Natural Language Processing
Assignment 2
Due Wednesday, April 26 (by midnight)

A2: N-Gram Language Models

Authors. *Klinton Bicknell, Harry Eldridge, Nathan Schneider, Lucia Donatelli, Alexander Koller*

Submission. Each group member should upload a .zip file to Canvas with:

1. All of your code and output files (named as instructed below);
2. A PDF with all written responses (separate from your code; see word limits below) and a brief description of how each group member contributed to the assignment.

Starter code. A2.zip

Grading. The assignment is worth 100 points, distributed as follows: Q0: 15 (5 code, 5 plots, 5 written); Q1: 5, Q2: 10 (5 code, 5 written), Q3: 10, Q4: 15 (5 code, 10 written), Q5: 10, Q6: 20 (10 code, 10 written), Q7: 15; Bonus: 10.

The assignment assumes you are using at least Python 3.5, preferably with the Anaconda distribution. Prior to completing the assignment, you should install and familiarize yourself with the spaCy, NLTK, and numpy libraries, which will be reviewed during Thursday practice sessions.

I. Inspecting the data & Zipf's law

0. Empirically verify Zipf's law on the Brown corpus.

[Coding and written answer]

To access the *full* Brown corpus, import it from NLTK. Compute a list of unique words sorted by descending frequency for (i) the whole corpus and (ii) two different genres of your choice.

Before moving on, extract the following information (should be visible in your code and output files): number of tokens; number of types; number of words; average number of words per sentence; average word length. You should also run a default part-of-speech tagger on the dataset and identify the ten most frequent POS tags.

Next, Use the Python library `matplotlib` to plot the frequency curves for the corpus and two genres you choose: i.e. x-axis is position in the frequency list, y-axis is frequency. Provide both a plot with linear axes and one with log-log axes.

Provide your source code for all of the above in problem0.py.

Written QA:
Research the origin of the corpus and provide a brief discussion of the findings in light of what you find out. Be sure to highlight any key important linguistic observations that relate to class discussions. (word limit: 200)

II. Unigram model

1. Creating the `word_to_index` dictionary

[Coding only: use starter code problem1.py]

The first step in building an n-gram model is to create a dictionary that maps words to indices (which we'll use to access the elements corresponding to that word in a vector or matrix of counts or probabilities). You'll create this dictionary from a vocabulary file that lists each word in the corpus exactly once: brown_vocab_100.txt

This file lists the 811 word types used in the first 100 sentences of the Brown corpus and includes the special word types `<s>` and `</s>`, for a total of 813 word types. In this version of the Brown corpus, punctuation marks are also treated as words, so they are also included in this total. In the corpus itself, punctuation words are separated from other words by a space, just like other words are, so you won't need to treat them differently at all.

The brown_vocab_100.txt file is formatted to contain one word per line. We want to create a dictionary in python that maps each word to its order of occurrence in this file, starting with 0. For example, the first word in the file is 'all' and this should map to 0. The last word is 'resolution' and this should map to 812. To create the dictionary in python, you'll iterate over each line (which is a word plus a newline character), use `rstrip()` to remove the trailing newline character, and then add the word to the dictionary mapped to its appropriate index. To do this, you'll need to keep track of which line of the file you're in using `enumerate()`.

After creating this dictionary, write the dictionary to a file called `word_to_index_100.txt`. Because `wf.write()` only writes strings, you will need to convert the dictionary to a string before you can write it using the `str()` function. To check that you've done this correctly, verify in the file output that 'all' is mapped to 0 and 'resolution' to 812.

2. Building a MLE unigram model

[Coding and written answer: use starter code problem2.py]

Now you'll build a simple MLE unigram model from the first 100 sentences in the Brown corpus, found in: brown_100.txt

This corpus is represented as one sentence per line with a space separating all words, as well as the end-of-sentence word `</s>`. You'll need to split the sentences into a list of words (e.g., using the string's `.split()` member function) and convert each word to lowercase (e.g., using the string's `.lower()` member function).

First, copy your code from problem 1 to load the dictionary mapping words to indices.
Then `import numpy as np` and initialize the numpy vector of `counts` with zeros. Finally, iterate through the sentences and increment counts for each of the words they contain.

Now print the `counts` vector. (Do this in the python interpreter.)

Written QA:
Estimate (just by eyeballing) the proportion of the word *types* that occurred only once in this corpus and explain your estimate. Do you think the proportion of words that occur only once would be higher or lower if we used a larger corpus (e.g., all 57000 sentences in Brown)? Use concepts discussed in class with examples for your answer. (word limit: 200)

Finally, to normalize your counts vector and create probabilities, you can simply divide the counts vector by its sum in numpy like so:

```
probs = counts / np.sum(counts)
```

Write your new `probs` vector to a file called `unigram_probs.txt` and verify that the first probability in it (the probability of 'all') is 0.00040519 and that the last probability (probability of 'resolution') is 0.00364668. (Note that our model trained on this small corpus has estimated that 'resolution' is about 10 times as frequent as 'all'! Models trained on very small corpora are very noisy.)

III. Bigram Models

3. Building an MLE bigram model

[Coding only: use starter code problem3.py]
For a review on Maximum Likelihood Estimation (MLE), see SLP chapter 3.

Now, you'll create an MLE bigram model, in much the same way as you created an MLE unigram model. We recommend writing the code again from scratch, however (except for the code initializing the mapping dictionary), so that you can test things as you go. The main differences between coding an MLE bigram model and a unigram model are:

- you'll initialize a numpy matrix instead of a numpy vector
- you'll increment counts for a combination of word and previous word. This means you'll need to keep track of what the previous word was. I recommend doing this by (1) initializing `previous_word = '<s>'` just prior to the `for` loop iterating through a line (skipping the `'<s>'` at the 0th index), and then (2) inside the `for` loop, adding a line that sets `previous_word = word` *after* incrementing the counts for the current `previous_word, word` pair. Think this through to make sure you understand why this ensures that the variable `previous_word` will always be the appropriate previous word when incrementing counts.
- you'll now normalize each row of the counts matrix instead of just normalizing a single counts vector. To do that, replace the line involving `np.sum()` with the lines

```
from sklearn.preprocessing import normalize
probs = normalize(counts, norm='l1', axis=1)
```

- Make sure the matrix is arranged so each row represents the probability of a word given the seen word so that it works with the generator. For more on this see generate.py

When complete, add code to write the following probabilities to `bigram_probs.txt`, one per line:

- p(the | all)
- p(jury | the)
- p(campaign | the)
- p(calls | anonymous)

Make sure that the first two of these probabilities are 1.0 and about .08333.

4. Add-α smoothing the bigram model

[Coding and written answer: save code as problem4.py]

This time, copy `problem3.py` to `problem4.py`. We'll just be making a very small modification to the program to add smoothing. In class, we discussed two types of smoothing in detail: Laplace smoothing, in which 1 *pseudocount* is added to every bigram count, and add-α smoothing, in which some amount α is added to every bigram count. (Laplace smoothing is thus a special case of add-α smoothing.)

You should modify your problem to use add-α smoothing with α, i.e., pretending that we saw an extra one-tenth of an instance of each bigram. With numpy, you can very simply add 0.1 to every cell of a matrix like so:

```
counts += 0.1
```

Now also change the program to write the same four probabilities as before to a file called `smooth_probs.txt`. To verify that your program is working correctly, check that the first probability in the file is about 0.0133657.

Finally, compare `smooth_probs.txt` to `bigram_probs.txt`.

Written QA:
Why is smoothing useful when calculating probabilities related to language? Why did all four probabilities go down in the smoothed model? (word limit: 200)

Now note that the probabilities did not all decrease by the same amount. In particular, the two probabilities conditioned on 'the' dropped only slightly, while the other two probabilities (conditioned on 'all' and 'anonymous') dropped rather dramatically.

Written QA:
Why did add-α smoothing cause probabilities conditioned on 'the' to fall much less than these others? And why is this behavior (causing probabilities conditioned on 'the' to fall less than the others) a good thing? (word limit: 200)

In figuring this out, you may find it useful to look at the relevant individual rows of the counts matrix (prior to adding the 0.1) to see how they're different. In numpy, you can look at nth row of the `counts` matrix using `counts[n,]`.

IV. Using n-gram models

5. Experimenting with an MLE trigram model

[Coding only: save code as problem5.py]

Using your knowledge of language models, compute what the following probabilities would be in both a smoothed and unsmoothed trigram model (note, you should not be building an entire model, just what you need to calculate these probabilities):

- p(past | in, the) (should be 0.0625 for unsmoothed, and ~0.011305 for smoothed)
- p(time | in, the)
- p(said | the, jury)
- p(recommended | the, jury)
- p(that | jury, said)
- p(, | agriculture, teacher)

6. Calculating sentence probabilities

[Coding and written answer]

For this problem, you will use each of the three models you've constructed in problems 2–4 to evaluate the probability of a toy corpus of sentences (containing just the first two sentences of the Brown corpus) found at: toy_corpus.txt

The sentences are contained in this corpus in the same format as the other corpora you've been using so far: one sentence per line and words separated by a space. As before, you'll need to remove the end-of-line character.

First, you'll edit `problem2.py`, and add code at the bottom of the script to iterate through each sentence in the toy corpus and calculate the joint probability of all the words in the sentence under the unigram model. Then write the probability of each sentence to a file `unigram_eval.txt`, formatted to have one probability for each line of the output file.

To do this, you'll be writing a loop within a loop very similarly to how you iterated through the training corpus to estimate the model parameters. But instead of incrementing counts after each word, you'll be updating the joint probability of the sentence (multiplying the probabilities of each word together). One easy way to do this is to initialize `sentprob = 1` prior to looping through the words in the sentence, and then just update `sentprob *= wordprob` with the probability of each word. At the end of the loop, `sentprob` will contain the total joint probability of the whole sentence. To verify that this worked correctly, note that the joint probability of the second sentence under this model should be 4.84008387782e-99

Next, you'll transform this joint probability into a perplexity (of each sentence), and write that to the file instead. To calculate the perplexity, first calculate the length of the sentence in words (be sure to include the end-of-sentence word) and store that in a variable `sent_len`, and then you can calculate `perplexity = 1/(pow(sentprob, 1.0/sent_len))`, which reproduces the definition of perplexity we discussed in class.

Now, write the perplexity of each sentence to the output file instead of the joint probabilities. To verify that you've done this correctly, note that the perplexity of the second sentence with this model should be about 153.

Now, you'll do the same thing for your other two models. Add code to `problem3.py` to calculate the perplexities of each sentence in the toy corpus and write that to a file `bigram_eval.txt`. Similarly, add code to `problem4.py` and write the perplexities under the smoothed model to `smoothed_eval.txt`. To verify that you did these correctly, note that the perplexity of the second sentence should be about 7.57 with the MLE bigram model and about 54.28 for the smoothed bigram model. Note when calculating perplexity that the number of bigrams is different from the number of tokens. Compare the perplexities of these two sentences under all three models.

Written QA:
Compare the performance of the different models. What do you notice? How can we evaluate the performance of each in relation to another? (word limit: 200)

Written QA:
Did smoothing help or hurt the model's 'performance' when evaluated on this corpus? Why might that be? (word limit: 200)

7. Generation

[Written answer]

Edit `problem2-4.py` to use the provided generator code to generate some (~10) sentences from each of your models.

Store the sentences in `unigram_generation.txt`, `bigram_generation.txt`, and `smoothed_generation.txt` to go with their respective models.

Written QA:
Compare the models qualitatively based on their generation. How does each perform? Cite examples that illustrate linguistic principles discussed in class (ambiguity, compositionality, creativity, etc.) and compare these across models. Attempt to explain the performance of the models given what you know about how they work. (word limit: 500)

(Bonus on next page!)

V. Bonus (up to 10 points)

In statistical NLP we frequently make independence assumptions about relevant events which are not actually correct in reality. We are asking you to test the independence assumptions of unigram language models.

Pointwise mutual information,

$$\mathrm{pmi}(w_1, w_2) = \log \frac{P(X_t = w_1, X_{t+1} = w_2)}{P(X_t = w_1) \cdot P(X_{t+1} = w_2)} \approx \log \frac{C(w_1 w_2) \cdot N}{C(w_1) \cdot C(w_2)},$$

is a measure of statistical dependence of the events $X_t = w_1$ and $X_{t+1} = w_2$; *C(w)* is the absolute frequency and *N* is the size of the corpus. If the probability of the next word in the corpus being w_2 is unaffected by the probability of the previous word being w_1, then pmi(w_1, w_2) = 0; otherwise the pmi is positive or negative.

Calculate the pmi for all successive pairs (w_1, w_2) of words in the Brown corpus. Words (not word pairs!) that occur in the corpus less than 10 times should be ignored. List the 20 word pairs with the highest pmi value and the 20 word pairs with the lowest pmi value. Document and submit your observations and code.

Written QA:
Discuss the validity of the independence assumption for unigram models**.** (word limit: 200)