Lab 2: Bigram Language Model

Introduction

In this practice, we delve into the world of Natural Language Processing (NLP) by undertaking a creative task: generating new names from an existing dataset. This exercise serves as an introduction to the foundational concepts and methodologies of NLP. By processing and analyzing the dataset `nombres_raw.txt`, we explore how textual data can be transformed into a format that algorithms can interpret and utilize. This not only reinforces our understanding of data preprocessing, but also demonstrates the practical application of NLP techniques in generating human-like text, a fundamental step towards building more advanced language models.

Learning Outcomes

By the end of this lab, students will be able to:

- Convert textual data into a numerical format that can be interpreted by machine learning models.
- Create and utilize mappings between characters and indices to facilitate the processing of language data.
- Generate new text sequences by learning and applying probability distributions of characters from a given dataset.

Preparation

Ensure you have Python and necessary libraries (like torch and matplotlib) installed. Familiarize yourself with basic Python programming and PyTorch basics.

Deliverables

For the successful completion of this laboratory session, you are required to submit the following files:

1. `data_processing.py` - This Python file contains all the functions related to data loading, preprocessing, and bigram visualization.
2. `bigram_model.py` - This Python file includes the functions for bigram counting, probability calculations, and other functionalities relevant to the bigram model.

These files should be compressed into a single `.zip` archive and named with your student identifier. The `.zip` file must then be uploaded to the designated Moodle assignment task before the deadline.

Exercises

Exercise 1: A journey begins

In this exercise, you will create a function, `load_and_preprocess_data`, that prepares the raw text data for further processing. The function will read from a file and transform each line of text into a series of bigrams, i.e., into pairs of adjacent characters, including special tokens to denote the beginning and end of a word.

The process to complete this function is:

1. Open the file, read each line and split the line into words.
2. For each word
   (a) Convert it to lowercase.
   (b) Prepend the start token and append the end token.
   (c) Generate bigrams by pairing each character with its subsequent character.
3. Return the list of bigrams.

Hint #1:

Assume the start and end tokens are single characters. However, many times these tokens are more than one character, such as '<S>' and '<E>'.

Exercise 2: From character to index...

After processing the data into bigrams, the next step is to create a mapping of each character to a unique index. This mapping is vital for numerical representation of textual data. Implement the `char_to_index` function that receives the alphabet as a string and the start and end tokens. It should return a dictionary where each character, including the start and end tokens, is assigned a unique index. The start token is expected to have index 0 and the end token is expected to have the last index.

Exercise 3: ... and vice-versa

Complementary to the previous function, `index_to_char` converts indices back to characters. This reverse mapping is used for interpreting the numerical data as text, which would be useful when predicting new characters. Implement the `index_to_char` function that takes the dictionary created by `char_to_index` and returns a new dictionary mapping each index back to its corresponding character.

Hint #2:

List (or dict) comprehension might be specially useful in this case.

Exercise 4: Welcome to Sesame Street

With the mappings in place, we move on to quantifying the relationships between characters. Implement the function `count_bigrams` which will count the frequency of each bigram in our dataset. This function takes the list of bigrams and the dictionary created with the `char_to_index` function as inputs and returns a 2D tensor representing the bigram counts. This tensor is foundational for understanding the common patterns in the dataset and will be crucial for the subsequent probability modeling.

Hint #3:

A special function called `plot_bigram_counts` has been prepared for you to visualize the bigram counts. Check it out!

Exercise 5: From Counts to Probabilities

Now that we have a count of how frequently each bigram appears, the next step is to convert these counts into probabilities. Implement the function `bigrams_count_to_probabilities` which takes the bigram counts tensor and transforms it into a probability matrix. Each row of this matrix represents how probable is one character to follow a given character, setting the stage for predictive modeling.

Exercise 6: Nothing is Impossible

A crucial aspect in the transformation of bigram counts to probabilities is the concept of smoothing, specifically through the addition of a smooth factor. In real-world datasets, some bigrams may not appear, but assigning a probability of zero would be misleading (and potentially disastrous). This is where smoothing comes in.

Modify the `bigrams_count_to_probabilities` function to include a smoothing parameter. This addition ensures that even unseen bigrams have a small, non-zero probability, reflecting the real-world scenario where any bigram could theoretically occur, albeit with a low probability. This concept is fundamental in probabilistic models and will aid in creating a more robust NLP application.

Hint #4:

The `smooth_factor` shall be added to all bigram counts in order for the probability distribution to sum 1.

Food for thought #1:

You have obtained the probabilities based on the bigram counts directly from the corpus. How is this related to a Machine Learning model that predicts the probability of a character following your current character?

Exercise 7: The Most Important Step is Always the Next One

The next step in your journey involves the `sample_next_character` function. This function is pivotal in constructing names character by character, based on the probabilities you have calculated from the

corpus. In this task, you'll use a probabilistic approach to determine each subsequent character in a name. The function takes the index of the current character and a probability distribution that represents the likelihood of each possible next character. Your challenge is to sample from this distribution to decide the next character in the sequence.

Hint #5:

Check the torch.Multinomial function to sample characters based on a probability distribution.

Exercise 8: Elon Musk uses this

Finally, you will implement the `generate_name` function, the culmination of all your prior work. This function brings together the components you've developed, using the probabilities and sampling techniques to create entirely new names.

In order to complete this function, your first character shall be the start token, and you shall obtain new characters using the `sample_next_character` function. As we might want to limit the length of the generated names, you shall end generating characters once you obtain the end token or the name length reach the limit.

Exercise 9: You are so likeable!

The `calculate_log_likelihood` function plays a pivotal role in understanding how likely a name is, based on its constituent bigrams and the probabilities obtained from your corpus. In this task, your goal is to implement a function that computes the log likelihood of a given word. This function might be later used to calculate how likely is a name to be produced by your corpus based on its bigrams.

This function will require you to iterate through each bigram in the word and sum the log probabilities of each bigram, obtained from the bigram probabilities matrix. This concept is essential for grasping the probabilistic nature of language and forms the basis for many advanced NLP models. Remember, the log-likelihood helps in dealing with very small probability values, common in language processing tasks.

Hint #6:

The log-likelihood of a bigram ab can be calculated as:

$$L(ab) = log_e(P(ab))$$

Exercise 8: You are so likeable (on average)!

The task of implementing the `calculate_neg_mean_log_likelihood` function takes you a step further in evaluating the effectiveness of your NLP model. In this section, you will calculate the mean log likelihood across a set of words, providing a broader assessment of the model's performance on a given dataset. This metric is crucial for understanding the overall behavior of your model on real-world data.

In this case, it is expected that you use the function developed in the previous exercise, iterating over each word. Once you have the log likelihood of all the words, calculate the mean and returned it multiplied by -1.

Food for thought #2:

Why do we want the *negative* log likelihood? To maximize the likelihood

Test your work!

After implementing each function, it's crucial to validate your work. We have prepared test functions for each required function. These tests will help ensure that your implementations meet the specified requirements. If a test function returns `None`, it indicates that the corresponding function might and the test is skipped. It is essential to pass all tests to ensure the functionality and accuracy of your code.

Additionally, a `main.py` file is provided. This file runs the necessary functions to generate names based on your implementations. After completing all exercises, run `main.py` to see your NLP model in action.