Lab 2

13 Sep 2007

Following on our discussion of conditional and joint probabilities, today's lab will have you gathering various statistics on a large chunk of text input. You can use any high-level language you feel comfortable with. "High-level", in this context, means it must include modern libraries (maps/dictionaries, sets, etc)—I don't want you spending the lab period reinventing the wheel.

Task 1: warming up

To get started, just make sure you can read the data in and count it. The input is simply plain old raw text, to be read in from standard input. I've put two files (`lab2-vicar.txt` and `lab2-excerpt.txt`) in the course directory (`/home/courses/cs262/`) for you to test your code with: the former is the full text, courtesy Project Gutenberg, of *The Vicar of Wakefield* by Oliver Goldsmih. The second is a one-paragraph excerpt therefrom, which you can use to debug your code. Feel free to find and use other texts if you like.

Of note is that the text is not preprocessed in any way. For now, just treat every whitespace-separated string as a word, without making a particular effort to handle punctuation. You should, however, operate case-insensitively, so that "the" and "The" and "THE" are all the same word.

Your initial output, then, will be a simple report containing every "word" in the input, paired with the number of times it occurs. Print them in a two-column table, with words and their frequencies, sorted alphabetically by word, e.g. the table

```
 2 a
 1 and
23 fnord
 2 parrot
```

(reporting on a very strange piece of text indeed).

Task 2: simple probabilities

Modify your program so that it calculates and prints, not the frequencies of the words, but their empirically-observed probability. That is, if I pick one word at random from the input text, what is the probability that the outcome would be this particular one? This is calculated as the ratio of the count C of a given word to the total number of words in the corpus:

$$p(W = w) = \frac{C(W = w)}{N}$$

Keep in mind when writing the printf formatting that the probability of some of the words will be very, very small, on the order of 10^{-5} or 10^{-6}.

Task 3: preprocessing

Almost any application involving information processing will need to do a little bit of preprocessing before the clever algorithm can do its best (and in some cases, before the algorithm will work at all). Often this preprocessing step is to bring all the data into similar formats—converting any GIFs or JPGs into PNGs, resampling to match bitrates—but in a lot of cases it has more to do with cleaning up the data and removing outliers and noise.

That's basically what we're doing here. If we want to gather statistics about word frequency, then we'd rather if a "word" like

```
country.'--'Ay,
```

(actually present in the `lab2-vicar.txt` file) were broken up into a few pieces first. As with many preprocessing algorithms, the following rules are based on the particularities of the file we're processing; if you use data from somewhere else, you'll probably have to tweak them.

- First, if you see a two-hyphen string "`--`" in the middle of a string, break that string into three words: everything before the dash, the dash itself, and everything after the dash.
- Next, if the word begins with a single quote, simply remove it.
- If the word ends with a single quote that is preceded by a non-letter, remove it. (Single-quotes are represented by the same character as

apostrophes, and we want to keep apostrophes. Actual close-quote marks are always preceded by punctuation, usually a comma or period.)

- At this point, you are left with a word possibly ending with a punctuation mark (comma, question mark, etc). If the last character is a non-letter, separate it into its own word.

The above string would thus be broken into five words:

```
country . -- Ay ,
```

The exact details of your implementation don't matter as long as the eventual word-stream is correct.

Note, by the way, that the above rules aren't perfectly complete even for just this text. I *think* that all the exceptions involve typoes in the source, but you're welcome to deal with them or ignore them. We'll discuss the problem of noisy data in class in a week or two.

Homework preview: conditional probability

This or something very like it will be on the homework that goes out tomorrow, so you might as well get started on it.

Next, you'll add data structures that will let you calculate the probability of a word given the word preceding it:[1]

$$p(W|P)$$

To see how to calculate it, let's look at a specific example. (Read this through, but even if it doesn't quite click yet, just follow the instructions at the end. We'll discuss it tomorrow.) To calculate the single probability of the word "were" given that the previous word was "we", we relate it to the corresponding joint and marginal probabilities:

$$p(W = \text{were}|P = \text{we}) = \frac{p(W = \text{were}, P = \text{we})}{p(P = \text{we})}$$

[1] These kinds of statistics are usually called "bigram" statistics, because *bi-* is a Latin-derived prefix for "two" and *gram* is from the Greek word for "word". Never accuse linguists of being consistent.

(That's just the definition of conditional probability from yesterday.) Those probabilities are themselves ratios of counts to total words in the corpus, whose denominators cancel:[2]

$$p(W = \text{were}|P = \text{we}) = \frac{\frac{C(W=\text{were},P=\text{we})}{N}}{\frac{C(P=\text{we})}{N}} = \frac{C(W = \text{were}, P = \text{we})}{C(P = \text{we})}$$

In the brief excerpt file, the word "were" preceded by "we" occurs twice. The word "we" precedes a total of five words. So

$$p(\text{were}|\text{we}) = \frac{2}{5}.$$

Since the counts for the previous words distribution P will be identical to the counts for the current words distribution W,[3] we've already done all the work for the denominator. So, add data structures to collect the counts for the joint distribution: the number of times every *pair* of words occurs. Careful with the structures here, though. While you could map pairs of strings to integers, it's usually cleaner and easier to map the first ("previous") string to a nested map, which itself maps the second ("current") string to an integer.[4] The nested map then corresponds to the entire adds-to-1 conditional probability distribution conditioned on that specific previous word: $p(W|P = \text{we})$, for instance.

Once you've written the code to store the counts, update the output to print a three column table. (Keep the old two-column table too; just add this at the end.) The first column prints the conditioning (previous) word once, then for every word that *can* follow it, the second and third columns contain the probability and the word. For instance:

```
we          0.20  had
            0.20  might
            0.20  suffered
            0.40  were
```

[2]Technically, the denominator on that joint probability should be $N - 1$, since if there are N words, there are only $N - 1$ word/previous-word pairs. As a special case, we can pretend that the first word was preceded by a period, giving us N actual data points and representing the fact that "starts a sentence" is approximately equivalent to "follows a period".

[3]Might not be true if the text doesn't end in a period. As a practical matter, this doesn't make enough of a difference to worry about. If we were really mathematicians we would probably care more.

[4]The type will thus be `Map<String, Map<String, Integer>>` or `map<string, map<string, int> >`.