NLP Assignment 1

Group Members : Bhavinkumar R. , Aastha K.

Task 1

Here we will be discussing the choice of Tokenizer from NLTK and the difference between outputs from GPT2 and NLTK tokenizer.

About choice of Tokenizer from NLTK :

After researching the possible tokenizers available from NLTK, we decided to use the WordPunct Tokenizer as this one tokenizes Punctations and Numerical objects separately and could possibly add more contextual information to the encoded output related to the original text.

Differences between the tokenizer outputs :

1. GPT Tokenizer output tokens differ from the NLTK tokenizer output in the way that GPT Tokenizer returns special types of tokens consisting of 'Ġ', 'Ċ' characters. Upon researching, we realized that these tokens have special meaning. A. 'Ċ' : represents line break / separator tokens which gives additional contextual information regarding the encoded text. This feature is not available in the word Punt tokenizer from NLTK. B. 'Ġ' : In order to encode spaces between words, the GPT Tokenizer uses this character to represent that the word that follows this character had white spaces before them in the original text. Again, this can be useful to encode contextual information from the original text. NLTK doesn't have any such functionality.
2. About NLTK WordPunct Tokenizer : This tokenizer goes beyond the regular str.split(" ") type tokenization by adding tokens for punctuations which adds more information about original text in the encoding by signifying where punctuations lie.

Task 2

Implementing our version of N-Gram models.

Describe N gram models and how they function

As per our understanding, N Gram models are probabilistic Markov models that predict the next item in the sequence as a function of the previous N-1 items. Meaning, for sentence completion, it predicts the Nth word based on the previous N-1 set of words in the sentence.

Give the implementation of how we did it.

To make an N gram Model, we went through the following steps :

1. Read Data : Here we created a function that takes file name as input and outputs the complete data as a single string. This is to remove duplication of code when reading multiple files.
2. Cleaning Data : In order to clean the data, we did the following : A. Remove the Special Characters B. Remove the New Line characters C. Remove Extra Spaces D. Convert all data into Lowercase

3. Tokenize Data : Here since we have already completed the first task, we reuse the tokeniezr functions to generate tokens from NLTK as well GPT2Fast
4. N-Gram Calculation : For each value of N, calculate sets of N and N-1 Grams. Here we iterate through the complete Tokenized Dataset and use a **Sliding Window approach** to store tokens from [i:i+N] as an Ngram and [i:i+N-1] as N-1 Grams.
5. Probaility Calculations : Here our task is that given the set of N Grams generated, we have to calculate the probability of this sequence of N words occuring in exact sequence in the dataset. To do this, we simply counted the frequency of each N gram and we divide it by the total len of N Gram list.

   P(Given NGram) = Freq(Given NGram) / Total Number of NGrams
6. Evaluation : For evaluating the Model based on Test Data, we basically followed this formula

   P(ith Word | Previous N-1 Words) = Count (ith Word Following Previous N-1 Words) / Count (Previous N-1 Words)

Results from 2 different Tokenizers for the 1,2,3,7 gram models

```
WordPunctTokenizer and 1-grams: 795.855313548824
GPT2FastTokenizer and 1-grams: 696.0489735021165

WordPunctTokenizer and 2-grams: 3659.6679274231783
GPT2FastTokenizer and 2-grams: 892.7512810352328

WordPunctTokenizer and 3-grams: 3219629.016859747
GPT2FastTokenizer and 3-grams: 263882.11846897326

WordPunctTokenizer and 7-grams: 9003668643.138819
GPT2FastTokenizer and 7-grams: 5992289762.698238
```

Analysis on Results :

- As the Number of Tokens (N) in sequence increases, the probability of finding the same exact sequence from Test in Training decreases exponentially and thus we see that results are really poor for any N greater than 3.
- Better (Lower) Perplexity with GPT2 Tokenizer indicates that the tokenizer is objectively better at encoding contextual information than simple NLTK base tokenizers.

Task 3

Here the task was to resolve unseen data's probability calculation.

We implemented a simple (**Bonus**) apporach where if NGram is in probability Dictionary (Which Maps NGrams to its Probability from Training Data), we let it calculate the Perplexity based on regular formula but, if we don't have its probabilities, then we append (1/ Total Number of N Grams). The resultant perplexities are as follows.

Code Changes :

```
    logProb = 0
    for ngram in nGrams:
        if ngram in probabilities:
            logProb += m.log(probabilities[ngram])
        else:
            logProb += m.log(1/len(nGrams))
    perplexity = m.exp(-logProb / len(nGrams))
```

```
WordPunctTokenizer and 1-grams: 795.855313548824
GPT2FastTokenizer and 1-grams: 696.0489735021165

WordPunctTokenizer and 2-grams: 432.1985037476832
GPT2FastTokenizer and 2-grams: 193.21067155713155

WordPunctTokenizer and 3-grams: 5304.665673831564
GPT2FastTokenizer and 3-grams: 1660.5139160157796

WordPunctTokenizer and 7-grams: 218315.560249373
GPT2FastTokenizer and 7-grams: 210047.1558563303
```

Task 3A

The Task here was to add Laplace Smoothing to the Probability Calculation.

We modified the Probability Calculation as Follows :

P (Wi:Wi-N+1) = (Count (Wi-N+1,...Wi) + 1)/ (Count(Wi-N+1,...Wi-1) + Number of unique words)

```
    nGramCounts = Counter(ngrams)
    probabilities = {}

    if (n == 1):
        total = sum(nGramCounts.values())
        for ngram in nGramCounts:
            probabilities[ngram] = nGramCounts[ngram] / total
    else:
        n1Grams = [' '.join(ngram.split()[:-1] ) for ngram in ngrams]
        n1GramCounts = Counter(n1Grams)

        # All the unique words encountered across N Grams
        unique_words = set(word for ngram in nGramCounts for word in
ngram)

        # taken Len to get the number of unique Words
```

```
        v = len(unique_words)

        for ngram in nGramCounts:
            n1gram = ' '.join(ngram.split()[:-1] )
            # Adding V to the Denominator and 1 to Numerator
            probabilities[ngram] =
            (1 + nGramCounts[ngram]) / (n1GramCounts[n1gram] + v)


    return probabilities
```

```
WordPunctTokenizer and 1-grams: 795.855313548824
GPT2FastTokenizer and 1-grams: 696.0489735021165

WordPunctTokenizer and 2-grams: 512.5797874130968
GPT2FastTokenizer and 2-grams: 215.55432414980064

WordPunctTokenizer and 3-grams: 8326.03063686746
GPT2FastTokenizer and 3-grams: 2573.128529202333

WordPunctTokenizer and 7-grams: 222074.12760260416
GPT2FastTokenizer and 7-grams: 222818.55649805462
```

Analysis :

- Because we are essentially increasing Denominator more than we are increasing Numerator, (+1 vs +V), we get generally lower Probabilities for our N Grams and thus we can see marginally worse Perplexity.

Task 4

In this task we are evaluating the performance of pretrained GPTLNHeadModel on wiki2test text file on the evaluation metric "Perplexity".

Process:

1. We imported the model and the tokeniser from the HuggingFace's transformers library and set the model to the evaluation mode. Basically we are just using the pre trained model to evalute the text in the wiki2test file. We didn't preform any training of our own.
2. After reading the text file we created the tokens, one thing of note is that we processed the tokens in the chunk size of 1024 to stay in the maximum input length of the model.

```
def wiki2testGPTLMHeadModel():
    # Load the pre-trained GPT-2 model and tokenizer
    model, tokenizer = load_model_and_tokenizer()

    # Read the test data from a file
```

```
    data = read_data_wiki2test('../Data/wiki2.test.txt')

    # Encode the data and split into chunks of max_length = 1024
    encoded_chunks = encode_data(tokenizer, data)

    # Calculate the perplexity for each chunk and return the average
    avg_perplexity = calculate_perplexity_wiki2test(model, encoded_chunks)

    # Print the average perplexity
    print(f'Average Perplexity for wiki2test: {avg_perplexity}')
```

The Preplexity came out to be 24.91500014664805

Analysis:

We got better perplexity results for GPTLMHeadModel because of the following reasons:

GPTLNHeadModel uses transformer architecture with self-attention mecahnism which makes the model capable of considering the entire textual input for context in comparison to the limited n-1 word context of the n gram models. Additionaly the GPT model is pre trained on diverse data, it allows better genralization across different texts.

For unseen words, GPT model utilised subword tokenisation as compared to the limited handeling of n grams for out of vocabulary words.

Task 5

```
Example 1:
 Example Text : 01. Best known for developing the theory of relativity,
Einstein also made important contributions to quantum mechanics, and was
thus a central figure in the revolutionary reshaping of the scientific
understanding of nature that modern physics accomplished in the first
decades of the twentieth century.

Perplexity using GPTLNHeadModel = 18.37134552001953
WordPunctTokenizer and 1-grams: 960.1347729921475
WordPunctTokenizer and 2-grams: 112.54147433855947
WordPunctTokenizer and 3-grams: 65.31389719311872
WordPunctTokenizer and 7-grams: 44.491356310559304


Example 2:
 Example Text : 02. Everyone looks to the Stark banners, with their
direwolf crest-of-arms. We see their opinions about the pups change, as
they come to understand the import of this omen.

Perplexity using GPTLNHeadModel = 96.34398651123047
WordPunctTokenizer and 1-grams: 497.62303029785073
WordPunctTokenizer and 2-grams: 112.8694726435921
WordPunctTokenizer and 3-grams: 36.803207516125426
```

```
WordPunctTokenizer and 7-grams: 31.999999999999943


Example 3:
 Example Text : 03. I do not like them in a house. I do not like them with
a mouse. I do not like them here or there. I do not like them anywhere. I
do not like green eggs and ham. I do not like them, Sam-I-am.

Perplexity using GPTLNHeadModel = 12.52757740020752
WordPunctTokenizer and 1-grams: 864.6764400533297
WordPunctTokenizer and 2-grams: 123.86991682364616
WordPunctTokenizer and 3-grams: 62.74836856992189
WordPunctTokenizer and 7-grams: 51.0


Example 4:
 Example Text : 04. I am Sam.  Sam I am.  I do not like green eggs and
ham.

Perplexity using GPTLNHeadModel = 139.3201141357422
WordPunctTokenizer and 1-grams: 1744.8465912528027
WordPunctTokenizer and 2-grams: 83.69193849874935
WordPunctTokenizer and 3-grams: 26.282763892448568
WordPunctTokenizer and 7-grams: 13.0


Example 5:
 Example Text : 05. I am Sam.       Sam I am.       I do not like green
eggs and ham.

Perplexity using GPTLNHeadModel = 36.378211975097656
WordPunctTokenizer and 1-grams: 1708.005592434021
WordPunctTokenizer and 2-grams: 85.09981322939151
WordPunctTokenizer and 3-grams: 26.282763892448568
WordPunctTokenizer and 7-grams: 13.0


Example 6:
 Example Text : 06. <s> I am Sam. </s> <s> Sam I am. </s> <s> I do not
like green eggs and ham. </s>

Perplexity using GPTLNHeadModel = 19.55457305908203
WordPunctTokenizer and 1-grams: 926.5156280364979
WordPunctTokenizer and 2-grams: 135.76468730441684
WordPunctTokenizer and 3-grams: 33.379921293054856
WordPunctTokenizer and 7-grams: 19.00000000000002


Example 7:
 Example Text : 07. <s>

Perplexity using GPTLNHeadModel = 151.07847595214844
WordPunctTokenizer and 1-grams: 486.43145240099125
WordPunctTokenizer and 2-grams: 142.48944876990956
```

```
WordPunctTokenizer and 3-grams: 1.0
Cannot calculate perplexity for 7-grams as the test data is too short


Example 8:
 Example Text : 08. Natural Language Processing (NLP) is a branch of
artificial intelligence that focuses on techniques that enable computers
to understand, interpret and manipulate human language.

Perplexity using GPTLNHeadModel = 12.873590469360352
WordPunctTokenizer and 1-grams: 2035.724418734341
WordPunctTokenizer and 2-grams: 145.13537032394106
WordPunctTokenizer and 3-grams: 28.893655281587026
WordPunctTokenizer and 7-grams: 21.999999999999964


Example 9:
 Example Text : 09. Common NLP tasks include question answering, text
classification (including fakes detection), text summarization, text
generation (including dialogue, translation and program synthesis),
natural language inference and knowledgebase completion, among others.

Perplexity using GPTLNHeadModel = 124.01785278320312
WordPunctTokenizer and 1-grams: 2165.4552607939063
WordPunctTokenizer and 2-grams: 56.95908820639672
WordPunctTokenizer and 3-grams: 33.85655195196138
WordPunctTokenizer and 7-grams: 31.999999999999943


Example 10:
 Example Text : 10. The Steelers, whose history may be traced to a
regional pro team that was established in the early 1920s, joined the NFL
as the Pittsburgh Pirates on July 8, 1933. The team was owned by Art
Rooney and took its original name from the baseball team of the same name,
as was common practice for NFL teams at the time.

Perplexity using GPTLNHeadModel = 20.022005081176758
WordPunctTokenizer and 1-grams: 545.6311039878337
WordPunctTokenizer and 2-grams: 88.91072658721832
WordPunctTokenizer and 3-grams: 58.230341192618916
WordPunctTokenizer and 7-grams: 61.99999999999979


Example 11:
 Example Text : 11. Northwestern University (NU) is a private research
university in Evanston, Illinois, United States. Established in 1851 to
serve the historic Northwest Territory, it is the oldest chartered
university in Illinois. The university has its main campus along the
shores of Lake Michigan in the Chicago metropolitan area.

Perplexity using GPTLNHeadModel = 14.56400203704834
WordPunctTokenizer and 1-grams: 912.0980918238084
WordPunctTokenizer and 2-grams: 95.00751882753366
WordPunctTokenizer and 3-grams: 58.64549084736445
```

```
WordPunctTokenizer and 7-grams: 48.99999999999993
```

Analysis :

- GPT2 Pretrained model outperforms the N Gram Models significantly for all examples. This is primarily due to the Attention mechanism and the transformer architecture at play which Takes Positional Encoding as well Attention mechanism that gives added context for sentences. As opposed to N Gram that only takes probabilistic approach to predicting tokens. Since the amount of context that model has for these 2 approaches are vastly different, it is expected that Simple Probability based models will perform poorly.
- Increased Performance as N increases. Since with Increasing N, we get better context of sentence, the model is able to progressively able to predict better.
- Another reason why Higher N grams are better able to predict tokens (on paper at least) is that since we are dependent on purely the counts of N grams, as N increases, we have fewer Total Number of distinct N Grams for sentences and as a result, our denominator is much lower even if our numerator values are low.