CHAPTER 4

Spam Filters, Naive Bayes, and Wrangling

The contributor for this chapter is Jake Hofman. Jake is at Microsoft Research after recently leaving Yahoo! Research. He got a PhD in physics at Columbia and regularly teaches a fantastic course on data-driven modeling at Columbia, as well as a newer course in computational social science.

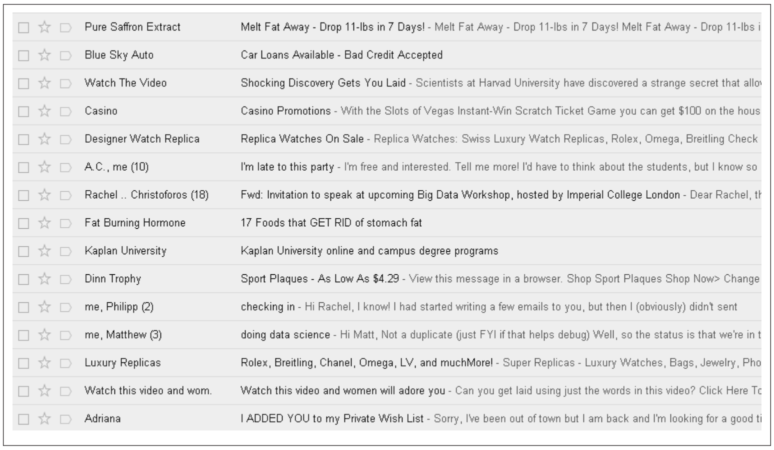
As with our other presenters, we first took a look at Jake's data science profile. It turns out he is an expert on a category that he added to the data science profile called "data wrangling." He confessed that he doesn't know if he spends so much time on it because he's good at it or because he's bad at it. (He's good at it.)

Thought Experiment: Learning by Example

Let's start by looking at a bunch of text shown in Figure 4-1, whose rows seem to contain the subject and first line of an email in an inbox.

You may notice that several of the rows of text look like spam.

How did you figure this out? Can you write code to automate the spam filter that your brain represents?



Pure Saffron Extract	Melt Fat Away - Drop 11-lbs in 7 Days! - Melt Fat Away - Drop 11-lbs in 7 Days! Melt Fat Away - Drop 11-lbs i
Blue Sky Auto	Car Loans Available - Bad Credit Accepted
Watch The Video	Shocking Discovery Gets You Laid - Scientists at Harvad University have discovered a strange secret that allo
Casino	Casino Promotions - With the Slots of Vegas Instant-Win Scratch Ticket Game you can get $100 on the hous
Designer Watch Replica	Replica Watches On Sale - Replica Watches: Swiss Luxury Watch Replicas, Rolex, Omega, Breitling Check
A.C., me (10)	I'm late to this party - I'm free and interested. Tell me more! I'd have to think about the students, but I know so
Rachel .. Christoforos (18)	Fwd: Invitation to speak at upcoming Big Data Workshop, hosted by Imperial College London - Dear Rachel, th
Fat Burning Hormone	17 Foods that GET RID of stomach fat
Kaplan University	Kaplan University online and campus degree programs
Dinn Trophy	Sport Plaques - As Low As $4.29 - View this message in a browser. Shop Sport Plaques Shop Now> Change
me, Philipp (2)	checking in - Hi Rachel, I know! I had started writing a few emails to you, but then I (obviously) didn't sent
me, Matthew (3)	doing data science - Hi Matt, Not a duplicate (just FYI if that helps debug) Well, so the status is that we're in t
Luxury Replicas	Rolex, Breitling, Chanel, Omega, LV, and muchMore! - Super Replicas - Luxury Watches, Bags, Jewelry, Pho
Watch this video and wom.	Watch this video and women will adore you - Can you get laid using just the words in this video? Click Here To
Adriana	I ADDED YOU to my Private Wish List - Sorry, I've been out of town but I am back and I'm looking for a good ti

Figure 4-1. Suspiciously spammy

Rachel's class had a few ideas about what things might be clear signs of spam:

- Any email is spam if it contains Viagra references. That's a good rule to start with, but as you've likely seen in your own email, people figured out this spam filter rule and got around it by modifying the spelling. (It's sad that spammers are so smart and aren't working on more important projects than selling lots of Viagra…)
- Maybe something about the length of the subject gives it away as spam, or perhaps excessive use of exclamation points or other punctuation. But some words like "Yahoo!" are authentic, so you don't want to make your rule too simplistic.

And here are a few suggestions regarding code you could write to identify spam:

- Try a probabilistic model. In other words, should you not have simple rules, but have many rules of thumb that aggregate together to provide the probability of a given email being spam? This is a great idea.
- What about k-nearest neighbors or linear regression? You learned about these techniques in the previous chapter, but do they apply to this kind of problem? (Hint: the answer is "No.")

In this chapter, we'll use Naive Bayes to solve this problem, which is in some sense in between the two. But first…

Why Won't Linear Regression Work for Filtering Spam?

Because we're already familiar with linear regression and that's a tool in our toolbelt, let's start by talking through what we'd need to do in order to try to use linear regression. We already know that's *not* what we're going to use, but let's talk it through to get to why. Imagine a dataset or matrix where each row represents a different email message (it could be keyed by email_id). Now let's make each word in the email a *feature*—this means that we create a column called "Viagra," for example, and then for any message that has the word Viagra in it at least once, we put a 1 in; otherwise we assign a 0. Alternatively, we could put the number of times the word appears. Then each column represents the appearance of a different word.

Thinking back to the previous chapter, in order to use linear regression, we need to have a training set of emails where the messages have *already* been labeled with some outcome variable. In this case, the outcomes are either spam or not. We could do this by having human evaluators label messages "spam," which is a reasonable, albeit time-intensive, solution. Another way to do it would be to take an existing spam filter such as Gmail's spam filter and use those labels. (Now of course if you already had a Gmail spam filter, it's hard to understand why you might also want to build another spam filter in the first place, but let's just say you do.) Once you build a model, email messages would come in without a label, and you'd use your model to predict the labels.

The first thing to consider is that your target is binary (0 if not spam, 1 if spam)—you wouldn't get a 0 or a 1 using linear regression; you'd get a number. Strictly speaking, this option really isn't ideal; linear regression is aimed at modeling a continuous output and this is binary.

This issue is basically a nonstarter. We should use a model appropriate for the data. But if we wanted to fit it in R, in theory it could still work. R doesn't check for us whether the model is appropriate or not. We could go for it, fit a linear model, and then use that to predict and then choose a critical value so that above that predicted value we call it "1" and below we call it "0."

But if we went ahead and tried, it still wouldn't work because there are too many variables compared to observations! We have on the order of 10,000 emails with on the order of 100,000 words. This won't work. Technically, this corresponds to the fact that the matrix in the equation for linear regression is not invertible—in fact, it's not even close. Moreover, maybe we can't even store it because it's so huge.

Maybe we could limit it to the top 10,000 words? Then we could at least have an invertible matrix. Even so, that's too many variables versus observations to feel good about it. With carefully chosen feature selection and domain expertise, we could limit it to 100 words and that could be enough! But again, we'd still have the issue that linear regression is not the appropriate model for a binary outcome.

Aside: State of the Art for Spam Filters

In the last five years, people have started using stochastic gradient methods to avoid the noninvertible (overfitting) matrix problem. Switching to logistic regression with stochastic gradient methods helped a lot, and can account for correlations between words. Even so, Naive Bayes is pretty impressively good considering how simple it is.

How About k-nearest Neighbors?

We're going to get to Naive Bayes shortly, we promise, but let's take a minute to think about trying to use k-nearest neighbors (k-NN) to create a spam filter. We would still need to choose features, probably corresponding to words, and we'd likely define the value of those features to be 0 or 1, depending on whether the word is present or not. Then, we'd need to define when two emails are "near" each other based on which words they both contain.

Again, with 10,000 emails and 100,000 words, we'll encounter a problem, different from the noninvertible matrix problem. Namely, the space we'd be working in has *too many dimensions*. Yes, computing distances in a 100,000-dimensional space requires lots of computational work. But that's not the real problem.

The real problem is even more basic: even our nearest neighbors are really far away. This is called "the curse of dimensionality," and it makes k-NN a poor algorithm in this case.

Aside: Digit Recognition

Say you want an algorithm to recognize pictures of hand-written digits as shown in Figure 4-2. In this case, k-NN works well.

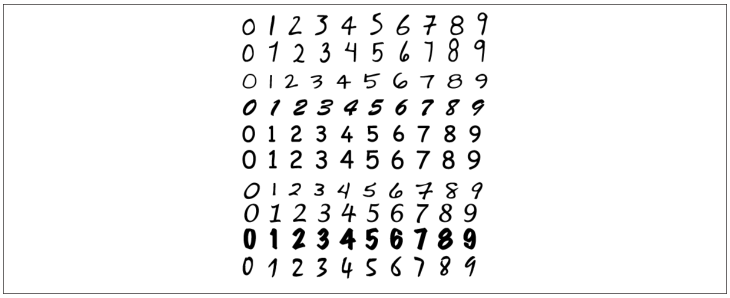


Figure 4-2. Handwritten digits

To set it up, you take your underlying representation apart pixel by pixel—say in a 16x16 grid of pixels—and measure how bright each pixel is. Unwrap the 16x16 grid and put it into a 256-dimensional space, which has a natural archimedean metric. That is to say, the distance between two different points on this space is the square root of the sum of the squares of the differences between their entries. In other words, it's the length of the vector going from one point to the other or vice versa. Then you apply the k-NN algorithm.

If you vary the number of neighbors, it changes the shape of the boundary, and you can tune k to prevent overfitting. If you're careful, you can get 97% accuracy with a sufficiently large dataset.

Moreover, the result can be viewed in a "confusion matrix." A *confusion matrix* is used when you are trying to classify objects into k bins, and is a $k \times k$ matrix corresponding to actual label versus predicted label, and the (i, j)th element of the matrix is a count of the number of items that were actually labeled i that were predicted to have label j. From a confusion matrix, you can get *accuracy*, the proportion of total predictions that were correct. In the previous chapter, we discussed the misclassification rate. Notice that accuracy = 1 - misclassification rate.

Naive Bayes

So are we at a loss now that two methods we're familiar with, linear regression and k-NN, won't work for the spam filter problem? No! Naive Bayes is another classification method at our disposal that scales well and has nice intuitive appeal.

Bayes Law

Let's start with an even simpler example than the spam filter to get a feel for how Naive Bayes works. Let's say we're testing for a rare disease, where 1% of the population is infected. We have a highly sensitive and specific test, which is not quite perfect:

- 99% of sick patients test positive.
- 99% of healthy patients test negative.

Given that a patient tests positive, what is the probability that the patient is actually sick?

A naive approach to answering this question is this: Imagine we have 100 × 100 = 10,000 perfectly representative people. That would mean that 100 are sick, and 9,900 are healthy. Moreover, after giving all of them the test we'd get 99 sick people testing sick, but 99 healthy people testing sick as well. If you test positive, in other words, you're equally likely to be healthy or sick; the answer is 50%. A tree diagram of this approach is shown in Figure 4-3.

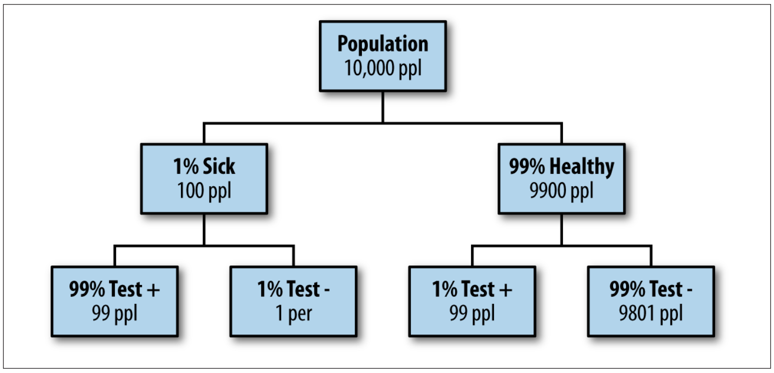


Figure 4-3. Tree diagram to build intuition

Let's do it again using fancy notation so we'll feel smart.

Recall from your basic statistics course that, given events x and y, there's a relationship between the probabilities of either event (denoted $p(x)$ and $p(y)$), the joint probabilities (both happen, which is denoted $p(x,y)$), and conditional probabilities (event x happens given y happens, denoted $p(x|y)$) as follows:

$$p(y|x)p(x) = p(x,y) = p(x|y)p(y)$$

Using that, we solve for $p(y|x)$ (assuming $p(x) \neq 0$) to get what is called *Bayes' Law*:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

The denominator term, $p(x)$, is often implicitly computed and can thus be treated as a "normalization constant." In our current situation, set y to refer to the event "I am sick," or "sick" for shorthand; and set x to refer to the event "the test is positive," or "+" for shorthand. Then we actually know, or at least can compute, every term:

$$p(sick|+) = \frac{p(+|sick)p(sick)}{p(+)} = \frac{0.99 \cdot 0.01}{0.99 \cdot 0.01 + 0.01 \cdot 0.99} = 0.50 = 50\%$$

A Spam Filter for Individual Words

So how do we use Bayes' Law to create a good spam filter? Think about it this way: if the word "Viagra" appears, this adds to the probability that the email is spam. But it's not conclusive, yet. We need to see what else is in the email.

Let's first focus on just one word at a time, which we generically call "word." Then, applying Bayes' Law, we have:

$$p(spam|word) = \frac{p(word|spam)p(spam)}{p(word)}$$

The righthand side of this equation is computable using enough prelabeled data. If we refer to nonspam as "ham" then we only need compute $p(word|spam)$, $p(word|ham)$, $p(spam)$, and $p(ham) = 1\text{-}p(spam)$, because we can work out the denominator using the formula we used earlier in our medical test example, namely:

$$p(word) = p(word|spam)p(spam) + p(word|ham)p(ham)$$

In other words, we've boiled it down to a counting exercise: $p(spam)$ counts spam emails versus all emails, $p(word|spam)$ counts the prevalence of those spam emails that contain "word," and $p(word|ham)$ counts the prevalence of the ham emails that contain "word."

To do this yourself, go online and download Enron emails. Let's build a spam filter on that dataset. This really this means we're building a new spam filter on top of the spam filter that existed for the employees of Enron. We'll use their definition of spam to train our spam filter. (This does mean that if the spammers have learned anything since 2001, we're out of luck.)

We could write a quick-and-dirty shell script in bash that runs this, which Jake did. It downloads and unzips the file and creates a folder; each text file is an email; spam and ham go in separate folders.

Let's look at some basic statistics on a random Enron employee's email. We can count 1,500 spam versus 3,672 ham, so we already know $p(spam)$ and $p(ham)$. Using command-line tools, we can also count the number of instances of the word "meeting" in the spam folder:

```
grep -il meeting enron1/spam/*.txt | wc -l
```

This gives 16. Do the same for his ham folder, and we get 153. We can now compute the chance that an email is spam only knowing it contains the word "meeting":

$$\hat{p}(spam) = 1500 / (1500 + 3672) = .29$$

$$\hat{p}(ham) = .71$$

$$\hat{p}(meeting|spam) = 16 / 1500 = .0106$$

$$\hat{p}(meeting|ham) = 153 / 3672 = .0416$$

$$\hat{p}(spam|meeting) = \hat{p}(meeting|spam) * \hat{p}(spam) / \hat{p}(meeting) = (.0106 * .29) / (.0106 * .29 + .0416 * .71) = 0.09 = 9\%$$

Take note that we didn't need a fancy programming environment to get this done.

Next, we can try:

- "money": 80% chance of being spam
- "viagra": 100% chance
- "enron": 0% chance

This illustrates that the model, as it stands, is overfitting; we are getting overconfident because of biased data. Is it really a slam-dunk that any email containing the word "Viagra" is spam? It's of course possible to write a nonspam email with the word "Viagra," as well as a spam email with the word "Enron."

A Spam Filter That Combines Words: Naive Bayes

Next, let's do it for all the words. Each email can be represented by a binary vector, whose jth entry is 1 or 0 depending on whether the jth word appears. Note this is a huge-ass vector, considering how many words we have, and we'd probably want to represent it with the indices of the words that actually show up.

The model's output is the probability that we'd see a given word vector given that we know it's spam (or that it's ham). Denote the email vector to be x and the various entries x_j, where the j indexes the words. For now we can denote "is spam" by c, and we have the following model for $p(x|c)$, i.e., the probability that the email's vector looks like this considering it's spam:

$$p(x|c) = \Pi_j \theta_{jc}{}^{x_j} \left(1 - \theta_{jc}\right)^{(1-x_j)}$$

The θ here is the probability that an individual word is present in a spam email. We saw how to compute that in the previous section via counting, and so we can assume we've separately and parallelly computed that for every word.

We are modeling the words *independently* (also known as "independent trials"), which is why we take the product on the righthand side of the preceding formula and don't count how many times they are present. That's why this is called "naive," because we know that there are actually certain words that tend to appear together, and we're ignoring this.

So back to the equation, it's a standard trick when we're dealing with a product of probabilities to take the log of both sides to get a summation instead:

$$log(p(x|c)) = \sum_j x_j log(\theta_j / (1-\theta_j)) + \sum_j log(1-\theta_j)$$

It's helpful to take the log because multiplying together tiny numbers can give us numerical problems.

The term $log(\theta_j / (1-\theta_j))$ doesn't depend on a given email, just the word, so let's rename it w_j and assume we've computed it once and stored it. Same with the quantity $\sum_j log(1-\theta_j) = w_0$. Now we have:

$$log(p(x|c)) = \sum_j x_j w_j + w_0$$

The weights that vary by email are the x_js. We need to compute them separately for each email, but that shouldn't be too hard.

We can put together what we know to compute $p(x|c)$, and then use Bayes' Law to get an estimate of $p(c|x)$, which is what we actually want —the other terms in Bayes' Law are easier than this and don't require separate calculations per email. We can also get away with not computing all the terms if we only care whether it's more likely to be spam or to be ham. Then only the varying term needs to be computed.

You may notice that this ends up looking like a linear regression, but instead of computing the coefficients w_j by inverting a huge matrix, the weights come from the Naive Bayes' algorithm.

This algorithm works pretty well, and it's "cheap" to train if we have a prelabeled dataset to train on. Given a ton of emails, we're more or less just counting the words in spam and nonspam emails. If we get more training data, we can easily increment our counts to improve our filter. In practice, there's a global model, which we personalize to individuals. Moreover, there are lots of hardcoded, cheap rules before an email gets put into a fancy and slow model.

Here are some references for more about Bayes' Law:

- "Idiot's Bayes - not so stupid after all?" (The whole paper is about why it doesn't suck, which is related to redundancies in language.)
- "Naive Bayes at Forty: The Independence Assumption in Information"
- "Spam Filtering with Naive Bayes - Which Naive Bayes?"

Fancy It Up: Laplace Smoothing

Remember the θ_j from the previous section? That referred to the probability of seeing a given word (indexed by j) in a spam email. If you think about it, this is just a ratio of counts: $\theta_j = n_{jc} / n_c$, where n_{jc} denotes the number of times that word appears in a spam email and n_c denotes the number of times that word appears in any email.

Laplace Smoothing refers to the idea of replacing our straight-up estimate of θ_j with something a bit fancier:

$$\theta_{jc} = \left(n_{jc} + \alpha\right) / \left(n_c + \beta\right)$$

We might fix $\alpha = 1$ and $\beta = 10$, for example, to prevent the possibility of getting 0 or 1 for a probability, which we saw earlier happening with "viagra." Does this seem totally ad hoc? Well, if we want to get fancy, we can see this as equivalent to having a prior and performing a maximal likelihood estimate. Let's get fancy! If we denote by ML the maximal likelihood estimate, and by D the dataset, then we have:

$$\theta_{ML} = argmax_{\theta} p(D|\theta)$$

In other words, the vector of values $\theta_j = n_{jc} / n_c$ is the answer to the question: for what value of θ were the data D most probable? If we assume independent trials again, as we did in our first attempt at Naive Bayes, then we want to choose the θ_j to separately maximize the following quantity for each j:

$$log\left(\theta_j^{n_{jc}}\left(1 - \theta_j\right)^{n_c - n_{jc}}\right)$$

If we take the derivative, and set it to zero, we get:

$$\hat{\theta}_j = n_{jc} / n_c$$

In other words, just what we had before. So what we've found is that the maximal likelihood estimate recovers your result, as long as we assume independence.

Now let's add a prior. For this discussion we can suppress the j from the notation for clarity, but keep in mind that we are fixing the jth word to work on. Denote by MAP the *maximum a posteriori* likelihood:

$$\theta_{MAP} = argmax\ p(\theta | D)$$

This similarly answers the question: given the data I saw, which parameter θ is the most likely?

Here we will apply the spirit of Bayes's Law to transform θ_{MAP} to get something that is, up to a constant, equivalent to $p(D|\theta) \cdot p(\theta)$. The term $p(\theta)$ is referred to as the "prior," and we have to make an assumption about its form to make this useful. If we make the assumption that the probability distribution of θ is of the form $\theta^{\alpha}(1-\theta)^{\beta}$, for some α and β, then we recover the Laplace Smoothed result.

Is That a Reasonable Assumption?

Recall that θ is the chance that a word is in spam if that word is in some email. On the one hand, as long as both $\alpha > 0$ and $\beta > 0$, this distribution vanishes at both 0 and 1. This is reasonable: you want very few words to be expected to *never* appear in spam or to *always* appear in spam.

On the other hand, when α and β are large, the shape of the distribution is bunched in the middle, which reflects the prior that most words are equally likely to appear in spam or outside spam. That doesn't seem true either.

A compromise would have α and β be positive but small, like 1/5. That would keep your spam filter from being too overzealous without having the wrong idea. Of course, you could relax this prior as you have more and better data; in general, strong priors are only needed when you don't have sufficient data.

Comparing Naive Bayes to k-NN

Sometimes α and β are called “pseudocounts.” Another common name is “hyperparameters.” They’re fancy but also simple. It’s up to you, the data scientist, to set the values of these two hyperparameters in the numerator and denominator for smoothing, and it gives you two knobs to tune. By contrast, k-NN has one knob, namely *k*, the number of neighbors. Naive Bayes is a linear classifier, while k-NN is not. The curse of dimensionality and large feature sets are a problem for k-NN, while Naive Bayes performs well. k-NN requires no training (just load in the dataset), whereas Naive Bayes does. Both are examples of supervised learning (the data comes labeled).

Sample Code in bash

```
#!/bin/bash
#
# file: enron_naive_bayes.sh
#
# description: trains a simple one-word naive bayes spam
# filter using enron email data
#
# usage: ./enron_naive_bayes.sh <word>
#
# requirements:
#   wget
#
# author: jake hofman (gmail: jhofman)
#

# how to use the code
if [ $# -eq 1 ]
    then
    word=$1
else
    echo "usage: enron_naive_bayes.sh <word>"
    exit
fi

# if the file doesn't exist, download from the web
if ! [ -e enron1.tar.gz ]
    then
    wget 'http://www.aueb.gr/users/ion/data/
    enron-spam/preprocessed/enron1.tar.gz'
fi

# if the directory doesn't exist, uncompress the .tar.gz
if ! [ -d enron1 ]
```

```
    then
    tar zxvf enron1.tar.gz
fi

# change into enron1
cd enron1

# get counts of total spam, ham, and overall msgs
Nspam=`ls -l spam/*.txt | wc -l`
Nham=`ls -l ham/*.txt | wc -l`
Ntot=$Nspam+$Nham

echo $Nspam spam examples
echo $Nham ham examples

# get counts containing word in spam and ham classes
Nword_spam=`grep -il $word spam/*.txt | wc -l`
Nword_ham=`grep -il $word ham/*.txt | wc -l`

echo $Nword_spam "spam examples containing $word"
echo $Nword_ham "ham examples containing $word"

# calculate probabilities using bash calculator "bc"
Pspam=`echo "scale=4; $Nspam / ($Nspam+$Nham)" | bc`
Pham=`echo "scale=4; 1-$Pspam" | bc`
echo
echo "estimated P(spam) =" $Pspam
echo "estimated P(ham) =" $Pham

Pword_spam=`echo "scale=4; $Nword_spam / $Nspam" | bc`
Pword_ham=`echo "scale=4; $Nword_ham / $Nham" | bc`
echo "estimated P($word|spam) =" $Pword_spam
echo "estimated P($word|ham) =" $Pword_ham

Pspam_word=`echo "scale=4; $Pword_spam*$Pspam" | bc`
Pham_word=`echo "scale=4; $Pword_ham*$Pham" | bc`
Pword=`echo "scale=4; $Pspam_word+$Pham_word" | bc`
Pspam_word=`echo "scale=4; $Pspam_word / $Pword" | bc`
echo
echo "P(spam|$word) =" $Pspam_word

# return original directory
cd ..
```

Scraping the Web: APIs and Other Tools

As a data scientist, you're not always just handed some data and asked to go figure something out based on it. Often, you have to actually figure out how to go get some data you need to ask a question, solve a problem, do some research, etc. One way you can do this is with an API. For the sake of this discussion, an API (application programming interface) is something websites provide to developers so they can download data from the website easily and in standard format. (APIs are used for much more than this, but for your purpose this is how you'd typically interact with one.) Usually the developer has to register and receive a "key," which is something like a password. For example, the *New York Times* has an API here.

Warning about APIs

Always check the terms and services of a website's API before scraping. Additionally, some websites limit what data you have access to through their APIs or how often you can ask for data without paying for it.

When you go this route, you often get back weird formats, sometimes in JSON, but there's no standardization to this standardization; i.e., different websites give you different "standard" formats.

One way to get beyond this is to use Yahoo's YQL language, which allows you to go to the Yahoo! Developer Network and write SQL-like queries that interact with many of the APIs on common sites like this:

```
select * from flickr.photos.search where text="Cat"
and api_key="lksdjflskjdfsldkfj" limit 10
```

The output is standard, and you only have to parse this in Python once.

But what if you want data when there's no API available?

In this case you might want to use something like the Firebug extension for Firefox. You can "inspect the element" on any web page, and Firebug allows you to grab the field inside the HTML. In fact, it gives you access to the full HTML document so you can interact and edit. In this way you can see the HTML as a map of the page and Firebug as a kind of tour guide.

After locating the stuff you want inside the HTML, you can use curl, wget, grep, awk, perl, etc., to write a quick-and-dirty shell script to grab what you want, especially for a one-off grab. If you want to be more systematic, you can also do this using Python or R.

Other parsing tools you might want to look into include:

lynx and lynx --dump
: Good if you pine for the 1970s. Oh wait, 1992. Whatever.

Beautiful Soup
: Robust but kind of slow.

Mechanize (or here)
: Super cool as well, but it doesn't parse JavaScript.

PostScript
: Image classification.

Thought Experiment: Image Recognition

How do you determine if an image is a landscape or a headshot?

Start with collecting data. You either need to get someone to label these things, which is a lot of work, or you can grab lots of pictures from flickr and ask for photos that have already been tagged.

Represent each image with a binned RGB (red, green, blue) intensity histogram. In other words, for each pixel, and for each of red, green, and blue, which are the basic colors in pixels, you measure the intensity, which is a number between 0 and 255. Represent each image with a binned RGB (red, green, blue) intensity histogram. In other words, for each pixel, and for each of red, green, and blue, which are the basic colors in pixels, you measure the intensity, which is a number between 0 and 255.

Then draw three histograms, one for each basic color, showing how many pixels had which intensity. It's better to do a binned histogram, so have counts of the number of pixels of intensity 0-51, etc. In the end, for each picture, you have 15 numbers, corresponding to 3 colors and 5 bins per color. We are assuming here that every picture has the same number of pixels.

Finally, use k-NN to decide how much "blue" makes a landscape versus a headshot. You can tune the hyperparameters, which in this case are the number of bins as well as k.

Jake's Exercise: Naive Bayes for Article Classification

This problem looks at an application of Naive Bayes for multiclass text classification. First, you will use the *New York Times* Developer API to fetch recent articles from several sections of the *Times*. Then, using the simple Bernoulli model for word presence, you will implement a classifier which, given the text of an article from the *New York Times*, predicts the section to which the article belongs.

First, register for a *New York Times* Developer API key and request access to the Article Search API. After reviewing the API documentation, write code to download the 2,000 most recent articles for each of the Arts, Business, Obituaries, Sports, and World sections. (Hint: Use the nytd_section_facet facet to specify article sections.) The developer console may be useful for quickly exploring the API. Your code should save articles from each section to a separate file in a tab-delimited format, where the first column is the article URL, the second is the article title, and the third is the body returned by the API.

Next, implement code to train a simple Bernoulli Naive Bayes model using these articles. You can consider documents to belong to one of C categories, where the label of the ith document is encoded as $y_i \in 0, 1, 2, \ldots C$—for example, Arts = 0, Business = 1, etc.—and documents are represented by the sparse binary matrix X, where $X_{ij} = 1$ indicates that the ith document contains the jth word in our dictionary.

You train by counting words and documents within classes to estimate θ_{jc} and θ_c:

$$\hat{\theta}_{jc} = \frac{n_{jc} + \alpha - 1}{n_c + \alpha + \beta - 2}$$

$$\hat{\theta}_c = \frac{n_c}{n}$$

where n_{jc} is the number of documents of class c containing the jth word, n_c is the number of documents of class c, n is the total number of documents, and the user-selected hyperparameters α and β are pseudocounts that "smooth" the parameter estimates. Given these estimates and the words in a document x, you calculate the log-odds for each class (relative to the base class c = 0) by simply adding the class-specific weights of the words that appear to the corresponding bias term:

$$\log\left(\frac{p(y=c|x)}{p(y=0|x)}\right)=\sum_j \hat{w}_{jc}x_j+\hat{w}_{0c}$$

where

$$\hat{w}_{jc}=\log\frac{\hat{\theta}_{jc}\left(1-\hat{\theta}_{j0}\right)}{\hat{\theta}_{j0}\left(1-\hat{\theta}_{jc}\right)}$$

$$\hat{w}_{0c}=\sum_j \log\frac{1-\hat{\theta}_{jc}}{1-\hat{\theta}_{j0}}+\log\frac{\hat{\theta}_c}{\hat{\theta}_0}$$

Your code should read the title and body text for each article, remove unwanted characters (e.g., punctuation), and tokenize the article contents into words, filtering out stop words (given in the stopwords file). The training phase of your code should use these parsed document features to estimate the weights $\hat{w}$, taking the hyperparameters α and β as input. The prediction phase should then accept these weights as inputs, along with the features for new examples, and output posterior probabilities for each class.

Evaluate performance on a randomized 50/50 train/test split of the data, including accuracy and runtime. Comment on the effects of changing α and β. Present your results in a (5×5) confusion table showing counts for the actual and predicted sections, where each document is assigned to its most probable section. For each section, report the top 10 most informative words. Also present and comment on the top 10 "most difficult to classify" articles in the test set.

Briefly discuss how you expect the learned classifier to generalize to other contexts, e.g., articles from other sources or time periods.

Sample R Code for Dealing with the NYT API

```
# author: Jared Lander
#
# hard coded call to API
```

```
theCall <- "http://api.nytimes.com/svc/search/v1/
article?format=json&query=nytd_section_facet:
[Sports]&fields=url,title,body&rank=newest&offset=0
&api-key=Your_Key_Here"

# we need the rjson, plyr, and RTextTools packages
require(plyr)
require(rjson)
require(RTextTools)

## first let's look at an individual call
res1 <- fromJSON(file=theCall)
# how long is the result
length(res1$results)
# look at the first item
res1$results[[1]]
# the first item's title
res1$results[[1]]$title
# the first item converted to a data.frame, Viewed in the data
viewer
View(as.data.frame(res1$results[[1]]))

# convert the call results into a data.frame, should be 10
rows by 3 columns
resList1 <- ldply(res1$results, as.data.frame)
View(resList1)

## now let's build this for multiple calls
# build a string where we will substitute the section for the
first %s and offset for the second %s
theCall <- "http://api.nytimes.com/svc/search/v1/
article?format=json&query=nytd_section_facet:
[%s]&fields=url,title,body&rank=newest&offset=%s
&api-key=Your_Key_Here"
# create an empty list to hold 3 result sets
resultsSports <- vector("list", 3)
## loop through 0, 1 and 2 to call the API for each value
for(i in 0:2)
{
    # first build the query string replacing the first %s with
    Sport and the second %s with the current value of i
    tempCall <- sprintf(theCall, "Sports", i)
    # make the query and get the json response
    tempJson <- fromJSON(file=tempCall)
    # convert the json into a 10x3 data.frame and
    save it to the list
    resultsSports[[i + 1]] <- ldply(tempJson$results,
    as.data.frame)
}
# convert the list into a data.frame
resultsDFSports <- ldply(resultsSports)
```

```
# make a new column indicating this comes from Sports
resultsDFSports$Section <- "Sports"

## repeat that whole business for arts
## ideally you would do this in a more eloquent manner, but
this is just for illustration
resultsArts <- vector("list", 3)
for(i in 0:2)
{
    tempCall <- sprintf(theCall, "Arts", i)
    tempJson <- fromJSON(file=tempCall)
    resultsArts[[i + 1]] <- ldply(tempJson$results,
    as.data.frame)
}
resultsDFArts <- ldply(resultsArts)
resultsDFArts$Section <- "Arts"

# combine them both into one data.frame
resultBig <- rbind(resultsDFArts, resultsDFSports)
dim(resultBig)
View(resultBig)

## now time for tokenizing
# create the document-term matrix in english, removing numbers
and stop words and stemming words
doc_matrix <- create_matrix(resultBig$body, language="english",
removeNumbers=TRUE, removeStopwords=TRUE, stemWords=TRUE)
doc_matrix
View(as.matrix(doc_matrix))

# create a training and testing set
theOrder <- sample(60)
container <- create_container(matrix=doc_matrix,
labels=resultBig$Section, trainSize=theOrder[1:40],
testSize=theOrder[41:60], virgin=FALSE)
```

Historical Context: Natural Language Processing

The example in this chapter where the raw data is text is just the tip of the iceberg of a whole field of research in computer science called natural language processing (NLP). The types of problems that can be solved with NLP include machine translation, where given text in one language, the algorithm can translate the text to another language; semantic analysis; part of speech tagging; and document classification (of which spam filtering is an example). Research in these areas dates back to the 1950s.

CHAPTER 5

Logistic Regression

The contributor for this chapter is Brian Dalessandro. Brian works at Media6Degrees as a VP of data science, and he's active in the research community. He's also served as cochair of the KDD competition. M6D (also known as Media 6 Degrees) is a startup in New York City in the online advertising space. Figure 5-1 shows Brian's data science profile —his y-axis is scaled from Clown to Rockstar.

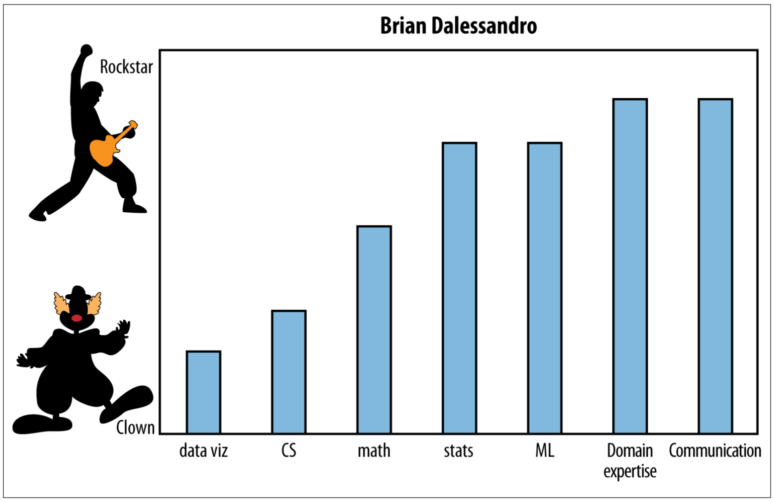


Figure 5-1. Brian's data science profile

Brian came to talk to the class about logistic regression and evaluation, but he started out with two thought experiments.