CS 224N / Ling 237

Assignment 2: Word Alignment Models

Due Wed, 20 April 2005, 5pm

0 Reading

Make sure that you've read Knight's workbook before attempting this assignment.

1 Setup

We've put everything for this assigment (both Java starter code and data files) in the directory `/afs/ir/class/cs224n/hw2/`. The Java starter code is in `hw2/java/`, and the data is in `hw2/data/`. Copy over the new code for this homework (you may want to first save what you did for HW#1).

Begin by ensuring that you can compile the starter code. Remember that for this assignment (and later ones), we'll be using Java 1.5, so you'll need JDK 1.5 (which includes the `javac` compiler and the `java` virtual machine). Unfortunately JDK 1.5 is not yet standard issue on the Leland system. We've put a copy of JDK 1.5 in the class bin directory at

```
/afs/ir/class/cs224n/bin/jdk1.5.0/solaris/
/afs/ir/class/cs224n/bin/jdk1.5.0/linux/
```

The solaris version works on Solaris machines like the `elaine` machines (use the shell command `uname` to find if you are using a Linux or Solaris (SunOS) machine).

For example, assuming use of `csh`, you might put the following in your `.cshrc` so it will see the new Java:

```
setenv PATH /afs/ir/class/cs224n/bin/jdk1.5.0/solaris/bin:$PATH
```

Remember that if you're compiling with Ant, you may need to modify your `build.xml` file to tell it where to find the Java 1.5 compiler. The build file that we include in the Java directory now works with 1.5. (The old 1.4 compiler will break on 1.5 code.)

If you want to use JDK 1.5 on your own machine, you can download it directly from Sun at `http://java.sun.com/j2se/1.5.0/download.jsp`.

The most important class for this assignment is `cs224n.assignments.WordAlignmentTester`. The `main()` method of this class takes several command line options. Try running it with:

```
java -server -Xmx500m cs224n.assignments.WordAlignmentTester \
    -path hw2/data/ -model baseline -data miniTest -verbose
```

You should see a few toy sentence pairs fly by, with alignments from the baseline (diagonal) model. The `-verbose` flag controls whether the alignment matrices are printed. For the `miniTest` dataset, the baseline model isn't so bad. Look in the data directory to see the source `miniTest` sentences. They are:

English	*French*	*Alignments*
`<s snum=1> A B C </s>`	`<s snum=1> X Y Z </s>`	`1 1 1 S`
`<s snum=2> A B </s>`	`<s snum=2> X Y </s>`	`1 2 2 S`
`<s snum=3> B C </s>`	`<s snum=3> Y W Z </s>`	`1 3 3 S`
`<s snum=4> D F </s>`	`<s snum=4> U V W </s>`	`2 1 1 S`
		`2 2 2 S`
		`3 1 1 S`
		`3 2 3 S`
		`4 1 1 S`
		`4 2 2 S`

In the alignments file, the fields are `snum`, `e`, `f`, and `S`(ure)/`P`(ossible). A 'Sure' alignment is where an alignment is clearly right, whereas a 'Possible' alignment is one that it is reasonable to make but not so clear. (Commonly this is function words associated with the main semantic content of a word. So, for the French word *mangeaient* '(they) were eating', if it is part of a sentence translated as *The children were eating when we arrived*, one would expect a Sure alignment between *mangeaient* and *eating* and a Possible alignment between *mangeaient* and *were*.) For our toy example, the intuitive alignment is `X=A`, `Y=B`, `Z=C`, `U=D`, `V=F`, and `W=null`. The baseline model will get most of this set right, missing only the mid-sentence null:

```
[#]   | Y
    # | W
   [ ]| Z
------'
 B  C
```

The hashes indicate the proposed alignment pairs, while the brackets indicate reference pairs. That is, the hashes are the alignments that your model is suggesting are correct, while the brackets show the alignments that the supplied gold standard data think are correct. Square brackets show Sure alignments, and parentheses – shown in the next alignment matrix, below – indicate possible alignment positions. Your program only produces a binary notion of aligned or not. Your goal is a model that puts a hash inside all square brackets, and it is okay but not necessary if it also puts a hash inside parentheses. Any other hashes placed by your program are deemed errors. At the end of the test output you get overall precision (with respect to possible alignments), recall (with respect to sure alignments), and alignment error rate (AER). This quantity can be thought of as similar to $1.0 - \mathrm{F}_1$, where F_1 is a harmonic mean of precision and recall, but the Sure/Possible distinction complicates things. Put precisely, we have 3 sets of alignments: the proposed alignment set of the model A, the Sure set S, and the Possible set P, which we define as the union of the Sure and Possible alignments discussed above (i.e., everything in either square brackets or parentheses). Following Och and Ney (2000),[1] we then conceptually define:

$$\text{Recall} = \frac{|A \cap S|}{|S|} \qquad\qquad \text{Precision} = \frac{|A \cap P|}{|A|}$$

[1] F. J. Och and H. Ney. 2000. Improved statistical alignment models. In ACL 38, pp. 440–447. http://acl.ldc.upenn.edu/P/P00/P00-1056.pdf

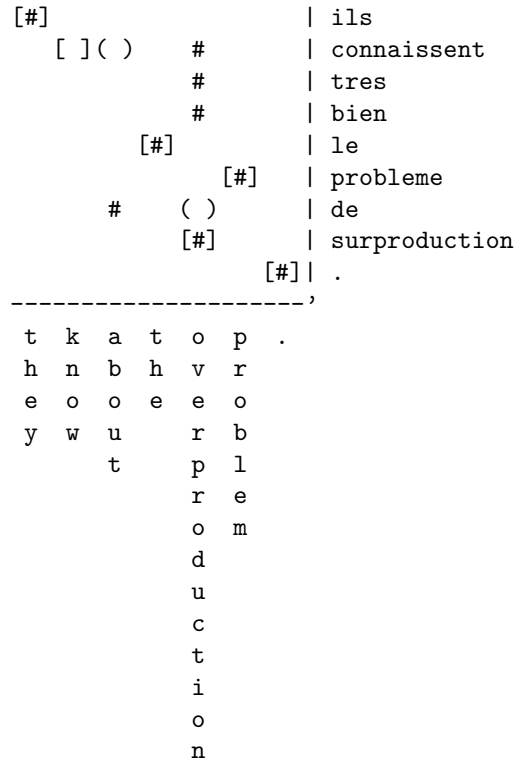
and the measure we hence use is:

$$AER(S, P, A) = 1 - \frac{|A \cap S| + |A \cap P|}{|S| + |A|}$$

Note that for this quantity to be zero, all sure alignments must be proposed, and all proposed alignments must be possible ones.

You should also try running the code with `-data validate` and `-data test`, which will give the validation set and test set respectively. Baseline AER on the test set should be 68.7 (lower is better). If you want to learn alignments on more than the test set, you can get an additional k sentences with the flag `-sentences k`. Maximum values of k usable by your code will probably be between 10,000 and 100,000, depending on how much memory you have, and how you encode things. (There are a million or so sentence pairs there if you want them — to use anywhere near that many, you'll have to change some of the starter code.)

You'll notice that the code is hardwired to English-French, just as the examples in class were presented. If you don't speak any French, this assignment may be a little less fun, but French has enough English cognates that you should still be able to sift usefully through the data. For example, if you see the matrix



you should be able to tell that "problem" got handled correctly, as did "overproduction", but something went very wrong with the "know about" region, even without speaking any French.[2]

2 Three Basic Alignment Models

In this assignment, you will build several word-level alignment systems. As a first step, and to get used to the data and support classes, build a replacement for `BaselineWordAligner` which matches up words based on superficial statistics. One common stab is to pair each French word f with the English word e for which the ratio

$$\frac{P(f,e)}{P(f)P(e)}$$

is greatest (the log of this measure is often referred to as "pointwise mutual information" in the NLP literature). Other possibilities exist; play a little and see if you can detect any useful translation pairs with such methods.

Once you've gotten a handle on the data and code, the first real model to implement is IBM Model 1. Recall that in Models 1 and 2, the probability of an alignment a for a sentence pair (f,e) is

$$P(f,a|e) = \prod_{i=1}^{len_f} P(a_i|i,len_f,len_e)P(f_i|e_{a_i})$$

[2]Franz Och, a well-known statistical MT researcher who now works on Google, is well-known for saying that it's preferable to work on statistical MT systems for languages you don't know, as it leads you to concentrate on the statistical essence of the data. We tend to disagree, but at any rate, this shows that trying to do this assignment with no knowledge of French isn't hopeless.

where the null English word is at position 0 (or -1, or whatever is convenient in your code). The simplifying assumption in Model 1 is that

$$P(a_i|i, len_f, len_e) = P(a_i) = \frac{1}{len_e + 1}$$

That is, all positions are equally likely. In practice, the null position is often given a different likelihood, say 0.2, which doesn't vary with the length of the sentence, and the remaining 0.8 is split evenly amongst the other locations.

The iterative EM update for this model is very simple and intuitive. For every English word type e, you count up how many French words are aligned with tokens of e (the answer will be a fraction), as well as the distribution over those French words' types. That will give you a new estimate of the translation probabilities $P(f|e)$, which leads to new alignment posteriors, and so on. For the `miniTest` dataset, your Model 1 should learn most of the correct translations, including aligning `W` with `null`. However, it will be confused by the `DF / UV` block, putting each of `U` and `V` with each of `D` and `F` with equal likelihood (probably resulting in a single error, depending on numerical precision issues).

Look at the alignments produced on the validation or test sets with your Model 1. You can improve performance by training on additional sentences as mentioned above. However, even if you do, you will still see many alignments which have errors sprayed all over the matrices, errors which would be largely fixed by concentrating guesses near the diagonals. To address this trend, you should now implement IBM Model 2, which changes only a single term from Model 1: $P(a_i|i, len_f, len_e)$ is no longer independent of i. A common choice is to have

$$P(a_i|i, len_f, len_e) \propto d\left(bucket\left(a_i - i\frac{len_e}{len_f}\right)\right)$$

Here, $P(a_i|i, len_f, len_e)$ is parameterized by a rough displacement from the diagonal. Again, to make this work well, one generally needs to treat the null alignment as a special case, giving a constant chance for a null alignment (independent of position), and leaving the other positions distributed as above. The function *bucket* maps real numbered displacements (normalized for overall sentence length) to one of a number of categorical buckets. For instance, it might be formed by just rounding the argument to the nearest integer, except for placing all displacements greater than 5 in absolute value into two buckets for large positive and negative displacements. How you bucket those displacements is up to you; there are many choices and most will give similar behavior. If you run your Model 2 on the `miniTest` dataset, it should get them all right (you may need to fiddle with your null probabilities).

3 Evaluation criteria

You should now have three functional models: a non-iterative surface-statistics model, an implementation of Model 1, and an implementation of Model 2. Solid implementations of these models, along with a good write-up, is sufficient to earn a full grade. Your grade will depend on:

- Effective implementation of those models to get good results.
- A clear presentation of the algorithms you used and alternatives you tried.
- A discussion of the motivations for choices that you made and a description of the testing that you did.

- Showing that your models use proper probability distributions.
- Insightful commentary on what kinds of things your models get right and get wrong, and possible reasons why.
- Ideas as to how to best improve the system, based on the above.

4 Extra Credit: Further Investigation

We will give up to 10% extra credit for innovative work going substantially beyond the minimal requirements. The field of possibilities is wide open. Some options:

- Implement IBM Model 3 as described in Kevin Knight's *A Statistical MT Tutorial Workbook.*
- Implement the competitive linking algorithm from I. Dan Melamed, "Models of Translational Equivalence among Words", *Computational Linguistics*, 2000. See `http://www.cs.nyu.edu/~melamed/ftp/papers/clmote.pdf`.
- (Probably the easiest for code-optimizers.) Scale your system up to a large amount of data (defined as, say, running on at least 100K training sentences). Investigate the learning curve as a function of training size.

Using some extra sentences and Model 2 or better, you should be able to get your AER down below 40% very easily and below 35% fairly easily, but getting it much below 30% will likely require greater effort.

5 Submitting the assignment

5.1 The program: electronic submission

Submission of the program code will be via a script. To submit your program, put all the needed files in one directory on a Sweet Hall machine (or, possibly, another machine with AFS on it and with you being correctly authenticated) and type:

```
/afs/ir/class/cs224n/bin/submit-hw2
```

If you need to resubmit it type

```
/afs/ir/class/cs224n/bin/submit-hw2 -replace
```

Make sure that you include all the source code for your programs. We will run programs on the Sweet Hall systems, so they should run without problems there. Please don't include large data files in your submission. Your code doesn't have to be beautiful but we should be able to scan it and figure out what you did without too much pain.

5.2 The report: turn in a hard copy

You should turn in a write-up of the work you've done, as well as the code. **Your write-up must be submitted as a hard copy.** There is no set length for write-ups, but a ballpark length might be 4 pages, including your evaluation results, a graph or two, and some interesting examples.