Natural Language Processing
1st Assignment
N-gram Language Models

Marina Samprovalaki *f3322310*
Petros Chanas *f3322314*

October 30, 2023

Dataset Selection

We have selected as our coprus a sample of the book *Alice in Wonderland* by Lewis Carrol. This was done on purpose as we wanted a fully diverse coprus in matters of word complexity and sentence synthesis in order for our model to be as multi-layered as possible. You can find our notebook with the full code here.

Pre-Processing the Corpus

Once we had chosen our corpus, the initial step in our research involved modifying the text to create a list of strings. Each string in this list represents a segment of the original text, separated by newline characters. We first convert all text to lowercase. Lowercasing plays a pivotal role in maintaining uniformity throughout the corpus, as it mitigates issues associated with case sensitivity. For instance, ”Apple” and ”apple” would be considered separate words in a case-sensitive context, but lowercasing harmonizes them as identical. Additionally, this procedure is of utmost importance for generalization, as it normalizes the text.

Splitting the Dataset

To facilitate the training and testing of our model, we employed techniques to partition our corpus into distinct segments, allowing them to be processed separately and independently. It’s important to note that we began this process by shuffling the data for randomness and bias avoidance. We created three sets: a training set, a development set (also known as a validation set), and a test set. The training set is used to train the model, while the development (validation) set is used for hyperparameter tuning, model selection, and assessing model performance during training. As for the testing set, is used for evaluation.

Having a separate development set is crucial because it enables us to experiment with different configurations, adjust model parameters, and select the best model without contaminating the test set. By monitoring model performance on the development set, we can make adjustments to prevent overfitting. Many machine learning algorithms have hyperparameters (e.g., learning rate, regularization strength) that need to be set before training. The development set is used to fine-tune these hyperparameters to achieve the best possible model performance, as is the case in our project.

The dataset was divided into three sets according to our specific prerequisites:

- The training set with a said ratio of 0.6 ($P_{train} = 0.7$)
- The development set with a said ratio of 0.15 ($P_{dev} = 0.15$)
- The test set with a said ratio of 0.15 ($P_{test} = 0.15$)

Vocabulary and OOV words

It's crucial to highlight that our model had to be specifically designed to handle unknown words. To address this, we adopted a prudent approach of constructing a vocabulary. In this process, words with a frequency exceeding 10 were included in the vocabulary, while the remaining words were labeled with the token `'UNK'` to signify unknown terms. It's important to note that this vocabulary creation and UNK replacement is executed prior to dividing our corpus into the three subsets. This strategic approach enabled our model to be trained while effectively managing UNK words. Out-of-Vocabulary (OOV) words are important in the context of N-gram language models and natural language processing for several reasons:

1. **Robustness to new data**: OOV words are words that the model hasn't seen during training. In real-world applications, it's common to encounter words that were not present in the training data. OOV handling ensures that the model doesn't break or perform poorly when dealing with previously unseen words.
2. **Smoothing technique**: N-gram models often use smoothing techniques like Laplace (add-one) smoothing or Good-Turing smoothing to handle unseen N-grams, which include OOV words. These techniques allocate some probability mass to unseen events, making it possible to estimate probabilities for OOV words and avoid zero probabilities.
3. **Vocabulary expansion**: As language evolves, new words are constantly introduced. OOV handling allows a model to adapt to these changes without needing retraining. By having a mechanism to deal with OOV words, N-gram models can make informed predictions for words that were not in the original training data.

Tokenization

We had to tokenize the text stored in the training dataset. Tokenization is the process of splitting text into individual words or tokens. In this case, the code uses nltk's `word_tokenize` function.

Tokens Frequency Distribution

Token frequency is a useful concept when building an n-gram language model, as it helps in estimating the probabilities of n-grams within a given text corpus. N-gram language models are statistical models that capture the co-occurrence of sequences of n tokens (words, characters, etc.) in a text.Token frequency is important for several reasons:

(a) **Probability Estimation**: In an n-gram language model, you estimate the probability of a specific n-gram by counting how often that n-gram appears in your training data. The frequency of a token (or word) is a part of these counts, which helps in calculating the probability of a specific n-gram occurring in a given context.

(b) **Smoothing**: Token frequencies are also essential for smoothing techniques in language modeling. Smoothing methods help address the problem of unseen n-grams or n-grams with low frequencies in the training data. Techniques like Laplace (add-one) smoothing or Kneser-Ney smoothing rely on token frequencies to redistribute probability mass.

(c) **Language Model Evaluation**: When evaluating the quality of an n-gram language model, token frequencies can help you measure how well the model fits the training data and how well it generalizes to unseen data.

N-grams frequency

It is crucial to perform n-gram counting for unigrams, bigrams, and trigrams within the given sentences. This is a common pre-processing step when building language models, as it helps in understanding the distribution of n-grams in the text. After the tokenization of the sentences, we count individually the occurence of unigrams (single tokens), bigrams (pairs of consecutive tokens) and trigrams (triplets of consecutive tokens) within those sentences and we print the most common n-gram for each n-gram type.

$$P(w), P(w_i|w_i - 1), P(w_i|w_i - 2, w_i - 1), \forall w_n \in C$$

bi-gram probability

Now that we have the frequency of said tokens it is time to compute the corresponding probability for each counter.

Using Laplace Smothing

Laplace smoothing, also referred to as add-one smoothing or add-k smoothing, is a technique employed to tackle the challenge of zero probabilities in probability distributions. To address this issue, Laplace smoothing involves adding a small, constant value (typically 1 or a small positive integer) to the count of each event in the dataset.

We use the following formula to compute:

$$P(w_2|w_1) = \frac{C(w_1, w_2) + \alpha}{C(w_1) + \alpha \cdot |V|}$$

- $C(w_1, w_2)$: bigram count
- $C(w_1)$: unigram count
- $0 \leq \alpha \leq 1$: smoothing hyper-parameter
- $|V|$: vocabulary size

tri-gram probability

Using Laplace Smoothing

When it comes to trigrams, the previously mentioned formula takes on the following form:

$$P(w_3|w_1, w_2) = \frac{C(w_1, w_2, w_3) + \alpha}{C(w_1, w_2) + \alpha \cdot |V|}$$

- $C(w_1, w_2, w_3)$: Trigram count, the number of times the trigram $(w1, w2, w3)$ appears in the corpus.
- $C(w_1, w_2)$: Bigram count, the number of times the bigram $(w1, w2)$ appears in the corpus.
- $0 \leq \alpha \leq 1$: Smoothing hyper-parameter.
- $|V|$: vocabulary size

Using Kneser-Ney smoothing

This probability for a given token w_i is proportional to the **number of bigrams which it completes**:

$$P_{continuation}(w_i) \propto \ |w_{i-1} : c(w_{i-1}, w_i) > 0|$$

This quantity is normalized by dividing by the total number of bigram types (note that j is a free variable):

$$P_{continuation}(w_i) = \frac{|w_{i-1} : c(w_{i-1}, w_i) > 0|}{|w_{j-1} : c(w_{j-1}, w_j) > 0|}$$

So implementing this idea in our code we get the following relation:

$$P(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) - \delta_+}{C(w_{i-1})} + \lambda(w_{i-1}) \cdot P_{\text{continuation}}(w_i)$$

Kneser-Ney Smoothing

Kneser-Ney smoothing is a more sophisticated smoothing technique compared to Laplace (add-one) smoothing. It aims to address some of the limitations of Laplace smoothing, particularly in modeling n-gram probabilities. Below is the mathematical representation of Kneser-Ney smoothing and a comparison with Laplace smoothing.

The Kneser-Ney smoothing method uses several components and formulas to estimate n-gram probabilities. For simplicity, let's consider the case of bigram probabilities and it evolved from **absolute-discounting interpolation**, which makes use of both higher-order (i.e., higher-*n*) and lower-order language models, reallocating some probability mass from 4-grams or 3-grams to simpler unigram models. The formula for absolute-discounting smoothing as applied to a bigram language model is presented below:

$$P_{abs}(w_i|w_{i-1}) = \frac{\max((c(w_{i-1}w_i) - \delta), 0)}{\sum_{w'} c(w_{i-1}w')} + a \cdot p_{abs}(w_i)$$

Here δ refers to a fixed **discount** value, and α is a normalizing constant.

The essence of Kneser-Ney is in the clever observation that we can take advantage of this interpolation as a sort of backoff model. When the first term (in this case, the discounted relative bigram count) is near zero, the second term (the lower-order model) carries more weight. Inversely, when the higher-order model matches strongly, the second lower-order term has little weight.

Kneser-Ney:

$$P_{KN}(w_i|w_{i-1}) = \frac{\max((c(w_{i-1}w_i) - \delta), 0)}{\sum\limits_{w'} c(w_{i-1}w')} + \lambda\frac{|w_{i-1} : c(w_{i-1}, w_i) > 0|}{|w_{j-1} : c(w_{j-1}, w_j) > 0|}$$

where λ serves as a normalizing constant

$$\lambda(w_{i-1}) = \frac{\delta}{c(w_{i-1})}|w' : c(w_{i-1}, w') > 0|$$

Note that the denominator of the first term can be simplified to a unigram count.

Kneser-Ney Interpolated

$$P_{KN}(w_i|w_{i-1}) = \frac{\max((c(w_{i-1}w_i) - \delta), 0)}{c(w_{i-1})} + \lambda\frac{|w_{i-1} : c(w_{i-1}, w_i) > 0|}{|w_{j-1} : c(w_{j-1}, w_j) > 0|}$$

Create and count n-grams frequency

We needed to count the occurrences of n-grams in the training data. Then, we calculated the probabilities associated with each n-gram and used these probabilities to construct the n-gram language model itself.

Bigram

To train the bigram language model, our approach involved several key steps. Firstly, we pre-processed each sentence by adding "¡start¿" at the beginning and "¡end¿" at the end. This is a common practice in language modeling and is used to demarcate the start and end of sentences. Next, we implemented a nested loop to iterate through the bigrams in order to calculate the bigram probability for each pair of tokens. To account for unseen or low-frequency bigrams, we applied Laplace smoothing.

Trigram

To train the trigram language model, we follow a similar process. However, with trigrams, we had to go a step further and add "¡start1¿" and "¡start2¿" at the very beginning of the sentence. This is because trigrams use the last two words to calculate the probability. The rest of the process, including iterating through trigrams and applying Laplace smoothing, is consistent with the approach used for bigrams.

Interpolation

Interpolation allows us to seamlessly combine the strengths of different n-gram models to enhance predictive accuracy and adapt to various text data scenarios. Interpolation provides a parameter ”lambda” (λ) representing the weight assigned to trigram models compared to bigram models. By employing interpolation, we can simultaneously consider both trigram and bigram models when calculating the probability of word sequences. This balance between the two models helps mitigate data sparsity issues and allows for improved predictions. The result is a more accurate language model, as indicated by cross entropy and perplexity values.

$$P(w_n|w_{n-2}, w_{n-1}) = \lambda \cdot P(w_n|w_{n-2}, w_{n-1}) + (1 - \lambda) \cdot P(w_n|w_{n-1})$$

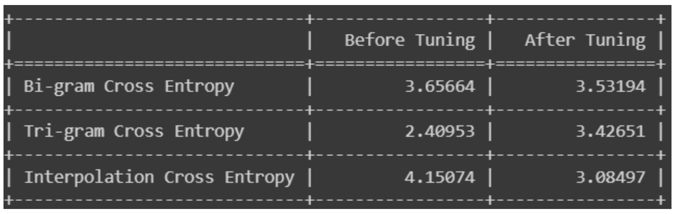
Hyper-parameters Tuning

Frequently, our models do not perform as effectively as we initially anticipate after training. To enhance their performance, we employ a crucial procedure known as Hyperparameter Tuning.

Hyperparameter tuning is the methodical process of searching for the best hyperparameter values that result in the most accurate and effective models. This process involves fine-tuning these hyperparameters to identify the optimal combination that minimizes errors and maximizes the model’s predictive power on unseen data. To achieve this, we utilize the validation subset created in the initial step. The process requires the selection of specific hyperparameters to fine-tune. In our case, we opted to focus on alpha for Laplace and lambda for Interpolation.

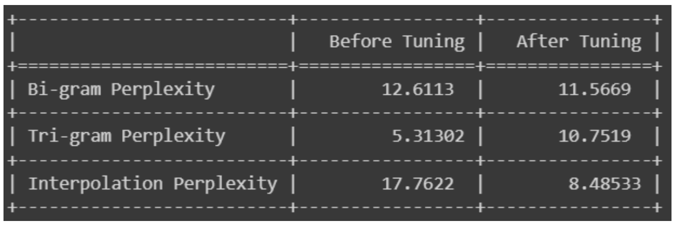
After-tuning

The figures below depict the cross-entropy and perplexity values before and after tuning. The observed decrease in the trigram model’s performance after tuning can be attributed to a combination of factors. One of the most important ones is that the trigram model, being more complex and considering the conditional probability of a word based on the previous two words, is prone to overfitting. Our corpus is not so extended so the training and validation sets are not so big enough. As a result, it is easy for a model to be overfitted. As a result, the trigram model may struggle to generalize effectively to unseen data, leading to higher perplexity scores in the evaluation.



```
+-----------------------------+----------------+---------------+
|                             |  Before Tuning |  After Tuning |
+=============================+================+===============+
| Bi-gram Cross Entropy       |        3.65664 |       3.53194 |
+-----------------------------+----------------+---------------+
| Tri-gram Cross Entropy      |        2.40953 |       3.42651 |
+-----------------------------+----------------+---------------+
| Interpolation Cross Entropy |        4.15074 |       3.08497 |
+-----------------------------+----------------+---------------+
```

Figure 1: Cross Entropy before and after Tuning



```
+--------------------------+----------------+----------------+
|                          |  Before Tuning |   After Tuning |
+==========================+================+================+
| Bi-gram Perplexity       |       12.6113  |       11.5669  |
+--------------------------+----------------+----------------+
| Tri-gram Perplexity      |        5.31302 |       10.7519  |
+--------------------------+----------------+----------------+
| Interpolation Perplexity |       17.7622  |        8.48533 |
+--------------------------+----------------+----------------+
```

Figure 2: Perplexity before and after Tuning

Generate Completions

In both bigram and trigram models, sentence completion involves estimating the conditional probability of the next word in a sentence given the context provided by the preceding words. This probability is used to suggest the most likely word to complete the sentence. While bigram models are simple and computationally efficient, trigram models offer more context-aware predictions, which can be valuable in tasks like text generation, autocomplete, and machine translation. The choice between bigram and trigram models often depends on the specific application and the trade-off between model complexity and accuracy.

In our model we have the following approach:

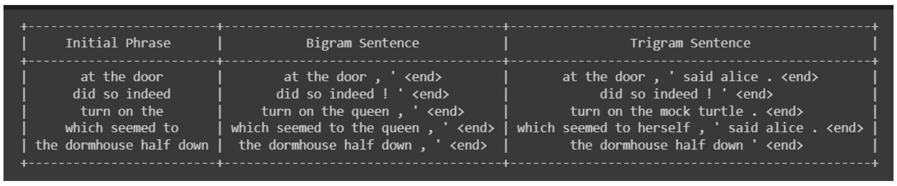
(a) We eliminate all `UNK` tokens from our corpus as it is the most prevalent one by occurrence

(b) We find the bigram with the greatest probability in the filtered corpus

(c) We select the next bigram to be added to the sentence based on the bigram's probability, ensuring that the sentence generation favors bigrams with higher probabilities.

(d) We update the list of the already stored words for the next iteration to be the second word of the selected bigram.

The same procedure is strictly followed to the trigram model with the same modular steps.



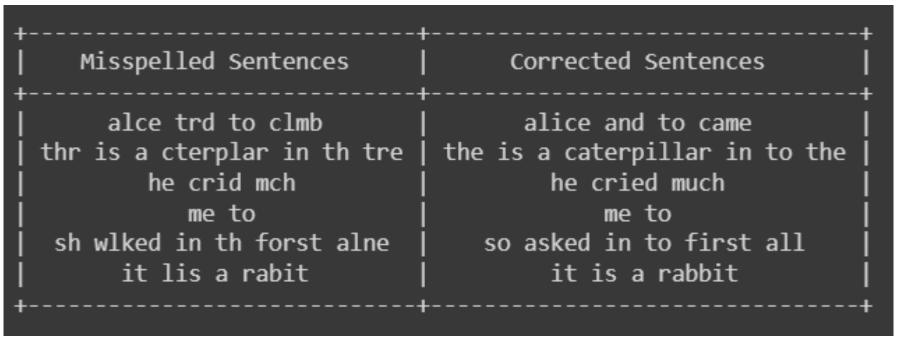
```
+------------------------+------------------------------------+---------------------------------------------+
|     Initial Phrase     |          Bigram Sentence           |              Trigram Sentence               |
+------------------------+------------------------------------+---------------------------------------------+
|      at the door       |       at the door , ' <end>        |     at the door , ' said alice . <end>      |
|     did so indeed      |      did so indeed ! ' <end>       |           did so indeed ! ' <end>           |
|      turn on the       |      turn on the queen , ' <end>   |       turn on the mock turtle . <end>       |
|     which seemed to    | which seemed to the queen , ' <end> | which seemed to herself , ' said alice . <end> |
| the dormhouse half down |  the dormhouse half down , ' <end>  |       the dormhouse half down ' <end>       |
+------------------------+------------------------------------+---------------------------------------------+
```

Figure 3: Prediction examples for both bigram and trigram models

Context-aware Spelling Corrector

The subsequent phase of our project involved the development of a context-aware spelling corrector, which utilizes our trigram model, a beam search decoder, and the Levenshtein distance This spelling corrector is designed to rectify any errors or mistakes in words, enhancing the accuracy and readability of text. As for the algorithm, we first generate candidate corrections for the given word based on their Levenshtein distance from the original word. Then, we compute a score for each candidate correction based on a combination of the negative Levenshtein distance and our trigram language model score. Last but not least, we implement a Beam Search decoder for each word. Then, we select the top candidates based on their scores and narrows down the sequences to the best ones. The best correction for the word is the one with the maximum score.

As observed, there are instances where our corrector performed as anticipated, while



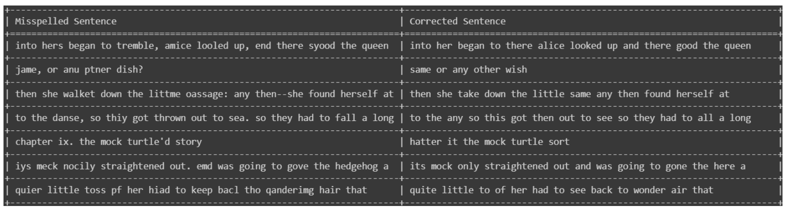
```
+----------------------------+------------------------------+
|    Misspelled Sentences    |     Corrected Sentences      |
+----------------------------+------------------------------+
|      alce trd to clmb      |      alice and to came       |
| thr is a cterplar in th tre | the is a caterpillar in to the |
|         he crid mch        |         he cried much        |
|            me to           |             me to            |
|   sh wlked in th forst alne |    so asked in to first all   |
|        it lis a rabit      |        it is a rabbit        |
+----------------------------+------------------------------+
```

Figure 4: Misspelled and corrected examples

in some cases, it did not yield the expected results. This variability can be attributed to either the model's inability to identify the most suitable correction or the fact that the word in question did not exist in the vocabulary.

Evaluate the Corrector

The next phase involved the utilization of an extended dataset to evaluate our Corrector. We chose to use the same test dataset that we previously employed for assessing our language models. However, we introduced a modification by replacing each non-space character of every test sentence with another random non-space character, with a small probability. Additionally, we attempted to replace each letter with another random letter. In cases where the word was represented as the ”UNK” token, we replaced it with ”y” in the list. We specifically selected 20 misspelled sentences and applied the Corrector to rectify them. As observed, the Corrector was successful in correcting some of the errors, but some mistakes still remained.



```
+-------------------------------------------------------------------------+------------------------------------------------------------------------+
| Misspelled Sentence                                                     | Corrected Sentence                                                     |
+=========================================================================+========================================================================+
| into hers began to tremble, amice looled up, end there syood the queen  | into her began to there alice looked up and there good the queen       |
+-------------------------------------------------------------------------+------------------------------------------------------------------------+
| jame, or anu ptner dish?                                                | same or any other wish                                                 |
+-------------------------------------------------------------------------+------------------------------------------------------------------------+
| then she walket down the littme oassage: any then--she found herself at | then she take down the little same any then found herself at           |
+-------------------------------------------------------------------------+------------------------------------------------------------------------+
| to the danse, so thiy got thrown out to sea. so they had to fall a long | to the any so this got then out to see so they had to all a long       |
+-------------------------------------------------------------------------+------------------------------------------------------------------------+
| chapter ix. the mock turtle'd story                                     | hatter it the mock turtle sort                                         |
+-------------------------------------------------------------------------+------------------------------------------------------------------------+
| iys meck nocily straightened out. emd was going to gove the hedgehog a  | its mock only straightened out and was going to gone the here a        |
+-------------------------------------------------------------------------+------------------------------------------------------------------------+
| quier little toss pf her hiad to keep bacl tho qanderimg hair that      | quite little to of her had to see back to wonder air that              |
+-------------------------------------------------------------------------+------------------------------------------------------------------------+
```

Figure 5: Misspelled and corrected sentences from the test set

Metrics

WER and CER are metrics that indicate the amount of text in a handwriting that the applied HTR model did not read correctly

Word Error rate (WER)

WER is a metric used to evaluate the performance of systems that generate or correct sequences of words, such as automatic speech recognition systems or spelling correction systems. It measures the number of word-level errors made by a system when comparing its output to the reference or ground truth text.

$$WER = \frac{S + I + D}{N}$$

(a) S : the number of substitutions

(b) i : the number of insertions

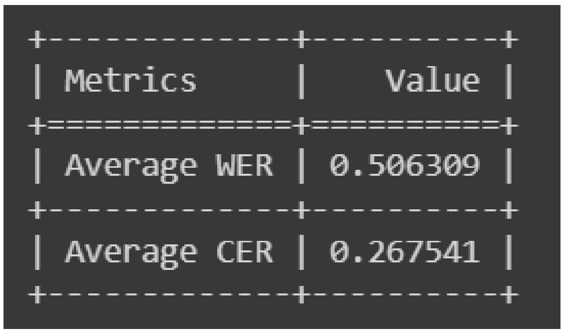
(c) D : the number of deletions

(d) N: the total number of words in the reference

Character Error Rate (CER)

CER is similar to WER but operates at the character level. It measures the accuracy of character-level transcriptions or corrections. CER quantifies the number of character-level errors made by a system when comparing its output to the reference text. Errors include substitutions, insertions, and deletions of individual characters.

$$WER = \frac{S + I + D}{N}$$



```
+-------------+----------+
| Metrics     |    Value |
+=============+==========+
| Average WER | 0.506309 |
+-------------+----------+
| Average CER | 0.267541 |
+-------------+----------+
```

Figure 6: Average WER and CER