CHAPTER 13
Naive Bayes

It is well for the heart to be naive and for the mind not to be.
—Anatole France

A social network isn't much good if people can't network. Accordingly, DataSciencester has a popular feature that allows members to send messages to other members. And while most members are responsible citizens who send only well-received "how's it going?" messages, a few miscreants persistently spam other members about get-rich schemes, no-prescription-required pharmaceuticals, and for-profit data science credentialing programs. Your users have begun to complain, and so the VP of Messaging has asked you to use data science to figure out a way to filter out these spam messages.

A Really Dumb Spam Filter

Imagine a "universe" that consists of receiving a message chosen randomly from all possible messages. Let S be the event "the message is spam" and B be the event "the message contains the word *bitcoin*." Bayes's theorem tells us that the probability that the message is spam conditional on containing the word *bitcoin* is:

$$P(S|B) = [P(B|S)P(S)]/[P(B|S)P(S) + P(B|\neg S)P(\neg S)]$$

The numerator is the probability that a message is spam *and* contains *bitcoin*, while the denominator is just the probability that a message contains *bitcoin*. Hence, you can think of this calculation as simply representing the proportion of *bitcoin* messages that are spam.

If we have a large collection of messages we know are spam, and a large collection of messages we know are not spam, then we can easily estimate $P(B|S)$ and $P(B|\neg S)$. If

we further assume that any message is equally likely to be spam or not spam (so that $P(S) = P(\neg S) = 0.5$), then:

$$P(S|B) = P(B|S)/[P(B|S) + P(B|\neg S)]$$

For example, if 50% of spam messages have the word *bitcoin*, but only 1% of nonspam messages do, then the probability that any given *bitcoin*-containing email is spam is:

$$0.5/(0.5 + 0.01) = 98\,\%$$

A More Sophisticated Spam Filter

Imagine now that we have a vocabulary of many words, w_1 ..., w_n. To move this into the realm of probability theory, we'll write X_i for the event "a message contains the word w_i." Also imagine that (through some unspecified-at-this-point process) we've come up with an estimate $P(X_i|S)$ for the probability that a spam message contains the ith word, and a similar estimate $P(X_i|\neg S)$ for the probability that a nonspam message contains the ith word.

The key to Naive Bayes is making the (big) assumption that the presences (or absences) of each word are independent of one another, conditional on a message being spam or not. Intuitively, this assumption means that knowing whether a certain spam message contains the word *bitcoin* gives you no information about whether that same message contains the word *rolex*. In math terms, this means that:

$$P(X_1 = x_1, \ldots, X_n = x_n|S) = P(X_1 = x_1|S) \times \cdots \times P(X_n = x_n|S)$$

This is an extreme assumption. (There's a reason the technique has *naive* in its name.) Imagine that our vocabulary consists *only* of the words *bitcoin* and *rolex*, and that half of all spam messages are for "earn bitcoin" and that the other half are for "authentic rolex." In this case, the Naive Bayes estimate that a spam message contains both *bitcoin* and *rolex* is:

$$P(X_1 = 1, X_2 = 1|S) = P(X_1 = 1|S)P(X_2 = 1|S) = .5 \times .5 = .25$$

since we've assumed away the knowledge that *bitcoin* and *rolex* actually never occur together. Despite the unrealisticness of this assumption, this model often performs well and has historically been used in actual spam filters.

The same Bayes's theorem reasoning we used for our "bitcoin-only" spam filter tells us that we can calculate the probability a message is spam using the equation:

$$P(S|X = x) = P(X = x|S)/[P(X = x|S) + P(X = x|\neg S)]$$

The Naive Bayes assumption allows us to compute each of the probabilities on the right simply by multiplying together the individual probability estimates for each vocabulary word.

In practice, you usually want to avoid multiplying lots of probabilities together, to prevent a problem called *underflow*, in which computers don't deal well with floating-point numbers that are too close to 0. Recalling from algebra that $\log(ab) = \log a + \log b$ and that $\exp(\log x) = x$, we usually compute $p_1 * \cdots * p_n$ as the equivalent (but floating-point-friendlier):

$$\exp(\log(p_1) + \cdots + \log(p_n))$$

The only challenge left is coming up with estimates for $P(X_i|S)$ and $P(X_i|\neg S)$, the probabilities that a spam message (or nonspam message) contains the word w_i. If we have a fair number of "training" messages labeled as spam and not spam, an obvious first try is to estimate $P(X_i|S)$ simply as the fraction of spam messages containing the word w_i.

This causes a big problem, though. Imagine that in our training set the vocabulary word *data* only occurs in nonspam messages. Then we'd estimate $P(\text{data}|S) = 0$. The result is that our Naive Bayes classifier would always assign spam probability 0 to *any* message containing the word *data*, even a message like "data on free bitcoin and authentic rolex watches." To avoid this problem, we usually use some kind of smoothing.

In particular, we'll choose a *pseudocount*—k—and estimate the probability of seeing the ith word in a spam message as:

$$P(X_i|S) = (k + \text{number of spams containing } w_i)/(2k + \text{number of spams})$$

We do similarly for $P(X_i|\neg S)$. That is, when computing the spam probabilities for the ith word, we assume we also saw k additional nonspams containing the word and k additional nonspams not containing the word.

For example, if *data* occurs in 0/98 spam messages, and if k is 1, we estimate $P(\text{data}|S)$ as 1/100 = 0.01, which allows our classifier to still assign some nonzero spam probability to messages that contain the word *data*.

Implementation

Now we have all the pieces we need to build our classifier. First, let's create a simple function to tokenize messages into distinct words. We'll first convert each message to lowercase, then use `re.findall` to extract "words" consisting of letters, numbers, and apostrophes. Finally, we'll use `set` to get just the distinct words:

```
from typing import Set
import re

def tokenize(text: str) -> Set[str]:
    text = text.lower()                         # Convert to lowercase,
    all_words = re.findall("[a-z0-9']+", text)  # extract the words, and
    return set(all_words)                       # remove duplicates.

assert tokenize("Data Science is science") == {"data", "science", "is"}
```

We'll also define a type for our training data:

```
from typing import NamedTuple

class Message(NamedTuple):
    text: str
    is_spam: bool
```

As our classifier needs to keep track of tokens, counts, and labels from the training data, we'll make it a class. Following convention, we refer to nonspam emails as *ham* emails.

The constructor will take just one parameter, the pseudocount to use when computing probabilities. It also initializes an empty set of tokens, counters to track how often each token is seen in spam messages and ham messages, and counts of how many spam and ham messages it was trained on:

```
from typing import List, Tuple, Dict, Iterable
import math
from collections import defaultdict

class NaiveBayesClassifier:
    def __init__(self, k: float = 0.5) -> None:
        self.k = k  # smoothing factor

        self.tokens: Set[str] = set()
        self.token_spam_counts: Dict[str, int] = defaultdict(int)
        self.token_ham_counts: Dict[str, int] = defaultdict(int)
        self.spam_messages = self.ham_messages = 0
```

Next, we'll give it a method to train it on a bunch of messages. First, we increment the `spam_messages` and `ham_messages` counts. Then we tokenize each message text, and for each token we increment the `token_spam_counts` or `token_ham_counts` based on the message type:

```
    def train(self, messages: Iterable[Message]) -> None:
        for message in messages:
            # Increment message counts
            if message.is_spam:
                self.spam_messages += 1
            else:
                self.ham_messages += 1

            # Increment word counts
            for token in tokenize(message.text):
                self.tokens.add(token)
                if message.is_spam:
                    self.token_spam_counts[token] += 1
                else:
                    self.token_ham_counts[token] += 1
```

Ultimately we'll want to predict $P(\text{spam} \mid \text{token})$. As we saw earlier, to apply Bayes's theorem we need to know $P(\text{token} \mid \text{spam})$ and $P(\text{token} \mid \text{ham})$ for each token in the vocabulary. So we'll create a "private" helper function to compute those:

```
    def _probabilities(self, token: str) -> Tuple[float, float]:
        """returns P(token | spam) and P(token | ham)"""
        spam = self.token_spam_counts[token]
        ham = self.token_ham_counts[token]

        p_token_spam = (spam + self.k) / (self.spam_messages + 2 * self.k)
        p_token_ham = (ham + self.k) / (self.ham_messages + 2 * self.k)

        return p_token_spam, p_token_ham
```

Finally, we're ready to write our `predict` method. As mentioned earlier, rather than multiplying together lots of small probabilities, we'll instead sum up the log probabilities:

```
    def predict(self, text: str) -> float:
        text_tokens = tokenize(text)
        log_prob_if_spam = log_prob_if_ham = 0.0

        # Iterate through each word in our vocabulary
        for token in self.tokens:
            prob_if_spam, prob_if_ham = self._probabilities(token)

            # If *token* appears in the message,
            # add the log probability of seeing it
            if token in text_tokens:
                log_prob_if_spam += math.log(prob_if_spam)
                log_prob_if_ham += math.log(prob_if_ham)

            # Otherwise add the log probability of _not_ seeing it,
            # which is log(1 - probability of seeing it)
            else:
                log_prob_if_spam += math.log(1.0 - prob_if_spam)
                log_prob_if_ham += math.log(1.0 - prob_if_ham)
```

```
    prob_if_spam = math.exp(log_prob_if_spam)
    prob_if_ham = math.exp(log_prob_if_ham)
    return prob_if_spam / (prob_if_spam + prob_if_ham)
```

And now we have a classifier.

Testing Our Model

Let's make sure our model works by writing some unit tests for it.

```
messages = [Message("spam rules", is_spam=True),
            Message("ham rules", is_spam=False),
            Message("hello ham", is_spam=False)]

model = NaiveBayesClassifier(k=0.5)
model.train(messages)
```

First, let's check that it got the counts right:

```
assert model.tokens == {"spam", "ham", "rules", "hello"}
assert model.spam_messages == 1
assert model.ham_messages == 2
assert model.token_spam_counts == {"spam": 1, "rules": 1}
assert model.token_ham_counts == {"ham": 2, "rules": 1, "hello": 1}
```

Now let's make a prediction. We'll also (laboriously) go through our Naive Bayes logic by hand, and make sure that we get the same result:

```
text = "hello spam"

probs_if_spam = [
    (1 + 0.5) / (1 + 2 * 0.5),      # "spam"  (present)
    1 - (0 + 0.5) / (1 + 2 * 0.5),  # "ham"   (not present)
    1 - (1 + 0.5) / (1 + 2 * 0.5),  # "rules" (not present)
    (0 + 0.5) / (1 + 2 * 0.5)       # "hello" (present)
]

probs_if_ham = [
    (0 + 0.5) / (2 + 2 * 0.5),      # "spam"  (present)
    1 - (2 + 0.5) / (2 + 2 * 0.5),  # "ham"   (not present)
    1 - (1 + 0.5) / (2 + 2 * 0.5),  # "rules" (not present)
    (1 + 0.5) / (2 + 2 * 0.5),      # "hello" (present)
]

p_if_spam = math.exp(sum(math.log(p) for p in probs_if_spam))
p_if_ham = math.exp(sum(math.log(p) for p in probs_if_ham))

# Should be about 0.83
assert model.predict(text) == p_if_spam / (p_if_spam + p_if_ham)
```

This test passes, so it seems like our model is doing what we think it is. If you look at the actual probabilities, the two big drivers are that our message contains *spam*

(which our lone training spam message did) and that it doesn't contain *ham* (which both our training ham messages did).

Now let's try it on some real data.

Using Our Model

A popular (if somewhat old) dataset is the SpamAssassin public corpus (*https://spamassassin.apache.org/old/publiccorpus/*). We'll look at the files prefixed with *20021010*.

Here is a script that will download and unpack them to the directory of your choice (or you can do it manually):

```
from io import BytesIO  # So we can treat bytes as a file.
import requests         # To download the files, which
import tarfile          # are in .tar.bz format.

BASE_URL = "https://spamassassin.apache.org/old/publiccorpus"
FILES = ["20021010_easy_ham.tar.bz2",
         "20021010_hard_ham.tar.bz2",
         "20021010_spam.tar.bz2"]

# This is where the data will end up,
# in /spam, /easy_ham, and /hard_ham subdirectories.
# Change this to where you want the data.
OUTPUT_DIR = 'spam_data'

for filename in FILES:
    # Use requests to get the file contents at each URL.
    content = requests.get(f"{BASE_URL}/{filename}").content

    # Wrap the in-memory bytes so we can use them as a "file."
    fin = BytesIO(content)

    # And extract all the files to the specified output dir.
    with tarfile.open(fileobj=fin, mode='r:bz2') as tf:
        tf.extractall(OUTPUT_DIR)
```

It's possible the location of the files will change (this happened between the first and second editions of this book), in which case adjust the script accordingly.

After downloading the data you should have three folders: *spam*, *easy_ham*, and *hard_ham*. Each folder contains many emails, each contained in a single file. To keep things *really* simple, we'll just look at the subject lines of each email.

How do we identify the subject line? When we look through the files, they all seem to start with "Subject:". So we'll look for that:

```
import glob, re
```

```
# modify the path to wherever you've put the files
path = 'spam_data/*/*'

data: List[Message] = []

# glob.glob returns every filename that matches the wildcarded path
for filename in glob.glob(path):
    is_spam = "ham" not in filename

    # There are some garbage characters in the emails; the errors='ignore'
    # skips them instead of raising an exception.
    with open(filename, errors='ignore') as email_file:
        for line in email_file:
            if line.startswith("Subject:"):
                subject = line.lstrip("Subject: ")
                data.append(Message(subject, is_spam))
                break  # done with this file
```

Now we can split the data into training data and test data, and then we're ready to build a classifier:

```
import random
from scratch.machine_learning import split_data

random.seed(0)      # just so you get the same answers as me
train_messages, test_messages = split_data(data, 0.75)

model = NaiveBayesClassifier()
model.train(train_messages)
```

Let's generate some predictions and check how our model does:

```
from collections import Counter

predictions = [(message, model.predict(message.text))
               for message in test_messages]

# Assume that spam_probability > 0.5 corresponds to spam prediction
# and count the combinations of (actual is_spam, predicted is_spam)
confusion_matrix = Counter((message.is_spam, spam_probability > 0.5)
                           for message, spam_probability in predictions)

print(confusion_matrix)
```

This gives 84 true positives (spam classified as "spam"), 25 false positives (ham classified as "spam"), 703 true negatives (ham classified as "ham"), and 44 false negatives (spam classified as "ham"). This means our precision is 84 / (84 + 25) = 77%, and our recall is 84 / (84 + 44) = 65%, which are not bad numbers for such a simple model. (Presumably we'd do better if we looked at more than the subject lines.)

We can also inspect the model's innards to see which words are least and most indicative of spam:

```
def p_spam_given_token(token: str, model: NaiveBayesClassifier) -> float:
    # We probably shouldn't call private methods, but it's for a good cause.
    prob_if_spam, prob_if_ham = model._probabilities(token)

    return prob_if_spam / (prob_if_spam + prob_if_ham)

words = sorted(model.tokens, key=lambda t: p_spam_given_token(t, model))

print("spammiest_words", words[-10:])
print("hammiest_words", words[:10])
```

The spammiest words include things like *sale*, *mortgage*, *money*, and *rates*, whereas the hammiest words include things like *spambayes*, *users*, *apt*, and *perl*. So that also gives us some intuitive confidence that our model is basically doing the right thing.

How could we get better performance? One obvious way would be to get more data to train on. There are a number of ways to improve the model as well. Here are some possibilities that you might try:

- Look at the message content, not just the subject line. You'll have to be careful how you deal with the message headers.
- Our classifier takes into account every word that appears in the training set, even words that appear only once. Modify the classifier to accept an optional `min_count` threshold and ignore tokens that don't appear at least that many times.
- The tokenizer has no notion of similar words (e.g., *cheap* and *cheapest*). Modify the classifier to take an optional `stemmer` function that converts words to *equivalence classes* of words. For example, a really simple stemmer function might be:

  ```
  def drop_final_s(word):
      return re.sub("s$", "", word)
  ```

  Creating a good stemmer function is hard. People frequently use the Porter stemmer (*http://tartarus.org/martin/PorterStemmer/*).
- Although our features are all of the form "message contains word w_i," there's no reason why this has to be the case. In our implementation, we could add extra features like "message contains a number" by creating phony tokens like *contains:number* and modifying the `tokenizer` to emit them when appropriate.

For Further Exploration

- Paul Graham's articles "A Plan for Spam" (*http://www.paulgraham.com/spam.html*) and "Better Bayesian Filtering" (*http://www.paulgraham.com/better.html*) are interesting and give more insight into the ideas behind building spam filters.

- scikit-learn (*https://scikit-learn.org/stable/modules/naive_bayes.html*) contains a `BernoulliNB` model that implements the same Naive Bayes algorithm we implemented here, as well as other variations on the model.