**CS 6320.001: Natural Language Processing**
**Spring 2026**

**Homework 1 Programming Component – 50 points**
**Issued 21 Jan. 2026**
**Due 11:59pm CDT 04 Feb. 2026**

**Deliverables:** Your completed `hw1.py` file, uploaded to Gradescope.

## 0 Getting Started

Make sure you have downloaded the data for this assignment:

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

Make sure you have installed the following libraries:

- NLTK, `https://www.nltk.org/`
- NLTK corpora, `https://www.nltk.org/data.html`

## 1 A Basic N-Gram Language Model (30 points)

We will start with a very basic n-gram language model. Open the provided skeleton code `hw1.py` in your favorite text editor.

**Fill in the generator function `get_ngrams(n, text)`.** The argument `n` is an int that tells you the size of the n-grams (you can assume $n > 0$), and the argument `text` is a list of strings (words) making up a sentence. The function should do the following:

- Pad `text` with enough start tokens `'<s>'` so that you're able to make n-grams for the beginning of the sentence, plus a single end token `'</s>'`, which we will need later in Part 3.
- For each "real," non-start token, yield an n-gram tuple of the form `(word, context)`, where `word` is a string and `context` is a tuple of the $n-1$ preceding words/strings.

Next, let's work on the class `NGramLM`, which will keep track of counts from the training data. Look over the initialization method `__init__(self, n)`. The argument `n` is an int that tells you the size of the n-grams used in this `NGramLM` (as before, you can assume $n > 0$). The initialization method saves `n` as an internal variable `self.n` and initializes three other internal variables:

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

**Fill in the method `update(self, text)`.** The argument `text` is a list of strings. The function should do the following:

- Use `get_ngrams(n, text)` to get n-grams of the appropriate size for this `NGramLM`.

- For each n-gram, update the internal counts as needed. (Think about how we should handle the start and end tokens, `'<s>'` and `'</s>'`...)
  - The keys of `ngram_counts` should be tuples of the form `(word, context)` where `context` is a tuple.
  - The keys of `context_counts` should be tuples of strings.

Now we're ready to load the data and train the `NGramLM`.

**Fill in the function `load_corpus(corpus_path)`.** The argument `corpus_path` is a string giving the path to a corpus file. The function should do the following:

- Open the file at `corpus_path` and load the text.
- Tokenize the text into sentences.
  - Split the text into paragraphs. The corpus file has blank lines separating paragraphs.
  - Use the NLTK function `sent_tokenize()` to split the paragraphs into sentences.
- Use the NLTK function `word_tokenize()` to split each sentence into words.
- Return a list of all sentences in the corpus, where each sentence is a list of words.

**Fill in the function `create_ngramlm(n, corpus_path)`.** The argument `n` is an int that tells you the size of the n-grams (you can assume $n > 0$), and the argument `corpus_path` is a string giving the path to a corpus file. The function should do the following:

- Use `load_corpus` to load the data from `corpus_path`.
- Create a new `NGramLM` and use its `update()` function to add each sentence from the loaded corpus.
- Return the trained `NGramLM`.

Now that we can train a model, we need to be able to use it to predict n-gram and sentence probabilities. Let's go back to the `NGramLM` class.

**Fill in the method `get_ngram_prob(self, word, context, delta=0)`.** The argument `word` is a string, and the argument `context` is a tuple of strings; the argument `delta` is used in Part 2. The method should do the following:

- Use the counts stored in the internal variables to calculate and return $p_{MLE}(\text{word}|\text{context})$.
- If `context` is previously unseen (ie. not in the training data), return $1/|V|$, where $V$ is the model's vocabulary. (Why do we do this? Food for thought...)

To predict the probability of a sentence, we multiply together its individual n-gram probabilities. This can be a very small number, so to avoid underflow, we will report the sentence's log probability instead.

**Fill in the method `get_sent_log_prob(self, sent, delta=0)`.** The argument `sent` is a list of strings; the argument `delta` is used in Part 2. The method should do the following:

- Use `get_ngrams()` to get the n-grams in `sent`.
- For each n-gram, use `get_ngram_prob()` to get the n-gram probability and take the

base-2 logarithm using `math.log()`.
- Return the sum of the n-gram log probabilities.

Now you can try running `main()` to train a trigram NGramLM on `warpeace.txt` (make sure it's in the same directory as your code!) and use it to predict the probabilities of these two sentences:
- God has given it to me, let him who touches it beware!
- Where is the prince, my Dauphin?

Does anything unusual happen?

## 2 Smoothing (5 points)

We need to add support for out-of-vocabulary words. Let's implement Laplace smoothing.

**Update** `NGramLM.get_ngram_prob_()` to support Laplace-smoothed probabilities using the `delta` argument. The new version of the function should do the following:
- Check if `delta` is 0. If so, it should return the same probability as it would have before.
- If `delta` is not 0, apply Laplace smoothing and return the smoothed probability.

Now try running `main()` again with non-zero `delta`!

## 3 Evaluation (15 points)

**Fill in the method** `NGramLM.get_perplexity(self, corpus)`. The argument `corpus` is a list of lists of strings, representing sentences in a test corpus. The method should do the following:
- Use `NGramLM.get_sent_log_prob()` to get corpus-level log probability.
- Divide by the total number of tokens in the corpus to get the average log probability.
- Use `math.pow()` to calculate the perplexity.

Finally, let's generate some text!

**Fill in the method** `NGramLM.generate_random_word(self, context, delta=0)`. The argument `context` is a tuple of strings, and the argument `delta` is an int. The method should do the following:
- Sort `self.vocabulary` according to Python's default ordering (basically alphabetically order).
- Generate a random number $r \in [0.0, 1.0)$ using `random.random()`. This value $r$ is how you know which word to return.
- Iterate through the words in the sorted vocabulary and use `NGramLM.get_ngram_prob()` to get their probabilities given `context`. Make sure to pass `delta`.
- 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 in the vocabulary. 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 also generate sentences.

**Fill in the method** `NGramLM.generate_random_text(self, max_length, delta=0).` The argument `max_length` is an int representing the maximum number of words to generate. The method should do the following:

- Generate the first word using `NGramLM.generate_random_word()` with a context consisting of start tokens `'<s>'`.
- Continue generating using the previously generated words as context for each new word.
- Stop generating when either `max_length` is reached, or if the stop token `'</s>'` is generated.
- Return the generated sentence as a single string.

We are all set! You can modify `main()` and use the provided data files `warpeace.txt` and `shakespeare.txt` to test your code and make sure it gives reasonable outputs.