CIS 430/530 Fall 2012 HW 2

Instructor: Ani Nenkova
TA: Kai Hong, Jessy Li, Achal Shah

Released: October 5, 2012
Due: 11:59PM October 18, 2012

Overview

This assignment focuses on helping you get started with language models and clustering.

Deliverables

You will submit the code for the functions you have implemented. There are some comparison and thinking questions, you are required to do them but you do not need to hand in anything for these questions. There are in total 100 points for the three homework problems.

NOTE: If you do not have an account on Eniac or if you cannot access Biglab, please contact the TA immediately. **Large jobs on Eniac will likely get terminated, so make sure you can access Biglab and run your code there.**

Submitting your work

The code for your assignment should be placed in a single file called hw2_code_yourpennkey.py where yourpennkey is your Penn Key. Since my (Jessy) pennkey is "ljunyi", I would submit the following files: hw2_code_ljunyi.py. You should also submit two additional files, one is your cluster result file, which should be called "cluto.rs"; the other is the SRILM language model output, which should be called "lm.rs".

To electronically submit homework, if you are not already working on Eniac, you need to place the file containing your solution on your SEAS account storage. One way to do this would be to use an SFTP client such as FileZilla or WinSCP.

Then connect via ssh to seas.upenn.edu and use the turnin command to submit your files for grading:

```
% turnin -c cis530 -p hw2 hw2_code_yourpennkey.py cluto.rs lm.rs
```

This should print out a confirmation message. If you are prompted for a section name, type 'ALL. 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 file(s) you have submitted.

Code Guidelines

You can use in-built *NLTK and numpy modules* whenever possible unless specified otherwise. **You are in fact encouraged to do so, because you will likely notice speed improvements by using functions or classes inside these modules**. However, only import the modules that you are actually using. For example:

```
from nltk import * # Bad!
from nltk import FreqDist, ConditionalFreqDist # Good!
```

You may write any extra helper functions that you think are necessary, but all the functions defined in this document should be present.

** **Ambiguity is a very common problem in NLP. For this homework, we will only show you how the answer should LOOK LIKE instead of giving you the answer ran on a smaller corpus. Your code will be both automatically and manually graded, so be sure to put comments and justify the way you solve a problem, especially whenever you feel there can be an ambiguity. Try to write clean code as well.**

Data

The data for this homework is accessible in `/home1/c/cis530/hw2/data`. The training corpus for the language model is in the `corpus/` subdirectory. We will be using corpus from the same source as last time, but in this homework, the corpus consists of 10 companies with 100 documents from each company. In subsequent questions you will be asked to evaluate language models, data for these tasks is put under `test/` subdirectory. This homework also involves word prediction, and the files related to this task is under the `wordfit/` subdirectory.

1 Modeling and Predicting

Based on the company data, we will build language models, and use these to compute for a set of given sentences with blanks within them, the most likely word to fill in the blanks.

1.1 Some Preparation

We will first further clean up the numbers in the data before training a language model.

1. (2 points) Create a function `word_transform(word)`, such that when given a word, it would return either the word itself in lower-case, or if the word is a number, just return "num".

   ```
   >>> word_transform('34,213.397')
   'num'
   >>> word_transform('General')
   'general'
   ```

2. (2 points) Next, create a similar function `sent_transform(sent_string)` that returns the sentence as a formatted list of words (excluding the `None` types).

   ```
   >>> sent_transform("Mr. Louis's company (stock) raised to $15 per-share, growing 15.5%
   at 12:30pm.")
   ['mr.', 'louis', "'s", 'company', '(', 'stock', ')', 'raised', 'to', '$', 'num',
    'per-share', ',', 'growing', 'num', '%', 'at', 'num', ':', '30pm', '.']
   ```

1.2 Building Language Models

1. (8 points) Create a function `make_ngram_tuples(samples, n)` that returns a sequence of all the n-grams seen in the input, in order. The function returns a sequence of tuples where each tuple is of the form (context, event). The context is None (the Python built-in value None, not the string "None") in the case of $n = 1$ (unigrams), and for larger values of n it is a tuple of the preceding $n - 1$ words for each sample.

```
>>> samples = ['her', 'name', 'is', 'rio', 'and', 'she', 'dances', 'on', 'the', 'sand']
>>> make_ngram_tuples(samples, 1)
[(None, 'her'), (None, 'name'), (None, 'is'), (None, 'rio'), (None, 'and'), (None, 'she'),
(None, 'dances'), (None, 'on'), (None, 'the'), (None, 'sand')]
>>> make_ngram_tuples(samples, 2)
[(('her',), 'name'), (('name',), 'is'), (('is',), 'rio'), (('rio',), 'and'),
(('and',),  'she'), (('she',), 'dances'), (('dances',), 'on'), (('on',), 'the'),
(('the',), 'sand')]
```

   Note that in Python a tuple displayed as `(x,)` is a tuple consisting of one element, `x`. The comma is there simply to establish that this is a tuple, and not just parentheses around a value.

2. (8 points) We will implement a language model with Laplace smoothing (add 1) over training data. Define a class `NGramModel` with the following instance methods. In this problem, you **cannot** use the ngram module in NLTK.

   - `def __init__(training_data, n)`: builds an n-order language model using the list of tokens supplied in training data.
   - `logprob(context, event)`: returns the log probability of the event given the context.

```
>>> words = ['the', 'fulton', 'county', 'grand', 'jury', 'said', ...]
>>> model = NGramModel(words,2)
>>> model.logprob( ('the',), 'fulton' )
-1.70474809224
```

1.3 Fill in the Blanks

In this question you will be given a set of sentences under `wordfit/` with file names such as "sent1.txt" or "sent2.txt". Each of the file contains one sentence, and in each sentence, there will be a `-blank-`. For example, `'Stocks -blank- this morning.'`. Under the sentence there is a list of words, one word per line. The task is to find the best fit of the blank in a sentence file by choosing a word from the corresponding word list file.

1. (5 points) First, write a function `build_bigram_from_files(file_names)` to train a bigram language model using the NGramModel class, with all 1000 documents in the corpus, tokenized and formatted as in section 1.1. You do not need to consider sentence or document boundaries.

```
>>> file_names = ['file1','file2','file3',...]
>>> lm = build_bigram_from_files(file_names)
>>> lm.logprob(('her','name'),'is')
-1.70474809224
```

2. (3 points) Write a function `get_fit_for_word(sentence, word, langmodel)` that given a formatted sentence, gives the log probability of the sentence with the word substituting `-blank-` using our bigram language model.

```
>>> get_fit_for_word('stocks -blank- this morning','rose', lm)
-1.70474809224
```

3. (5 points) Finally, write a function `get_all_bestfits(dir)` to be used with the data inside `wordfit/`, and return a list of best fit words, one for each sentence (file). (Don't forget to format to be compatible with the bigram language model).

```
>>> get_all_bestfits('/home1/c/cis530/hw2/data/wordfit/'):
['calls','vice','considered']
```

2 Clustering Documents

In this problem we will use a clustering software called CLUTO[1] to cluster similar documents together.

2.1 Generate the Matrix

(10 points) Say there are n documents. First, we need to produce a $N \times N$ similarity matrix for each pair of them. We will calculate document pair similarities in the same way as the last homework, i.e. define a common dictionary consisting of top words from all documents, and take cosine similarity with every pair of documents. In this question, we will be using the top frequent words without stopwords method from the last homework to find the top 200 words for each company. You should only keep counts, do not do TF.IDF weightings. Your top words dictionary should consist of the 10 companies under `corpus/`.

Write a function `get_cluto_matrix(file_names)` that generates a similarity matrix according to the instructions above. (`file_names` is a `list` of files). This function should return the square matrix. Also, make sure you keep track of which row/column in the matrix corresponds to which files, let's call it `label_arr` and it will be used later. We will use (the faster and quicker) *numpy*'s `zeros()` function to define the matrix.

```
>>> file_names = ['file1','file2','file3']
>>> gen_cluto_matrix(file_names)
array([[1.0,        0.24532325, 0.73121242],
       [0.24532325, 1.0,        0.23456434],
       [0.73121242, 0.23456434, 1.0]])
```

2.2 Running CLUTO

(10 points) We will cluster all our documents into 15 clusters, including the testing document. Later on we will use the test document's cluster to evaluate language models. The location of CLUTO's binary executable is at `/home1/c/cis530/hw2/cluto-2.1.1/Linux/scluster`. (You will find two executables, but we will be using `scluster`).

Refer to CLUTO's manual (at `/home1/c/cis530/hw2/cluto-2.1.1/manual.pdf`) and do the following:

1. Look up scluster's input format in section 3.3.1 from the manual, and produce (write) the corresponding matrix (graph) file for all companies under `corpus/` as well as the one document under `test/`, a total of 1001 documents, using the matrix generation function from the last problem. Notice here that we can view our matrix as a complete graph.

2. Cluster the 1001 documents into 15 clusters.

   You can use the following command:

[1] http://glaros.dtc.umn.edu/gkhome/cluto/cluto/overview

```
$ CLUTO_S -clustfile=cluto.rs graph_file 15
```

where `CLUTO_S` is the path to scluster, `cluto.rs` is the output file for CLUTO, which you will hand in, and `graph_file` is the matrix output file we did previously.

2.3 Match the clusters

1. (6 points) We are interested in finding out which cluster our test document belongs to.

   Write a function `find_doc_cluster(cluster_file, label_arr, file_name)` that finds the cluster id `file_name` belongs to. `cluster_file` will be CLUTO's output, in our case, `cluto.rs`. `label_arr` is just simply a list (or a numpy array) of files each row or column of the matrix corresponds to as we mentioned earlier.

   ```
   >>> label_arr = ['file1','file2','file3',...]
   >>> find_doc_cluster('cluto.rs', label_arr, 'testfile.txt')
   5
   ```

2. (7 points) We would also like to know the documents each cluster has. Write a function `rebuild_clusters(cluster_file, label_arr, excl_file=None)` that returns a dictionary whose keys are cluster ids, and the values are the list of documents in that cluster. We have a new `excl_file` parameter here, such that when specified, the file would be excluded from the dictionary returned. We will need this functionality later.

   ```
   >> rebuild_clusters('cluto.rs', label_arr, excl_file='testfile.txt')
   {0: ['/home1/c/cis530/hw2/data/corpus/starbucks/101230.txt', ...],
    ...
    12: ['/home1/c/cis530/hw2/data/corpus/starbucks/10339.txt', ...],
    ...}
   ```

3 Language Models with SRILM

3.1 Preparation

(5 points) SRILM is a language model tool developed by SRI. You can find its documentations at http://www.speech.sri.com/projects/srilm/manpages/.

The software's executables are located at `/home1/c/cis530/hw2/srilm/`.

`ngram-count` is used to build language models. The input text file it takes is a single file with one sentence per line. We will format it in the same way as section 1.1, and we will "stick" the words in a sentence together by separating each word with a space. For example, the sentence "Mr. Louis's company (stock) raised to $15 per-share, growing 15.5% at 12:30pm.", will now become "mr. louis 's company (stock) raised to $ num per-share , growing num % at num : 30pm ." You might find the sentence and word tokenization functions in homework 1 to be useful. **Make sure you convert words to lowercase after sentence tokenization.**

Define a function `print_sentences_from_files(file_names, outfilename)`, where `file_names` is a list of the files we want to transform into a single file according to SRILM's format, and `output_filename` is the corresponding output file we generate and will feed into SRILM. Use this function and generate the following SRILM input files:

- An input file with all documents from `test/118742636.txt`'s cluster, but excluding the file itself (we'll call it `cluster_text`).

- An input file with all documents *except* the ones in `test/118742636.txt`'s cluster (we'll call it `exclclus_text`).
- An input file for `test/118742636.txt` (we'll call it `test_text`).

3.2 Impact of Clustering

1. (4 points) Now we are ready to generate our models using the following command:

   ```
   $ NGRAM_COUNT_LOC -text YOUR_TEXT_FILE -lm YOUR_MODEL_FILE
   ```

   Where `NGRAM_COUNT_LOC` is just the location of `ngram_count`; `YOUR_TEXT_FILE` corresponds to the files we just generated as input; and `YOUR_MODEL_FILE` is SRILM's language model output. Take a look at the output file and you will find that SRILM has generated all uni-, bi-, and trigrams. Each line of the file is of the format "`log-prob ngram backoff-weight`". You can refer to *http://www.speech.sri.com/projects/srilm/manpages/ngram-format.5.html* for more details about the format.

   In this problem we will compare the two language models we built previously: one for all documents from our test data's cluster (but excluding our test data), i.e. from `cluster_text`, and one from all documents but our test data's cluster, i.e. from `exclclus_text`.

   Write a function `gen_lm_from_file(input, output)` to call `ngram_count` with the above command, such that `input` is `YOUR_TEXT_FILE` and `output` is `YOUR_MODEL_FILE`.

2. (7 points) Perplexity over unseen test data is an evaluation method for a language model. We are going to use SRILM's `ngram` program and our `test_text` to calculate the perplexity for the two language models we generated. The perplexity can be obtained using the following command:

   ```
   $ NGRAM_LOC -lm YOUR_MODEL_FILE -ppl YOUR_TEST_FILE
   ```

   Write a function `get_lm_ranking(lm_file_list, test_text_file)` that when given a list of language model files from `ngram_count`, outputs a list of tuples (lm_file, ppl) sorted from the best language model to the worse, evaluated with `test_text_file`.

   Now, which language model do you find is better? The clusters, or the ones not in the cluster? *Write it down as comment in your code.*

3.3 Comparing Smoothing Methods

1. There are many techniques for smoothing. We will explore the ones covered in class here. Generate language models with `ngram_count` over all files in our corpus - that is, the 1000 documents under `corpus/`, using the following smoothing methods:

   (a) The default discounting method is Good-Turing, we can get this just by using the command in section 3.2. *Name this language model output as "lm.rs" which you will hand in.*

   (b) Laplace for all uni-, bi-, and trigrams, by specifying `-addsmooth 1`.

   (c) Ney's absolute discounting of 0.75, by specifying `-cdiscount 0.75`.

   (d) Ney's absolute discounting with interpolation of both bigrams and trigrams, by specifying `-interpolate`.

2. Use the function `get_lm_ranking` to get a ranking of the above language models with different smoothing methods, evaluated with `test_text`. Which smoothing method is the best here? And which one is the worst?

3. (10 points) We will further investigate different smoothing methods by getting the top 10 ngrams in the language model file with the best smoothing method obtained from above, and see their ranks in models with other smoothing methods. Write a function `get_rank_differences(ref_lm_file, lm_files, n)` that returns a tuple `(tops, ranks)`, where `tops` is the top n ngrams in `ref_lm_file`, `ranks` contains one list per `lm_file`, each list is the rank of each of the top ngrams in one `lm_file`.

```
>>> get_rank_differences('my_best_lm', ['lm_smooth1','lm_smooth2','lm_smooth3], 10)
(['thenew york times', 'lowecompanies inc', '<s> lowecompanies inc', 'period fiscal num',
  '<s> heinz </s>', 'ended september num', 'num shareholders record', 'from consolidated net',
  'ended march num', 'ended july num'],
 [[1, 2, 3, 5, 8, 14, 15, 16, 21, 22], [42, 57, 58, 95, 129, 233, 234, 239, 258, 264],
  [15, 72, 1, 19, 57, 11, 51, 29, 4, 3]])
```

3.4 Prediction Again

(8 points) Re-do the "fill in the blank" task in 1.4 using the language model with the best smoothing method obtained from above.

Write a function `get_all_bestfits(dir, lm_file)` to be used with the data inside `wordfit/`, and return a list of best fit words, one for each sentence (file). Don't forget to format to be compatible with the language model.

```
>>> get_all_bestfits('/home1/c/cis530/hw2/data/wordfit/', 'my_best_lm'):
['calls','vice','considered']
```

Now, compare with our answer for 1.4. Do you think the prediction is better?