LIN 353C: Introduction to Computational Linguistics, Spring 2017, Erk

Homework 6: Language Models

Due: Tuesday April 4, 2017

This homework comes with a file, `lastname_firstname_hw6.py`. This is a (basically empty) file, called a *stub file*. Please record all the answers in the appropriate places of this file. Please make sure that you save the file in *plain text*, not something like the Word format.

For submission, please rename the file such that it reflects your name. Also do not forget to put your name and EID in the second line of the file.

Please put your Python code into the answers file. You can omit statements that produced an error or that did not form part of the eventual solution, but please include all the Python code that formed part of your solution. **Please submit your homework solution electronically using Canvas.**

If any of these instructions do not make sense to you, please get in touch with the instructor right away.

A perfect solution to this homework will be worth *100* points.

1. **Trigram language models (10 pts.)**

   The Markov assumption says that when determining the probability of the next word in long sequence of items, as in $P(w_{10,001}|w_1, w_2, w_3, w_4, \ldots, w_{10,000})$, it is not the whole sequence that matters; we can approximate this probability by looking only at a few items directly preceding $w_{10,001}$. A bigram language model makes the assumption that it suffices at one directly preceding item. A trigram language model assumes that one only needs to look at the two directly preceding items, that is $P(w_{10,001}|w_1, w_2, w_3, w_4, \ldots, w_{10,000}) \approx P(w_{10,001}|w_{9,999}, w_{10,000})$.

   A bigram language model estimates the probability of the word sequence

   `<s> I know </s>`

   as

   $$P(\texttt{<s> I know </s>}) \approx P(\texttt{I}|\texttt{<s>}) \cdot P(\texttt{know}|\texttt{I}) \cdot P(\texttt{</s>}|\texttt{know})$$

   (where we omit the $P(\texttt{<s>})$ in the beginning because we assume that the probability of starting with `<s>` is one.)

   Now assume the word sequence is

   `<s> My iguana is on fire </s>`

   How does a *trigram* language model estimate the probability of this word sequence? That is, how is $P(\texttt{<s> My iguana is on fire </s>})$ approximated by a trigram language model? Write down a formula analogously to the one for "I know" above, but for a trigram language model and for the given word sequence.

2. **Estimating trigram probabilities (10 pts.)**

   In class we have discussed that to estimate probabilities like $P(\texttt{know}|\texttt{I})$ from data, we often use *Maximum Likelihood Estimation*, which in this case means that we estimate the probability as relative frequency.

   For a bigram language model, this means

   $$P(\texttt{know}|\texttt{I}) = \frac{\text{count}(\texttt{I}, \texttt{know})}{\text{count}(\texttt{I}, \_)}$$

   For a *trigram* language model, how would you estimate $P(\texttt{fire}|\texttt{is}, \texttt{on})$?

3. **Understanding language models (10 pts.)**

You turn on the radio as it is broadcasting an interview. Assuming a trigram model, match up expressions (A), (B), (C) with descriptions (1), (2), (3):

The expression

(A) $P(\texttt{Do}) \cdot P(\texttt{you}|\texttt{Do}) \cdot P(\texttt{think}|\texttt{Do you})$

(B) $P(\texttt{Do}|\texttt{<s>}) \cdot P(\texttt{you}|\texttt{<s> Do}) \cdot P(\texttt{think}|\texttt{Do you}) \cdot P(\texttt{</s>}|\texttt{you think})$

(C) $P(\texttt{Do}|\texttt{<s>}) \cdot P(\texttt{you}|\texttt{<s> Do}) \cdot P(\texttt{think}|\texttt{Do you})$

represents the probability that

(1) the first complete sentence you hear is "Do you think" (as in, "D'ya think?")

(2) the first 3 words you hear are "Do you think"

(3) the first complete sentence you hear starts with "Do you think"

Explain your answers briefly.

This problem is from Jason Eisner's homework on language modeling.

4. **Maximum likelihood estimation (35 pts.)**

In class, we used Python code to compute bigram probabilities $P(word_2|word_1)$ as follows: First, count word bigram frequencies using `nltk.ConditionalFreqDist()`, second, convert counts to probabilities using `nltk.ConditionalProbDist()`.

In this problem, we will do these two steps by hand.

For the first step, we need to store counts that depend on two words, $word_1$ and $word_2$. In Python, we can do that as follows: We have a dictionary with an entry for each $word_1$. The entry for $word_1$ then needs to map any possible $word_2$ to the count of "$word_1$ $word_2$", so it is a dictionary again. That is, we have a dictionary that maps a word to a dictionary that maps a second word to a count.

Here is the code for this. This code is also provided in a separate file `countbigrams.py` with this homework.

```
import nltk

count = { }

corpus = """
<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>
"""

words = corpus.split()

# nltk.bigrams(words) yields a list of pairs,
# [('<s>', 'I'), ('I', 'am'), ...]
# I could iterate over them by saying
# 'for pair in nltk.bigrams(words):...'
# Or I can use assignment to multiple
# variables at once, and that is what I will do.
# As a reminder, assingment to multiple variables
# looks like this:
# word1, word2 = ('I', 'am')
# and that will assign 'I' to word1 and 'am' to word2.
# I am just doing this in a for-loop.
#
```

```
for word1, word2 in nltk.bigrams(words):
    if word1 not in count:
        count[word1] = { }

    if word2 not in count[word1]:
        count[word1][word2] = 1
    else:
        count[word1][word2] += 1

# This printout just demonstrates
# that the code does the right thing
for word1 in count:
    for word2 in count[word1]:
        print(word1, word2, count[word1][word2])
```

For this problem, extend the program to compute probabilities using Maximum Likelihood Estimation: Use the `count` dictionary with the computed frequencies from the "Sam corpus", and compute another dictionary `prob` that will contain the estimated probability for each bigram. It should map a word *word*1 to a dictionary that maps a word *word*2 to the probability $P(word_2|word_1)$. (See problem 1 for a reminder of how to estimate probabilities using Maximum Likelihood.)

You do not have to do any smoothing.

The file `prob4result.py` contains the dictionary that this should give you.

5. **Generating random text (35 pts.)**

   (a) NLTK's `ConditionalProbDist` has a function `generate()` to generate random text. For this problem, we take a closer look at what it does.

   In the "Sam corpus", the probability of beginning a sentence with 'I' is 2/3, that is, $P(\texttt{I}|\texttt{<s>}) = 2/3$, while the probability of beginning a sentence with 'Sam' is 1/3.

   To generate random text, we could start with `<s>` and then always continue with the most likely word. In this case, that would mean we would start every sentence with "I", and the sentence would also always continue in the same way.

   This is not what the `generate()` function does. It will start a sentence with 'I' in about 2/3 of all cases, and with 'Sam' in about 1/3 of all cases.

   You can imagine what it does like this: You have a stick with a mark on it at 2/3 of the stick's length. The part that takes up the first 2/3 is labeled 'I'. The part that takes up the remaining 1/3 is labeled 'Sam':

   Now you throw a dart at this stick. If you hit the stick in the first, larger part, you say "I". If you hit the stick in the last third, you say "Sam".

   The easiest way to implement this in Python is to use the package `random`. It has a function called `random()`, which will output a random number between 0 and 1. Here is an example of what this function does:

```
>>> import random
>>> random.random()
0.7221120542995859
>>> random.random()
0.15412333377125453
>>> random.random()
0.7693526277446497
```

```
>>> random.random()
0.5272664572462025
>>>
```

So to simulate you throwing a dart at the stick above, you generate a number using `random.random()`. If the number you generated is less than 2/3, you say "I". If the number is greater or equal 2/3, you say "Sam".

Using your dictionary `prob` (computed from the Sam corpus) from the previous problem, write code that, given a word, will generate a random next word using the "stick-throwing" method sketched above. Use this to generate a random text of 50 words starting in `<s>`, using your probabilities calculated from the Sam corpus.

(If you did not complete problem 4, use the dictionary in `prob4result.py`.)

(b) The file `alicecorpus.txt` that comes with this homework contains the beginning of "Alice in Wonderland", processed to have each sentence start with `<s>` and end with `</s>`. Apply your code from problem 4 to it to get probabilities $P(word_2|word_1)$ for this corpus. (If you did not complete problem 4, use the dictionary in `prob5result.py`.)

While in the "Sam corpus", most words only had a single other word that could follow, and a few words had a choice of two words that can follow, this corpus has words that can be followed by more than two other words. For example, your newly computed dictionary of probabilities should show that the word "no" can be followed by "one" with a probability of 0.4, or "idea" with a probability of 0.2, or "pictures" or "mice", each again with a probability of 0.2.

The same idea of "throwing a dart at a stick" applies, it is just that the stick now has more sections:

To simulate the dart-throwing, you can again use `random.random()`:

If it returns a number between 0 and 0.4, you say "one". If it returns a number between 0.4 and 0.6, you say "idea". If the number is between 0.6 and 0.8, you say "pictures". And finally, if the number is greater than 0.8, you say "mice".

Using your probability dictionary `prob` for the Alice corpus, write code that, given a word, will generate a random next word using the "dart-throwing" method sketched above. Use this to generate a random text of 50 words starting in `<s>`, this time using probabilities from the Alice corpus.