Northeastern University, Khoury College of Computer Science

CS 6120 — Assignment 4

Due: March 9, 2026(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/cs6120s26/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. Your training data will consist of tweets in `en_US.twitter.txt`. because each tweet is delineated by a line break, `\n`, split the data into sentences using the `\n` as the delimiter, where each training sample is a line in the data. You will be filling out the sub-functions calls (`read_and_tokenize_sentences`, `get_words_with_nplus_frequencies` and `replace_oov_words_by_unk`) in the following `preprocess_data` method.

```
def preprocess_data(filename, count_threshold, special_tokens,
                    sample_delimiter='\n', split_ratio=0.8):
    """
    Ungraded: You do not need to change this function.

    Preprocess data, i.e.,
        - Find tokens that appear at least N times in the training data.
        - Replace tokens that appear less than N times by "<unk>" .
    Args:
        count_threshold: Words whose count is less than this are
                         treated as unknown.

    Returns:
        training_data = list of lists denoting tokenized sentence. This looks like
                        the following:
```

```
                        [ ["this", "<unk>", "example"],
                          ["another", "sentence", "<unk>", "right"],
                         ...
                        ]
        test_data = Same format as above.
        vocabulary = list of vocabulary words. This looks like the following:

                        ["vocab-word-1", "vocab-word-2", etc.]
    """

    # Create sentences and tokenize the data to create a list of strings.
    tokenized_data = read_and_tokenize_sentences(filename, sample_delimiter)

    # Create the training / test splits
    train_size = int(len(tokenized_data) * split_ratio)
    train_data = tokenized_data[0:train_size]
    test_data = tokenized_data[train_size:]

    # Get the closed vocabulary using the train data
    vocabulary = get_words_with_nplus_frequency(train_data, count_threshold)

    # For the train data, replace less common words with unknown token
    train_data_replaced = replace_oov_words_by_unk(
        train_data, vocabulary, unknown_token = special_tokens.unknown_token)

    # For the test data, replace less common words with "<unk>"
    test_data_replaced = replace_oov_words_by_unk(
        test_data, vocabulary, unknown_token = special_tokens.unknown_token)

    return train_data_replaced, test_data_replaced, vocabulary
```

Note: in the code that you are to implement, you will notice an argument called `special_tokens`. We can create such an object with:

```
class SpecialTokens:
  def __init__(self, start_token = "<s>", end_token = "<e>", unknown_token = "<unk>"):
    self.start_token = start_token
    self.end_token = end_token
    self.unknown_token = unknown_token

special_tokens = SpecialTokens()
```

Q 1.1: Read and Tokenize Data

Language Models rely on *tokens*, where algorithms (like ChatGPT) predict the next token based on what tokens appeared before it. There are several tools that are highly optimized and industry standard for removing punctuation and delimiting words. In this homework, build the language model for tweets by tokenizing each tweet with the NLTK library (i.e., `nltk.word_tokenize(tweet)`). While each token is typically delimited by spaces, punctuation, and numbers, we define a *data sample* as a *tweet* delimited by a newline.

You will notice that there can be an inordinate number of words in the English lexicon, but many don't often get used, and therefore we should not be recommending them as the next predicted word. That is, we should be practical by choosing words that occur frequently enough that they are worth modeling and adding to our *vocabulary*. Since the corpus is relatively small, let us use a minimum frequency of *2*.

We will be using words as tokens, but we must also add special tokens that are more for data processing and not a part of the English *language*. These denote specific contextual information about the sequence of words. We will use a start token (i.e., `<s>`) and an end token (i.e., `<e>`) for words that occur at the beginning or end of the sentence. These are the additional special tokens (along with the `<unk>` token.)

Q 1.2: Handling OOV

If your model 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

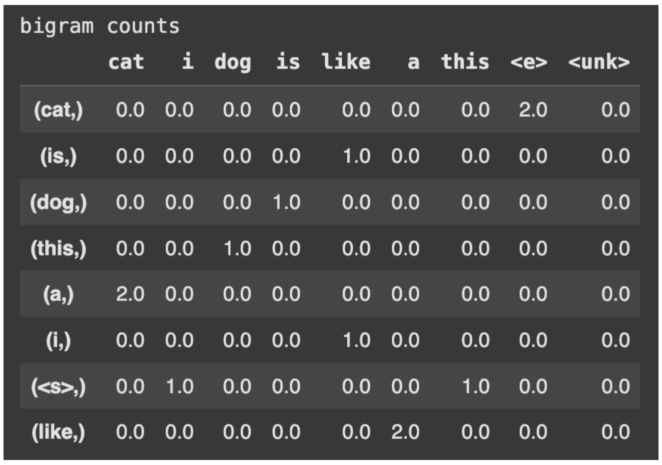
You will find that the easiest estimate of word distributions is simply to count the frequency of words relative to the corpus. In this section, you will implement a function that returns a dictionary of n-gram counts, given the tokens and data extracted in Q1.

```
def count_n_grams(data, n, special_tokens):
    """
    Count all n-grams in the data

    Args:
        data: List of lists of words
        n: Number of words in a sequence
        special_tokens: A structure that contains:
          - start_token = "<s>"
          - end_token = "<e>"
          - unknown_token = "<unk>"

    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
```

To visualize how one transitions from n-grams to $n+1$-grams, it is sometimes easier and more intuitive to print out a probability matrix, which can be constructed from counts. We have borrowed a function called `make_count_matrix` on the the course website, which you can leverage. Test your functions out with a probability matrix. The vertical axis of the matrix will be the current n-grams and the horizontal axis of the matrix will be a new word that is added to make the $n+1$-gram. Test using very small sets to understand if it is working. Once you've downloaded utils.py, the following code will help you with debugging:

```
!wget -nc https://course.ccs.neu.edu/cs6120s26/data/twitter/utils.py

import utils

def make_probability_matrix(n_plus1_gram_counts, vocabulary, k):
    count_matrix = utils.make_count_matrix(n_plus1_gram_counts, unique_words)
    count_matrix += k
    prob_matrix = count_matrix.div(
           count_matrix.sum(axis=1) + k*len(vocabulary), 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, SpecialTokens())

print("bigram counts")
display(utils.make_count_matrix(bigram_counts, unique_words))

print("bigram probabilities")
display(make_probability_matrix(bigram_counts, unique_words, k=1))
```

Your count matrix should look like this:



bigram counts

	cat	i	dog	is	like	a	this	<e>	<unk>
(cat,)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0
(is,)	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
(dog,)	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
(this,)	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
(a,)	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(i,)	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
(<s>,)	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
(like,)	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0

Question 3: Estimate the Probabilities

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

$$\hat{P}(w_t|w_{t-n}\dots w_{t-1}) = \frac{C(w_{t-n}\dots w_{t-1}, w_t) + k}{C(w_{t-n}\dots w_{t-1}) + k|V|} \tag{3.1}$$

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. We will incorporate k into our model object.

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(context_tokens, ngram_model):
    """
    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
        ngram_model: a structure that contains:
            - 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
            - k: positive constant, smoothing parameter

    Returns:
        A dictionary mapping from next words to probability
    """
    probabilities = {}
    <YOUR-CODE-HERE>
    return probabilities
```

In the above, we pass a model object structured called `ngram_model`. Include all the parameters into the object, including k. You can save the results from Q2 into the following model class.

```
class NGramModel:
  def __init__(self, n_gram_counts, n_plus1_gram_counts, vocab_size, k = 1.0):

    # dictionary of n grams counts
    self.n_gram_counts = n_gram_counts
    # dictionary of n+1 grams counts
    self.n_plus1_gram_counts = n_plus1_gram_counts
    # number of words
    self.vocabulary_size = vocab_size
    # positive constant, smoothing parameter
    self.k: k
```

Question 4: Infer N-Grams

Now to put it all together. Write a function that uses the models that you've created, and predicts the next word (the word with the maximum probability).

```
def predict_next_word(sentence_beginning, model):
  '''
  Argument:
    sentence_beginning: a string
    model: an NGramModel object

  Returns:
    a string with the next word that his most likely to appear after the
    sentence_beginning input using the define model
  '''
  return next_word
```

We will be using your function to find the next word that might appear from the beginning of a few sentences. You can simulate this testing process by predicting the next word from our understanding of trigrams and bigrams, which we can define via the `n_gram_model` model object as below.

```
train_data, test_data = preprocess_data("en_US.twitter.txt", 2)

trigram_model = NGramModel(
    count_n_grams(data, 2, SpecialTokens()),
    count_n_grams(data, 3, SpecialTokens()),
    vocabulary,
    k = 1.0
)

predict_next_word("The next word is", n_gram_model)
```

Question 5: Extra Credit: Stylistic N-Grams

Process the three datasets:

- Earnest Hemingway(https://course.ccs.neu.edu/cs6120s26/data/hemingway/hemingway-edit.txt)
- William Shakespeare: (https://www.gutenberg.org/cache/epub/100/pg100.txt)
- Emily Dickinson (https://www.gutenberg.org/cache/epub/12242/pg12242.txt)

Using a partial sentence as the context, provide the next most probable word. Identify the most likely writing style between the three authors, and subsequently use the n-gram modeling approaches to provide the ten words in the most likely style that have the highest probability of occurring next. To help with autograding, implement them in the following Class, where we will be initializing an object of type `StyleGram` and subsequently running the function `write_in_style_ngram`

```
class StyleGram:

    def __init__(self, style_files):
        """
        We will only be passing style_files in. All your processing and
        training should be done by the time this function retunrs.
        """
        self.style_files = style_files
        <YOUR-CODE-HERE>
        return

    def write_in_style_ngram(self, passage):
    """
    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 to be used to determine the style
    Returns:
         single word <string>
         probability associated with the word <float>
         index of "style" it originated from (e.g., 0 for 1st file) <int8>
         probability associated with the style <float>
    """
      % <YOUR-CODE_HERE>
      return word, probability_word, style_file, probability_style
```

Hint: With the preprocessing steps, you should be able to make a rudimentary classifier similar to what you've built in homework 2, that can take a single passage and determine which author produced it.

Submission Instructions

There are several artifacts that you will need to upload to Gradescope. These include:

- **Required** Your code with all of the functions above in a file called `assignment4.py`. The related functions and classes that we'll be expecting are:

  Q1 : Preprocessing

  - `preprocess_data`
  - `read_and_tokenize_sentences`
  - `get_words_with_nplus_frequency` `replace_oov_words_by_unk`

  Q2 : `count_n_grams`

  Q3 : `estimate_probabilities`

  Q4 : `predict_next_word`

  EC - 10pts The Class `StyleGram`

- **Optional** - Any descriptions of your algorithm, considerations, and specifics in a `assignment4.pdf`. If you attacked the extra credit problem, this is most useful for us to understand.