CS159 - Assignment 6

Part A: Monday, 10/26

Part B: Monday, 11/2 at 11:59pm (any time zone)

`http://mox.ingenierotraductor.com/`

In this assignment we will be implementing and experimenting with word alignment between sentence-aligned texts. For part A you will be getting a basic version of the alignment system working (and trying to convince me that it's correct). For part B, you will be examining one possible extension. Hopefully this flexibility in part B will allow you to investigate something that you find interesting!

As with previous assignments, you may (and I would encourage you to) work with a partner. You may use whatever programming language you'd like as long as I can reasonably get it running on my computer.

Part A: Model 1 Word Aligner

For the first part of this assignment you are to implement IBM Model 1 training algorithm that learns $p(f|e)$ from a data set of aligned sentences and then submit a short writeup evaluating the results.

Implementation details

- **Input**

  Create a script called `wordalign.sh` that takes *four* parameters:

`word-align.sh <english_sentences> <foreign_sentences> <iterations> <probability_threshold>`

- `<english_sentences>` A list of English sentences, one per line. You can assume the data has been preprocessed and should treat anything separated by a space as a "word". This will include numbers, punctuation, etc.
- `<foreign_sentences>` A list of foreign sentences, one per line. This file should have the same number of lines as `<english_sentences>` and there will be a line-wise correspondence between the files. The file has also been preprocessed, so just split on whitespace to get the "words".
- `<iterations>` The number of iterations of EM to run. An iteration includes one update of the model (i.e. $p(e|f)$).
- `<probability_threshold>` Any two words that co-occur[1] in the corpus will have a non-zero probability. However, when printing out the table, we're generally only interested in those that have some non-trivial probability. When printing out the table (see the output section) you should only print out entries that have a probability *greater than or equal* to `<probability_threshold>`.

- **Output**

  Your program should train the model using the data provided for the number of iterations and then print out to standard output the learned probability table (i.e. $p(f|e)$) in the form:

  ```
  <english_word>     <foreign_word>    <probability>
  ```

  The output should be sorted alphabetically based on the combination of English word and foreign word (this will group them by English word) and only include entries whose probability is greater than or equal to the probability threshold. For example, the following would be the output from our example in class with 3 iterations and a probability threshold of 0.0 *and NOT using the NULL alignment*[2]:

  ```
  green    casa    0.35999999999999993
  green    verde   0.6399999999999999
  house    casa    0.6923076923076924
  house    la      0.15384615384615385
  house    verde   0.15384615384615385
  the      casa    0.35999999999999993
  the      la      0.6399999999999999
  ```

- Your code should include the "`NULL`" alignment (printed out in the probability table as `NULL`) that allows for foreign words to appear without having to be aligned to an English word. The good news is that for Model 1 this can be easily be done by just adding a special string

[1] I'll use the term "co-occur" throughout this document to mean when an English word e and foreign word f occur in the corresponding aligned sentences.

[2] Your program should include NULL, so your output of your final program won't match this, though you can remove NULL insertion and check to make sure it matches.

representing `NULL` as an additional word in each English sentence. The most common practice is to put it at index 0, but any will do. Just be careful that whatever string you choose to represent `NULL` does *not* occur anywhere in the actual English corpus.

- Sentences should be treated as sets of words, i.e., ignore repetitions. Since IBM model 1 ignores word order, this will simplify the calculations.

- You will need to keep track of the probability table $p(f|e)$. In theory, this will contain all possible combinations of English and foreign words, initially with uniform probabilities. In practice, the number of words in each language can easily get into the 10K to 100K range, which makes storying all possible combinations impractical.

  The key to fixing this is noticing that after the first iteration of the algorithm, many of these entries will become zero, in particular, any pair that does not co-occur in a sentence in the corpus will be set to zero after the first iteration. In your implementation you *should not store all possible combinations of English and foreign words*, instead only store those pairs that co-occur.

  There are many ways to achieve this, but the easiest is to treat the first iteration of the EM training algorithm as a special case of the general iteration process. During the first iteration when you go through the corpus and are aggregating partial counts, rather than using entries in the word probability table (which would all be uniform anyway) just use a constant value for the probability (I used 0.01, but any constant that doesn't cause overflow is fine). You can use any constant since you will be normalizing the entries in the table at the end to be a proper probability distribution. The value of the constant will only change the value of this normalization constant, but not of the actual probabilities.

  For the remaining iterations you can then just use the entries that actually are in the probability table, which will be all words that actually co-occur in the corpus.

- Your program must be able to run in a reasonable amount of time and with a reasonable amount of memory. My "basic" implementation takes about 10 seconds per EM iteration on 10K sentence pairs and can run easily with 2GB of memory, though if you need 3GB that's fined as long as you're in this same ballpark, you're in good shape. If you have problems running this on your own computer, let me know and I can help you run it on one of the CS servers.

Data

I've put some sentence-aligned data for you to play with in the usual place online at:

`http://www.cs.pomona.edu/classes/cs159/assignments/assign6-starter.zip`

I've included two aligned data sets:

- `en-es.(en|es).zip` A parallel corpus of 100K English/Spanish sentences. This is a subset of the full Spanish/English corpus from the Europarl Corpus[3], which has aligned data from European Parliament proceedings. It has parallel corpora in 21 languages (all with English)!

[3] `http://www.statmt.org/europarl/`

- `test.(en|fr)` A very simple (three sentences) English/French corpus for testing your algorithm on.

Comments/Advice

- Think about what data your program needs to store and how to store this most efficiently. For this assignment, you'll need to worry both about runtime efficiency as well as memory efficiency, though, you don't have to go overboard; just make reasonable decisions and you should be ok.

- As part of your algorithm you'll need to calculate:

$$p(f \rightarrow e) = \frac{p(f|e)}{\sum_{\hat{e} \text{ in } E} p(f|\hat{e})}$$

  Make sure you calculate the denominator of this in a way that avoids repeated calculations for any given sentence pair.

- Since we're dealing with foreign text, you may have to deal with text encoding. For our basic Spanish data, you will have accented words. Most programming languages handle these things naturally, though you may have to explicitly process it using a particular encoding (in our case, UTF-8). For example, if you use the `Scanner` class, you will need to specify it.

- As always, develop your program incrementally and check things along the way. I'd advise first trying to get it working without the `NULL` alignment. Check your counts and other miscellaneous meta-data to make sure they're right and then work from there. You can then check your answer against the class notes. Once you have this working, insert the `NULL` feature. Think about how inserting `NULL` changes your answer and possibly work through it by hand.

- It's easy to get the EM algorithm "almost" right. Find ways to double-check your answers, e.g. a) work through one more example by hand (with or without the `NULL`) and b) do some sanity checks at different stages of the algorithm, like making sure that $\sum_f p(f|e) = 1$ for all e.

Writeup

Once you've got your basic version of EM training working, create a writeup that includes the following three things, explicitly numbered:

1. The output of your program on the `test` sentences for 10 iterations and probability threshold of 0. You should also include a short (2-4 sentences) describing why the output makes sense. Describe not only whether the highest probabilities look reasonable (or not), but also other interesting characteristics like the probability associated with those that don't look good.

2. Run your program on the `en-es` data set with 10K sentence pairs (remember your command-line friend `head`) for 10 iterations and probability threshold of 0.3. Include 20 random entries from the output (when I say "random" I mean some computerized notion of random). Check the correctness of these values using a translation dictionary or online translation service (or even better, somebody who knows English and Spanish). Include a few sentences talking about the correctness/quality of your output. Highlight any mistakes/anomalies.

3. Include a strong argument for why your program is correct. This may include walking through additional examples, analyzing code fragments, etc. I want you to convince me (and yourselves) with whatever means necessary that your version of the program is correct. You will be graded based on how compelling your argument is. This is an important part of this assignment, so please dedicate some time and thought to it.

When you're done with part A

When you're all done, submit via the normal submission mechanism.

Include all of your code and your writeup. Your script `wordalign.sh` should be in the base directory and should work from there. Please **do not submit any of the data files**.

Part B: Extending Your Word Aligner

For part B of the assignment you need to extend your word aligner in *one additional way* listed below and then evaluate the impact of that extension. **You only need to do one extension**, so pick which one seems the most interesting to you.

Extension option 1: memory efficiency

If you try and run the current version of your program on larger pieces of the `en-es` data set, you'll see the memory quickly climb to the point that eventually it won't function efficiently (or at all). My basic version can run the 100K sentences with 8GB of memory, but that's a lot. If you download the entire Spanish/English data set (1.9M sentences) and try it, you're not going to have a lot of success[4].

Note, some of the things below may work better in some languages versus others. For example, if everything is an object (like in python) you can still get some savings from the tricks below, but not as much as if the language has "special" built-in data types like Java or C/C++.

There are a few ways that you can improve the memory efficiency (and to some extent, run-time efficiency) of your training algorithm:

[4] I ran my basic version with 12GB of memory and it still ended up hitting the memory wall.

- The biggest memory problem with the program is most likely because you left everything as a `String`. Strings are represented as an array of characters which is generally a byte per character (though it does depend on character encoding). In addition, Strings are classes in most languages and also incur additional memory overhead because of that.

  A common fix for this is create an index mapping that associates each word in your corpus with a unique integer. Then, do all of your processing with integers (in Java use `int` not `Integer`) rather than Strings. First, do a preliminary pass and read through your corpus and create this vocabulary mapping. Then, start your actual algorithm and replace all your strings with `int`s.

  If you want to be extra hardcore, you can actual do the preprocessing step as a separate program which generates the vocabulary file and then rewrites the corpus out as its integer representation. Your word alignment program can then take in these integer files as input. This is what most actual systems do. Then, you also have to post-process the output.

- Things like lists (e.g. `ArrayList`) are very convenient, but they're very wasteful in memory since they reserve extra space to allow for amortized constant time adding (among other things). Arrays, on the other hand, are contiguous chunks in memory and generally don't use much more than the exact space required to store the data.

- Words that only occur a small number of times in a corpus don't provide much signal to the EM algorithm. So, rather than use all words that occur, you can filter your data set to only include words that occur X times or more (say 3-5 times). For training model 1, if all you're interested in is the $p(f|e)$ table, you can actually just remove these words. If you're trying to do other things, it's better to replace these words with an unknown symbol (e.g. `<UNK>`) like we did with language modeling.

Pick the first memory fix and at least one other fix to implement. Implement them and provide a short writeup with two things:

1. The output from the test example showing the same results as your basic version with your new, memory efficient version.
2. A table or graph comparing the memory (and run-time) usage of your basic input versus the improved version.

Extension option 2: actual alignments

For part A, all that we've calculated is the $p(f|e)$ table, but not any actual alignments. Create a version of your program that can also calculate the alignment of any sentence pair in your training corpus. Then include a writeup that has the following information:

1. Your program should be able to print out the alignments in some reasonable format. Pick a few sentences from the corpus and show what their alignment printout looks like.

2. Pick 10 random sentence pairs (you can pick random sentences under a certain length, to make this easier). Hand-align these sentences. Run them through your model and evaluate the precision and recall of your model. You may coordinate with other groups on generating the 10 random sentences and definitely can get help from a Spanish expert (or Google translate).

Extension option 3: more experimentation

Experiment and create a writeup examining at least two of the components below. Describe your experimental setup, your results and some text interpreting the results.

- How does the number of iterations affect the performance of the algorithm? Calculate some metric that you can monitor as the number of iterations increases, e.g. sum of the difference in $p(f|e)$ between iterations or the actual likelihood of the data under the model and plot this as the number of iterations increases. For the 10K en-es data, what would you suggest as a reasonable number of iterations.
- How does performance vary across languages? The Europarl corpus has 21 languages in it. Pick 2 others besides Spanish, train models and then compare the performance across some evaluation metric. You can use the random sampling approach from part A or some other reasonable metric. You'll need to preprocess the data in the other languages (lowercase and strip off punctuation as separate characters).
- In class I claimed that the initialization doesn't matter, i.e. initializing the $p(f|e)$ with something besides uniform still gets you the same answer. Is this true? Verify this experimentally.
- Something of your own choosing. If you have questions about appropriateness, come talk to me.

Extension option 4: translation

Even though Model 1 is almost never used as an actual translation model, it is a proper translation model. When paired with a proper English language model, it can produce reasonable translations. If you'd like to go this route, come talk to me and we can talk about different ways of tackling this. If you do this option, I'll give you extra credit.

Extension option 5: ???

If you have an idea for something you'd like to investigate or implement related to part A, I'm willing to entertain other options besides those listed here. If you want to go this route, you need to talk with me before part A is due and get approval.

Extra credit

You may implement one additional extension for 5 points of extra credit (or do extension 4).

Submission of part B

When you're all done, submit via the normal submission mechanism.

Include all of your code and your writeup. I will be looking at your code, but the most important part will be your writeup so make sure that you spend time working on this.

If you worked with a partner, put both people's last names on the submitted directory, but only submit one copy.

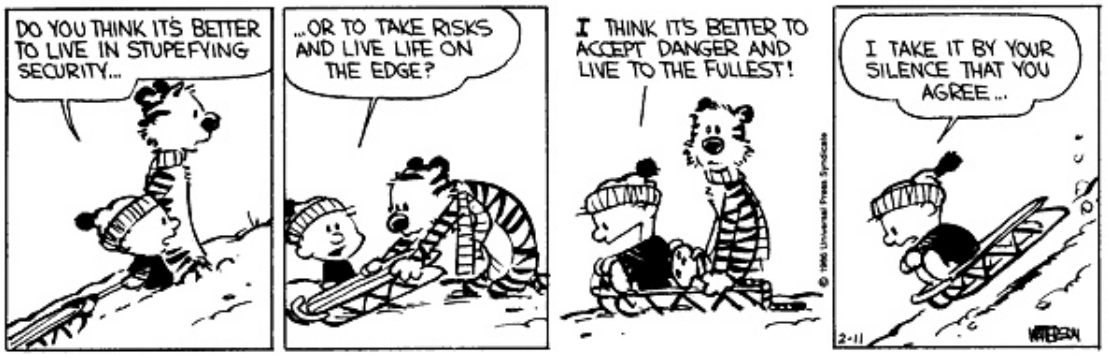
If the naming of the files isn't obvious about where various things are being done, please include a `README` file explaining your organization.

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file you modify.
- Each class and method should have appropriate JavaDoc comments (doc strings if you do it in Python).
- If anything is complicated, put a short note in there to help me out if there are any issues.

This is a non-trivial assignment and it can get complicated, which makes code style and comments very important so that I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.



Grading

Part		points
Part A		
	Correctness	20
	Efficiency	5
	Writeup	15
Part B		25
Comments/style		5
extra credit		6
total		70 + 5 extra