machine learning can help you with questions of

colon

- Classification
- Quantitative prediction

- Inference
- Exploration and discovery

The opening example in this chapter already introduced the concept of classification, where you tried to determine if the hosts were infected or not. ***Classification*** is the process of identifying the category something belongs in, or determining which label should be applied. Classification always begins with a list of possible categories and known data that describes those categories (so they are supervised algorithms). Many of the tactical challenges within information security revolve around a single classification problem, such as "Is this malicious or not?" Mechanisms exist to authenticate and authorize users, but do their actions match that of a normal user or a malicious user? Is this HTTP request valid or is the source attempting something they shouldn't be? These are all questions that classification algorithms are best at tackling.

What if you wanted to forecast a quantity instead? Machine learning (and classical statistics) offers methods to do ***quantitative prediction***. The overall approach may make people with a strong engineering background a bit uneasy thinking that prediction is impossible. But relax—nobody is claiming that the precise future is hidden in the data. However you can use the data to make a pretty good estimate. Given a set of observations and the outcome that resulted (so again, these are supervised methods), you can build a predictive model that will provide estimates of future values.

OK

Think back to the linear regression analysis performed in Chapter 5. If by some strange turn of events another state appears with 6 million people, the regression analysis using just population would predict just under 5,000 zero access infections in that state. Although that example isn't exactly practical, you could use techniques to estimate bandwidth usage next month, or even forecast the probable magnitudes for the next DDoS attack.

ZeroAccess

Sometimes the end result isn't a prediction of a quantity or category. Sometimes you just want to know about the variables you observe and determine how they contribute to and interact with the outcome. In these cases you'll want to apply methods for ***inference***. Inferential methods allow you to describe your environment. How important are these variables? Are data around processor and memory usage the best predictors of an infected machine? For example, linear regression enables you to toss multiple variables into a single analysis and see how each of them contributes to the outcome and see the quantitative relationships around them. Both supervised and unsupervised methods support inference about the variables, and that inference is an important part of any model or algorithm.

The last application of machine learning is for ***exploration and discovery***. This is an area where unsupervised algorithms truly excel, but supervised methods can also support exploration. Sometimes you may find yourself just sitting on a mound of data and you want to know what sort of relationships or patterns exists in the data. Using methods like multidimensional scaling and hierarchical clustering will help you explore and gain perspectives on the data that just aren't possible with simple descriptive statistics.

comma

exist

Measuring Good Performance

At the core of good learning is good feedback. If you're creating models and algorithms and never check if they are doing well, you're doomed to repeat the same mistakes and improvement is nigh impossible. This is such a fundamental concept that several techniques have been developed to measure performance within supervised algorithms. It's important to understand that unsupervised algorithms are generally not used to prove (or disprove) a theory. We don't have the space to go into the mathematical details for each method; instead this section explains a few basic approaches and some of the terms for further exploration.

Following common sense, the best way to measure the performance of any predictive algorithm is to simply see how well it predicts (or how poorly it predicts if you are a pessimist). There is no single, perfect approach, so you will want to choose an approach that performs better than all the available approaches (don't toss out a helpful approach simply because it's imperfect). All of the fancy math formulas that describe this process are just variations on a simple theme: If you are working with quantitative values, select the approach in which predictions are the closest to the observations. If you are working with a classification system, choose the model with the highest number of correct classifications.

Within classic regression analysis, the difference between the calculated prediction and the observed value is squared for each of the values and then added up. When the difference is squared, it amplifies the larger distances and rewards the smaller values and gives a better indication of quality. The fancy term for this is the ***sum square of errors*** (***SSE***). In the grand tradition of multiple ways to express the same thing, this is also called the error sum of squares, sum square of residuals (SS residual), or the residual sum of squares (RSS).

Since calculating SSE involves adding the squared differences, larger sample sizes have larger SSE values. That makes it impossible to compare between a training data set and the test data set when they don't have the same number of data points. To standardize the SSE, it is divided by the number of data points (sample size), and the result can be compared when the sample size is not the same. That result is called the ***mean squared error*** (***MSE***). Prior to the concept of a training data set and test data set, this was (and still is in default classic approaches) calculated on the data set used to train the model. The challenge with just relying on the MSE of the training data is that it is prone to ***overfitting*** (see the sidebar on overfitting). One approach to comparing quantitative models and algorithms is to calculate the MSE and compare it across multiple approaches and feature selections. You'll read more about this process in the section ~~about cross-validation and bootstrapping later in the chapter.~~

comma

hyphen

Au: Is it clear to the reader where to find this section?

"…in the section titled "Validating your model" later in this chapter.

should be "cross-validation" (no comma), but may be removed (See other note)

Overfitting

Since learning algorithms "learn" what to do from the data, it's possible that they'll learn too much or put too much confidence in the data. When this happens, the algorithm may do very well on the training data, but fail miserably when run on real data. This is called ***overfitting*** and occurs when the training algorithm is too aggressive in fitting to the training data. Because it's a sample, the training data will have it's own quirks and characteristics that may not match the population. Ideally, you want the learning process to ignore the quirks of the training data and just focus on the characteristics that apply to the general population. It's a good thing to be aware of overfitting, but awareness alone doesn't help all that much. Several approaches exist to help detect and avoid overfitting, and you'll read about a few in the next section.

Selecting Features

Before you can train an algorithm and measure its performance, you need to have data to run on. One of the less talked about topics within machine learning is how you go about selecting the data to collect and include in your analysis. The variables that you collect and use within your algorithm are called ***features***. Within classic statistics they are also called explanatory, independent, or predictor variables (and a few other things).

The processor and memory usage in the opening example of this chapter were the features you used to train your algorithm.

The tricky part with feature selection is that there are no guidelines in selecting the initial set of features, so this is where your domain expertise comes into play. You collect the data points that may be important and then (as discussed in the previous section) run them through the algorithm and check if they are actually contributing to the outcome. (By doing this relatively simple step that is supported in almost every statistical approach, you will have surpassed every risk analysis model within information security; congratulations.) Although it's tempting to grab everything and anything, remember that data collection and cleaning has a cost (at least in time and resources). And be aware that many approaches benefit from fewer variables and may perform quite poorly with a lot of variables.

As an example, if you were thinking of improving on the malware classifier, you could pull variables from network traffic logs, such as the ports and protocols used, how often, and how much, as a starting point. The first pass of variables doesn't really matter all that much because whatever you choose initially will undoubtedly be wrong-—and that's okay. Grab data that makes sense and then try to make sense of it. You may find that only some of the variables are helpful, or that none of the variables do well, or variables do well only when used in combination. But the point is that feature selection is an iterative process. In the end, you should not only look at how well the features contribute to the outcome, but also how well (or accurately) the whole algorithm performs and then try to improve on it.

comma

Using the Best Subset

Given a bunch of features, how could you determine which ones to include or exclude? One approach is to try every possible combination of features and select the subset of features that performs the best. This technique is rather appropriately named the ***best subset*** approach. The benefit and drawback of this approach is the same—every possible combination is tested. On one hand, this approach may discover a combination of features that you wouldn't have found without this brute force method. On the other hand, you have to run through all the combinations of features and that may take considerable time. As a reference point, using the best subset selection method on 20 variables will require well over one million iterations through the algorithm and validation steps.

There is another caveat with the best subset approach. As the number of features increases, the probability of finding bogus relationships in the features also increases. The good news is that it will generally not happen silently. Overfitting with this method may look "best" on the training data, but perform very poorly on the test data. One way to tackle that problem is to apply several of the best subsets to the test data or move to another technique.

Using Stepwise Comparison

When the brute force method of the best subset is infeasible or undesirable, the stepwise approach may be a good compromise. Rather than tossing everything in, you build up the correct set of features by stepping over them. In a forward stepwise process, the method begins by training with each of the features individually. Whichever feature performs the best is kept and the process is repeated by adding one more feature to the previous results. The features are added in one by one based on their contribution. Once all of the features have been added, all of the best performing algorithms at each of the steps are compared. The overall best set of features is selected as the final set.

The benefit of this method is that it constitutes an enormous reduction in the number of iterations compared to the best subset method. But the drawback is that not all the combinations are tried and so the best combination may be hidden. In some cases, a feature that performs best alone may not perform best when other features are added. You can also perform a reverse stepwise comparison, where you start with all the features and then sequentially step backward, removing the least helpful feature until you're down to one feature again. Then, you look at all of the best combinations and select the best that way.

Validating Your Model

However you go about selecting the features to include, you still need to validate how well the approach performs. Each algorithm may have subtle differences in how it works and the test statistics it generates and focuses on. Still, there are a few general approaches for validating how you're doing, and they apply to almost all the methods. The most widely used method is ***cross-validation***, and it's discussed here. As a second validation pass, you could look at any other resampling methods, such as bootstrapping or the jackknife method.

The opening example started with 247 observations and then was split into a training set to train the algorithm and a test set to test how the training set did. Recall that the data was arbitrarily split so that two thirds was training data and the remaining one third was test data. One drawback in doing that is that you can't train on the one third that you pulled out, and that introduces more variation in the outcome of the training process.

What if you were to repeat the splitting and testing process so all data could contribute to both the training and test of the algorithm? It is possible to increase the accuracy of the algorithm by generating multiple training and test data sets and comparing the results from all of the splits. This approach is called ***cross-validation***, and it works better than when you just split the data once.

The common method of performing cross-validation is to split the data into some number of equal partitions (more than a few) and then iterate over the data using each partition as the test data once. This is known as ***k-fold cross-validation***, because you ***fold*** the data ***k times*** (once for each partition). For example, you could go back to the data in the first example and divide it into 10 partitions and iterate through the process 10 times each, with a different test data set and slightly different training data set. By combining (averaging) the estimation of the accuracy across each of the iterations, you can have more confidence in how that approach will perform on new data.

a

A variation on the k-fold cross-validation is to set the number of partitions equal to the number of samples in the data. The result is the ***leave-one-out cross-validation***. Named because you can sequentially leave out one value from the training set and test against that one value across the whole data set. The results from this method are often more accurate in assessing the algorithm, but it comes at a computational cost.

Specific Learning Methods

There are a lot of learning methods, and it isn't possible to survey all of them in one chapter. We chose a handful of approaches and will briefly touch on what makes them unique, including their strengths, weakness, and so on. But the field of machine learning is as wide as it is deep, and there are many methods we don't touch on here. Do not take that to mean they are less important or not as good. On the contrary, a

es

method such as neural networks or support vector machines may perform better in some circumstances. We have just picked a few to serve as an introduction and overview.

Supervised

Supervised methods require that you begin with known or labeled data. You will not be able to apply supervised methods for malware detection unless you have data from known infection and known normal systems. While this may present a challenge in some circumstances, the power of supervised learning may make the extra effort well worth it.

Linear Regression (and Transformation)

Linear regression is a very popular approach when it comes to quantitative prediction and inference about the independent variables, and for good reason. Linear regression has been around since the late 1800s and has evolved into a robust and flexible approach. One of the early "a-ha" moments with linear regression is that it can be used on data that is not actually linear. For example, look at the line in Figure 9-3. That line was fit to the data with linear regression.

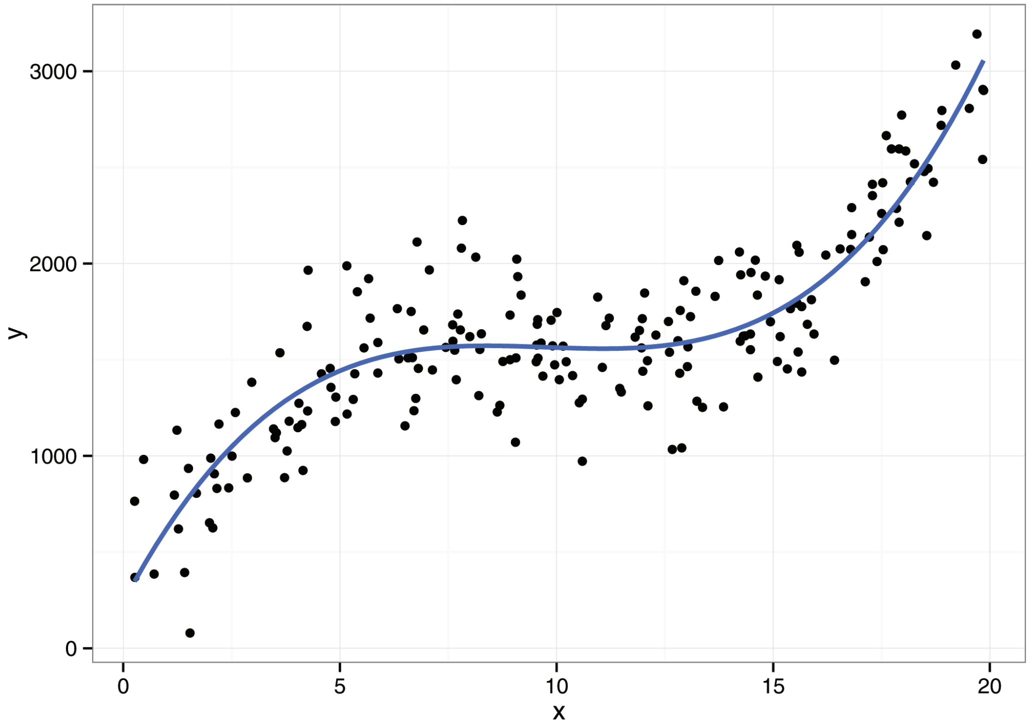


FIGURE 9-3 *Linear regression on non-linear data*

Can you see the linear relationship in Figure 9-3? Believe it or not, it's there. The ***linear*** part of linear regression is a reference to the ***linear coefficients*** estimated, not the data. In other words, you can use a linear model to describe non-linear data. The trick (although it's not really a trick) is to transform the data prior to running linear regression on it. Looking back at Figure 9-3, the relationship between x and y is a cubic polynomial, and some variation around y = x3. Therefore, you would want to transform the x variable and include that in the model so you can estimate the (linear) coefficients for each of those variables. When transforming the variables like this, you must be careful not to overfit the data. It would be possible to add enough transformed variables to perfectly fit the training data, but such an approach would perform horribly on the test or real data. superscript: 3

Linear regression has many variations and nuances that make it powerful, especially when combined to some of the techniques mentioned earlier in this chapter. Classic linear regression relies on computing a p-value (see Chapter 5) to assess the strength of the model and variables. The recent trend is to also integrate validation methods such as cross-validation to support model selection and validation. See the `lm()` and `glm()` commands within R for the specifics on how to execute linear regression.

OK

Logistic Regression

Although linear regression is designed for predicting quantitative variables, it isn't helpful when the problem isn't quantitative. For example, in the opening example, you needed to classify the hosts as infected or not; linear regression wouldn't be helpful in that circumstance. Instead, you can turn to logistic regression, which is an extension of linear regression. It models a yes/no output, that is, choosing between just two outcomes. Figure 9-4 shows logistic regression applied to that training data.

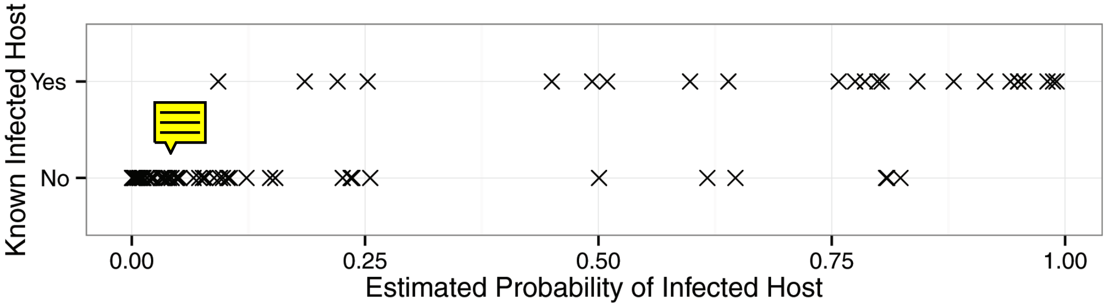


FIGURE 9-4 *Logistic regression on infection test data*

The output (on the x-axis) is an estimated probability of a host being infected based on the input variables. That output is plotted against the known value in the test data (on the y-axis, and remember the y-axis is not known in real life). It's clear that, given these input values, you would be able to estimate a large portion of the hosts correctly. No matter where the cutoff is set (for example, hosts above 0.4 are classified as "infected"), you will undoubtedly have some false positives (identifying hosts as infected when they are not) and false negatives (identifying hosts as not infected when they are). Traditionally, logistic regression is used to make a logical classification (this is or is not something). There are techniques for applying logistic regression to multiple categories, which we won't cover here.

Within R, there are several approaches to logistic regression; however, the `glm()` function can handle most situations.

K-Nearest Neighbors

The technique of k-nearest neighbors is best described using a generic sports analogy. Suppose you want to choose a person at random from anywhere in the world and predict his or her favorite sports teams. When you choose that person at random, you can ask his or her neighbors and friends ("k" of them, where k is any consistent number) which teams that person cheers for. Then, you determine which teams the majority of those neighbors cheer for and assume the person you plucked has similar taste as their neighbors.

The k-nearest neighbors algorithm does the same thing. Given a set of known (this is supervised algorithm) variables, for each new data point, this algorithm looks at the nearest k data points (you pick the value for k) and assumes that the new data point is like its neighbors. This gets away from the linear classification of the opening example. This approach increases in accuracy as the number of observations increases. One drawback is that it is sensitive to the selection of k. With very large values of k, this approach gets closer and closer to creating a linear boundary. Overall, the k-nearest neighbors can be a very effective classifier and outperform many other techniques. It's worth understanding.

Within R, the `class` package offers support for k-nearest neighbors (and other `knn` functions).

Random Forests

Random forests are built on the concept of the decision tree and excel at multidimensional data (data with a lot of features). The decision tree is what IT people think of as a flowchart. You start at the top of the tree and branch off in different directions, depending on the criteria within the tree compared to the observed features. Imagine the various types of decisions that could be built given data types—if the data is above average, fork here; if data fits into that category, go there. If you have complex data, you must use more than one decision tree. A technique called ***boosting*** was developed to create a whole lot of decision trees and then look at the aggregate result from all of them. Boosting provides a huge improvement. Although each individual tree performs poorly, they all performed poorly in a predictable spread around the best answer. Therefore, the best answer can be derived from looking at all the trees (see where this is going with the forest?).

Boosting decision trees works quite well, but it's influenced by noisy features. One or two bad eggs in the basket can bias the result by consistently pulling trees in a weird (and difficult to detect) direction. The random forest technique gets around that problem by growing the trees with only a small subset of the features. This makes each individual tree an even worse predictor, but the aggregate improves because the noisy variables are included only in a subset of the features selected for each tree, and not influenced every tree in the forest.

by

Random forests bring a new way of thinking and are squarely in the non-parametric camp. They do not attempt to create a model of reality and then derive the parameters of the model (such as with regression techniques). Instead, random forests create a huge set of relatively weak predictors and then aggregate across them all. This is like going to a new town and asking only tourists for directions. Many of the answers will be way off, but if you look at the aggregation of all the answers, you'll probably get where you're going.

As you may be thinking, you would never attempt to apply the random forest technique with pencil and paper. This technique grows hundreds or even thousands of multi-branched decision trees based on random points in random features and can be done only with the aide of a computer. Within R, random forests are available from the appropriately named `randomForest` package. It's also worthwhile to explore parallel processing solutions and the R package `doParallel` offers a good solution for spreading the processing across multiple cores and reducing the computational time needed for random forests.

aid

comma

Comprehension versus Performance

One of the challenges with machine learning is that some of the techniques are so complex and abstract that they push the boundaries of human comprehension. At some point, you'll reach a trade-off between the ability to comprehend an approach and the performance it brings. Neural networks are an example of this trade-off. In some cases, neural networks offer better performance, but they are rather complex and difficult to comprehend and difficult to tune properly without that comprehension. You may opt for an approach that is easier to comprehend, easier to explain to others, and easier to use at the expense of a slight deterioration in performance, and you should be comfortable with that. Given the complex nature of many decisions with information security, any approach with machine learning is better than any decision without machine learning.

Unsupervised

As we mentioned earlier in this chapter, unsupervised approaches are quite useful to find underlying patterns and relationships in the data. Given some pile of data, what kinds of trends exist in there?

K-Means Clustering

K-means, like k-nearest neighbor, uses the "k" to represent a variable that you will set as part of the approach. The k in this technique represents the number of clusters to be generated. The k-means approach follows the following algorithm:

1. Set k "center points" randomly among the data.
2. Assign all the data points to the nearest center point.
3. Calculate a new (mean) center of the data points assigned.
4. Move the center points to the new calculated (mean) center.
5. Repeat Steps 2-4 until all centers no longer move in Step 4.

Notice how Step 1 in the k-means technique uses randomness? This means that if you rerun a k-means clustering, you may get different clusters. Figure 9-5 shows the same data with multiple different k-values. The `kmeans()` function in base R will perform k-means clustering.

Hierarchical Clustering

The downside to k-means is that you have to specify the number of clusters. This is where hierarchical clustering can help by deriving all of the clusters within the data. The output of hierarchical clustering is called a ***dendrogram*** (see Figure 9-7 later in this chapter) and looks like a tree, starting at the top and branching off into two groups at a time until all of the objects are in their own cluster. Algorithmically, the approach actually starts at the bottom with everything in its own cluster. It then scans across all the pairs, comparing and looking for the most similar pairing. When it finds similar clusters, it will combine those two. This repeats until there is one cluster with everything at the top.

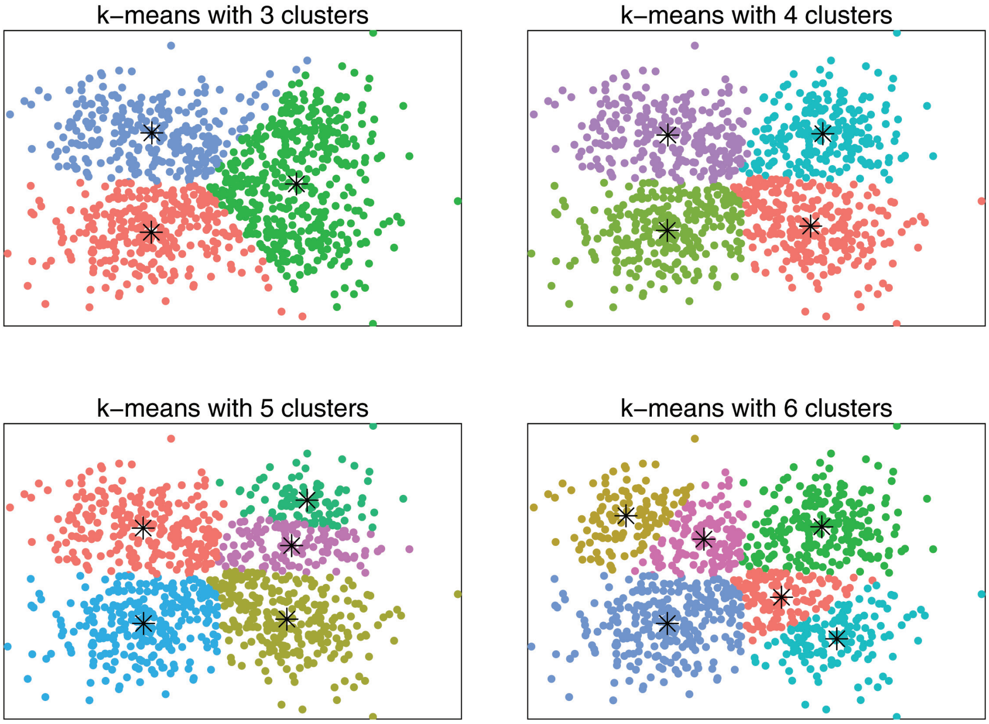


FIGURE 9-5 *K-means clustering with centers shown*

The advantage to hierarchical clustering is that it is possible to “cut” the tree down and inspect the clusters at any point in the tree, which is what you’ll do later in this chapter when you use the `hclust()` function on breach data.

Principal Component Analysis

Principal component analysis (PCA) reduces the number of features you look at to those that really matter and is one of a few techniques to perform this ***dimension reduction***. PCA works best on data that is highly correlated because it can capture most of the variation in the data with a reduced number of variables. The outcome of PCA is a list of derived components ordered by how much variance they describe in the data. Once the data is reduced to that format, you can pull out the significant components and use those moving forward rather than the larger (and possibly noisy) number of dimensions. Running PCA in R requires the single command of `prcomp()`.

Multidimensional Scaling

Sometimes you'll just want to see the clusters. This can be problematic with multidimensional data because you cannot visualize in more than three dimensions (and even that third dimension is tough on a flat screen or paper). The solution is to use a technique called ***multidimensional scaling*** (MDS). Like PCA, MDS performs dimension reduction. It can squish the multidimensional data into two dimensions so you can visualize the relative similarities between objects. You'll run through an example of this technique later in this chapter, using the R command `cmdscale()` for classic multidimensional scaling. In the next section you will see the output of multidimensional scaling when applied to the industries of breach data.

Hands On: Clustering Breach Data

OK

In this section, you revisit the VERIS community database (VCDB) data you used in Chapter 7 in order to see multidimensional scaling and hierarchical clustering in action.

The natural approach to breach data is to simply count the categories, see what occurs more often, and then draw some conclusions from that information. But the challenge with that approach is that any conclusions drawn may be applied too broadly if the conclusions don't apply across the board. After working with breach data for a while, it becomes clear that different industries have different problems. Each industry shares some common traits, like they all deal with the same type of information, causing some industries to be targeted more or less than others. Organizations in the same industry are more likely to copy others in the same industry, so the breach data may also show some type of pattern because of that.

line space
x2

The problem then is this:

Just how different (***or similar***) ***are the incidents across industries***?

This is a rather interesting challenge because the only thing you start with is a hunch that industries are in fact, different. The best approach to this question is some of clustering algorithm. If you can isolate variables across the industries, you can calculate a "distance" between the industries in order to determine which are alike and which display some unique traits.

in / the / ?

You'll begin this analysis by converting the VCDB data to a matrix (see Listing 9-8). You haven't read much about matrices in R, but they are similar to a data frame in that they have fixed row and column widths (think of ~~a~~ spreadsheet cells, which are "rows" long and "columns" wide). The unique aspect of matrices is that they can contain only one type of variable (such as just characters or just numbers). For this work, you will convert the VCDB to a numeric matrix. Luckily, the `verisr` package has a function for just such an occasion, and it's appropriately called `veris2matrix()`. Begin by loading the `verisr` package (see Listing 7-5 in Chapter 7 if you haven't installed it yet).

delete: a / comma

OK

LISTING 9-8

remove

```
Listing 9-8
# requires package : verisr (7-5)
# requires VCDB data from chapter 7 (see comments)
library(verisr)
# if you have grabbed the incidents from the VCDB repository at
# https://github.com/vz-risk/VCDB you can set the directory to that
# Otherwise, this should reference the data from chapter 7
jsondir <- '../ch07/data/vcdb/'
# create a veris instance with the vcdb data
vcdb <- json2veris(jsondir)
# finally, you can convert veris object into a numeric matrix
vmat <- veris2matrix(vcdb)
# you may look at the size of
# the matrix with the dim() command
dim(vmat)
## [1] 1643  264
```

Looking at the output from the `dim()` command, the data from VCDB at this point is providing 1,643 rows (one row per incident in the data repository) and 264 columns. Each column is a single enumeration in the data, and you can see what the columns are by looking at the column names with the `colnames()` command. When `veris2matrix()` creates the matrix, it will create a unique column for every enumeration it sees within the VERIS data. For example, if the hacking variety of a SQL injection attack is present, one column in the matrix will be `action.hacking.variety.SQLi` and the column will be a `0` or a `1`, depending on if that particular value was present in the incident, and will be set for all the incidents in the matrix. If none of the incidents is recorded with SQL injection, the whole column will not be present. The entire matrix is just a collection of ones and zeros at this point. This matrix isn't directly helpful as is, but it will serve as the base data from which you'll generate the training data.

Next, you'll identify the variables that you want to compare—the victim industries. In order to get that list, you can simply look at the column names, pull out columns with `victim.industry` in the title, and use them as the variables (see Listing 9-9). You will want to pass that into the function from `verisr` called `foldmatrix()`, which will take in the numeric matrix you just created and the list of variables you're going to fold this matrix on (the victim industries).

You will also pass in two other variables. The first variable is `min`, which enables you to set a minimum threshold for the number of incidents in each industry. If an industry has less than the minimum, it will not be included in the analysis. For this exercise, you'll set `10` as the minimum. The last variable to pass in is `clean`, which asks the function to clean up the final matrix by removing the rows less than the minimum and any columns that are all the same. You will need to clean it up since those variables will not contribute to the analysis. If you were using this approach to do PCA analysis, it would throw an error if you didn't first clean up the matrix.

LISTING 9-9

remove

```
Listing 9-9
# requires package : verisr (7-5),
# requires object : vmat (9-8)
# now pull the column names and extract industries
vmat.names <- colnames(vmat)
industry <- vmat.names[grep('victim.industry', vmat.names)]
# "fold" the matrix on industries
# this pulls all the incidents for the industry
# and compresses so the proportions of the features are represented.
```

continues

Listing 9-9 *(continued)*

```
imat <- foldmatrix(vmat, industry, min=10, clean=T)
dim(imat)
## [1]  17 251
```

OK

There were 17 industries (actually 17 unique two-digit industry codes from the NAICS specification discussed in Chapter 7). It also looks like the function cleaned up 13 columns after folding the matrix. Now you have one row per industry; the columns represent a VERIS variable,and the value represents the proportion of incidents in the industry with the VERIS variable present.

comma

For example, if you were looking at healthcare and SQL injection (again), and 40 of the 100 healthcare incidents involved SQL injection, you would see a 0.4 in the column of `action.hacking.variety.SQLi` in the healthcare row. This is where the comparison occurs. You compare the differences in all of these variables across the industries.

Multidimensional Scaling on Victim Industries

The purpose of all that prep work was to get the data ready to apply some multidimensional scaling to the industries. And finally, this is where the magic happens! As with many tasks within data analysis, you spent more time preparing the data than you spend actually running the analysis. The first command converts your matrix of industries and variables into a distance matrix. This matrix uses the Canberra metric of distance (it does better with values around the origin) to calculate a distance metric between each pair of industries. Then you can feed that distance matrix into the `cmdscale()` function, which projects it onto a two-dimensional plane for plotting (see Listing 9-10).

LISTING 9-10

remove

```
Listing 9-10
# requires object : imat (9-9), vmat (9-8),
# convert the distance matrix
idist <- dist(imat, method='canberra')
# run it through classical MDS
cmd <- cmdscale(idist)
# and take a look at the first few rows returned
head(cmd)
##                            [,1]       [,2]
## victim.industry2.32 -75.080869 -50.662403
## victim.industry2.33 -29.457487  -2.942502
## victim.industry2.42 -24.727909  21.751872
## victim.industry2.44   3.692422   7.840992
## victim.industry2.45 -18.855236  93.787627
## victim.industry2.48 -54.382350  23.166301
```

Look at what is returned from `cmdscale()`. It looks ready to be visualized because those are x and y points. In fact, at this point you could run `plot(cmd)` and see where those points are. However, the points would be unlabeled, and it's worth it to spend some time to create a good-looking plot. In a final plot, it'd be nice if you gave some indication of size per industry, and since you still have that original `vmat` matrix, you should be able to pull out a count of incidents in each industry. Then you should fix those labels because the VERIS data deals with the NAICS industry codes. Although very helpful, the industry codes are not all that user friendly. You can get nicer labels by loading the `industry2` data in the `verisr` package and mapping the industry codes to the shorter labels (see Listing 9-11).

LISTING 9-11

remove

```
Listing 9-11
# requires package : verisr (7-5),
# requires object : cmd (9-10), vmat (9-8),
# get the size of bubbles
ind.counts <- colSums(vmat[ , rownames(cmd)])
# extract the industry label
ind.label <- sapply(rownames(cmd), function(x) {
  tail(unlist(strsplit(x, "[.]")), 1)
})
# load up industry data, included with verisr package
data(industry2)
# create a new list of short tet
txt.label <- industry2$short[which(industry2$code %in% ind.label)]
```

And now you have variables called `ind.counts` and `txt.label` in the same order as the `cmd` object. Now you can create a data frame and create a plot with `ggplot2` (Listing 9-12).

LISTING 9-12

remove

```
Listing 9-12
# requires package : ggplot2
# requires object : cmd (9-10), ind.counts, txt.label (9-11)
library(ggplot2)
indf <- data.frame(x=cmd[ ,1], y=cmd[, 2], label=txt.label, size=ind.
counts)
gg <- ggplot(indf, aes(x, y, label=label, size=size))
gg <- gg + scale_size(trans="log2", range=c(10,30), guide=F)
gg <- gg + geom_point(fill="lightsteelblue", color="white", shape=21)
gg <- gg + xlim(range(indf$x)*1.1) # expand x scale
gg <- gg + geom_text(size=4)
gg <- gg + theme(panel.grid = element_blank(),
                 panel.border = element_blank(),
                 panel.background = element_blank(),
                 axis.text = element_blank(),
                 axis.title = element_blank(),
                 axis.ticks = element_blank())
print(gg)
```

Au: One line of code. OK as set?

Break after comma, put “size=ind.counts” on new line and under the “x=cmd[,1]”

You use the ggplot `theme()` command to strip out everything because the scales and labels are irrelevant for viewing. You want to view the relative location of the industries in respect to other industries. In this plot the x- and y-axes are a distance measurement using the Canberra metric, and the numbers don’t have any meaning or significance for a person viewing them.

Figure 9-6 is rather interesting. You can see that healthcare and government (public) victims appear to be similar (probably due to the large amount of lost devices and error that is reported within those demographics). The little cluster on the top of accommodation and retail is interesting. Those two industries see the bulk of the "point of sale smash and grab" attacks. The cluster of three in the lower-left corner might be worth more investigation. It's hard to say exactly why those three are grouped up there without looking further into the data.

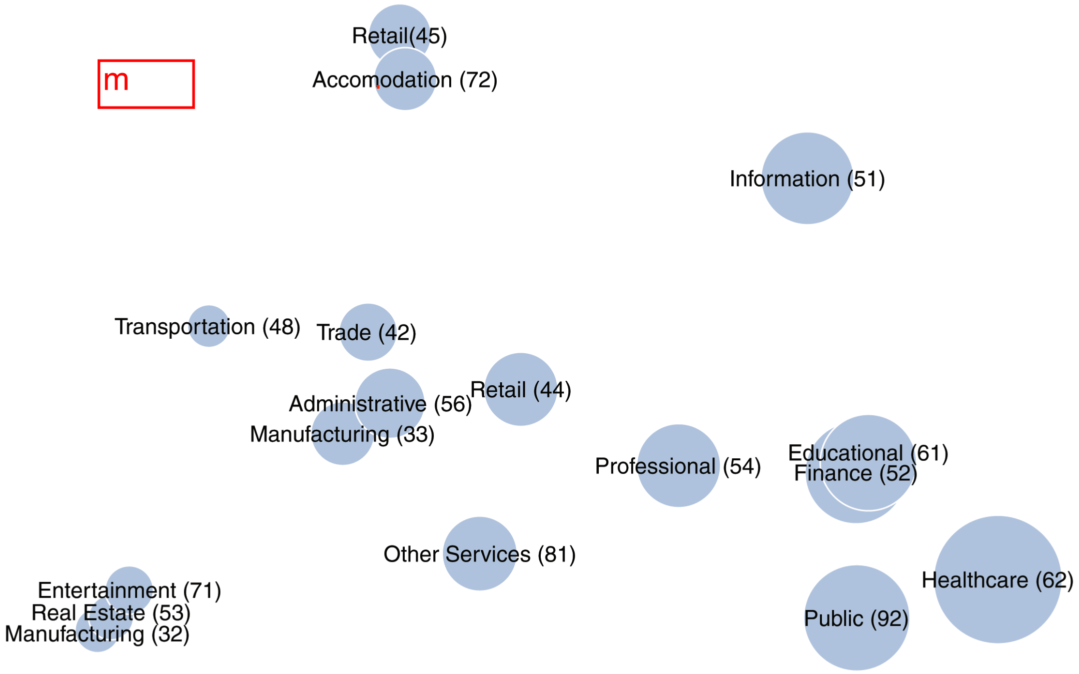


FIGURE 9-6 *Basic MDS plot of breaches within an industry*

Hierarchical Clustering on Victim Industries

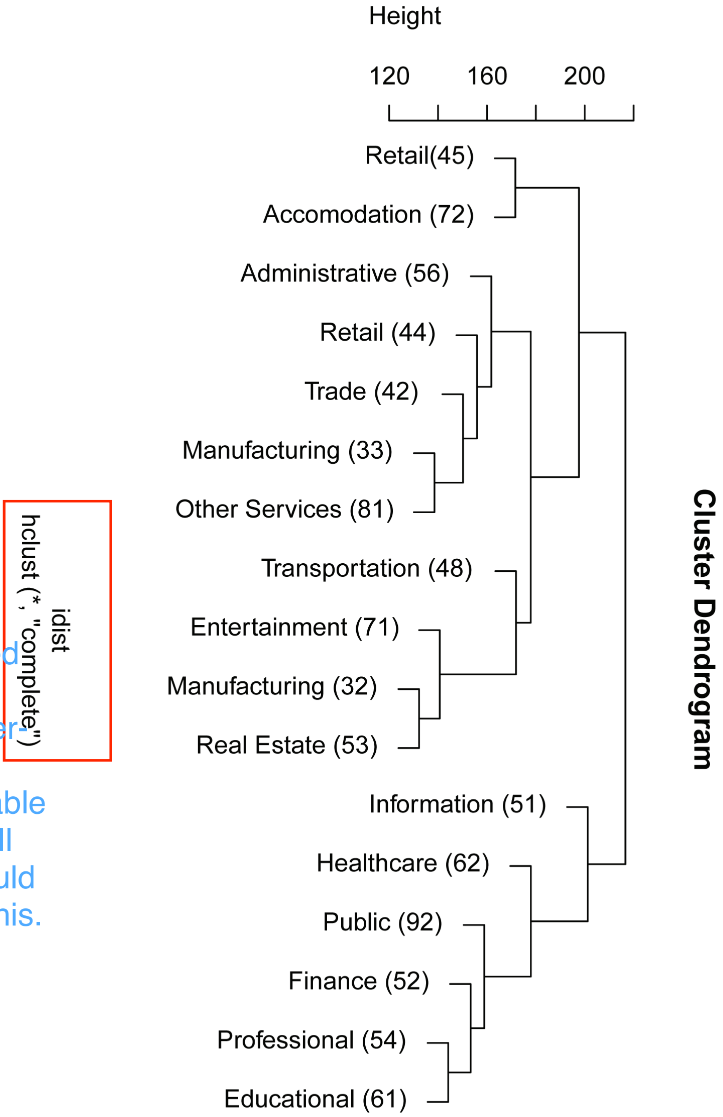
Although it's possible to look at Figure 9-6 and make some clusters visually, you should be careful in doing so. MDS reduces (approximates) a multidimensional object into two dimensions so there will be some perspective and detail lost. Figure 9-6 can serve as a visual to some talking points or, better yet, a point from which to jump into more analysis.

So let's keep going with this data and apply some hierarchical clustering on this data to derive the clusters mathematically. You can simply feed the `idist` distance matrix right into the `hclust` command and plot it (Listing 9-13). To make the labels on the plot user friendly, you should relabel the rows of the original industry matrix and rerun the `dist()` command to recreate the `idist` object with readable labels:

LISTING 9-13

```
Listing 9-13
# requires object : imat (9-9), txt.label (9-11)
# go back and relabel imat
```

```
rownames(imat) <- txt.label
# rerun idist
idist <- dist(imat, 'canberra')
# hclust couldn't be easier
hc <- hclust(idist) # , method="complete")
plot(hc)
```



Au: as meant?

This entire image is rotated incorrectly. It should be rotated 90 degrees counter-clockwise so the “Cluster Dendrogram” title is readable left-to-right. Doing that will match what he reader would see when they generate this.

Figure 9-7 *Hierarchical clustering on victim industries*

Au: OK to cite Fig.9-7 after the figure?

I'm okay with it

From Figure 9-7, you can see how and when things are split off into clusters. The end result is that they are all "clustered" into their own groups since each is unique. Now you can use the `cutree()` command to cut the hierarchical tree down into an appropriate number of clusters. You can try whatever number you like, but this example shows six clusters. Since you are subjectively choosing where to cut the tree, you cannot use this approach to prove there are six clusters here (or however many you choose). But what you can say is that, if the hierarchical cluster is cut at six, these are clusters it produces. Of course, many people won't have a clue what you're talking about, but at least now you do.

the

When you run the `cutree()` command, it will take in the output from the `hclust()` command and the number of clusters to cut it off at. It will return a vector of the numbered clusters that each industry is assigned. You can then use that vector to assign a unique fill color per cluster so the plot will visually be clustered by color (Listing 9-14). You do this by converting the `cuttree()` command to a factor and then adding it to the `indf` object created previously. Then plot it again with the colors, as shown in Figure 9-8.

Listing 9-14

remove

```
Listing 9-14
# requires package : ggplot2
# requires object : indf (9-12), hc (9-13)
# we can now cut off the heirarchical clustering at some level
# and use those levels to color the MDS plot
indf$cluster <- as.factor(cutree(hc, 6))
gg <- ggplot(indf, aes(x, y, label=label, size=size, fill=cluster))
gg <- gg + scale_size(trans="log2", range=c(10,30), guide=F)
gg <- gg + geom_point(color="gray80", shape=21)
gg <- gg + scale_fill_brewer(palette="Set2")
gg <- gg + xlim(range(indf$x)*1.06) # expand x scale
gg <- gg + geom_text(size=4)
gg <- gg + theme(panel.grid = element_blank(),
                 panel.border = element_blank(),
                 panel.background = element_blank(),
                 axis.text = element_blank(),
                 axis.title = element_blank(),
                 legend.position="none",
                 axis.ticks = element_blank())
print(gg)
```

Remember earlier we mentioned that this process converts multi-dimensional data into two dimensions? In looking at Figure 9-8, you can see how the transportation industry is clustered with the industries in the lower left, instead of the visually closer trade industry. Feel free to try changing experimenting with the number of clusters fed into the `cuttree` command. What you can take away from this particular visualization is that there are a lot more questions than answers from it. Why is healthcare in its own cluster? Why is the retail industry with NAICS code 44 so distant from the retail industry with NAICS code 45? What is

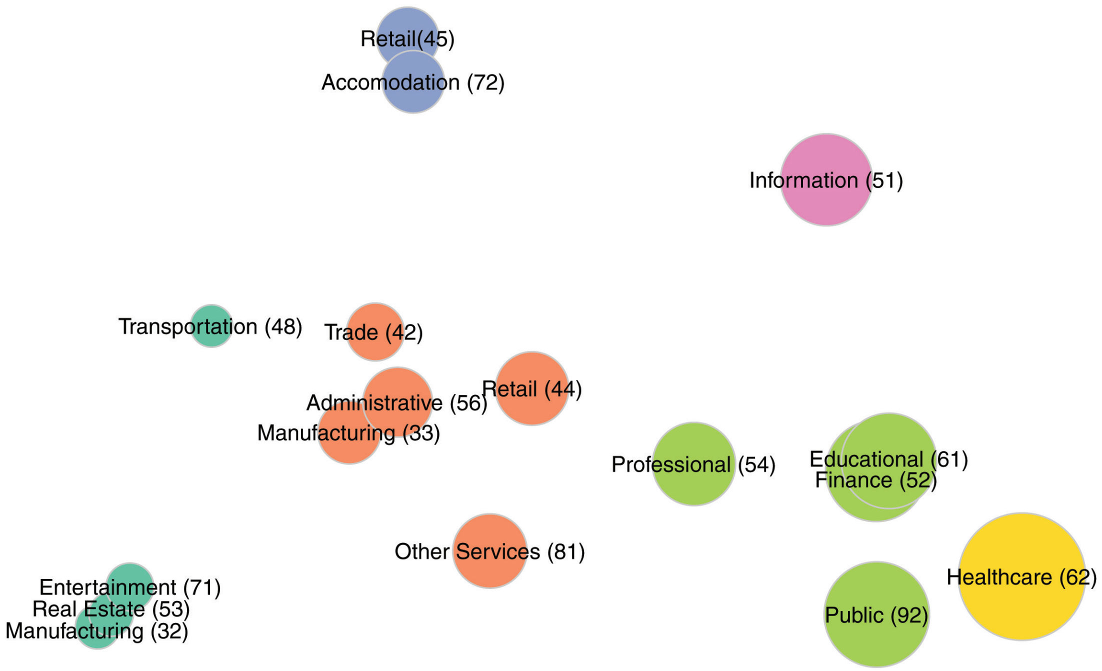


FIGURE 9-8 *Clustered MDS plot of victim industries*

going on in the Information industry that they are out on their own like that? The good news is that we've got the data, and the answers to these questions are just waiting to be discovered.

Please move to previous page

Summary

Open source applications like R and Python have made running machine learning algorithms accessible and relatively easy. However, there is a big difference between running a machine learning algorithm and running a machine learning algorithm ***well***. Like it or not, machine learning has very deep roots in statistics and mathematics. Attempting to dive into these techniques without an understanding of the subtleties and nuances may create more problems than they solve. Having said that, the best way to learn is to jump in head first and splash around. Grab (or generate) data, read the blogs, books, and documentation, and try several approaches. I can guarantee there will be some frustration along the way, but the outcome will be better learning and an overall better understanding of the data and, thus, the world around you. Such knowledge can feed directly into the security decisions you and your organization are facing on a daily basis.

semicolon x2

Recommended Readings

The following are some recommended readings that can further your understanding on some of the topics we touch on in this chapter. For full information on these recommendations and for the sources we cite in the chapter, please see Appendix B. OK

Machine Learning for Hackers **by Drew Conway and John Myles White**—There aren't many machine learning books for beginners, but this book is one of them. It does a very good job at giving hands-on examples in both R and Python. They avoid most of the math but not the challenges with the approaches. Overall, this is a good first book to purchase.

An Introduction to Statistical Learning with Applications in R **by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani**—As you progress beyond the basics and begin looking for that next step, this book is a fantastic next resource. It doesn't shy away from the math, but at the same time, it doesn't dive too deep into it and provides just enough explanation to make sense. The authors spend quite a bit of time on the algorithms, including covering resampling methods, model-selection techniques, and the foundations of all the algorithms.