Chapter 9: Machine Learning

“They know enough who know how to learn.”

Henry Adams

There are two types of people within 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”.[[1]](#footnote-1) How can a computer do anything with being explicitly programmed? Or better yet, this rather well known definition from Tom Mitchell in his 1997 book:

“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? At the off chance that things aren’t clear, we are going to jump right into machine learning with a simplified example. In other words, we will start this chapter off with some data and then not explicitly program the computer to learn from experience E and perform a task T, sound good?

Detecting Malware

Let’s 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’ve got 445 other systems that haven’t been inspected and you’d like to save some time and use the data you have to determine if the other 445 systems you have are infected or not.[[2]](#footnote-2) First start by loading up the data on the hosts you know about and inspecting it.

**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

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). But that will keep the numbers on the same scale. Scaling the variables like this is important when comparing across variables in some machine learning approaches. In order to explore this a bit more, let’s plot this data comparing the processor data to the memory and differentiate based on the malware state.

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)

Figure 9.1 Processor and Memory across systems [FILENAME 793725c06f001]

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 of the known hosts. But before you get too far, you’ll want to do a little planning. First you’ll want to figure out what machine learning algorithm you will 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 and we will touch on model and feature selection later in this chapter.

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) and then how to apply that learning to the systems you don’t know about.

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 up 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. 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. With machine learning, you are asking the computer to learn from the data and then apply that learning. The computer is far more capable and uncovering the differences and subtleties in the data that you would never be able to find by yourself.

type="note"

Throughout this chapter, we will use the term “model” and “algorithm” somewhat interchangeably. The subtle difference is that an algorithm is a series of instructions while we think of a model as a mathematical framework. You will train an algorithm and populate a model. There is a parametric model for linear and logistic regression while random forests are based on a non-parametric algorithm. Though it is also common to see linear regression algorithms or references to a random forest model. We will be using both terms here.

Getting back at the data 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 compare them to the processor and memory usage on the infected hosts. Once the machine has some notion of a difference between the two sets, you can give it some instructions on how to apply that information to the unclassified systems. Remember the goal here is to have the computer give its best guess on whether or not a system is infected with malware or not. Let’s create a short algorithm that is easy to understand and easy to follow:

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

Validating the Algorithm

Before you go off thinking that this algorithm is helpful, you should probably make sure. 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? 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 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 to not perform those steps as part of the model selection. For our example, you will just keep this simple and split the original data into two datasets. In reality, you would probably create multiple datasets from the original data, and train the data over multiple iterations (and validations).

Once you split our data into two groups, you’ll call the first group the training data, since you’ll use that to train the algorithm, and the second you’ll call the test data, since you’ll use that to yup, you guessed it, test your approach. To split the data randomly, make use of the sample() command. You will be pulling a random sample of the indexes of the original data and using that to split into the train and test data. While there’s no definitive rule on where to make the split (different techniques split in different ways), you will simply take one-third for the test data and train the algorithm on the other two-thirds.

# make this repeatable

**set.seed(1492)**

# get 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, size=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, just splitting the data like this once to measure its accuracy is better than just assuming the algorithm is good, but as we’ve mentioned there are more robust methods for validation you’ll want to leverage.

Implementing the Algorithm

We stated earlier that the first step in training this algorithm was to calculate the average (mean) for the infected processor and memory usage and the normal processor and memory usage. You’ll 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 for just 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.

**inf <- colMeans(train[train$state=="Infected", c("proc", "mem")])**

**nrm <- colMeans(train[train$state=="Normal", c("proc", "mem")])**

# view the results

**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. This will take in a single proc and mem value and 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: a2 + b2 = c2, where a and b are the two sides of the triangle and c is the hypotenuse. 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 and which ever is closer (smaller) is the one you will predict.

**predict.malware <- function(proc, 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 and see how it does if you’d like. But 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, the second argument being a “1” to denote to apply over the rows (instead of a “2” for columns), and then you’ll pass in the function. In this case, the function takes in a variable called row, which is character vector in this case. Since it is reduced to a vector, and there is a character in the vector (state and host), the whole vector is converted to a character vector. Therefore, the first thing the function does is convert the two values you’ll want back to numeric values.

prediction <- apply(test, 1, function(row) {

proc <- as.numeric(row[['proc']])

mem <- as.numeric(row[['mem']])

predict.malware(proc, mem)

})

type="note"

First, do no harm. Second, do better than the null model.

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 Zero Access infection data in Chapter 5, the null model could be the calculation of the average (mean) infection across all the states (5253 infections). The null model (for prediction) would estimate 5253 infections for any new state regardless of any data about that state. In this case we are omitting or “nullifying” our 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 are taller.

Once the test data is run through that code, you’ll have a set of predictions and the ability to compare to the real values (see the power of this method?). To look at 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 and divide it by the total number of predictions.

**sum(test$state==prediction)/nrow(test)**

## [1] 0.8780488

This very simple algorithm predicted almost 88% 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% is pretty good for your first machine learning algorithm, congratulations! The results are also pictured in figure 9.2. The way this classifier works is it 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 on the graphic, you can see how any normal systems above the line are mislabeled as well as any infected systems below the line.

Figure 9.2 Predictions from our algorithm [FILENAME 793725c06f002]

type="note"

Spam, Spam, Spam

Open any non-infosec book on machine learning (which may be all of them) and you will probably see a mention of spam filtering if not 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 (and a simplified walk through) 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?

Benefiting from Machine Learning

Now that we’ve walked through 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 towards continual adaptation based on the computers learning directly from the data. Hopefully, the days of thresholds and regular expressions rules are behind us.

Before we can talk about the benefits of machine learning, we should talk about the two types of machine learning algorithms: *supervised* and *unsupervised* and which approach to use is determined more by the type of data you have than any personal preference. **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 either infected with malware or not. Another example is the Zero Access data in Chapter 5, where you knew how many infections in each state and county and you could correlate that with other data about the states and counties. Supervised learning is only possible when you have labeled or known data.

**Unsupervised algorithms** are usually applied to data when what you’d like to know is unknown. For an example of this, think of the recommendation systems at Amazon or Netflix. Those systems begin with a 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. 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. Unsupervised methods enable you to discover relationships and explore the data like no other approach and as we’ll see, you can discover some interesting things with unsupervised learning methods.

type="concept"

To parametric or not to parametric, that is the question

Besides 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 we 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 we grow a series of decision trees that are then used for further classification. Oh, and we were kidding with the title, you will never make a decision between parametric or non-parametric, should decide between parametric models and non-parametric algorithms based on how well they perform and not on the existence of parameters.

Answering Questions with Machine Learning

What types of questions can machine learning answer? What sort of problems can it solve? The opening example in this chapter already introduced the concept of classification, when we tried to determine if the hosts were infected or not. **Classification** is the process of identifying the category something belongs in, or 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 challenges within information security revolve around a single classification problem, “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 what you are more interested in is forecasting a quantity? 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), we can build predictive models make estimates for known future states. Think back to the linear regression analysis we did in Chapter 5. If by some strange turn of events another state appears with 6 million people, that regression analysis using just population would predict just under 5,000 Zero Access infections in that state. While that example isn’t exactly practical, you could use the technique to estimate bandwidth usage next month, or even forecast the size of the next DDoS attack.

Sometimes the end result isn’t prediction of a quantity or category. Sometimes we just want to know about the variables we observe and how they contribute and interact. For these cases we 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 the quantitative relationships. Both supervised and unsupervised methods support inference about the variables and it’s an important part of any model or algorithm.

The last application of machine learning is for **exploration and discovery**. This is an area that 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’d like to know what sort of relationships or patterns exists in the data. Using methods like mutidimensional scaling and principal component analysis will help you explore and gain perspectives of the data that just isn’t possible with simple descriptive statistics.

Measuring good performance

At the core of good learning is good feedback. If you’re creating models and algorithms and never checked if they were doing well, you’d be doomed to repeat the same mistakes and improvement would be nigh impossible. This is such a fundamental concept that several techniques have been developed to measure performance within supervised algorithms and also why unsupervised algorithms are generally not used to prove (or disprove) a theory. We won’t have the space to go into the mathematical details for each method, instead we will discuss 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 approach that will be perfect, so you will want to choose an approach that performs better than all the approaches available (and not toss out a helpful approach simply because its imperfect). All of the fancy math formulas that describe this are just variations on a simple theme: if you are working with quantitative values, you will select the approach where predictions are the closest to the observations. If you are working with a classification system you will want to choose the model with the highest number of correct classifications. We will spend a little time on these two approaches.

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 emphasizes 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 may also be called the error sum of squares, sum square of residuals (SSResidual) or the residual sum of squares (RSS).

Since the SSE is just adding things up, larger sample sizes have larger SSE values. Which makes them impossible to compare between a training data set and the test data set. To standardize the SSE with the sample size, the SSE can be dividing by the sample size and the result can be compared when the sample size is not the same. That term 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. This would make the approach prone to overfitting (see note on overfitting). One approach to comparing quantitative models and algorithms is calculate the MSE and compare across multiple approaches and feature selections as we’ll see when talk about cross validation and bootstrapping below.

type="note"

Overfitting

Since learning algorithms do in fact “learn” what to do from the data, it’s possible that they’ll learn too much from 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. 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 we’ll briefly discuss a few in the next section.

Selecting Features

Before you can train an algorithm and measure its performance, you’ll need to have data to run on. One of the less talked about things within machine learning is how you’ll 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), but within machine learning these variables are referred to as the features of the algorithm. We could say that the processor and memory usage in the opening example were the features we used to train our algorithm.

The tricky part with feature selection is that there are no guidelines to select the initial set of features so this is where your domain expertise will come into play. You’ll collect the data points that may be important and then (as we discussed in the previous section), run them through the algorithm and check if they are actually contributing in the outcome[[3]](#footnote-3). While it’d be 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 some models benefit from fewer variables while some recent approaches do well with everything tossed in.

As an example, if you were thinking of improving on our malware classifier, you could pull variables from netflow 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 you will undoubtedly be wrong and that’s okay. Grab data that makes sense 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 only do well 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 (accurately) the whole algorithm performs and then try to improve.

Using the Best Subset

Given a bunch of features, how could you determine which to include or exclude? One approach is to simply 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 run. On one hand, this approach may discover a combination of features that wouldn’t have been found without this brute force method. However on the other hand, you’ll 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 twenty variables will require well over one million iterations through the algorithm.

There is another caveat with the best subset selection. As the number of features increases, the probability of finding bogus relationships in the features also increase. 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, it will attempt to build up the correct set of features by stepping over them. In a forward stepwise process, the method will begin 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 and the overall best set of features is selected as the final set.

The benefit of this is that it is an enormous reduction in the number of iterations compared to the best subset method. But the draw back is that not all the combinations are tried and so the best may be hidden. In some cases a feature that performs best alone, may not perform best when other features are added. Another approach to help discover a better approach is to do a reverse stepwise comparison, where you’ll start with all the features and then sequentially step backwards removing the least helpful feature until you’re down to one feature again. Then 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 will still need to validate how well the approach performs. While each method may have subtle differences in how they work and the test statistics they generate and focus on, there are a few general approaches for validating how you’re doing that apply to almost all the methods. The most widely used is a method called cross-validation which we will discuss, but as a follow-up you could also look at any other resampling methods such as bootstrapping and the jackknife method.

Applying Cross-Validation

In our opening example we started with 247 observations and then split it into a training set to train our algorithm and a test set to test how the training set did. We arbitrarily split the data so two-thirds was in the training data and the remaining one-third was in the test data. One drawback in doing that is that we were not able to train on the one-third we pulled out, and that introduces more variation in the outcome of the training process. But what if you were to repeat the splitting and testing process so all data was considered and was able to contribute to both the training and test of the algorithm? It is possible to increase the accuracy of our 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 out better than just splitting 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 we will *fold* the data *k times* (once for each partition). For example, you could go back to the data from the first example and divide the data 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 our accuracy across each of the iterations, we will be able to have more confidence in how that approach will perform on new data.

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

Specific Learning Methods

Supervised

There are a lot of methods and it just won’t be possible to survey all of them. We have chosen a handful of approaches and we will briefly touch on what makes them unique, their strengths and weakness, and so on.

Linear Regression (and transformation)

Linear regression is a very popular approach when it comes to either quantitative prediction or inference about the independent variables and for good reason. Linear regression has been around since the late 1800’s and has evolved to be 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 linear itself. 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 [FILENAME 793725c06f003]

Can you see the linear relationship in figure 9.3? Believe it or not, it’s in 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 (thought 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, we would want to create separate input variable for x3, x2 and x and estimate the (linear) coefficients for each of those variables. When transforming the variables like this, care must be taken not to overfit. It would be possible to add enough transformed variables to perfectly fit the training data, but it’d perform horribly on the test or real data.

Linear regression has many variations and nuances that make it powerful especially when combined with some of the techniques we mentioned earlier in this chapter. Classic linear regression would rely on computing a p-value (see chapter 4) to assess the strength of the model and variables. Recently the trend is to also integrate validation methods such as cross-validation to support model selection and validation. See the lm() and glm() command within R for the specifics on how to execute linear regression.

Logistic Regression

While linear regression is designed for predicting quantitative variables, that isn’t all that helpful when the problem isn’t quantitative. For example, in our opening example we want to classify our hosts into infected or not and linear regression wouldn’t be helpful in that circumstance. Instead you can turn to logistic regression, which is a extension of linear regression to model logical outcomes and you can use that to classify between two outcomes. We went back to the opening example and applied logistic regression to the processor and memory usage to the training data, see figure 9.4 for the output.

Figure 9.4 Logistic Regression on infection test data [FILENAME 793725c06f004]

The output (on the x axis) is an estimated probability of a host being infected based on the input variables. We plotted that output 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 to see that given these input values, you would be able to estimate a large portion of the hosts correctly. Though no matter where the cutoff is set (e.g. 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 to apply logistic regression to classify on multiple categories.

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 through a generic sports analogy. Suppose you’d like to pluck a person at random and perform a magic trick (machine learning is seen as magic by many) where you predict his or her favorite sports teams. Suppose when you pluck a person at random, you can ask their neighbors and friends (“k” of them, where k is any consistent number) which teams they cheer for. Then you could look at who the majority of those neighbors cheer for and assume the person you plucked is similar to their neighbors in their sports allegiance.

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

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

Random Forests

Random forests are built on the concept of a decision tree and excel at multi-dimensional data (data with a lot of features). The decision tree is what I.T. people may think of them as a flow chart. We 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. Picture the various types decisions that could be built given data types: if above average for this fork here, if that category go there. Just one decision tree will perform quite poorly. But a technique called boosting was developed to create a whole lot of decision trees and look at the aggregate result from all of them. This provided a huge improvement and worked out quite well. While each individual tree performed 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 worked quite well, but was influenced by noisy features. One or two bad eggs in the basket could bias the result by consistently pulling trees in a weird (and difficult to detect) direction. The random forest technique got around that problem by growing the trees with only a small subset of the features. This made each individual tree an event worse predictor, but the aggregate was improved because the noisy variables were only included in a subset of the features selected for each tree.

Random forests brought 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 whole, you’ll get to 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 will grow hundreds or even thousands of multi-branched decision trees based on random points in random features and can only be done with the aide of a computer. Within R, random forests are available from the appropriately named randomForest package. It’s also worthwhile exploring 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.

type="note"

Comprehension versus Performance

One of the challenges with machine learning is that some of the techniques are so complex and abstract that they are pushing the boundaries of human comprehension. At some point, a trade-off is going to occur where you’ll have to balance the ability to comprehend an approach with the performance it brings. Neural networks are an example of this. In some cases neural networks offers better performance, but they are rather complex and difficult to comprehend (and difficult to tune properly). Because of that, you may opt for an approach that is easier to comprehend and use at the expense of a slight improvement in performance and you should be comfortable with that. Given the complex nature of many decisions with information security, a simple approach with machine learning may be better than a 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 kind of trends exists 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 how many clusters will 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 re-run a k-means clustering you may see get different clusters. If you’d like the k-means to be repeatable be sure the set a static seed prior to running. Figure 9.5 shows the same data with multiple different k-values. The kmeans() function within base R will perform k-means clustering.

Figure 9.5 K-means clustering with centers shown [FILENAME 793725c06f005]

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 x 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. But 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 and when it finds the two most similar, it will cluster those two. This repeats until there is one big cluster at the top.

The advantage you’ll get from 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 we’ll do later in this chapter with breach data when we use the hclust() function on breach data.

Principal Component Analysis

Principal component analysis (PCA) is used to reduce the number of features we 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 will be able to 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 reduced to that format, you’ll be able to pull out the significant components and use those moving forward rather than the larger (and possibly noisy) number of dimensions. Running PCA, like most things in R is the single command of prcomp().

Multidimensional scaling

Sometimes you’ll just want to see the clusters. This is very problematic with multidimensional data because you cannot visualize in more than 3 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 is a technique to perform dimension reduction, because it can squish the multidimensional data into two dimensions so you can visualize the relative similarities between objects. We will run through an example of this technique later in this chapter using the R command cmdscale() for classic multidimensional scaling.

Hands on: clustering breach data

We are going to revisit the data we used in chapter 7 (the VERIS community database data) in order to show multidimensional scaling and hierarchical clustering in action.

The natural approach to breach data is to simply count up the categories, see what occurs more often and then draw some conclusions from that. But the challenge with that approach is that any conclusions drawn may be applied too broadly when the conclusions may not apply to everyone. 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 too.

The problem we are facing then is this:

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

This is a rather interesting challenge because the only thing we start with is a hunch that industries are in fact, different. The best approach then is some of clustering algorithm. If we are able to isolate variables across the industries than we will be able to calculate a distance between the industries in order to determine which are alike and which may display some unique traits.

You’ll begin this analysis by converting the VCDB data to a matrix. We haven’t worked much with 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 only contain one type of variable (such as just characters or just numeric). For this work you will convert the VCDB to a numeric matrix. Luckily the verisr package has a function for just such an occasion appropriately called veris2matrix(). Begin by loading up the verisr package (see chapter 7 if you haven’t installed it yet).

# if you haven't installed verisr yet, do these two steps:

**library(devtools)**

# now let's install the verisr package

**install\_github("verisr", "jayjacobs")**

# and load up the verisr library

**library(verisr)**

# grab the incidents from the VCDB repository

# https://github.com/vz-risk/VCDB

# set the dir to the incidents/ directly of that

**jsondir <- '../VCDB/incidents'**

# 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] 1733 272

Looking at the output from the dim() command, the data from VCDB at this point is providing 1,733 rows (one row per incident in the data repository) and 272 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 zero or a one 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, than 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 to you, but it will serve as the base data from which we’ll generate our training data. Next you’ll want to identify the variables that you want to compare and that is all of the victim industries. In order to get that list, you can simply look at the column names and pull out columns with “victim.industry” in the title and use those as the variables. We 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 we’re going to fold this matrix on (the victim industries).

You will also pass in two other variables the first will be min, so we can 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 (if they exist). You will need to clean it up since those variables will not contribute to the analysis. If you were using this to do PCA analysis, it would throw an error if you didn’t first clean up the matrix.

# 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

**imat <- foldmatrix(vmat, industry, min=10, clean=T)**

**dim(imat)**

## [1] 17 260

There were 17 industries (actually 17 unique 2-digit industry codes from the NAICS specification discussed in chapter 7). It also looks like the function cleaned up 12 columns that had all the same value once we folded the matrix. Now what we have is 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. For example, if we 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 is going to occur. You’ll be comparing 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 just spent more time preparing the data than you will to actually run the analysis. The first command will convert your matrix of industries and variables into a distance matrix. This will use 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 that will project it onto a two-dimensional plane for plotting.

# convert the industry matrix to a 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 80.170772 69.46704

## victim.industry2.33 10.506350 -15.67658

## victim.industry2.42 30.600109 -14.18568

## victim.industry2.44 -8.841317 -20.57215

## victim.industry2.45 34.683463 -88.69814

## victim.industry2.48 53.410562 -17.15443

Looking 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 spend some time to create a good-looking 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’d want to fix those labels because the VERIS data deals with the NAICS industry codes. While very helpful, the industry codes are not all that user friendly. You can get nicer labels by loading up the industry2 data in the verisr package and mapping the industry codes to the shorter labels.

# get a count of incidents per industry

**ind.counts <- colSums(vmat[ , rownames(cmd)])**

# extract the industry label

**ind.label <- sapply(rownames(cmd), function(x) {**

# split, convert to vector and take the last (tail) element

# which is a 2-digit NAICS code

**tail(unlist(strsplit(x, "[.]")), 1)**

**})**

# load up industry data from verisr package

**data(industry2)**

# create a new list of short names

**txt.label <- industry2$short[which(industry2$code %in% ind.label)]**

And now you have variables called ind.counts and txt.label both in the same order as the cmd object. Now you can create a data frame and create a plot with ggplot2.

**indf <- data.frame(x=cmd[ ,1], y=cmd[, 2], label=ind.label, size=ind.counts)**

**library(ggplot2)**

**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="black", shape=21)**

**gg <- gg + xlim(range(df$x)\*1.04)** # 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())**

We use the ggplot theme() command to strip out everything because the scales and labels are somewhat irrelevant. We will just 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 won’t have any meaning or significance for a person viewing it.

Figure 9.3 Basic MDS plot [FILENAME]

Figure 9.3 is rather interesting to look at. You can see that healthcare and government (public) victims appear to be similar (probably due to large amount of lost devices and error that is reported from those demographics. The little cluster on the bottom 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 upper right might be worth more investigation. It’s hard to say exactly why those three are grouped up there without looking further into the data.

Hierarchical Clustering on Victim Industries

While it’d be possible to look at figure 9.3 and make some clusters visually, you should be careful in doing so. MDS reduces (approximates) a multidimensional object into 2-dimensions so there will be some perspective and detail lost. Figure 9.3 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 and let’s apply some hierarchical clustering on this data and derive the clusters mathematically. We can just simply feed the idist distance matrix right into the hclust command and plot it. To make the labels on the plot easy to look at, you should relabel the rows of in the original industry matrix and re-run the dist() command to recreate the idist object with readable labels.

# go back and relabel imat with txt.label

# (both created above)

**rownames(imat) <- txt.label**

# rerun idist

**idist <- dist(imat, 'canberra')**

# hclust couldn't be easier

**hc <- hclust(idist) # , method="complete")**

**plot(hc)**

Figure 9.4 Hierarchical Clustering on Victim Industries [FILENAME 793725c06f004]

Using figure 9.4 we can see how things are split off into clusters but we can also see when. The end result is that they are all “clustered” into their own group since each is unique. Now you can use a command called cutree() to cut that hierarchical tree down into how many clusters you think would be appropriate. You can try whatever you’d like, but we’ll demonstrate showing 5 clusters. Since we’re are subjectively choosing where to cut the tree, we cannot use this approach to say, “see there are clearly 5 clusters here” (or how many you choose). But what we can say is “if the hierarchical cluster is cut at 5, these are clusters it produces.” Of course, many people won’t have a clue what you’re talking about, but now you do.

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 cluster each industry is assigned. You would like to use that vector to assign a unique fill color per cluster so the plot will visually be clustered by color. You’ll do this by converting the cuttree() command to a factor and then adding it to the ()indf object we created above and plot it again with the colors.

# 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, 5))**

**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="black", shape=21)**

**gg <- gg + xlim(range(df$x)\*1.04)** # 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())**

Figure 9.5 Clustered MDS plot of victim industries [FILENAME 793725c06f005]

Summary

Open source work like R and python have made running machine learning algorithms relatively easy. But 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 and 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. We 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 us.

For Further Reading

*Machine Learning for Hackers* by Drew Conway and John Myles White

There are not 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 and they avoid most of the math but not the challenges with the approaches. Overall this would be a good first book to purchase. Also, keep an eye on the book website for updates and changes.

*An Introduction to Statistical Learning with Applications in R* by James, Witten, Hastie and Tibshirani.

As you progress beyond the basics and are looking for that next step with machine learning (and statistics in general) this book is a fantastic book for you. It won’t shy away from the math, but at the same time, it will not dive too deep into it and will provide just enough explanation to make sense. The authors spend quite a bit of time around the algorithms and talk about resampling methods, model selection techniques and the foundations of all the algorithms.

1. <http://whatis.techtarget.com/definition/machine-learning> retrieved 10/5/2013 [↑](#footnote-ref-1)
2. Please keep in mind that this is a contrived demonstration of a machine learning approach, for a much better application of machine learning for malware detection, see Bilge, Leyla, et al. [↑](#footnote-ref-2)
3. By doing this relatively simple step that is supported in most every statistical approach, you will have surpassed most every risk analysis model within information security, congratulations. [↑](#footnote-ref-3)