CIS 530 Fall 2014 Homework 2

Instructor: Ani Nenkova
Head TAs: Jessy Li and Shruthi Gorantala

Released: October 14, 2014
Due: 11:59PM October 26, 2014

Overview

The focus of this assignment is language models. You will first build your own bi-gram language model, then create a language model with a state-of-the-art tool which provides felxible options for dealing with unknown words and smoothing. We ask you to compare the effectiveness of different n-gram and smoothing methods using perplexity on test data and two task-based evaluations.

To run your code, please use the Biglab machines. Large jobs will be automatically terminated on Eniac. For instruction on how to use Biglab, please refer to the link provided below:

`http://www.seas.upenn.edu/cets/answers/biglab.html`

Submitting your work

The assignment should be done individually, so each student should make one submission with their own code. The code for your assignment should be placed in a **single file** named `hw2_code_yourpennkey.py`, where `yourpennkey` is the first part of your Penn email address. For this assignment you will also submit: language model files generated by SRILM; `results.txt` that records the results of several problems. *Please make sure you submit all requested files in the specified format.*

All submissions will be electronic, done from your Eniac account. You need to copy your files to Eniac if you choose to work elsewhere. To copy files you can use a SFTP client such as FileZilla, WinSCP or Cyberduck, or secure copy `% scp local_file yourpennkey@eniac.seas.upenn.edu:PATH` where `local_file` is the path to your homework on the local machine and PATH is the path to the location on Eniac where you wish to copy it.

Once you have the file in place in your Eniac account, connect via ssh to `seas.upenn.edu` and use the `turnin` command to submit your files for grading:

```
% turnin -c cis530 -p hw2 hw2_yourpennkey.zip
```

You will get a confirmation message. You can run `turnin` multiple times before the deadline. Each time you run `turnin`, it overwrites your previous submission for that assignment. You can check that the homework was submitted successfully:

```
% turnin -c cis530 -v
```

This will show you the list of files you have submitted.

Code Guidelines

In the assignment below we specify a number of functions. *Your submission should include implementation for each of these required functions, with the specified number and types of arguments.* In addition, you may write any extra functions that you find necessary. Avoid doing any preprocessing of the input text besides that explicitly described in the assignment. This will make grading easier.

Your code will be graded automatically. It is your responsibility to make sure your function can be called correctly and that they return exactly the type of values specified in the assignment. To make sure that your code will run correctly during grading, please include **only** function declarations in your source code or above the line `if __name__ == "__main__"`, and prefix **all** your global variables with `yourpennkey`.

1 Building Language Models

1.1 Named Entity tagging with Stanford CoreNLP (14 Points)

Before collecting the statistics for our language model, we will preprocess the data to substitute all references to named entities with their named entity tags.

The main reason for doing this preprocessing is to introduce you to the Stanford coreNLP tool suite which contains state-of-the-art (or very close to that) tools for most of the NLP tasks that are currently considered solved with practical accuracy.

A second important reason to do this preprocessing is related to data sparsity issues, i.e. to the fact that specific names and dates are repeated rarely in text, so statistics on these rare tokens will not be reliable. The different classes of names entities however occur often and may give a much more robust and predictive counts.[1]

For example the sentence

> Prof. Nenkova has been a member of the Association for Computational Linguistics since 2001.

is likely to contain unknown tokens for a language model trained on news data. In contrast, the preprocessed version is likely to correspond to robust counts in the same language model:

> PERSON has been a member of ORGANIZATION since TIME.

To obtain these tags, we will use the Stanford CoreNLP package. Stanford CoreNLP is a suite of tools that includes most state of the art systems for POS tagging, parsing, NER and coreference resolution. We highly recommend that you get familiar with the system. You are likely to find it very helpful if you work on any NLP-related task in your future work.

To run Stanford CoreNLP, we have to specify a list of annotations which we want it to produce, a list of input files and the output directory. You can access the Stanford CoreNLP toolkit in the homework folder: `/home1/c/cis530/hw3/corenlp/stanford-corenlp-2012-07-09`

You can either call the system from that directory or you can download CoreNLP (225MB) on your local machine or Eniac. From the directory which includes `stanford-corenlp-2012-07-09.jar`, you can execute the following command to generate xml files with the required annotation:

```
java -cp stanford-corenlp-2012-07-09.jar:stanford-corenlp-2012-07-06-models.jar:xom.jar:
joda-time.jar -Xmx3g edu.stanford.nlp.pipeline.StanfordCoreNLP -annotators tokenize,ssplit,
pos,lemma,ner -filelist file_list.txt -outputDirectory <OUTPUT DIRECTORY PATH>
```

Here `-filelist` specifies a raw text file: `file_list.txt`, which includes all the files you want to process. Each line contains a filename with an absolute path to the file. `-outputDirectory` specifies the directory where the output will be saved. `-annotators` list specifies the task which you want CoreNLP to execute. With the list passed on above, we enable tokenization, sentence splitting, POS tagging, lemmatization and NER.

You may find more details regarding how to run the CoreNLP toolkit at:
`http://nlp.stanford.edu/software/corenlp.shtml`

[1] We are not going to ask you to evaluate if preprocessing leads to a better language model than using raw text but with the code that you write for the assignment you can easily compare the two and find out the answer with minimum extra effort.

After running the tool, you will see a list of `.xml` files in the output directory. They contain the annotations for the respective files listed in `file_list.txt`. Here is a sample output file: `http://nlp.stanford.edu/software/corenlp_output.html`.

Make sure that you run the coreNLP toolkit on BigLab. It requires a lot of memory and the process will be terminated automatically if you try to run it on Eniac.

1. Write a function `preprocess(file_list.txt, corenlp_output_dir)` which makes a system call to coreNLP on the `file_list.txt` and stores the output in the `corenlp_output_dir` which contains the coreNLP annotations for files listed in `file_list.txt`.

2. Write a function `process_file(input_xml,output_file)` that reads `input_xml`(xml file generated by StanforCoreNLP) and creates `output_file` that contains the same text but with named entities replaced by their tag and marking the ends of sentences with the word `"STOP"` (added as an explicit, all-caps word). The beginning of file should be with the word `"STOP"` . Remove all punctuations.

1.2 Building your own bigram language model (22 points)

We are now ready to estimate a language model from the training data. We will use Laplace smoothing (add-1). In `/home1/c/cis530/hw2/data/processed_train_set` you will find the pre-processed training data with replaced named entity tags and the `STOP` symbols.

1. Define a class `BigramModel` with the following instance methods:

   - `__init__(self, trainfiles)`: the constructor that builds a bigram language model from the `list` of files specified in `trainfiles`. Assume absolute paths and assume the trainfiles are pre-processed according to 1.1.2.

     The language model ought to be able to handle words not seen in the training data. Such words will most certainly appear if the LM is used in some application to estimate the likelihood of new data. There are many ways to incorporate unknown vocabulary in a language model. In this problem, we will take all words that appear only once and replace them with the symbol `<UNK>`.
   - `logprob(self, context, event)`: returns the log probability of the event (i.e., current word, type `str`) given its context (type `str`). Note that out of vocabulary words should be replaced by `<UNK>`s.
   - `print_model(self, outputfile)` where`outputfile` contains information of the language model in the following format for each line:

     `context1:  word1 logprob(word1) word2 logprob(word2) ...`

     In other words, each line of the file correspond to a word as context, followed by the possible continuations and their log probabilities. The line begins with the context followed by a colon and a single space. All words and log probabilities are separated by a single space.

2. An intrinsic evaluation measure that allows us to compare the quality of language models is perplexity on testing data. The perplexity of a text is inversely proportional to the text likelihood, so the smaller perplexity means the model assigned higher likelihood to the test data (or intuitively speaking the text was what the model expected).

   Computing the likelihood of the test data involves multiplying many probabilities (real numbers between 0 and 1), which in practice is guaranteed to result in underflow. Therefore we will calculate the log likelihood. Note that apart from the practical convenience, logarithms play part in the formal definition of entropy.

   Perplexity is defined as:

$$ppl = 10^{-l}, \text{ where}$$
$$l = \frac{1}{|T|} log_{10} P(T)$$

Note that the J&M book introduces l as the formula for perplexity; It is more standardly called cross-entropy. We also use a log base of 10 instead of 2 such that the results are on the same scale with that of SRILM (see below). Here, $|T|$ is the total number of words in the test file, $log_{10}P(T)$ is the log probability of the test text T.

Add a function `getppl(self, testfile)` to the `BigramModel` class that takes the absolute path to a text file and returns the perplexity of the model computed for that particular text file.

1.3 Language modeling using SRILM (12 Points)

SRILM is a language model tool developed by SRI. It can perform all basic language modeling tasks and includes various smoothing options. Unless one is working on specific improvements of aspects of language modeling, one would be better off using the tool than implementing language models themselves. You can find the documentation for the tool at `http://www.speech.sri.com/projects/srilm/manpages/`.

On eniac, the SRILM executables are located at `/home1/c/cis530/hw2/srilm/`.

1. `ngram-count` is used to build a language model. It takes a single argument, a text file containing the data from which the language model is to be estimated. The data should be formatted one sentence per line, that is sentence boundaries are marked by new line and there are no new lines which occur at other places in the sentence. `ngram-count` can be called using the following command:

   ```
   ngram_count -unk -text YOUR_TEXT_FILE -lm YOUR_MODEL_FILE
   ```

   This is the default setting for a back-off language model, with the maximum order of N-grams to count being 3; Good-Turing discounting is applied. The `-unk` option builds an open-vocabulary language model. Therefore with SRILM you do not need to pre-process the vocabulary for `<UNK>`.

   Here `YOUR_TEXT_FILE` is the data on which the model should be estimated. The estimated model will be saved in `YOUR_MODEL_FILE`, this is SRILM's language model output. Take a look at the output file and you will find that SRILM has generated a list of all uni-, bi-, and trigrams from the training data. Each line of the file contains the log probability and back-off weight for the ngram, "`log-prob ngram backoff-weight`". More details can be found here:
   `http://www.speech.sri.com/projects/srilm/manpages/ngram-format.5.html`

   First, you need to process the data files in the `data/processed_train_set` in the following way:

   - Concatenate all files;
   - Substitute all `STOP` symbols with the new line symbol (`\n`).

   Note that you need to merge the files in the directory into a single file.Do not add `<UNK>` tags as srilm takes cares of it. Each line in the file corresponds to a single sentence.

   Now, build the following three language models using all the files under the training folder `/home1/c/cis530/hw2/data/processed_train_set`:

   - A unigram model with the default discounting method, i.e. Good-Turing, by specifying `-order 1`
   - A bigram model with Ney's absolute discounting of 0.75 and interpolation, by specifying `-order 2 -cdiscount 0.75 -interpolate`.
   - A trigram model with Ney's absolute discounting of 0.75 and interpolation, by specifying `-cdiscount 0.75 -interpolate`.

   Extract the *last 100 lines* of the resulting language model files. Name them as `unigram.srilm`, `bigram.srilm` and `trigram.srilm`, and submit these files with your code.

2. We are going to use SRILM's `ngram` program to calculate the perplexity for the two language models on our test files. The perplexity can be obtained using the following command from the SRILM directory: `ngram -lm YOUR_MODEL_FILE -ppl YOUR_TEST_FILE`

   Write a function `get_srilm_ppl_for_file(lm_file, test_file)` where `lm_file` is the language model generated by SRILM. The test file directory is `/home1/c/cis530/hw2/data/processed_test_set`.

   The `STOP -> newline` substitution should be carried out for the test files as well.

1.4 Comparing language models (16 Points)

So far, we have built four language models:

- **LMID: 0** Your own bigram language model with Laplace smoothing;
- **LMID: 1** SRILM generated unigram model with Good-Turing smoothing;
- **LMID: 2** SRILM generated bigram model with Ney's discounting and interpolation;
- **LMID: 3** SRILM generated trigram model with Ney's discounting and interpolation.

We can compare these language models by finding out which one gives a smaller perplexity on our test data.

Write a function `get_all_ppl(bigrammodel, directory)` that takes an instance of `BigramModel`, a test directory, and returns the perplexity for the combined text of all files in the test directory.

You can now run your language model on test files located at `/home1/c/cis530/hw2/data/processed_test_set`. This directory contains our test data with named entity tag replacements and `STOP` symbols.

Write a function `get_all_ppl_srilm(lm_file, directory)` that takes a language model file generated by SRILM, a test directory, and returns the perplexity for the combined text of all files in the test directory.

List the language models from best to worst in a text file named `results.txt`. Result of the comparison should be provided in a single line starting with "LM ranking:" followed by space-separated LMIDs; for example, `LM ranking: 0 1 2 3` if the best language model was your on bigram language model and the second best was the SRILM unigram model with Good-Turing smoothing and so on. This file should be submitted with your code.

2 Memorable movie quotes. Or not. (16 Points)

Have you ever wondered why you remember some quotes in a movie or a book, but not others? A group of NLP researchers in Cornell did an interesting study on the distinctive measures for movie quotes that are memorable vs. those that are not. In this problem, we will reproduce their experiments that led them to conclude that "memorable quotes use less common word choices", using language models.

Their paper can be found at
`http://www.cs.cornell.edu/~cristian/memorability_files/memorability.pdf`

The authors collected a list of memorable quotes and paired each one of them with a line from the same movie character uttered near the memorable quote. Then they set out to study what characteristics distinguish the memorable quotes from the non-memorable ones.

Specifically, we will compare the perplexity of our language models for memorable and non-memorable quotes. In the `/home1/c/cis530/hw2/data/quotes` directory you will find files containing these quotes. Each pair of memorable and non-memorable lines from the movie script are associated with a pair ID, i.e. for each ID, there are two quotes, one memorable and one non-memorable. The file names are formatted as `QuoteID_mem.txt` and `QuoteID_not_mem.txt`. Note that these quotes are not pre-processed

1. Write a function `get_distinctive_measure(lm_file, mem_quote_file, nonmem_quote_file)` to calculate the perplexity of each quote. Here `mem_quote_file` contains a memorable quote and `nonmem_quote_file` contains a non-memorable quote. `lm_file` is a language model file generated by SRILM. The function return a tuple of the form `(ppl for memorable quote, ppl for non-memorable quote)`. Assume the quote files is pre-processed as 1.1.2

2. Write a function `distinctive_highppl_percentage(lm_file, directory)` that returns the percentage of quote pairs where the memorable quote has higher perplexity. Here `lm_file` is a language model file from SRILM, and `directory` is the quote directory.

   The language models were trained on data preprocessed in a certain way. Make sure that the quotes are preprocessed in the same way before querying the language model.

   Use the trigram language model, report in the second line of `results.txt` the percentage of quote pairs where the memorable quote has higher perplexity. This line should start with "Percentage of memorable quotes from LM ID with higher perplexity: ", followed by the percentage you computed. For example the line could be
   "Percentage of memorable quotes from LM 0 with higher perplexity: 57.32%".

3 Fill in the blanks (20 Points)

For this task, you will be given sentences with blanks and a set of choices for words to fill in. You should predict which word among the given options best fits a particular blank. The decision will be based on your own bigram language model. An example sentence:

`Stocks <blank> this morning.`

i) plunged ii) walked iii) discovered iv) rise

1. Write a function `get_bestfit(sentence, wordlist, bigrammodel)`. Parameter `sentence` is a sentence string such as the one in our example above and `wordlist` is a `list` of words provided for that sentence as choices for the blanks. The location of the blank is marked by the tag `<blank>`. The wordlist is given as comma separated words. Remember to pre-process the sentence by calling CoreNLP and substituting each named entity with its tag and the specifications of 1.1.2

   Using the bigram language model supplied in `bigrammodel`, return the word best fits the blank for the sentence. Remember that the language model was trained on text preprocessed in a certain way. Before applying the language model, the same pre-preprocessing should be applied to the test sentence.

2. Here are a few sentences with some blanks and a set of words which could probably fill in this space. The first choice is the correct one. In the third line of `results.txt`, report the accuracy with which the language model fills the blanks for the sentences given below.

   - `Stocks <blank> this morning.`
     i) plunged ii) walked iii) discovered iv) rise
   - `Stocks plunged this morning, despite a cut in interest <blank> by the Federal Reserve.`
     i) rates ii) patients iii) researchers iv) levels
   - `Stocks plunged this morning, despite a cut in interest rates by the <blank> Reserve.`
     i) Federal ii) university iii) bank iv) Internet
   - `Stocks plunged this morning, despite a cut in interest rates by the Federal Reserve, as Wall Street began <blank> for the first time.`
     i) trading ii) wondering iii) recovering iv) hiring
   - `Stocks plunged this morning, despite a cut in interest rates by the Federal Reserve, as Wall Street began trading for the first time since last Tuesday's <blank> attacks.`
     i) terrorist ii) heart iii) doctor iv) alien

3. Finally, language models can be used to directly predict words that is most likely to occur after a given content, without explicit choices provided ahead of time. Here we will test the extent to which language model predictions are capable of completing a context the same way a person would.

Write a function `fill_blank(sentence, bigrammodel)` that takes in a sentence (with a blank). It uses your own bigram language model (passed as a second argument) to fill in the blank. Return the word of your language model's choice.

Note that as before you would need to first call the preprocessing function to normalize the input sentence in the same way as the data for the language model.

For the following quotes, regard the underlined words as blanks and report words predicted by the language model in `results.txt`. These results should start on a new line directly following the previously recorded results for problems above. The format should be "original:predicted" where original is the underlined word in the example below and predicted is the word most likely to complete this context according to your language model. Each pair is listed on separate line.

- With great powers comes great responsibility
- Say hello to my little friend
- Hope is the quintessential human delusion, simultaneously the source of your greatest strength, and your greatest weakness
- You either die a hero or you live long enough to see yourself become the villain
- May the Force be with you
- Every gun makes its own tune