Word Alignment Models for Machine Translation

Programming Assignment 2
CS 224N / Ling 284

Due: 5pm October 26, 2011

This assignment may be done individually or in groups of two. We strongly encourage collaboration; however your submission must include *a statement describing the contributions of each collaborator*. See the collaboration policy on the website (`https://courseware.stanford.edu/pg/pages/view/214507/grading-overview`).

Please read this assignment soon. Go through the Setup section to ensure that you are able to access the relevant files and compile the code. Especially if your programming experience is limited, start working early so that you will have ample time to discover stumbling blocks and ask questions. **Make sure that you have read Knight's workbook and chapter 25 of J&M before attempting this assignment**.

1 Setup

We've put everything for this assignment in the directory:

`/afs/ir/class/cs224n/pa2/java/` : the Java code provided for this course
`/afs/ir/class/cs224n/pa2/data/` : the data sets used in this assignment

Copy over the new code to your local directory and make sure you can compile the code without errors.

```
cd
mkdir -p cs224n/pa2
cd cs224n/pa2
cp -r /afs/ir/class/cs224n/pa2/java .
cd java
./ant
```

The most important class for this assignment is `cs224n.assignments.WordAlignmentTester`. The `main()` method of this class takes several command line options. Set the `CLASSPATH` and then try running it with:

```
java -Xmx500m cs224n.assignments.WordAlignmentTester \
  -path /afs/ir/class/cs224n/pa2/data/ \
  -model baseline -data miniTest -verbose
```

This will run a baseline model, which simply assigns alignments along the diagonal, on a toy set of sentence pairs (see the description of miniTest below). The script `./run-wa` may be useful for saving options. The `-verbose` flag controls whether the alignment matrices are printed. For the `miniTest` dataset, the baseline model isn't so bad. Of course, it doesn't work nearly as well on real data which consists of parallel texts in English and French from the Canadian Parliament. Your goal is to design and implement a WordAligner class that can be passed into the WordAlignmentTester to accurately align words between parallel sentences.

Start by looking 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 these are alignments of function words with the translation of a content word to which they are associated. 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*, because *were* is associated with *eating*.) 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 68.6 (lower is better). You should try learning alignments on more than the test set, at least for simple models (to investigate how much extra data helps). 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 might have to change some of the starter code !)

In WordAlignmentTester the code is currently hardwired to align English-French, but the constants you need to change for trying out other combinations are highlighted. 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

```
[#]                   | ils
   [ ]( )    #        | connaissent
             #        | tres
             #        | bien
         [#]          | le
               [#]    | probleme
       #    ( )       | de
            [#]       | surproduction
                 [#]  | .
---------------------'
 t  k  a  t  o  p  .
 h  n  b  h  v  r
 e  o  o  e  e  o
 y  w  u     r  b
       t     p  l
             r  e
             o  m
             d
             u
             c
             t
             i
             o
             n
```

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] And other languages are available for the decoding step ... see below!

[2]Franz Och, a well-known statistical MT researcher who now works at 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.

A WordAligner comes with a static final String NULL_WORD which you should use to represent the null word when aligning French words to English words. To implement a WordAligner class, you must fill in the following methods:

- train(List⟨SentencePair⟩ trainingPairs). Trains the model from the supplied collection of parallel sentences.
- getProbSourceGivenTarget(). Returns a `CounterMap⟨String,String⟩ probSourceGivenTarget`. This CounterMap should be structured so that the first word is from the source language and the second from the target language. Therefore, if the goal is to translate from French to English, the code `probSourceGivenTarget.getCount(FrenchWord, EnglishWord)` represents $P(FrenchWord|EnglishWord)$, the probabilities estimated by your alignment model. This method is used by the Decoder (see next section), so it doesn't play a role in building or testing the WordAligner (however, since you will probably have such a structure in your WordAligner, it should be an easy function to write).
- getAlignmentProb(List⟨String⟩ targetSentence, List⟨String⟩ sourceSentence, Alignment alignment). Returns the probability, according to the model, of the specified alignment between targetSentence and sourceSentence.
- alignSentencePair(SentencePair sentencePair). Returns the best Alignment between the two sentences in sentencePair.

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 (some of the above functions will just be blank). 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

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

where a_i is the index of the word in e which is aligned with the ith word in f, and 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.

For each bucket j, you should have a parameter $d(j)$ to indicate the probability of that distortion. These parameters should be learned during the EM process. (It turns out that hand-setting reasonable functions for d, instead of trying to learn parameters, can also work pretty well. We'd like you to try to learn distortion parameters with the EM algorithm, but if you're having trouble getting that to work, try it with a hand-set distortion function first. Or possibly fix the weights of the rare high distortion buckets and learn the rest. At any rate, report what you did. (Dealing with distortions in some other sensible way could warrant credit.) 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 Decoding

Aligning words between parallel sentences is cool and all, but what we really want to see is how well the computer can translate a sentence in one language (say French) into another (say English). Recall from lecture that this is done by finding the English sentence

$$\hat{e} = \arg\max_e P(f|e)P(e)$$

$P(f|e)$ is estimated by the word alignment model you build for this assignment, and $P(e)$ is estimated by the language model you built for assignment one. All you need now is a decoder to find the most probable English sentence and you will have a complete statistical machine translation package!

Luckily for you guys, we provide you with such a decoder. Because the search space is enormous it makes some simplifications. Instead of the above, it finds

$$\hat{e}, \hat{a} = \arg\max_{e,a} P(f, a|e)P(e)$$

Also, it does not actually find the optimal solution, but instead one that is pretty good. Our particular implementation is a Greedy Decoder which starts by translating each word to its most likely counterpart in the corresponding language; it then iteratively performs a series of mutations, at each step choosing the one that leads to the greatest increase in probability. The process stops once none of the mutations increase the translation's probability. [3]

We have tried to make the use of the Decoder as seamless as possible, but you will need to do a few things:

- Make sure you have implemented all the methods for the LanguageModel and the WordAligner that were mentioned in the handout, as the Decoder assumes they are there.
- Look in `assignments/DecoderTester.java` and add code in the indicated places to set options for your LanguageModel and WordAligner before training.
- Note the `zferts.e` file in the `pa2/java` directory. This file has a list of words that can be inserted as zero fertility words (to again limit the search space). This file *must* be present in the directory in which you are running the decoder for it to work properly.

Once you have done the above, you should be able to run DecoderTester. The relevant arguments are:

`-lmmodel`: the language model

`-lmsentences`: number of language model training sentences

`-wamodel`: word alignment model

`-wasentences`: number of word alignment model training sentence pairs

`-source`: source language

`-target`: target language, respectively (and of course any other options you include for your models)

`-lmweight`: a weight for the log(English LM prob) of a hypothesis

`-transweight`: a weight for the log(WordAlignment prob) of a hypothesis (source, target and alignment)

`-lengthweight`: a weight for the length(e) of the final score. This is to prevent the decoder from predicting really short sentences in order to get higher LM scores.

`-cache`: an optional argument which takes a directory as an argument, and loads/saves your trained language model and aligner from that directory.

`-zferts`: The file (without the language-specific extension) where probable zero-fertility words are stored. The default behavior is to look in the current directory (i.e. `-zferts ./zferts`).

For example, initially you might do `$ ./run-dec`
You'll notice that there are lots of "trans prob:-Infinity" in the output. It's because the `getAlignmentProb` method in `BaselineWordAligner` always returns 0, and the "trans prob" in the output is the logarithm of `getAlignmentProb`.

Once you have your Model 1 and Model 2 aligners implemented, try running:

```
$ ./run-dec -wamodel cs224n.wordaligner.YourModel1WordAligner
$ ./run-dec -wamodel cs224n.wordaligner.YourModel2WordAligner
```

[3] If you are interested you can read more about it in "Fast Decoding and Optimal Decoding in Machine Translation" available at http://www.isi.edu/natural-language/projects/rewrite/decoder.pdf.

For somewhat better results from the decoder, you can increase `beamSize` in `GreedyDecoder`. This will explore the search space somewhat better, at the cost of slowing down the decoding process by O(`beamSize`).

The languages currently available for translating are German, Spanish, and French (either the source or the target language must be English). If you have any other language requests, let us know and we will see if we can set you up with it. (In particular, anything else from EuroParl — Danish, Greek, Finnish, Dutch, Portuguese, or Swedish — would be easy to set up. Of course, parliamentary proceedings aren't the most thrilling of corpora so you could try to see what else is out there on the web.)

After running DecoderTester, you should see numbers that could be used to compare performance. "WER" is based on edit distance between the guessed and actual translations, which is a rather poor measure of translation quality. The number shown in "BLEU" is BLEU-4 (equation (25.35) in J&M where N=4). When the translation quality is poor, the higher order N-gram matches might be zero, therefore the log Ngram scores might be -Infinity, which cause the BLEU score to be zero. In that case, you can compute BLEU-3 (or even BLEU-2) based on the log Ngram scores and the brevity penalty (BP). An alternative solution to this problem is to use a *smoothed* `BLEU` metric, described in more detail in (Lin 2004). [4]

When you report numbers, remember to say if you're using BLEU with lower order N-grams.

4 Evaluation criteria

You should have three functional models: a non-iterative surface-statistics model, an implementation of Model 1, and an implementation of Model 2. You're required to compare the performance of your Model 1 and Model 2 on the decoder. Solid implementations of these models, along with a good write-up with interesting data analysis, are required 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. Remember we are looking for careful and thorough error analysis to be carried out during the process of improving your model, rather than being something tacked on at the end. Some of the things you might consider include:
  - Where word order differs between the languages does either model cope with this? When does it work and not work?
  - Does the model successfully insert needed function words into the output?
  - Does the model more often wrongly align function words or content words?
  - What are the most common sorts of wrong alignments? Aligning too many words to a word, or completely wrong words?
- Ideas as to how to best improve the system, based on the above.

[4] Chin-Yew Lin and Franz Josef Och. *ORANGE: a Method for Evaluating Automatic Evaluation Metrics for Machine Translation.* Found at `http://acl.ldc.upenn.edu/C/C04/C04-1072.pdf`

5 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 the competitive linking algorithm from I. Dan Melamed, "Models of Translational Equivalence among Words", *Computational Linguistics*, 2000. See

  `http://www.aclweb.org/anthology/J/J00/J00-2004.pdf`.

- Try out some of the ideas presented in Bob Moore's "Association-Based Bilingual Word Alignment" or "Improving IBM Word Alignment Model 1" (linked to from the course website, under the 1/12 readings).

- (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.

- Implement IBM Model 3 as described in Kevin Knight's *A Statistical MT Tutorial Workbook*.

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.

6 Submitting the assignment

6.1 The program: electronic submission

You will submit your program code using a Unix script that we've prepared. To submit your program, first put all the files to be submitted in one directory on a Leland machine (or any machine from which you can access the Leland AFS filesystem). This should include all source code files, but should not include compiled class files or large data files. Normally, your submission directory will have a subdirectory named `src` which contains all your source code. When you're ready to submit, **please cd to your submission directory**, and then type:

```
/afs/ir/class/cs224n/bin/submit-pa2
```

This will (recursively) copy everything in *your* submission directory into the official submission directory for the class. If you need to resubmit it type

```
/afs/ir/class/cs224n/bin/submit-pa2 -replace
```

We will compile and run your program on the Leland systems, using `ant` and our standard `build.xml` to compile, and using `java` to run. So, please make sure your program compiles and runs without difficulty on the Leland machines. If there's anything special we need to know about compiling or running your program, please include a `README` file with 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.

6.2 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. The write-up should specify what you built and what choices you made, and should include the accuracies, etc., of your systems. **If you are an on-campus student, your write-up must be submitted as a hard copy.** Hard copies may be submitted in class, or in the box outside Professor Manning's office. SCPD students may submit the write-up electronically on CourseWare.

There is no set length for write-ups, but a ballpark length might be 6 pages, including your evaluation results, a graph or two, error analysis, and some interesting examples. **Your write-up should not exceed 8 pages in length.**