Using N-grams as Tokens to Create Phrases

Mochalova Elizaveta

November 4, 2024

1 Introduction

In this project, we aim to develop a model that generates coherent, grammatically accurate phrases using N-gram modeling. The main process involves constructing trigrams from text data and then using these trigrams to generate phrases, which are subsequently refined using a T5-based transformer model. We measure the quality of generated phrases using different metrics. For additional refinement, we employ the HappyTextToText transformer model to correct and polish the generated phrases, aligning them more closely with natural language.

This approach combines the strengths of N-grams for phrase creating with the grammatical capabilities of transformers, resulting in generation of high-quality phrases.

2 Construction of N-grams

2.1 Dataset for N-grams

To start, we need to prepare N-grams so that they can be used as tokens later on to create whole phrases. This can be one using one of two approaches:

- using an existing dictionary of N-grams, such as Google Ngram
- creating a new dictionary from a large corpus of text.

We will be using the second approach since it would be easier to train our model on the topics we are interested in.

Another choice that has to be made is how many items do we want to consider for the N-grams? It is possible to use N-grams of characters or of words, in this project we will use words. We will create a dictionary of 3 words using NLTK library corpora. The main disadvantage of NLTK is that its corpora are quite limited in size and in applications. But they are already cleaned and tokenized. This saves time on preprocessing. In NLTK library there are multiple corpora inside, such as Brown (sources categorized by genre, has around 1.1 mil words), Gutenberg (selection of texts from the Project Gutenberg, around 2.5 mil words), etc. We use NLTK Gutenberg corpus in this project work:

```
nltk.download('gutenberg')
all_words = gutenberg.words()
```

If we want to work on a large-scale NLP project that requires substantial data, we might want to switch to a larger corpus like Common Crawl (hundreds of TB, but need to preprocess and clean), OpenWebText (about 40GB), or Wikipedia Dump (about 25GB). In this case we would need to:

- download the corpus;
- preprocess removing tags, non article sections;
- tokenize sentences into words.

A different process would be implemented if we were to use an available N-gram dataset, for example Google Ngram Viewer.

2.2 Building N-grams

We want to create an N-gram language model using the data prepared in the first step. NLTK library allows us to get the N-grams generated from a sequence of items, using *nltk.grams()*.

Since creating the model of N-grams with respective frequencies a special function was implemented that allows saving of the model as pickle file, to avoid the construction of the model every time.

3 Phrase generation

To generate complete sentences we start from 2 random consecutive words from the corpus, then we predict the next word by sampling from the probability distribution of possible next words given the previous n-1 words, in our case n=3. We continue this process until a phrase is of a certain length or a stop condition is met, such as punctuation. The choice of the next word can be done using one of these approaches:

- greedy - always pick the most probable word;
- random - sample words based on probability in the corpus. This approach is used to mitigate the use of some extremely common words.

To create phrases that flow better, we added some conditions, such as no repeating words in sequence, since it is something that very rarely happens in natural language.

4 Testing and evaluation

4.1 Metrics

After generating a phrase we pass it to an LLM to evaluate its grammar, the result can be taken as is or can be used to further increase the model's performance. Some of the metrics to evaluate the phrases can be:

- perplexity - the lower the perplexity the better;
- semantic coherence - use the LLM to assign a coherence score based on how logical or meaningful the phrase is;
- classifying phrases as coherent or not - fine-tune a classification head on the LLM to check whether phrases make sense;
- BLEU (Bilingual Evaluation Understudy) calculates the precision of N-grams generated by our custom model comparing them to some reference. It is then modified by a brevity penalty to account for phrases that are shorter than the reference ones. The formula is

$$BLEU = BP * exp(\sum pn)$$

.

- ROUGE (Recall-Oriented Understudy for Gisting Evaluation) measures the similarity between the machine-generated phrases and the reference ones using overlapping n-grams, word sequences that appear in both. The formula is

$$ROUGE = \sum(Recall)$$

.

We mostly concentrated on the last two metrics, using a library called evaluate, considering references as the corrected phrases and predictions as the original phrases produced by N-gram model.

```
bleu = evaluate.load("bleu")
rouge = evaluate.load('rouge')
```

4.2 Correction of phrases

In this project work we use HappyTextToText, which is a transformer base on T5, to check and correct the grammar of the phrase produced by the N-gram model. Here it is possible to see an example of how the model corrects a given phrase:

- phrase created by the N-gram *", or D ' ye do well : And court the fair girl ' s silver penny ."*
- phrase modified by HTTT *"And court the fair girl's silver penny."*

5 Results

In this project work we implemented a program that is able to create a model that composes complete sentences using trigrams, we correct these sentences with HappyTextToText transformer. In the image 1 we can see the evaluation of the 100 phrases created by the model and then corrected by HTTT. We can see that there are some outliers that have a particularly bad BLEU score, that is given by the penalty for short phrases, almost all of the outliers were in fact very short.

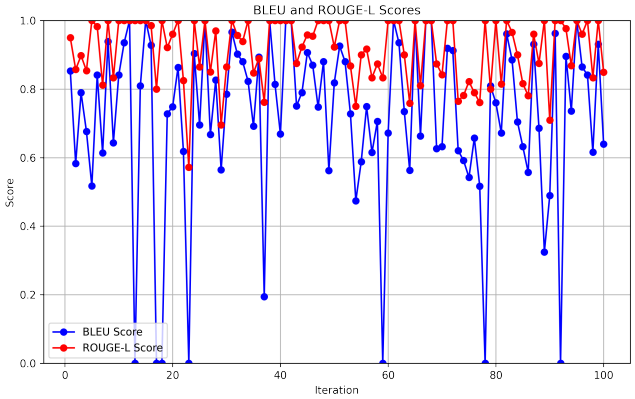

Figure 1: Phrase scores

Another test that was executed was to see the length of the generated phrases. The model used the limit of 50 new generated tokens after which it would stop always, but in case a punctuation such as *". ! ?"* was generated the phrase would be interrupted. As can be seen in the image 2 most of the phrases are generated fully for 50 words and for all the other lengths there is quite an even distribution.

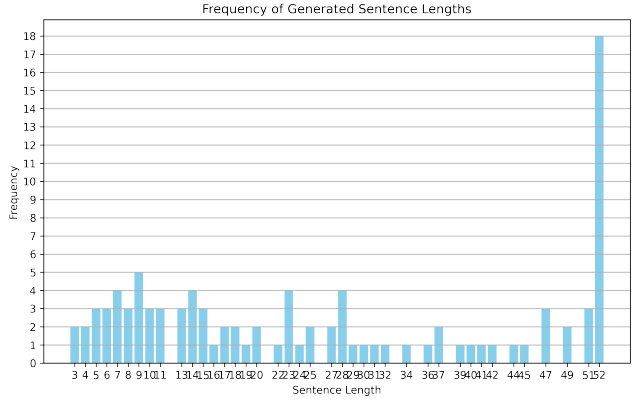

Figure 2: Length of the generated phrases