CS 6320.002: Natural Language Processing
Fall 2019

Homework 1 – 90 points
Issued 26 Aug. 2019
Due 8:30am 09 Sept. 2019

Deliverables: A tarball or zip file containing your code and your PDF writeup.

0 Getting Started

Make sure you have downloaded the data for this assignment:

- `shakespeare.txt`, the complete plays of Shakespeare
- `warpeace.txt`, Tolstoy's *War and Peace*
- `sonnets.txt`, Shakespeare's sonnets

1 A Basic N-Gram Language Model – 27 points

We will start with a very basic n-gram language model. Open a new file `ngram.py` and write a generator function `get_ngrams(n, text)`, where `n` is an int that tells you the size of the n-grams, and `text` is a list of words/strings (if you don't know what a generator function is, look up the `yield` keyword). The function's output should be n-gram tuples of the form `(word, context)`, where `word` is a string and `context` is a tuple of the n-1 preceding words/strings. Make sure to pad `text` with enough start tokens `'<s>'` to be able to make n-grams for the first n-1 words; also make sure to add stop token `'</s>'` (we will need it in Part 4).

Next, define a class `NGramLM`. Write its initialization method `__init__(self, n)`, which saves the int `n` and initializes three internal variables:

- `self.ngram_counts` for n-grams seen in the training data,
- `self.context_counts` for contexts seen in the training data,
- `self.vocabulary` for keeping track of words seen in the training data.

Writeup Question 1.1: What data types did you use for the two counters and the vocabulary, and did you initialize the vocabulary to be empty or already containing some token(s)? Explain why. You may want to wait to answer this question until after you complete the programming parts, in case you change your mind.

Add a method `update(self, text)` that updates the `NGramLM`'s internal counts and vocabulary for the n-grams in `text`, which is again a list of words/strings.

Now write a function `create_ngramlm(n, corpus_path)` that returns an `NGramLM` trained on the data in the file `corpus_path`. This is not a word tokenization homework, so simply use `split()` to tokenize lines using whitespace.

Now that we can train a model, we need to be able to use it to predict word and sentence probabilities. Write a method `word_prob(self, word, context)` that returns the probability of the n-gram `(word, context)` using the model's internal counters; the output should be a float. If `context` is previously unseen (ie. not in the training data), the probability should be $1/|V|$, where V is the model's vocabulary.

Writeup Question 1.2: Why do we have a special case for unseen contexts? Why do we set the probability to be $1/|V|$?

To predict the probability of a sentence, we multiply together its n-gram probabilities. This can be a very small number, so to avoid underflow, we will report the sentence's log probability instead. Import the `math` library and write a function `text_prob(model, text)` that returns the log probability of `text`, which is again a list of words/strings, under `model`, which is a trained `NGramLM`. The choice of base for the log doesn't matter as long as it's consistent with the base you use for perplexity in Part 3, so we will just use the library's default base e. The output of this function should be a (negative) float.

We are now ready to predict the probability of a sentence. Train a trigram `NGramLM` on `warpeace.txt` and use it to predict the probabilities of the following sentences:

- God has given it to me, let him who touches it beware!
- Where is the prince, my Dauphin?

Writeup Question 1.3: What are your model's predicted probabilities for these two sentences? Did anything unusual happen when you ran the second sentence? Explain what happened and why. (If you're not sure or can't remember from what we talked about in class, try stepping through your code to see what's going on.)

2 Out-of-Vocabulary Words and Smoothing – 30 points

First we will add support for out-of-vocabulary words. We need to add a special token `'<unk>'` to our vocabulary and get counts for it by replacing some of the words in the training data with `'<unk>'`.

Write a function `mask_rare(corpus)` that takes an entire training corpus (eg. all of `warpeace.txt`) and returns a copy of `corpus` with words that appear only once replaced by the `'<unk>'` token.

Update `create_ngramlm()` to use the masked version of the training corpus. You will also need to update `NGramLM.word_prob()` to use the `'<unk>'` token's counts whenever a word or context contains an out-of-vocabulary word.

Writeup Question 2.1: Try predicting the log probability of that second sentence again. Have we fixed whatever was going on? Why or why not?

Now we will implement smoothing. Add a new argument `delta=0` to the method `word_prob_()` (if you aren't familiar with this notation, look up argument default values) and update the method to return Laplace-smoothed probabilities. Be careful with this step! The formula for Laplace smoothing in the slides is specifically for bigrams only; it won't work correctly for larger n-grams. You will need to modify the formula to apply to larger n-grams.

(Hint 1: You are modifying a probability distribution. What does it mean to be a probability distribution?)

(Hint 2: You may want to add another internal variable to `NGramLM` to make this modification easier to implement.)

Writeup Question 2.2: How did you modify the Laplace smoothing formula? Explain why the modification was necessary.

Writeup Question 2.3: Try predicting the log probabilities of both sentences again using different values for delta. What do you get? How does the value of delta affect the predicted log probabilities? Based on these examples, do you think Laplace smoothing works well for n-gram language models? Why or why not?

Let's try another type of smoothing: linear interpolation. Define a class `NGramInterpolator` and write the following methods:

- `__init__(self, n, lambdas)`, where `n` is the size of the largest n-gram considered by the model and `lambda` is a list of length n containing the interpolation factors (floats) in descending order of n-gram size. This method should save `n` and `lambdas` and initialize n internal `NGramLM`s, one for each n-gram size.

- `update(self, text)` should update all of the internal `NGramLM`s.

- `word_prob(self, word, context, delta=0)` should return the linearly interpolated probability using `lambdas` and the probabilities given by the internal `NGramLM`s.

Writeup Question 2.4: Train a trigram `NGramInterpolator` with `lambdas = [0.33, 0.33, 0.33]` and use it to predict the log probabilities of the two example sentences. What do you get? How does its compare with the base `NGramLM`, both with and without smoothing?

3 Perplexity – 15 points

Write a function `perplexity(model, corpus_path)` that returns the perplexity of a trained model on the test data in the file `corpus_path`. You will need to load the test data from file, just like you loaded the training data (you don't need to mask rare words, though), and you will need to count N, the total number of tokens in the test data. Make sure you are using the same base e as in Part 1.

Writeup Question 3.1: Train two trigram `NGramLM`s on `shakespeare.txt`, one with smoothing (use `delta = 0.5`) and one without. (As you have probably noticed, the `NGramInterpolator` is slower because it builds multiple models, so we will be using plain `NGramLM`s for the rest of the homework.) Evaluate the two models' perplexities using `sonnets.txt` as the test data. What do you get? Does anything unusual happen? Explain what and why.

Writeup Question 3.2: Evaluate the perplexities of a smoothed (`delta = 0.5`) trigram `NGramLM` trained on `shakespeare.txt` and one trained on `warpeace.txt`. Use `sonnets.txt` as the test data for both. What do you get? Which one performs better, and why do you think that's the case?

Writeup Question 3.3: Authorship identification is an important task in NLP. Can you think of a way to use language models to determine who wrote an unknown piece of text? Explain your idea and how it would work (you don't need to implement it).

4 Generation – 15 points

Let's generate some text. Import the `random` library and set `random.seed(1)` (this will keep your random number generator consistent across different runs of your code). Add a method `random_word(self, context, delta=0)` to `NGramLM` that returns a word sampled from the model's probability distribution for `context`. Your method should perform the following steps:

1. Sort `self.vocabulary` according to Python's default ordering (basically alphabetically order).

2. Generate a random number $r \in [0.0, 1.0)$ using `random.random()`. This value `r` is how you know which word to return.

3. Iterate through the words in the sorted vocabulary and compute their probabilities given `context`. These probabilities all sum to 1.0, so if we imagine a number line from 0.0 to 1.0, the space on that number line can be divided up into zones corresponding to the words. For example, if the first words are "apple" and "banana," with probabilities 0.09 and 0.57, respectively, then [0.0, 0.9) belongs to "apple" and [0.9, 0.66) belongs to "banana," and so on. Return the word whose zone contains `r`.

Once we can generate words, we can generate sentences. Write a function `random_text(model, max_length, delta=0)` that generates up to `max_length` words, using the previously generated words as context for each new word. The initial context should consist of start tokens '`<s>`', and the function should return the generated string immediately if the stop token '`</s>`' is generated.

Writeup Question 4.1: Train a trigram model on `shakespeare.txt` and generate 5 sentences with `max_length = 10`. What did you generate? Are they good (Elizabethan) English sentences? What are some problems you see with the generated sentences?

Let's try a different way of sampling. Write a method `likeliest_word(self, context, delta=0` that returns the n-gram with the highest probability for `context`. You will also need a function `likeliest_text(model, max_length, delta=0)`, which will be almost identical to `random_text()`.

Writeup Question 4.2: Train four models – one bigram, one trigram, one 4-gram, and one 5-gram – on `shakespeare.txt` and generate the likeliest sentence for each one using `max_length = 10`. What did you generate? Do you notice anything about these sentences? How do they compare to each other? How do they compare to the randomly-generated sentences?

5 Meta Questions – 3 points

Writeup Question 5.1: How long did this homework take you to complete (not counting extra credit)?

Writeup Question 5.2: Did you discuss this homework with anyone?

6 Extra Credit – 10 points

Choose **one** of the following to implement. (You are welcome to do more than one if you want, but only one will be graded, and you must indicate which one you want me to grade. There is no extra-extra credit for doing more than one.)

- Katz backoff
- Kneser-Ney smoothing
- Beam search

You can use as many new variables, functions, and methods to do this as you like. Thoroughly comment your code! If I can't figure out what it's doing, I won't give credit.

Writeup Question 6.1a: For Katz backoff and Kneser-Ney, evaluate the perplexity of a trigram model trained on `shakespeare.txt` on `sonnets.txt`, using `beta = 0.75`, and report the result in your writeup. How does it compare with the unsmoothed and Laplace-smoothed perplexities from Question 3.1? (Keep in mind that the formulae for Katz backoff and Kneser-Ney in the slides are specifically for bigrams; you will need to generalize them to handle arbitrary n-grams.)

Writeup Question 6.1b: For beam search, generate 5 sentences with a trigram model trained on `shakespeare.txt` and `max_length = 10`, using `k = 5`, and put the generated sentences in your writeup. How do they compare to the generated sentences from Questions 4.1 and 4.2?