Lab 3

20 Sep 2007

This week we'll do more corpus work, and exercise some of the probability techniques we've been talking about. We'll be using a new corpus format, where each word has been annotated with its lexical category, or part of speech. You'll look at some existing code (probably similar to code you've already written) and then modifying it to improve the statistical models in the system.

Task 1: Comprehend the code, part 1

Into the course directory, I've put my solution for homework 2.3, which collected bigram statistics and reported on them. Read this code. Take ten or fifteen minutes to work through it and understand it; if you're not sure what something does or how something works or just why I did something a certain way, make sure to ask.

Let me answer a few likely questions in advance:

- All the work of the preprocessing was farmed out to a separate class, `StripScanner`, which is a wrapper for `Scanner` but does some extra stuff.
- Strictly speaking, the level of abstraction (helper methods etc) is a little overboard if this were the entire problem being solved. It's not outrageously so, though, and some of these extra methods will make your life easier in the second half of this lab (stay tuned).
- Note that there are two kinds of maps flying around: those with integer values (i.e. counts), for which I've used `HashMap`s since all I need to do is lookups and stores, and I have to do that a lot; and those with double values (i.e. probabilities), for which I've used `TreeMap`s since I'm actually printing a report from them.
- The prefix business on `simpleReport` is all so that I can reuse `simpleReport` in the course of `conditionalReport`. This is admittedly a little bit hacky.

But if there are other things that puzzle you, let me know.

Setting up the problem

In lecture we've talked a bit about the problem of tagging words with their appropriate lexical category. In the realm of statistical NLP, this problem is attacked by feeding in a big corpus of tagged text, using this to train a probability model, and then reading a stream of untagged text and marking it up.

There are probably as many ways to represent marked up data as there are corpora[1] that include markup, but our data will use a simple one: categories (i.e. POS tags) will simply alternate with the words they tag. No category or word contains a space. Whitespace is not significant except to separate tags and words.

Making it even easier for you, the punctuation will have already been separated out for you. You'll get to treat a comma as a "word" spelled "," (which has a lexical category also spelled ","). This does mean you can't assume that words will be alphanumeric, but it makes a ton of other things much simpler. A sample sentence: [2]

```
DT The JJ quick JJ brown NN fox VBD jumped IN over DT the
JJ lazy NN dog . .
```

Inside the `lab3` subdirectory are four files in this format, of varying size from "toy" (the above sentence) through "long" (some 58,000 words). The shortest ones may be helpful for debugging code, but the goal here is to use the long one to train a tagger and then see how it does on the short one (which is not a subset of lab3-long, but rather other text in the same genre).

[1] "Corpora" is the plural of "corpus". In English it's pronounced "COR–pruh".

[2] As an aside, in case you're curious, the POS tags needn't be completely opaque to you. The main ones are DT "determiner" (or "article"), NN "noun", JJ "adjective", and VB "verb". Many of the others are mnemonic according to what their members look like: IN means "preposition" because "in" is a preposition. NNS is "plural noun" because such words often end in "-s". VBD is "past-tense verb" because they often end in "-d", and VBN is "past participle of verb", because the ones that don't look just like the VBD form often end in "-n", like "chosen" or "gotten". You don't need to know all of this to complete the lab, but it might make the examples easier to follow.

Task 2: Comprehend the code, part 2

As a first pass at solving this problem, I've done a very thin adaptation of `BigramStats` to learn the distribution $p(T|W)$ based on this input format, and then to tag text. I've called it `DumbTagger`, and it's also available in the `lab3` directory. Spend a few minutes reading this code as well, and make sure you understand what all the pieces do.

By and large, the stats gathering is identical, or nearly so, to the previous program. The tagging code is new, so you should focus some attention on it, but it should hopefully be straightforward. If, perhaps, a bit dumb.

Setting up the program

For the remainder of the lab you will be building a system derived from `DumbTagger`. Set this up by copying it (and `SkipScanner`) into your home directory or some suitable subdirectory thereof. Rename the file (and the class) `NaiveTagger`, and add yourself as an author.

You'll also want to create symbolic links to the data files. If you `cd` into your working directory for this lab and type

```
ln -s /home/courses/cs262/lab3/lab3-tiny.pos
```

then you will get symbolic link named `lab3-tiny.pos` in your directory that you can refer to. Set these up for all four of the data files.

Also, compile and run the existing program. You'll have to provide the training text as a command-line argument, and the test text on standard input, so you'll want something like

```
java NaiveTagger lab3-long.pos < lab3-tiny.pos
```

although that doesn't wrap the text, and if it were longer it wold run off the top of the page, so you might want

```
java NaiveTagger lab3-long.pos < lab3-tiny.pos | fmt | less
```

or

```
java NaiveTagger lab3-long.pos < lab3-tiny.pos | fmt | vim -
```

instead.

As you modify this program for the rest of the lab, don't feel restricted just because a certain method is written a certain way or has a certain interface; add variables and change methods as necessary.

Task 3: Unknown word handling

Right now, any word the tagger has never seen before will be tagged as `UNK`, which isn't even a valid tag. Fix this by calculating a guess based on the previous tag: use the distributions $p(\text{PrevTag}|T)$ and $p(T)$ to figure out which value for T has the greatest likelihood/score.

When you are building the distribution, you know the actual previous tag and can store this value; when you are doing the tagging, though, all you have is what you *guessed* the previous tag to be. Go ahead and run with that; it's the best you've got for now. This is called a "greedy" algorithm—always taking the best option at the moment, locking it in, and then moving on, never backtracking—and to improve on it you need to do a bunch more work. Take NLP next term if you want to do better!

NOTE: both in constructing new distributions and in accessing them, make good use of the helper methods I've already written for you! You should only have to add a few lines of code in just a few places.

Save a copy of this version when you're done.

Task 4: Making it less dumb

Right now, the tagger only takes into account the current word in selecting the tag; its conditioning distribution is just $p(T|W)$. We have more information than that, though: we have the previous word and tag (not to mention the one before that, but let's not go crazy just yet). Using that would give us

$$p(\text{Tag}|\text{Word}, \text{PrevTag}, \text{PrevWord})$$

but of course this would be a very sparse distribution. The corresponding naïve Bayes classifier will instead compute the score

$$p(T)p(W|T)p(PT|T)p(PW|T)$$

for all values of T and select the highest one. You're going to implement this.

There are three potential gotchas here:

1. Remember that the current version computes and uses $p(T|W)$, but now you'll need $p(W|T)$ instead (in addition to some other new ones).

2. What happens if the previous word was a completely unknown word? Assuming W and PT were known, then you'll want to just use the score

   $$p(T)p(W|T)p(PT|T)$$

   instead. Indeed, this will just be the general case of how we handled unknown words before: if a word is completely unknown, then the relevant distribution is ignored when calculating the score. (Programmatically, you may find it more convenient to say that the relevant distribution multiplies the score by 1.0.)

3. Even trickier, and don't address this case until you've understood and gotten the previous ones working, but what if the previous word and tag and the current word are all *known* but have not been seen *together*? For instance, consider if we are tagging and come to the following situation:

...	VBD	PRP	IN	???	...
...	retrieved	it	from	under	...

   The tags are correct so far. The model has only seen the tag IN (preposition) for "under"—and this would be correct—but it also has only ever seen "from" followed by determiners (DT: "the", "a") and proper nouns (NNP: "Chicago", "Ipanema") before. So $p(T = \text{PRP}|PW = \text{from}) = 0$, but $p(T = \text{NNP}|W = \text{under}) = 0$, and since we're multiplying all these things together there will actually be no tag with a score higher than zero.

   There's a whole other body of research in addressing just this problem, but we're just going to take one of the easier solutions. Since we're just dealing with scores here anyway, and don't need to worry about busting the probability distribution, we can give a low but non-zero score to *all* combinations. When we go to look up a probability and the items are not present in the relevant map—and thus the nominal observed probability would be zero—we will instead report the probability as 0.000001.[3]

[3]You can actually type this value into your program as `1e-6`. Convenient, eh?