Assignment 2: Language Modeling

COSC 9/LING 50: Computational Linguistics (Winter 2013)

Posted on Jan 18. **Due Feb 01 before 12:30 pm.**

You may turn in your assignment as a hardcopy at the beginning of class on January 25, or to `cs9@cs.dartmouth.edu`. Code may be written in any language and submitted by e-mail. Indicate how many hours you spent on this assignment. If you discussed any of it with a classmate or got help, note that information.

1 Problem Set

1.1 Maximizing Entropy

(2 points) Consider a text with 104 characters drawn from a vocabulary of 26 letters [a-z]. What would the text have to be in order to have maximum possible unigram entropy under that vocabulary? What is the entropy of that text?

1.2 N-Gram Options

(18 points) Approximately what order n-grams would I have to use to model each of these problems? Would it have to be at the level of characters, words, or sentences? Would class-based models give a significant advantage?

There are no absolute right or wrong answers, just some that are more right than wrong. It's correct as long as you can convince me, so explain your reasoning.

1. Generating a nonsense English word that looks English-like in its spelling.
2. Predicting whether the English morpheme '-s' will be pronounced as `z` or `s`. (Non-linguists: listen carefully to some English plural nouns to figure out a pattern – `places`, `papers`, `tips`, `cats`, `dogs`, etc.)
3. Predicting the next character in a partially-typed word.

4. Context-sensitive spell-check.
5. Checking the coherence of a translation of a sentence from Chinese to English.
6. Checking the coherence of an OCRed sentence. (OCR = optical character recognition)
7. Generating a fake academic paper by training a language model on a corpus of papers. (See `http://pdos.csail.mit.edu/scigen/` for an actual example. They use a context-free grammar, but you could do a pretty good job with n-grams.)
8. Predicting whether an e-mail is spam.
9. Automatic capitalization in English. (Type in lowercase in a word processor like MS Word and observe where it auto-capitalizes.)

1.3 Soda-Pop-Coke

Your co-worker is about to order a sweet non-alcoholic carbonated beverage at lunch. You don't know much about her, but you have this information.

- There is a 45% chance she is from California, 25% that she's Texan, and 30% that she's from Illinois.
- Here's a breakdown of how each state orders its drinks.

	Soda	Pop	Coke
California	70%	25%	05%
Illinois	20%	60%	20%
Texas	10%	5%	85%

1. Is she most likely to call it a pop, soda, or coke?
2. She orders – you don't quite catch what she said, but you know it's one syllable (and therefore not soda). What's your guess now?
3. She clarifies that she ordered a pop. Where do you hypothesize she's from?

2 Programming

This section may look intimidating, but there actually isn't that much coding! The trick is to re-use your functions wherever you can.

2.1 Random Text Generation

(35 points) Write a program to build a random text generator as seen in class on Wednesday. You needn't make it as general as the program we saw – just have it model word trigrams. The user should be able to provide the name of the text file to train the model and the number of words to generate. You can re-use the code from Tutorial 2, but you will need to modify it to learn trigrams at the word level, rather than bigrams at the letter level. **Ignore the previous version of this assignment: do *not* implement add-σ smoothing in this section.**

Writing this will involve two main steps:

1. Read a plaintext file (it's up to you whether or not you want to lowercase or retain punctuation and non-alphabetic characters) and split it into words. Build a conditional probability table – using nested dictionaries in Python or some such – of trigram word probabilities.

2. Using the above trigram probability distribution, generate a random text of the appropriate length. That is, if we denote the random text by w, for each position i, generate a word w_i based on the previous 2 words according to the probability $P(w_i|w_{i-1}, w_{i-2})$.

   The paragraph below has also been updated from the previous version:

   How do you generate the first two words of the text? There are several ways, but a simple one is to just choose a context uniformly from the set of trigram contexts. You can do this in python with `random.choice(trigrams.keys())`, where `trigrams` is your language model.).

2.2 Perplexity

(25 points) Divide your-language-UDHR into an 80% training portion, which will be used to build a model, and a 20% test portion. You can do this by dividing the string that you read from the file into an 80-20 split.

Build a trigram word model with add-σ smoothing only on the training part of the text. That is, only add σ to the counts of the trigrams that have been seen in training, not to all possible zero-count trigrams.

Computing the perplexity on unseen data will be problematic now because there may be n-grams in your test data that were not seen in training! So augment the model by setting the count of all unseen trigrams in the *test* data to be σ. In effect, it is the same as adding σ to certain n-grams that originally had zero count.

Try calculating the perplexity of this model on the test portion. **For this problem, you can start computing cross-entropy from the *third* word in the test data; i.e.:**

$$PPL(M, w) = 2^{H(M,w)} \tag{1}$$

$$H(M, w) = \frac{1}{|w| - 2} \sum_{i=3}^{|w|} -\log P(w_i | w_{i-1}, w_{i-2}; M) \tag{2}$$

where M is the trigram model learned from the training data, and w is the test data. $|w|$ denotes the number of words in the test sample, and $P(w_i|w_{i-1}, w_{i-2}; M)$ implies the probability of the i^{th} word conditioned on the previous two words, with the probability computed according to the model M.

Experiment with different values of σ to find the one that gives the lowest perplexity. Letting σ be a command-line argument will make it easier to do this.

3 Choose Your Adventure

(10 points) Answer question 3.1, 3.2, or 3.3. The first question is more programming, the second involves linguistic analysis with a sprinkling of coding, and the third is a math problem. You may answer more than one of the questions: the highest score will count towards this assignment, and the lower scores will be applied to your extra credit/participation grade.[1]

3.1 Linear Interpolation

Retain your code for section 2 and write the solution to this question in a different file.

Implement linear interpolation. You will have to learn separate unigram and bigram models in addition to a trigram one. Do not use add-σ smoothing for now.

Train all these models on the training part of your-language-UDHR and calculate the perplexity on the test data. The perplexity will be in terms of P_{int}.

$$P_{\text{int}}(w_i|w_{i-1}, w_{i-2}) = \lambda_3 P(w_i|w_{i-1}, w_{i-2}) + \lambda_2 P(w_i|w_{i-1}) + \lambda_1 P(w_i) \tag{3}$$

You need not explicitly compute P_{int} for every possible w_i, w_{i-1}, and w_{i-2}. Instead, compute it on the fly when calculating perplexity.

Once again, there will be n-grams in the test data that you haven't seen in training. But because of the interpolation, we only need to smooth the unigram distribution in order

[1] As usual, the extra credit is undervalued, so do it mainly for your own satisfaction.

to avoid zero probability! So add σ to all your unigram counts *and* set the count of every unigram in the test data that you haven't seen in training to be σ. Fix σ **at 0.01 so you don't have to experiment with its value in this section.** Don't change any of the bigram or trigram counts.

Experiment with different values of λ_3, λ_2, and λ_1 to see what gives you the best perplexity. Make sure that $\lambda_3 + \lambda_2 + \lambda_1 = 1.0$.

3.2 Langu Model with Stemm

This problem will involve a very small amount of additional coding – most of it is a matter of experimenting with different values and analyzing the results.

Think about morphology and stemming. Say that a word stem is just the first 5 characters of the word, or the entire word if the word's length is less than 5.[2] For example, the stem of `language` would be `langu`, and the stem of `stemming` would be `stemm`.

Read the entire text of your-language-UDHR as a list of words, and transform it so all the words are converted to their stems in this fashion. Re-use your code from section 2.1 to train a trigram model over words on this new "stemmed" version of the text, with no smoothing. Let's call the model `stemmed-trigrams-yourlanguage`. Also train `trigrams-yourlanguage`, which is a trigram model on the original your-language-UDHR.

Compute the entropies of `stemmed-trigrams-yourlanguage` and `trigrams-yourlanguage`. How does the entropy change when using stemmed text?

Repeat the same experiment on English (i.e., train the models `stemmed-trigrams-english` and `trigrams-english`). Is the change in entropies similar to the change for your-language? What, if anything, does this say about your-language in comparison to English?

Try changing the number of prefix characters that define the stem in your-language (e.g., let the stem be the first 4 characters instead of 5). Train a new trigram model and compute the entropy. Does it increase or decrease? Do you have any other creative ways to do simple stemming that would make a text more or less predictable?

3.3 smarg-n

We know that the probability of a sentence w according to a trigram language model is

$$P(w) = P(w_0)P(w_1|w_0)P(w_2|w_1, w_0)P(w_3|w_2, w_1)\dots P(w_n|w_{n-1}, w_{n-2})$$

[2]Sadly/shockingly, this is how many 'real-world' applications do stemming!

This is a model for generating words from left to right. We can produce the same sentence by generating words from right to left, starting from w_n, w_{n-1}, etc., all the way to w_0. Then, the probability of the sentence according to the "reverse trigram" model is given by:

$$P_{\text{rev}}(w) = P(w_n)P(w_{n-1}|w_n)P(w_{n-2}|w_{n-1}, w_n)\dots P(w_0|w_1, w_2)$$

Prove that

$$P(w) = P_{\text{rev}}(w) \tag{4}$$