HOMEWORK 2

OVERVIEW

In this assignment you will be using n-gram models to model English phonotactics or 'wordlikeness'.

You will use your n-gram models in two ways: to generate random language (Part 1) and to assign scores to language data (Part 2).

There are three provided files. *word_transcriptions.txt* is a large phonetic dictionary of English, derived from the *CMU Pronouncing Dictionary.* The format is one word on each line, followed by a space and phonemes (represented as one or two letters) separated by spaces:

```
absorption AH B Z AO R P SH AH N
accountant AH K AW N T AH N T
actors AE K T ER Z
administering AE D M IH N AH S T ER IH NG
advised AE D V AY Z D
aged EY JH D
...
```

It uses an English-specific phonetic alphabet of 39 phonemes. You should check out the sounds that they represent by examining the file and by visiting the following website for examples of all the phonemes:

http://www.speech.cs.cmu.edu/cgi-bin/cmudict#phones

You will use this file for training your n-gram models. There are also two mystery files, *X.txt* and *Y.txt*. The made-up words in these files were used in an experiment where English speakers had to rate these words' English-likeness (e.g. phonotactic wellformedness). The words in one file were consistently rated poorly, while the words in the other file received high ratings.

PART 1

In this part of the assignment you will write a program (45pts) that generates random bi-grams and tri-grams. Call the program **ngram.py** and have it read in two arguments at the command line:

```
python ngram.py training_file N
```

where:

training_file is the phonetic dictionary
N is either '2' or '3' and stands for bi-gram (2) or tri-gram (3).

Given these arguments the program should train an n-gram model and then print out 25 random 'words' according to that model. Words should be generated starting from the special symbol '#' and generation should stop once '#' is randomly generated.

TIPS

I strongly suggest you approach the programming in small pieces. First, write the code that trains bi-grams and generates random bi-grams. You may want to consult the sample *word_bigram.py* program distributed in class. Make sure this part works correctly before moving on. Then expand your code so it can also deal with tri-grams. Although trigrams keep track of a longer history, the structure of the code

and the data structures used should be very similar. The units you will need to keep track of and iterate over will be *longer*, but no matter the length of the n, there are only two subsequences to keep track of: the n-gram sequence that goes in the numerator, and the (n-1)-gram sequence that goes in the denominator. If it is easier for you, it's ok to write separate code and functions for the bi-gram case and the tri-gram case and have your program walk through distinct code depending on whether N = 2 or 3. That is, you do not have to (although you can) write code that deals in a unified way with a generic N.

QUESTIONS

After you've written the program, use it to examine the output you get with bi-grams and tri-grams and answer the following questions:

Question 1 (10pts)
For both bi-grams and tri-grams, give 5 examples of words that look especially unnatural. For each word, explain why the model generates these unnatural strings.

Question 2 (3pts)
Which model (the bi-gram or tri-gram) generates words that look more natural and English-like? Why is this model better able to produce English-like words? Be specific!

PART 2

For this part you will extend your program (30pts) to implement add-1 smoothing and calculate the perplexity of words in an input file. Have your program check whether it is getting three command line arguments, and if so, have it calculate the perplexity of the words in the file specified as the last argument. So for this part, it should be run exactly as follows:

```
python ngram.py training_file N test_file
```

where:
training_file is once again the phonetic dictionary
N is 2 (bi-gram) or 3 (tri-gram)
test_file is the file whose perplexity should be calculated (either X.txt or Y.txt)

If your program gets only two arguments (a file and N), it should run *exactly* as before: printing 25 random words using unsmoothed ngrams.

When run with a third argument, the extended program should build a smoothed bi- or tri-gram using the training_file and then go through each of the words in the test file. For each word, it should print out the word followed by a tab and then the **probability** of that word on a single line, with one word per line. After printing all the words and their probabilities, it should print the **perplexity** of all the words in the file. That is, your program prints out one line for each line of the file, plus one additional line with a number representing the 'weighted average branching factor' of the whole corpus.

NOTE ON CALCULATING PROBABILITIES AND PERPLEXITY

You should *never* multiply probabilities in your code. Probabilities are extremely small, and when you multiply them together, they get even smaller. By round-off error the total probability will come out as 0 for any sizeable input. You should instead find the **sum of the log probabilities**, and use this sum to find the perplexity. To find the log of a number x in python, import math, then type "math.log(x, base)" so the log base 2 of 32 would be "math.log(32,2)" .

Specifically, you should find the log probability (base 2) of each word by summing the logs of each of the n-gram probabilities in the word. Then use the log probability of each word to calculate and print the corresponding probability. As you examine each word, also keep a running sum of the log probabilities of all the words (lets assume this sum is called "log_sum"). The perplexity for the corpus will then be:

Perplexity = 2**(log_sum*((-1)/N))

where N is the number of n-gram probabilities multiplied together. For both bi-grams and tri-grams, you should calculate the probability of each phoneme given the preceding context, and then calculate the probability of ONE final # following the last phoneme. This means that for both bi-grams and tri-grams, *N is the number of phonemes plus the number of words* in the file (there is one n-gram probability for generating each phoneme plus one n-gram probability for generating the final '#' on each line).

TIPS

You should re-use much of the code you wrote for Part 1 in the extended program. Again, I suggest you do this in pieces. First, modify the portions of code that calculate the probabilities in order to do add-1 smoothing whenever there is an input file to score. You should do this by having a variable that is set to 1 when you have three arguments and to 0 when you only have two arguments, and you should use this variable in the calculation of the n-gram probabilities. This part of the programming should be relatively minimal (hint: think of the formula for add-lambda smoothing), but it's necessary for calculating perplexity so that unseen n-grams don't cause the probabilities of all the words to be 0. Make sure you loop (and smooth) over all possible n-grams, not just those you encountered in the training file! The rest of the coding involves reading in the test file and calculating and printing its likelihood/perplexity.

QUESTIONS

Question 3 (6)
Use smoothed bigrams and trigrams to find the perplexity of *X.txt* and *Y.txt*. What is the perplexity of each corpus according to each model? How should the results be interpreted (that is, what does each model say about how English-like the two corpora are)?

Question 4 (6)
Look at the probabilities assigned to each word in the X and Y files by the two models. A model does a good job distinguishing between English-like words from non-English-like words if it is possible to choose a threshold such that the probabilities of all words in one file are below this threshold and the probabilities of all words are above it in the other file. Is it possible to do this for the bi-gram model? How about the tri-gram model? Which model does a better job distinguishing the two sets of words? Explain.

EXTRA CREDIT

Use the analogical model from HW1 to find the similarity of each word in the X and Y files to the English lexicon in the phonetic dictionary. Does the n-gram or analogical model do a better job of distinguishing the two sets of words? What are limitations of the analogical model – what kinds of words does it seem to rate incorrectly, and why does it do this?

WHAT TO TURN IN

Turn in your final **ngram.py** program. Make sure your code is well commented in general. Submit your answers to the questions in a separate file (in PDF format).