CS-5340/6340, Programming Assignment #1
Due: Monday, September 21, 2015 by 11:00pm

Your task for this assignment is to build an N-gram language model. Your language modeling program should accept two input files: (1) a training corpus file and (2) a test sentences file. Your program should accept these files as command-line arguments in the following order:

ngrams <training_file> <test_file>

Input Files

The *training_file* will consist of sentences, one sentence per line. For example, a training file might look like this:

> I love natural language processing .
> This assignment looks like fun !

You should divide each sentence into unigrams based solely on white space. Note that this can produce isolated punctuation marks (when white space separates a punctuation mark from adjacent words) as well as words with punctuation symbols that are still attached (when white space does <u>not</u> separate a punctuation mark from an adjacent word). For example, consider the following sentence:

“This is a funny-looking sentence” , she said !

This sentence should be divided into exactly nine unigrams:
(1) “This (2) is (3) a (4) funny-looking (5) sentence” (6) , (7) she (8) said (9) !

The *test_file* will have exactly the same format as the *training_file* and it should be divided into unigrams exactly the same way.

Building the N-gram Language Models

To create the N-gram language models, you will need to generate tables of frequency counts from the training corpus for unigrams (1-grams) and bigrams (2-grams). An N-gram should <u>not</u> cross sentence boundaries. All of your N-gram tables should be case-insensitive (i.e., “the”, “The”, and “THE” should be treated as the same word).

You should create three different types of language models:

(a) A unigram language model with no smoothing.
(b) A bigram language model with no smoothing.
(c) A bigram language model with add-one smoothing.

You can assume that the set of unigrams found in the training corpus is the entire universe of unigrams. We will not give you test sentences that contain unseen unigrams. So the vocabulary V for this assignment is the set of unique unigrams that occur in the training corpus.

However, we will give you test sentences that contain bigrams that did not appear in the training corpus. The n-grams will consist entirely of unigrams that appeared in the training corpus, but there may be new (previously unseen) combinations of the unigrams. The first two language models (a and b) do not use smoothing, so unseen bigrams should be assigned a probability of zero. For the last language model (c), you should use *add-one smoothing* to compute the probabilities for all of the bigrams.

Computing Sentence Probabilities

For each of the language models, you should create a function that computes the probability of a sentence $P(w_1...w_n)$ using that language model. Since the probabilities will get very small, you must do the probability computations in log space (as discussed in class, also see the lecture slides). **Please do these calculations using log base 2.** If your programming language uses a different log base, then you can use the formula on the lecture slides to convert between log bases.

Output Specifications

Your program should print the following information for each test sentence. When printing the logprob numbers, please only print 4 digits after the decimal point. For example, print -8.9753864210 as -8.9754. The programming language will have a mechanism for controlling the number of digits that are printed. If P(S) = 0, then the logarithm is not defined, so print logprob(S) = undefined.

Please print the following information, formatted like this:

```
S = <sentence>

Unigrams: logprob(S) = #
Bigrams: logprob(S) = #
Smoothed Bigrams: logprob(S) = #
```

For example, your output might look like this (the examples below are not real, they are just for illustration!):

```
S = Elvis has left the building .

Unigrams: logprob(S) = -9.9712
Bigrams: logprob(S) = -12.2933
Smoothed Bigrams: logprob(S) = -8.4819
```

FOR CS-6340 STUDENTS ONLY! (40 additional points)

CS-6340 students should also create a function that can automatically *generate* sentences using the **unsmoothed bigram language model**. Your program should accept an additional command-line argument that will be a *seeds_file*, which will contain a list of words to begin the language generation process. So your program should accept three command-line arguments:

ngrams <training_file> <test_file> <seeds_file>

First, your program should produce log probabilities for the test sentences using each of the language models, as described in the previous section. Next, your program should run the language generation process described below. The *seeds_file* will have one word per line, and each word should be used to start the language generation process.

Creating an N-gram Language Generator

Your language generator should use the unsmoothed bigram language model to produce new sentences, probabilistically! Given a *seed word*, the language generation algorithm is:

1. Find all bigrams that begin with the seed word - let's call this set B_{seed}. Probabilistically select one of the bigrams in B_{seed} with a likelihood proportional to its probability.

   For example, suppose "crazy" is the seed and exactly two bigrams begin with "crazy": "crazy people" (frequency=10) and "crazy horse" (frequency=15).
   Consequently, $P(people \mid crazy) = \frac{10}{25} = .40$ and $P(horse \mid crazy) = \frac{15}{25} = .60$.

   There should be a 40% chance that your program selects "crazy people" and a 60% chance that it selects "crazy horse".

   An easy way to do this is to generate a random number x between [0,1]. Then establish ranges based on the bigram probabilities. For example, if $0 \leq x \leq .40$ then your program selects "crazy people", but if $.40 < x \leq 1$ then your program selects "crazy horse".

2. Let's call the selected bigram $B' = w_0\ w_1$ (where w_0 is the seed). Generate w_1 as the next word in your new sentence.
3. Return to Step 1 using w_1 as the new seed word.

Your program should stop generating words when one of the following conditions exists:

- your program generates one of these words: . ? !
- your program generates 40 words (NOT including the original seed word)
- B_{seed} is empty (i.e., there are no bigrams that begin with the seed word).

IMPORTANT: For each seed word, your language generator should randomly generate 10 sentences that begin with that word. Since each sentence is generated probabilistically, the sentences will (usually) be different from each other.

Output Specifications

Your program should print each seed word followed by a blank line and then the 10 sentences generated from that seed word. You should format your output like this:

```
Seed = <seed>

Sentence 1: <sentence>
Sentence 2: <sentence>
Sentence 3: <sentence>
Sentence 4: <sentence>
Sentence 5: <sentence>
Sentence 6: <sentence>
Sentence 7: <sentence>
Sentence 8: <sentence>
Sentence 9: <sentence>
Sentence 10: <sentence>
```

For example, your output might look like this:

```
Seed = Elvis

Sentence 1: Elvis has gone fishing for french fries .
Sentence 2: Elvis has left McDonald's !
Sentence 3: Elvis sings to eat peanut butter sandwiches ?
Sentence 4: Elvis has sandwiches and left the building .
Sentence 5: Elvis is alive and this sentence is an example where the generator
keeps generating words without producing any punctuation marks corresponding
to the end of a sentence so it keeps going until the 40 word limit is reached and
it stops
Sentence 6: Elvis sings a lot .
Sentence 7: Elvis is buried at Graceland and likes to eat peanut butter .
Sentence 8: Elvis has left and lived in Mississippi .
Sentence 9: Elvis sang rock and roll and sings ?
Sentence 10: Elvis has left !
```

GRADING CRITERIA

We will run your program on the files that we give you as well as new files to evaluate the generality and correctness of your code. **So please test your program thoroughly!** Even if your program works perfectly on the examples that we give you, that does not guarantee that it will work perfectly on different test cases.

ELECTRONIC SUBMISSION INSTRUCTIONS
(a.k.a. "What to turn in and how to do it")

Please submit 3 things:

1. The source code for your program. Be sure to include all files that are needed to compile and run your program!
2. A README file that includes the following information:
   - how to compile and run your code
   - which CADE machine you tested your program on
     (this info may be useful to us if we have trouble running your program)
   - any known bugs, problems, or limitations of your program

   REMINDER: your program *must* compile and run on the linux-based CADE machines! We will not grade programs that cannot be run on these CADE machines.
3. Submit one trace file called **ngrams.trace** that shows the output of your program when given the input files on the CS-5340 web page.

   CS-6340 Students: your program should first process the test file to generate sentence probabilities according to the language models and then process the seeds file to generate new sentences.

You can generate a trace file in (at least!) 3 different ways: (1) print your output to a file called `ngrams.trace`, (2) print your output to standard output and then pipe it to a file (e.g., `ngrams training.txt test.txt > ngrams.trace`), or (3) print your output to standard output and invoke the unix *script* command before running your program. The sequence of commands to use is:

```
script ngrams.trace
ngrams training.txt test.txt
exit
```

This will save everything that printed to standard output during the session to a file called `ngrams.trace`.

To turn in your files, the CADE provides a web-based facility for electronic handin, which can be found here:

https://webhandin.eng.utah.edu/

Or you can log in to any of the CADE machines and issue the command:

handin cs5340 ngrams <filename>

HELPFUL HINT: you can get a listing of the files that you've already turned in via electronic handin by using the 'handin' command without giving it a filename. For example:

```
handin cs5340 ngrams
```

will list all of the files that you've turned in thus far. If you submit a new file with the same name as a previous file, the new file will overwrite the old one.