Northeastern University, Khoury College of Computer Science

CS 6120 — Assignment 4

Due: February 6, 2025 (100 points)

YOUR NAME + LDAP

In this assignment, we will implement an auto-correct algorithm that can be used (in real-time) to determine the most logical correct word substitute for a misspelled word. We will be using the Twitter corpus, which has nearly 50k lines of text to parse through. The dataset is at:

`https://course.ccs.neu.edu/cs6120s25/data/twitter/en_US.twitter.txt`

We will be preprocessing the sentences, building bigram and trigram distributions, and subsequently autocompleting words.

Question 1: Preprocessing and Vocabulary

In this question, you will prepare the data for training and inference.

Q 1.1: Tokenize Data

Language Models rely on *tokens*, where algorithms (like ChatGPT) predict the next token based on what appeared before it. There are several tools that are highly optimized and industry standard for removing punctuation and delimiting words. In our case, we will build a language model for tweets by tokenizing each tweet with the NLTK library (i.e., `nltk.word_tokenize(tweet)`).

We will need to be practical by choosing only those words that occur frequently enough that they are worth modeling. That is, we will create our vocabulary.
You won't use all the tokens (words) appearing in the data for training. Instead, you will use the more frequently used words.
You will focus on the words that appear at least N times in the data. First count how many times each word appears in the data. You will need a double for-loop, one for sentences and the other for tokens within a sentence

Q 1.2: Handling OOV

If your model is performing autocomplete, but encounters a word that it never saw during training, it won't have an input word to help it determine the next word to suggest. The model will not be able to predict the next word because there are no counts for the current word.

This 'new' word is called an 'unknown word', or out of vocabulary (OOV) words. The percentage of unknown words in the test set is called the OOV rate. To handle unknown words during prediction, use a special token to represent all unknown words 'unk'.

Modify the training data so that it has some 'unknown' words to train on. Words to convert into "unknown" words are those that do not occur very frequently in the training set. Create a list of the most frequent words in the training set, called the closed vocabulary . Convert all the other words that are not part of the closed vocabulary to the token 'unk'.

Question 2: N-Gram Counting

You will find that the easiest estimate of word distributions is simply to count the frequency of words relative to the corpus. In this section, you will implement a function that returns a dictionary of n-gram counts, given the tokens and data extracted in Q1.

```
def count_n_grams(data, n, start_token='<s>', end_token = '<e>'):
"""
Count all n-grams in the data

Args:
    data: List of lists of words
    n: number of words in a sequence

Returns:
    A dictionary that maps a tuple of n-words to its frequency
"""

# Initialize dictionary of n-grams and their counts
n_grams = {}
<YOUR-CODE-HERE>
return n_grams
```

Hint: Test your data out on very small sets to understand if it's working. The following code should help you with debugging:

```
def make_probability_matrix(n_plus1_gram_counts, vocabulary, k):
    count_matrix = make_count_matrix(n_plus1_gram_counts, unique_words)
    count_matrix += k
    prob_matrix = count_matrix.div(count_matrix.sum(axis=1), axis=0)
    return prob_matrix

sentences = [['i', 'like', 'a', 'cat'],
                ['this', 'dog', 'is', 'like', 'a', 'cat']]
```

```
unique_words = list(set(sentences[0] + sentences[1]))
bigram_counts = count_n_grams(sentences, 2)
print("bigram probabilities")
display(make_probability_matrix(bigram_counts, unique_words, k=1))
```

The above code should give you the following matrix:

bigram probabilities

	cat	i	dog	is	like	a	this	<e>	<unk>
(cat,)	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.272727	0.090909
(is,)	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000
(dog,)	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000	0.100000
(this,)	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000	0.100000	0.100000
(a,)	0.272727	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909
(i,)	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000
(<s>,)	0.090909	0.181818	0.090909	0.090909	0.090909	0.090909	0.181818	0.090909	0.090909
(like,)	0.090909	0.090909	0.090909	0.090909	0.090909	0.272727	0.090909	0.090909	0.090909

Q 2.1: Estimate the Probabilities

Estimate the probability of a word given the prior 'n' words using the n-gram counts.

$$\hat{P}(w_t|w_{t-n}\dots w_{t-1}) = \frac{C(w_{t-n}\dots w_{t-1}, w_t) + k}{C(w_{t-n}\dots w_{t-1}) + k|V|} \tag{2.1}$$

Note the introduction of the parameter $k = 1.0$. This is a *smoothing* parameter for when there are no occurrences of a sequence of words. Write the function that estimates the probabilities of the next possible word given a prior n-gram. That is, complete the code in `estimate_probabilities` with the following function signature:

```
def estimate_probabilities(previous_n_gram, n_gram_counts,
                           n_plus1_gram_counts, vocabulary_size,
                           k=1.0):
    """
    Estimate the probabilities of a next word using the n-gram counts
    with k-smoothing

    Args:
        word: next word
        previous_n_gram: A sequence of words of length n
        n_gram_counts: Dictionary of counts of n-grams
        n_plus1_gram_counts: Dictionary of counts of (n+1)-grams
        vocabulary_size: number of words in the vocabulary
        k: positive constant, smoothing parameter

    Returns:
        A dictionary mapping from next words to probability
    """
```

```
    probabilities = {}
    <YOUR-CODE-HERE>
    return probabilities
```

Question 3: Infer N-Grams

(Hint: With the preprocessed word frequencies, you should be able to make a rudimentary classifier that can take a single passage and determine which author produced it.)

Submit your code with the following function signature.

```
def write_in_style_bigram(passage, style_files):
  """
  Takes a passage in, matches it with a style, given a list of
  filenames, and predicts the next word that will appear
  using a bigram model.

  Args:
    passage: A string that contains a passage
    style_file: a list of filenames that will be used to determine the style

  Returns:
    a single word in the form of a string
  """
  % <YOUR-CODE_HERE>
  return []
```