iNLP - Assignment1

Mohit Sharma - 2022202022

1. Tokenizer:

A TextTokenizer class has been developed to tokenize input text effectively. Before the actual tokenization process, the TextTokenizer class incorporates several preprocessing steps to handle different types of data and simplify the text.

Preprocessing Steps

1. **Emails**: Email addresses are replaced with the token '<MAILID>'.
   Example: 'user@example.com' becomes '<MAILID>'..

2. **URLs**: Web addresses are replaced with the token '<URL>'.
   Example: 'https://www.example.com' becomes '<URL>'.

3. **Mentions**: User mentions (starting with '@') are replaced with the token '<MENTION>'.
   Example: '@username' becomes '<MENTION>'.

4. **Hashtags**: Hashtags (starting with '') are replaced with the token '<HASHTAG>'.
   Example: 'topic' becomes '<HASHTAG>'.

5. **Numbers**: Numeric values are replaced with the token '<NUM>'.
   Example: '123' becomes '<NUM>'.

Tokenization Process:

The text is split into sentences using a custom sentence-splitting regex. Words are extracted using a regex pattern for alphanumeric characters.

Example: 'This is a sample.' becomes '['<sos>', 'This', 'is', 'a', 'sample', '<eos>']'.

2. N-Gram Construction:

Finding common groups of words from a text or paragraph called N-grams. An N-gram is a set of N words that appear together. The goal is to see which word combinations are frequent. To perform this first breaks text into sentences then Counts how often groups of N words appear in each sentence.

3. Smoothing and Interpolation:

1. **Good-Turing Smoothing** : Good-Turing Smoothing is a method to guess how likely different groups of words (N-grams) are, especially when some groups don't show up much. It helps us make better guesses about the probability of less common N-grams.

   How it Works:

   1. Count of Counts: Figuring out how often each count of an N-gram appears in the text. Which is "count of counts."

   2. Adjusted Counts: I use a formula to adjust the counts of N-grams. This formula considers the count of the N-gram and looks at nearby N-grams to get a better idea.

$$p_r = \frac{(r+1)\, S(N_{r+1})}{N\, S(N_r)} .$$

   3. Smoothing Function: I have used a special function that smoothens the counts and helps us get more realistic numbers.

$$Z_r = \frac{N_r}{\frac{1}{2}(t-q)} ,$$

   and where q, r, and t are three consecutive subscripts with non-zero counts N_q, N_r, N_t . For the special case when r is 1, take q to be 0. In the

$$log(N_r) = a + b\ log(r)$$

   4. Probability Estimation: With the adjusted counts, I calculate the probability of each N-gram. This helps us understand how likely an N-gram is in the text.

2. **Linear Interpolation Smoothing:** Linear Interpolation Smoothing is like making a good guess about how likely a group of three words (trigram) is in a sentence. It does this by combining the chances of each word by itself, two words together, and all three words together.

   How it Works

   1. Weights(Lambda's): It gives different importance to the chances of each word group. Imagine adjusting how much you trust the chances of one word, two words, or all three words.

   2. Updating Weights: If I am learning from a set of sentences, I adjust these weights based on how often I see different three-word groups. Like this

```
set λ1 = λ2 = λ3 = 0
foreach trigram t1, t2, t3 with f(t1, t2, t3) > 0
    depending on the maximum of the following three values:
        case (f(t1,t2,t3)−1)/(f(t1,t2)−1): increment λ3 by f(t1, t2, t3)
        case (f(t2,t3)−1)/(f(t2)−1): increment λ2 by f(t1, t2, t3)
        case (f(t3)−1)/(N−1): increment λ1 by f(t1, t2, t3)
    end
end
normalize λ1, λ2, λ3
```

3. Probability Guessing: For each three-word group, it calculates how likely it is by looking at the chance of each word by itself, two words together, and all three words together. It combines these chances using the weights.

$$\hat{P}(w_n|w_{n-1}w_{n-2}) = \lambda_1 P(w_n|w_{n-1}w_{n-2}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n)$$

$$\sum_i \lambda_i = 1$$

4. Example: If you are trying to guess how likely "happy new year" is, I'd look at how likely "happy" is alone, "happy new" together, and "happy new year" as a whole.

4. Generation :

The Generator class is created for predicting the next word in a given sentence by using the language model's predictions. It generates a list of candidate words along with their associated probability scores.

How It Works: The Generator takes an input sentence, typically the context of a conversation or text. It interacts with the underlying language model to obtain predictions for the next word. Based on the language model's predictions, it generates a list of potential candidate words. Each candidate word is assigned a probability score, indicating the likelihood of it being the next word in the sentence.

Output: The final output is a dictionary where each candidate word is paired with its probability score.

5. Language Model:

The LanguageModel class is a basic template for overall experiments of this assignment. It handles training, saving, loading, and evaluation etc. of language models. Additionally, it provides functionality to calculate probabilities and perplexity.

Key Features:

- **Training**: The LanguageModel is responsible for training the language model based on a given corpus, which involves building N-grams, applying smoothing techniques(on of two given), and calculating probabilities.
- **Saving and Loading:** It enables the persistence of trained models by saving them to files in json format and later loading them when needed.
- **Evaluation**: The class facilitates the evaluation of language models by calculating perplexity scores, a measure of how well the model predicts the given text and also to store perplexities for each sentence in the test file .
- **Probability Calculation:** The LanguageModel can calculate the probability of a given sentence or sequence of words based on the trained model.
- **Perplexity Calculation:** It provides a method to calculate perplexity, a metric indicating how well the language model predicts a set of words.

6. Perplexity Scores:

1) **On "Pride and Prejudice" Corpus:**
    i. LM 1: Good-Turing Smoothing

        Train Set Avg Perplexity: **699.2316450778947**
        Test Set Avg Perplexity: **702.0698105047592**

    ii. LM 2: Linear Interpolation

        Train Set Avg Perplexity: **205.70975724719912**
        Test Set Avg Perplexity: **206.06493712222286**

2) **On "Ulysses" Corpus:**
    i. LM 3: Good-Turing Smoothing

        Train Set Avg Perplexity: **1606.4216505120885**
        Test Set Avg Perplexity: **1660.8031429880866**

    ii. LM 4: Linear Interpolation

        Train Set Avg Perplexity: **370.57673173386416**
        Test Set Avg Perplexity: **373.9713343735246**

7. Analysis of Scores:

The Good-Turing Smoothing model shows higher perplexity scores, suggesting some difficulty in capturing the complexity of language on corpuses However, it demonstrates consistency in performance across both the training and test sets. By the way, for ‘On "Ulysses" Corpus’ it gives better scores as compared to the previous one, but not for testing sets.

On the other hand, the Linear Interpolation model performs exceptionally well on corpuses boasting significantly lower perplexity scores on both the training and test sets. Its predictions are far better than Good turing smoothing in these cases.