Department of Informatics

M.Sc. in Data Science

Text Analytics

N-gram Language Models and Context-Aware Spelling Correction

Instructors: Ion Androutsopoulos, Manolis Kyriakakis

Authors: Sotirios Giogiakas (p3352204), Achilleas Matthaiou (p3352214), Antonios Mavridis (p3352215), Panagiotis Sakkis (p3352222)

30/04/2023

Table of Contents

1 Introduction

This project was a primer on n-grams. We had to work exclusively on bigrams and trigrams, studying how they effect entropy and attempting to develop a context aware spelling corrector from scratch to the best of our abilities. We tackled this assignment as a group. Given the nature of this exercise and how the project grows on itself, splitting the project into even halves would seem wasteful, as we would have group members sitting on their hands while the rest of us tackled the first few difficulties, and vice versa. As a result, we decided to proceed as follows: Every three days, we'd get together to discuss and set some goals for the following time. Then we would work on our own and compare outcomes at the next meet-up, while also attempting to improve our code and refresh our goals for the next time. This strategy was quite beneficial to us, but it makes it much more difficult to "describe in our report the contribution of each member" as desired. Finally, because we all contributed to their development, we should all have a good grasp of the algorithms employed.

You can find our code and our implementation in the below Google Drive folder: Source_Code

2 Exercise 1: Entropy

i) We have a collection of 800 training e-mail messages of which 200 are spam(C=1) and 600 are ham(C=0). The probabilities for each case, are

$$P(C=1) = \frac{200}{800} = \frac{1}{4}, \quad P(C=0) = \frac{600}{800} = \frac{3}{4}.$$

In addition, we calculate the entropy

$$\begin{aligned} H(C) &= -P(C=1)\log_2 P(C=1) - P(C=0)\log_2 P(C=0) \\ &= -\frac{1}{4}log_2\frac{1}{4} - \frac{3}{4}\log_2\frac{3}{4} \\ &= -\frac{1}{4}(-2)\log_2 2 - \frac{3}{4}\log_2 3 + \frac{3}{4}\log_2 4 \\ &\simeq 0.5 - 1.19 + 1.5 \simeq 0.81 \end{aligned}$$

ii) We repeat the same process when all the training messages are in one category. First, we assume that all the messages are spam:

$$P(C=1) = 1, \quad P(C=0) = 0.$$

The entropy is

$$\begin{aligned} H(C) &= -1\log_2 1 - P(C=0)\log_2 P(C=0) \quad (\text{Setting}\, y = P(C=0)) \\ &= -\lim_{y\to 0} y\log_2 y = -\lim_{y\to 0}\frac{\log_2 y}{\frac{1}{y}} \overset{L'Hospital}{=} -\lim_{y\to 0}\frac{\frac{1}{y}}{-\frac{1}{y^2}} = \lim_{y\to 0} y = 0 \end{aligned}$$

Then, $H(C) = 0$. Second, we assume that all messages are ham:

$$P(C=1) = 0, \quad P(C=0) = 1.$$

As above, $H(C) = 0$. iii) We repeat the same process when we have an equal number of training messages per category(400 spam, 400 ham) with probabilities $P(C=0) = P(C=1) = \frac{1}{2}$. Then, we calculate the entropy

$$\begin{aligned} H(C) &= -P(C=1)\log_2 P(C=1) - P(C=0)\log_2 P(C=0) \\ &= -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} \\ &= -\log_2\frac{1}{2} = 1. \end{aligned}$$

3 Exercise 2: Levenshtein Distance

You can find our code and our implementation in the below Google Colab Notebook: Exercise_2_Levenshtein_Distance.ipynb

3.1 i) Levenshtein Distance between two words

The `levenshtein_distance` function implements the Levenshtein distance algorithm, which is a measure of the difference between two strings. The function takes five arguments: s1 and s2 are the two strings to compare, `ins_cost`, `del_cost`, and `sub_cost` are the costs for inserting, deleting, and substituting a character, respectively.

The function first initializes a matrix dp with zeros of size m+1 by n+1, where m and n are the lengths of s1 and s2, respectively. It then initializes the first row and column of the matrix with the cost of deleting and inserting characters, respectively.

The matrix is then filled in using a nested loop over the indices i and j. Specifically, it iterates from 1 to m(inclusive), and for each i, it iterates from 1 to n (inclusive), and for each j, it computes the Levenshtein distance between the prefixes s1[0:i] and s2[0:j].

If the characters at position i-1 in s1 and position j-1 in s2 are the same, then the Levenshtein distance is the same as the distance between the prefixes s1[0:i-1] and s2[0:j-1]. Otherwise, the distance can be computed by taking the minimum of three possible distances:

The distance between s1[0:i-1] and s2[0:j], plus the cost of deleting the character at position i-1 in s1. The distance between s1[0:i] and s2[0:j-1], plus the cost of inserting the character at position j-1 in s2. The distance between s1[0:i-1] and s2[0:j-1], plus the cost of substituting/replace the character at position i-1 in s1 with the character at position j-1 in s2. After the matrix is filled in, the function prints out the Levenshtein table, which shows the values of the matrix dp.

Finally, the function returns the Levenshtein distance between s1 and s2, which is the value stored in dp[m][n].

Using the function that we created with the two below given words s1,s2 and for the below cost for insertion, deletion, substitution

```
s1 = "feinting"
s2 = "einstein"
distance = levenshtein_distance(s1, s2, ins_cost=1, del_cost=1, sub_cost=2)
print("\nLevenshtein distance between", s1, "and", s2, "is", distance)
```

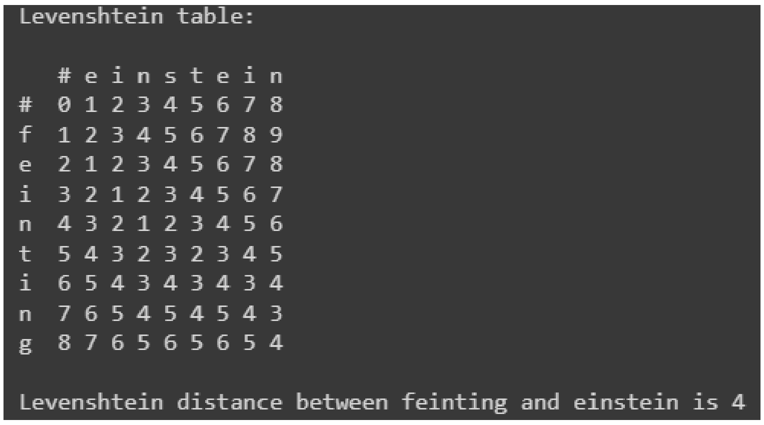
we have the below output:

3.2 i) Levenshtein Distance between a word and a vocabulary

The `levenshtein_distance_in_vocabulary` functions calculates the Levenshtein distance between a word w and all words in a vocabulary V, with a maximum allowed distance of d. The function takes in several optional arguments for the cost of different operations: `ins_cost` for the cost of an insertion, `del_cost`, for the cost of a deletion, and ,`sub_cost`, for the cost of a substitution.

The function first initializes an empty list matches to store all words in the vocabulary that have a Levenshtein distance less than or equal to d from w.

The function proceeds to calculate the Levenshtein distance between w and v using a dynamic programming approach. The function initializes a matrix dp with dimensions (m+1) x (n+1), where m is the length of w and n is the length of v. The function then fills in the matrix using a nested loop, similar to the first function we looked at.



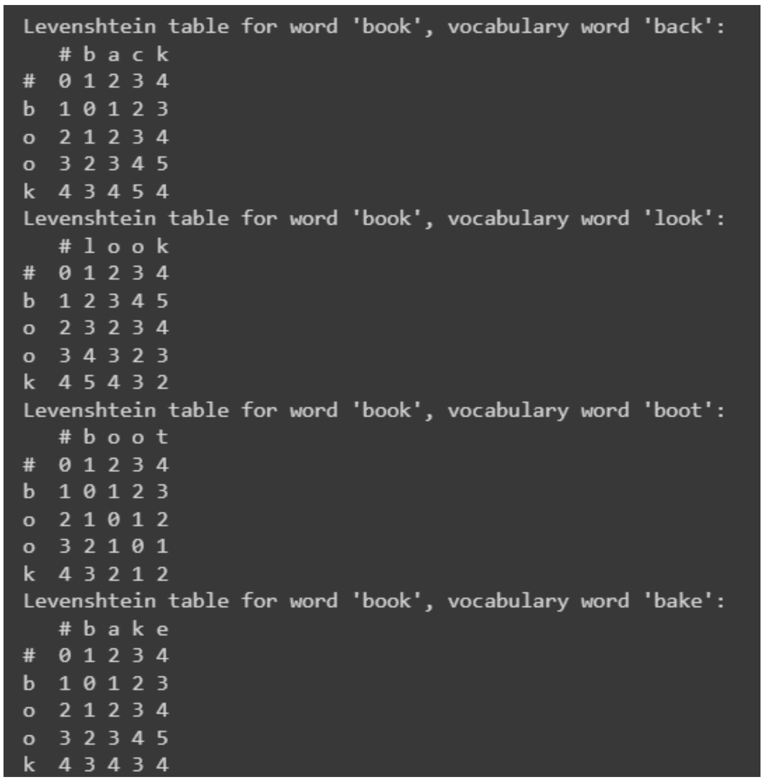
```
Levenshtein table:

   # e i n s t e i n
#  0 1 2 3 4 5 6 7 8
f  1 2 3 4 5 6 7 8 9
e  2 1 2 3 4 5 6 7 8
i  3 2 1 2 3 4 5 6 7
n  4 3 2 1 2 3 4 5 6
t  5 4 3 2 3 2 3 4 5
i  6 5 4 3 4 3 4 3 4
n  7 6 5 4 5 4 5 4 3
g  8 7 6 5 6 5 6 5 4

Levenshtein distance between feinting and einstein is 4
```

Figure 1: Levenshtein distance between s1 and s2

If the Levenshtein distance between w and v is less than or equal to d, the function adds v to the matches list. Finally, the function prints out the Levenshtein table for each pair of words and returns the matches list containing all words in the vocabulary that have a Levenshtein distance less than or equal to d from w. Using the function that we created with the below given word and vocabulary and for the below cost for insertion, deletion, substitution

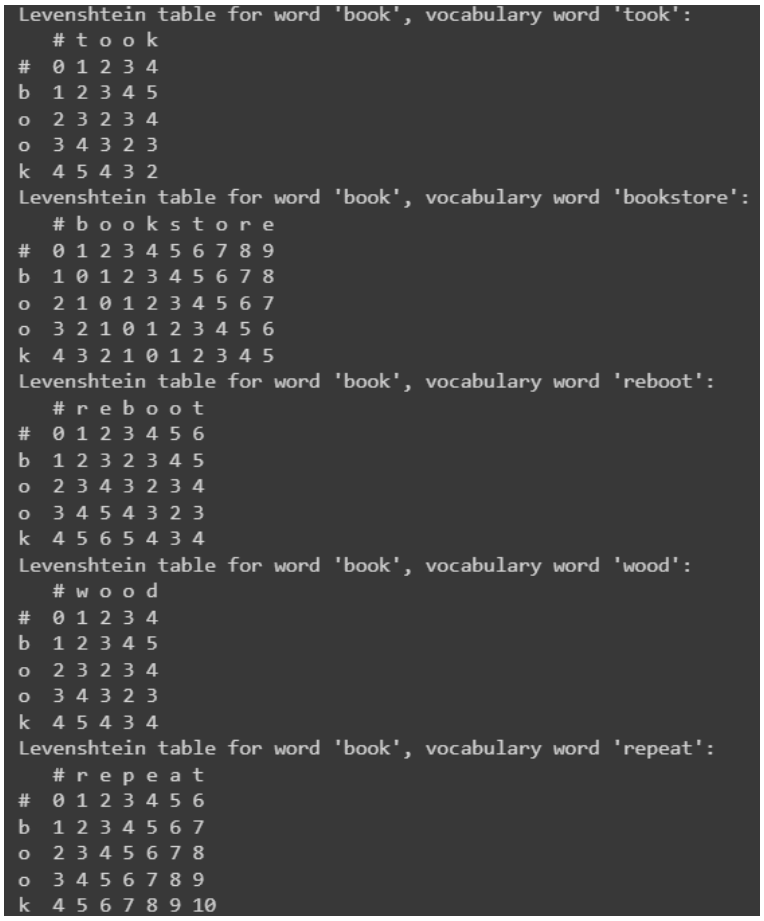
```python
w = "book"
V = ["back", "look", "boot", "bake", "took", "bookstore","reboot","wood","repeat","science"]
d = 2

matched_words = levenshtein_distance_in_vocabulary(w, V, d, ins_cost=1, del_cost=1, sub_cost=2)
print(f"\nThe words of V whose Levenshtein distance to w is up to d={d} are:", matched_words)
```

we have the below output:



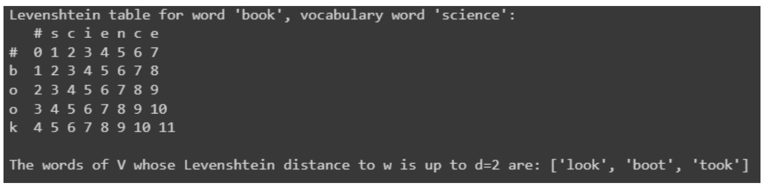
```
Levenshtein table for word 'book', vocabulary word 'back':
   # b a c k
#  0 1 2 3 4
b  1 0 1 2 3
o  2 1 2 3 4
o  3 2 3 4 5
k  4 3 4 5 4
Levenshtein table for word 'book', vocabulary word 'look':
   # l o o k
#  0 1 2 3 4
b  1 2 3 4 5
o  2 3 2 3 4
o  3 4 3 2 3
k  4 5 4 3 2
Levenshtein table for word 'book', vocabulary word 'boot':
   # b o o t
#  0 1 2 3 4
b  1 0 1 2 3
o  2 1 0 1 2
o  3 2 1 0 1
k  4 3 2 1 2
Levenshtein table for word 'book', vocabulary word 'bake':
   # b a k e
#  0 1 2 3 4
b  1 0 1 2 3
o  2 1 2 3 4
o  3 2 3 4 5
k  4 3 4 3 4
```

Figure 2: Levenshtein distance between ’book’ and ’back’,’look’,’boot’,’bake’



```
Levenshtein table for word 'book', vocabulary word 'took':
   # t o o k
#  0 1 2 3 4
b  1 2 3 4 5
o  2 3 2 3 4
o  3 4 3 2 3
k  4 5 4 3 2
Levenshtein table for word 'book', vocabulary word 'bookstore':
   # b o o k s t o r e
#  0 1 2 3 4 5 6 7 8 9
b  1 0 1 2 3 4 5 6 7 8
o  2 1 0 1 2 3 4 5 6 7
o  3 2 1 0 1 2 3 4 5 6
k  4 3 2 1 0 1 2 3 4 5
Levenshtein table for word 'book', vocabulary word 'reboot':
   # r e b o o t
#  0 1 2 3 4 5 6
b  1 2 3 2 3 4 5
o  2 3 4 3 2 3 4
o  3 4 5 4 3 2 3
k  4 5 6 5 4 3 4
Levenshtein table for word 'book', vocabulary word 'wood':
   # w o o d
#  0 1 2 3 4
b  1 2 3 4 5
o  2 3 2 3 4
o  3 4 3 2 3
k  4 5 4 3 4
Levenshtein table for word 'book', vocabulary word 'repeat':
   # r e p e a t
#  0 1 2 3 4 5 6
b  1 2 3 4 5 6 7
o  2 3 4 5 6 7 8
o  3 4 5 6 7 8 9
k  4 5 6 7 8 9 10
```

Figure 3: Levenshtein distance between ’book’ and ’took’,’bookstore’,’reboot’,’wood’,’repeat’



```
Levenshtein table for word 'book', vocabulary word 'science':
   # s c i e n c e
#  0 1 2 3 4 5 6 7
b  1 2 3 4 5 6 7 8
o  2 3 4 5 6 7 8 9
o  3 4 5 6 7 8 9 10
k  4 5 6 7 8 9 10 11

The words of V whose Levenshtein distance to w is up to d=2 are: ['look', 'boot', 'took']
```

Figure 4: Levenshtein distance between ’book’ and ’science’

Considering the above results, we conclude that the words ”back’, ’look’, ’boot’, ’took’, ’wood’ have distance two from the word ’book’.

4 Exercise 3: Bigram and Trigram Language Models

You can find our code and our implementation in the below Google Colab Notebook: Exercise_3_N-Grams.ipynb

4.1 i) Bigram and Tigram Language Model with Laplace Smoothing

To answer the initial inquiry, we were required to develop a bigram and trigram model utilizing Laplace smoothing on a corpus of our choice. The initial step to create the model was to obtain a corpus. We opted for 7 groups from the NLTK's "brown" corpus, which resulted in a total of 22143 sentences. We split our corpus into three sets: a training set, a validation set, and a testing set: 80% of the data is used for the training set and the remaining 20% is used for the validation sets with 10% of each set respectively. According to the requirement, we were instructed to substitute words that occur fewer than ten times with a special token "UNK". Prior to that, we converted our text into lowercase. This was done to ensure the proper identification of all n-grams and to reduce the cardinality of our vocabulary, as capitalization was neither significant for our task nor beneficial to our vocabulary size. To replace the out-of-vocabulary (OOV) words in our text, we adopted a fairly simple technique. Initially, we generated a list of all words, followed by a second list containing only the words identified as OOV based on their frequencies. Lastly, we utilized a nested loop and a search operation to substitute the OOV words in our training dataset with the **unk** token.

To develop various n-gram models, we initiated a Counter object for each n-gram, including all n-grams with lower orders. Additionally, a unigram counter was added, which was necessary for computing bigram probabilities. These three Counter objects were trained on the training dataset, with each sentence padded with a `<start>` and `<end>` token (or two in the trigram case) at the beginning and end. As the calculation of n-gram probabilities was critical to the project, two functions were created to accomplish that for the bigram and trigram probabilities, respectively.It is worth mentioning that we chose to output the base two logarithm of probabilities instead of the probabilities themselves in our functions. The rationale behind this decision is that we prefer to work with a sum of negative, relatively "large" numbers instead of a product that approaches zero as the number of probabilities increases. This approach is not only more visually appealing for the reader, but also easier for the computer to carry out the required computations.

4.2 ii) Fine-Tuning and Testing

Next, we adjust the hyperparameter alpha for Laplace Smoothing in our model through a process of fine-tuning during the validation process. For every alpha, we had to calculate the language cross-entropy and perplexity of the two models on our validation subset. Initially, we removed the OOV words from the validation subset using the OOV list we created earlier to clean the training set. Then, we added the necessary `<start>` and `<end>` tokens to each sentence and calculated the logarithmic probability of each bigram (or trigram, respectively) in a loop, followed by summing them. We divided that sum by the total number of n-grams, including `<end>` tokens as instructed, which gives us the cross-entropy of our model on the test set. To calculate the perplexity of our model, we simply raised 2 to the power of our calculated cross-entropy. For the bigram counter, we began iterating from the second word to avoid including probabilities of the form `P(<start>|...)`, and for the trigram loop, we began from the third word to avoid unnecessary calculations of the `P(<start1>|...)` and `P(<start2>|...)` probabilities. We also terminated the trigram loop one word earlier because `P(<end2>|...)` should not influence the entropy, since `P(<end1>|...)` already predicts if a word is at the end of a sentence or not. From our fine-tuning process, we had the below results:

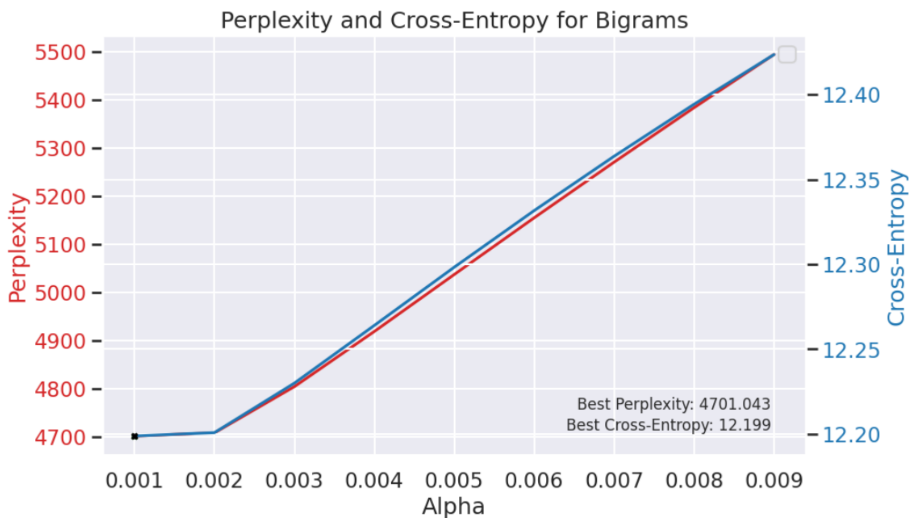


Figure 5: Perplexity and Cross-Entropy for Bigrams for different alphas.

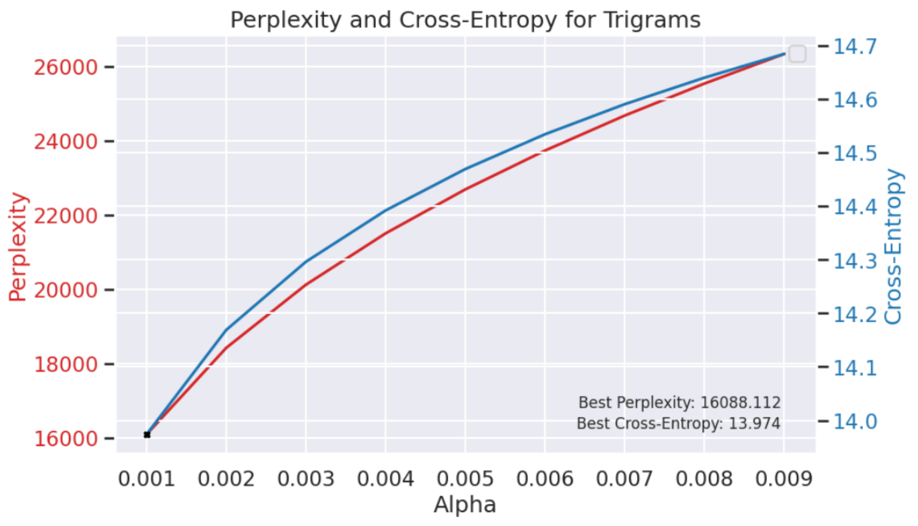


Figure 6: Perplexity and Cross-Entropy for Trigrams for different alphas.

According to our findings, the best Cross-Entropy and Perplexity for both models occurred when the alpha was equal to 0.001. So, in order to test our models, we repeated the validation set process in the test set with a fixed alpha of 0.001.

Model	Cross-Entropy	Perplexity
Bigram model with Laplace smoothing	11.854	3701.224
Trigram model with Laplace smoothing	17.457	179881.684

The outcomes of our two models turned out to be unexpected. Although the trigram model, which should theoretically perform better due to its capacity to consider more information in estimating probabilities, produced a higher entropy. At first, this result appeared counter-intuitive. However, upon reflection, it is not as complicated. The relatively small size of our dataset led to a significant number of words appearing less than ten times. This fact, combined with replacing them with unk tokens, further narrowed our vocabulary and increased the uncertainty of our context-sensitive model, as unk has no specific context. Another point to consider is that since the ”brown” corpus from NLTK includes several substantially different sub-corpora, the context of similar words in each could vary and ”confuse” our trigram model, which is much more sensitive to context.

4.3 iii) Context-Aware Spelling Corrector

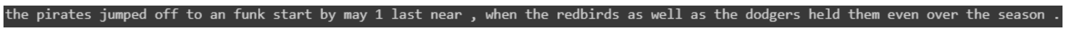
In this subsection, we had to create a context-aware spelling corrector using our n-gram models in a beam search decoder algorithm. The way we approached this algorithm was the following:

The code takes a sentence as input and tokenizes it using NLTK's TweetTokenizer. We then iterate over the tokenized sentence to find misspelled words that are not present in the vocabulary_train. For each misspelled word, we generate candidate corrections using the Levenshtein distance algorithm and score them using the bigram probability with Laplace smoothing.

Using the beam search algorithm, we select the top two candidate corrections with the highest probability. Starting at the misspelled word in the tokenized sentence, we generate all possible candidate corrections, score them using the bigram probability, and select the top beam_size candidates. We then expand the search to the next misspelled word and select the top beam_size candidates for each possible correction of the previous misspelled word. This process is repeated until all misspelled words in the sentence are corrected.

After generating the candidate corrections and their scores, we select the corrected sentence with the highest probability by multiplying the probabilities of each corrected word. If the probability of the corrected sentence using candidate correction 1 is greater than the probability of the corrected sentence using candidate correction 2, we output the first corrected sentence. Otherwise, we output the second corrected sentence. The output of our code is the corrected sentence with the highest probability.

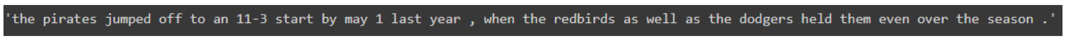
For example, the context-aware spelling corrector for the below sentence



```
the pirates jumped off to an funk start by may 1 last near , when the redbirds as well as the dodgers held them even over the season .
```

Figure 7: Input sentence in context-aware spelling corrector

has the below output:



```
'the pirates jumped off to an 11-3 start by may 1 last year , when the redbirds as well as the dodgers held them even over the season .'
```

Figure 8: Predicted sentence from context-aware spelling corrector

4.4 iv,v) Evaluation of the Context-Aware Spelling Corrector

Then we create an artificial test dataset to evaluate our context-aware spelling corrector based on Word Error Rate (WER) and Character Error Rate (CER), using the original (before character replacements) form of our test dataset as ground truth (reference sentences), and averaging WER (or CER) over the test sentences. WER (Word Error Rate) measures the percentage of incorrectly recognized words in the output compared to the ground truth. It's calculated by dividing the total number of word errors (insertions, deletions, and substitutions) by the total number of words in the ground truth. The result is multiplied by 100 to get a percentage. CER (Character Error Rate) measures the percentage of incorrectly recognized characters in the output compared to the ground truth. It's calculated by dividing the total number of character errors (insertions, deletions, and substitutions) by the total number of characters in the ground truth. The result is multiplied by 100 to get a percentage. The below table represents our Evaluation's result.

Word Error Rate	Character Error Rate
0.047619047619047616	0.09134615384615384

In your case, the WER score of 0.047 and CER score of 0.091 seem relatively low, which could be considered good performance.

5 Conclusion

This project involved a thorough exploration of the functioning of n-grams. We started by gaining a comprehension of n-gram probability and then moved on to building a beam-search decoder from the ground up. This project provided us with the opportunity to work on our individual shortcomings and grasp the various compromises that are involved in a project of this nature. In general, the experience of working on practical aspects such as exploring and testing sentence correction and gaining expertise in NLP libraries such as NLTK was fascinating.