CS 224N / Ling 280

Assignment 2: Word Alignment Models and MaxEnt Classifiers

Due Wed, 3 May 2006, 5pm

Part I
Word Alignment Model

0 Reading

Make sure that you've read Knight's workbook before attempting the first part.

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
```

Begin by ensuring that you can compile the starter code. It's Java 1.5, which is the version installed on the Leland machines.

The most important class for this part 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 -server -Xmx500m cs224n.assignments.WordAlignmentTester \
  -path /afs/ir/class/cs224n/pa2/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:

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

and the measure we hence use is:

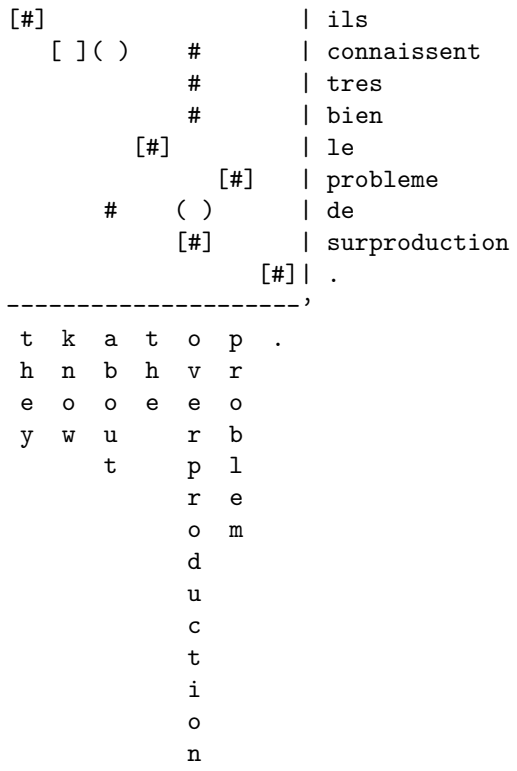
$$AER(S, P, A) = 1 - \frac{|A \cap S| + |A \cap P|}{|S| + |A|}$$

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

[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

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

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

[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.

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 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).

Part II
MaxEnt Classifiers

3 Setup

The principal source files for this assignment are:

```
src/cs224n/assignments/MaximumEntropyClassifierTester.java
```

You can do a quick test run for the first one using the command

```
java cs224n.assignments.MaximumEntropyClassifierTester -mini
```

Take a few minutes to look through the data directories, too.

4 A Maximum Entropy Classifier

4.1 Building a simple maxent classifier

Look through the code in `MaximumEntropyClassifierTester.java`. This class contains several subclasses. The most important two are:

`MaximumEntropyClassifier` (implements `classify.ProbabilisticClassifier`)
`MaximumEntropyClassifierFactory` (creates the former)

Look at the `main` method to see the overall program flow. There are two modes you can run this `main` method in. If you supply `-mini` as the first command line argument, you'll get a miniature classification problem from the `miniTest()` method. I recommend working with this branch first, since it's easier to debug. `miniTest()` creates several training datums and a test datum. Each datum represents either a cat or a bear and has several features (which are just strings). The training data are passed to a `MaximumEntropyClassifierFactory` which uses them to learn a `MaximumEntropyClassifer`. This classifier is then applied to the test data, and an accuracy (and distribution over labels) is printed out.

While the starter code contains a fully-functioning pipeline for training and testing a classifier, the classifier it builds is as dumb as a rock, and does *not* use maximum entropy. Your job is to turn the placeholder code into a maximum entropy classifier by filling in the two chunks of code marked by "`// TODO`" lines . First, look at

```
MaximumEntropyClassifier.getLogProbabilities()
```

This method takes a datum, and produces the (log) distribution, according to the model, over the possible labels for the datum. There will be some interface shock here, because you're looking at a method buried deep inside the implementation of the rest of the classifier. You are given several arguments, whose classes are defined in this same Java file:

```
EncodedDatum datum
Encoding<F, L> encoding
IndexLinearizer indexLinearizer
double[] weights
```

The `EncodedDatum` represents the input datum. It is a sparse encoding, which tells you which features were present in that datum, and with what counts. When you ask an `EncodedDatum` what features are present, it will return feature *indexes* instead of feature objects — for example it might tell you that feature 121 is present with count 1.0 and feature 3317 is present with

count 2.0. If you want to recover the original (`String`) representation of those features, you'll have to go through the `Encoding`, which maps between features and feature indexes. Encodings also manage maps between labels and label indexes. So while your `miniTest()` labels are "cat" and "bear", the `Encoding` will map these to indexes 0 and 1, and your returned log distribution should be a double array indexed by 0 and 1, rather than a hash on "cat" and "bear".

So, you first have a feature ("fuzzy") and a label ("cat") which map to some feature index (say 2) and some label index (say 0). As outlined above, these are managed by the `Encoding`. The job of the `getLogProbabilities()` method is to return an array of doubles, where the indexes are the label indexes, and the entries are the log probabilities of that label, given the current datum. To do this, you will need to properly combine the feature weights (the λs). The double vector `weights` contains these feature weights linearized into a one-dimensional array. To find the weight for the feature "fuzzy" and the label "cat," you'll need to take their indexes (2 and 0) and use the `IndexLinearizer` to find out what index in `weights` to use.

Your job here is to correctly fill in and return an array of log probabilities. Try to do this as efficiently as possible — this is the inner loop of the classifier training. Indeed, the reason for all this primitive-type array machinery is to minimize the amount of time it'll take to train large maxent classifiers.

(Here's a tip: you will likely find that your code spends much of its time taking logs and exps. You can often avoid a good amount of this work using the `logAdd(x, y)` and `logAdd(x[])` functions in `math.SloppyMath`.)

Run `miniTest()` again. Now that it's actually voting properly, you won't get a 0/1 distribution anymore — you'll get 50/50, because, while it is voting now, the weights are all zero. The next step is to fill in the weight estimation code. Look at

```
Pair<Double, double[]> calculate(double[] x)
```

buried all the way in

```
MaximumEntropyClassifierFactory.MaximumEntropyClassifier.ObjectiveFunction
```

This method takes a vector `x`, which is some proposed weight vector, and calculates the value of the objective function we're trying to optimize in training, along with its derivatives. Our objective function is the negative conditional log likelihood of the training data $\langle C, D \rangle$:

$$F(\vec{\lambda}) = -1 \cdot \left[\sum_{\langle c,d \rangle \in \langle C,D \rangle} \log p(c \mid d, \vec{\lambda}) \right]$$

$$p(c \mid d, \vec{\lambda}) = \frac{\exp\left[\sum_i \lambda_i f_i(c,d)\right]}{\sum_{c'} \exp\left[\sum_i \lambda_i f_i(c',d)\right]}$$

(Note that the is here correspond to the indexes generated by the `IndexLinearizer`.) The derivatives of the log likelihood are therefore:

$$\frac{\partial F(\vec{\lambda})}{\partial \lambda_i} = -1 \cdot \left[\left[\sum_{\langle c,d \rangle \in \langle C,D \rangle} f_i(c,d) \right] - \left[\sum_{\langle c,d \rangle \in \langle C,D \rangle} \sum_{c'} p(c' \mid d, \vec{\lambda}) f_i(c',d) \right] \right]$$

Recall that the left summation is the number of times the feature i actually occurs in examples with true class c in the training, while the right sum is the expectation of the same quantity using the label distributions the model predicts.

The current code just says that the objective is 42 and the derivatives are flat. Note that you don't have to guess at `x` — that's the job of the optimization code. All you have to do in the `calculate()` method is evaluate proposed `x` vectors. You have available as method arguments the data, the string-to-index encoding, and the linearizer we discussed before:

```
EncodedDatum[] data;
Encoding encoding;
IndexLinearizer indexLinearizer;
```

Write code to calculate the objective function and its derivatives, and return the `Pair` of those two quantities.

Now run `miniTest()` again. This time, the optimization should find a good solution, one that puts all of the mass onto the correct answer "cat."

Almost done! Remember that putting probability 1.0 on "cat" is probably the wrong behavior here. To smooth, or regularize, our model, we're going to modify the objective function to penalize large weights. In the `calculate()` method, you should now add code which modifies the objective function as follows:

$$G(\vec{\lambda}) = F(\vec{\lambda}) + \sum_i \frac{\lambda_i^2}{2\sigma^2}$$

The derivatives change correspondingly:

$$\frac{\partial G(\vec{\lambda})}{\partial \lambda_i} = \frac{\partial F(\vec{\lambda})}{\partial \lambda_i} + \frac{\lambda_i}{\sigma^2}$$

Run `miniTest()` one last time. You should now get less than 1.0 on "cat" (0.73 with the default sigma).

4.2 Feature engineering for a maxent classifier

Now that your classifier works, goodbye `miniTest()`! Run the `main` method with a single argument of

```
/afs/ir/class/cs224n/pa2/data/maxent/
```

or wherever you may have copied the data. The task is to classify each proper noun into one of these five categories: `company`, `drug`, `movie`, `person` and `place`. The command here loads the proper noun phrase data from the data directory and converts each data instance into a list of `String` features, one for each character unigram in the name. So "Xylex" will become

```
[``X'', ``y'', ``l'', ``e'', ``x'']
```

This should train relatively quickly (should be no more than a few minutes, possibly tens of seconds, and you can reduce the number of iterations for quick tests). It won't work well, though — which is unsurprising. You should get an accuracy of 63.7% using the default amount of smoothing (sigma of 1.0) and 40 iterations. This maxent classifier has the same information available as a class-conditional unigram model, though it'll probably work a little better.

Your job here is to flesh out the feature extraction code in

```
MaximumEntropyClassifierTester.extractFeatures(List<Character> characters)
```

You can take that list of characters and create any `String` features you want, such as `` ``BI-Xy'' `` to indicate the presence of the bigram `` ``Xy'' ``. Or `` ``Length<10'' ``, `` ``Length=5'' ``, `` ``WORD-Xylex'' ``. If you want bigrams (or longer n-grams), you might want to use a `util.BoundedList` to wrap the input list, which lets you ask for list items outside the list's range. Any descriptor of an aspect of the input that seems relevant is fair game (though add feature classes gradually so you can judge how slow you're making your training). Better indicators should raise the accuracy of the classifier. You should easily be able to get your classification accuracy over 70%, and possibly as high as 90% (a lot harder).

5 Evaluation criteria

For Part I, you should 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.

For Part II, you should describe:

- In your implementation of the maxent model code, we'll be looking first for correctness, and secondarily for efficiency.
- Describe the feature templates you used, and also how good and effective they are for the proper noun classification problem.

6 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.

7 Submitting the assignment

7.1 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, 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.

7.2 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.