Project 2: Spamassassin

Due: 28 September 2017

In this project you'll write a system that is essentially the core piece of Spamassassin (and some other spam filters): a naïve Bayes classifier.

The exact details of such classifiers will be covered in class over the next few days, but the basic idea in this application is that you have to answer the question, "based on the words in this email, is it more likely to be spam, or more likely to be non-spam?" To do this, your program will need to train itself on an established body of data—known as a *corpus*[1]—that contains both spam and ham,[2] and learn some statistics about the sorts of words that appear in the two different kinds of email.

Objectives

In the course of this project, the successful student will:

- implement a naïve Bayes classifier based on empirically-observed data,
- manage multiple open files to read data distributed throughout several directories in the filesystem,
- use maps/dictionaries to maintain frequency information of words in different contexts, and
- apply standard techniques for probabilistic computation in the presence of very small numbers and absent data.

The data sets

I've downloaded the *LingSpam* corpus (Androutsopoulos *et al* 2000) and made it accessible from the `/home/shared/262` directory. Inside the directory `lingspam_public` you'll find a few subdirectories with slightly different versions of the corpus; to begin with, use the `bare` directory, which has files with the smallest amount of preprocessing.

[1] Plural *corpora*, which is pronounced COR-prr-ruh, because Latin.
[2] That is, non-spam. Because ham is what you'd rather get.

The corpus is divided into ten roughly equal parts, and within each part there are two kinds of files: ham, whose names start with numbers, and spam, whose names start with `sp`. Within each file is three lines: one that gives the "Subject:" line of the email, a blank line, and then one that contains the entire body of the email, broken up into individual words (with punctuation marks counting as separate words), separated by spaces.

For your convenience (especially if you're not familiar with the system calls to list directories at runtime), I've put a full index of all the email messages in a file called `index`. That means that if your program knows to look in the directory `/home/shared/262/lingspam_public/bare/` and the index has a line like `part1/3-378msg2.txt` , you should be able to just concatenate those two to get a filename that you can open with an `ifstream` or a `Scanner` or whatever. (Even if you *do* know how to list directories at runtime, you should still use the index—it makes it a little easier to build and control your test cases.)

Prep work

Unlike the last project, I'm not giving you any code to start from. Your initial work involves getting the outer shell of the program running: getting a directory name *as a command line option*,[3] opening the `index` file in that directory, reading the filenames it lists, and looping through to open and process each of the files in directories labeled `part1` through `part9` (i.e. excluding files in `parttest`) of the corpus.

To demonstrate that you've done this, your checkpoint program will print (to standard output) the first "word" in the body (that's the third line) of each of these files, each on its own line.

Thus, if run with an argument of `/home/shared/262/lingspam_public/bare/` , the first few lines of output would be

```
>
the
.
"
could
```

[3] If you're not sure how, look it up or see me. In C++ this involves using `argc` and `argv`; in Java, the `args` array; and in other languages there are other useful incantations.

and there would eventually be a total of 2602 lines (because you should only be reading through part 9).[4]

This work is due next Thursday **at 4pm**. You'll have a chance to ask questions about it in class on Thursday to clear up any last-minute issues, but you really don't want to wait until then to start it. When you're ready to hand it in, use the `handin` script as described at the end of this document.

Design work

Once you've got the shell of the program running (or perhaps while you are working on that, but the design work depends on course content we won't cover until next week), you can start thinking about the algorithm and data structure design.

1. Devise an extremely short and simple data set that can serve to fuel test cases. It should have just a small handful of messages, each with just a few words; their collective contents should be able to test the different aspects of the empirical data collection system (in particular, some words should be repeated, and some not).

2. What built-in library data structures will you use to store the frequency information you'll need to compute the probabilities? Express the types of these data structures in the programming language you intend to use (and verify that its library includes those types!). Give descriptive names to them so you'll remember what they're for; and write out what the contents of those data structures would be given the data example you wrote out in the previous item.

3. The end goal will be to perform actual classification on messages, based on comparing the probability that a message is spam given the words it contains:
$$p(S|W) \propto p(S) \prod_{w \in W} p(w|S)$$

[4] And if you don't want to keep typing the full directory name, *do not* make a full copy of the files. Instead, you can make a symbolic link; inside your working directory, type
`ln -s /home/shared/262/lingspam_public/ lingspam`
and then you can refer to `lingspam/bare/` or whatever.

with the probability that it is not spam given the words it contains:

$$p(\neg S|W) \propto p(\neg S) \prod_{w \in W} p(w|\neg S)$$

In pseudocode or description, indicate how your stored data (from the previous item) would be accessed and used to compute these formulas.

4. Give some thought to your overall program design. You'll need to collect information about both ham and spam, and keep that all separate; and you'll need to use that information to classify test messages. How do you plan to organise that? Are there particular functions you might write (with what arguments and return types)? Are there particular classes you might define (with what instance variables and methods)?

Write your design work on paper (or do it on your laptop) and bring it to class; this work is due **at the start of class** on Tuesday the 20th. If you're really stuck on something, do your best, make a note of it, and move on; we'll be discussing this extensively in class.

Final version

A full-credit final version will be a complete, non-buggy, working implementation of a naïve Bayes classifier TOGETHER WITH convincing proof that it is correct. The program should be able to run on the provided training corpus (parts 1–9) of spam/ham email, then classify additional messages. When run, it should be provided with two command line arguments—the first is a directory name, and the second dictates subsequent behaviour:

prep tells it to read the training data (parts 1 to 9) and print out the first "word" on the third line of each message (as in the prep work). It may also print out other debugging information at the end.

single tells it to read the training data from their files, and then read a single message from standard input; and classifies that message and prints its judgement, either `HAM` or `SPAM`. It may also print out other debugging information, but the classification should be at the end.

test tells it to read the training data from their files (parts 1 to 9), and then the testing data from their files (`parttest`), and classify each

message independently, printing each filename followed by either `HAM` or `SPAM` (and possibly some numbers, but just one line per message). At the end of the test data, it should report how well it did, giving both precision and recall scores.

A full-credit version will compare the base version described above with an extension such as:

- counting the subject words (but separately from the body words)
- counting adjacent *pairs* of words (these are called "bigrams", i.e. "two words")
- ignoring punctuation (it's already broken into separate "words", so just skip them)
- incorporating *how many times* a word occurs in a document, not just *whether* it does so
- anything else you think might be neat to try (but check with me first)

and analyse whether the extension helped performance or hurt it.

Rubric

Note that I am not able to spend a ton of time with your program (and in fact may not read it at all, and definitely won't do your debugging for you), so your documentation will need to tell me anything I need to know to run and test your program. There need to be clear instructions on how to run it in general as well as how to run each/all of the tests and quickly verify that they ran correctly (and which rubric items each one corresponds to). Having complete and correct documentation is an easy 25 points, but if your documentation omits important info or tells me the wrong thing, you'll get less than full credit there.

After prep work (15 points), design work (10 points), and documentation (25 points), there remain 100 points in the rubric, which will be awarded according to the table below. Under each score, I show (for your convenience) the total cumulative points if you get that item plus *all* the previous points, and the letter grade this corresponds to. It is arranged roughly in the order I suggest you attempt them, with the earlier ones being easier or

enlightening with respect to the later ones, but you can in general get points for the later ones (if they work) without getting the earlier ones.

NOTE: if your code doesn't compile, or immediately crashes when it's run, you will get zero of these points. Don't let this happen to you!

Score	Description
10 (60/D−)	Compiles, runs, and handles `prep` option. (Extra debugging info ok.) All implemented options follow I/O spec.
10 (70/D)	Counts total number of words in *message bodies* in *training* corpus (i.e. one overall count)
10 (80/D+)	Gathers word counts for *each distinct* body word in training corpus (i.e. separate count for each word)
10 (90/C)	Gathers word counts separately for ham and spam
10 (100/C+)	Word counts represent frequency *of documents* containing words rather than overall word frequency
10 (110/B−)	On `single` option, reads test message and calculates $p(w\|S)$ and $p(w\|\neg S)$ for each word in message
5 (115/B)	Word probabilities account for unknown words (never seen in training, in either spam or ham)
5 (120/B+)	Word probabilities account for words with empirically-zero probability (seen in training in spam but not ham, or vice versa)
5 (125/B+)	Combines all conditional terms with prior to give classification for individual email
5 (130/A−)	Combination of probabilities accounts for and prevents arithmetic underflow
10 (140/A)	On `test` option, evals all messages in `parttest` directory of the corpus, checks classifier's answer, and produces score for classifier
10 (150/A+)	Extend classifier and compare results (did it help or hurt?)

Handing in

For both the checkpoint and the final version, hand it in as `proj2` using the handin script. The project is due at **4pm** on the due date.

Reference

I. Androutsopoulos, J. Koutsias, K.V. Chandrinos, George Paliouras, and C.D. Spyropoulos, "An Evaluation of Naive Bayesian Anti-Spam Filtering". In Potamias, G., Moustakis, V. and van Someren, M. (Eds.), *Proceedings of the Workshop on Machine Learning in the New Information Age*, 11th European Conference on Machine Learning (ECML 2000), Barcelona, Spain, pp. 9-17, 2000.