Project Report

Introduction:

This Project was an introduction to n-grams. We specifically had to work with bigrams and trigrams, study how they affect entropy and try to build a context aware spelling corrector from scratch to the best of our ability. We faced this project as a team. Given the nature of this exercise, and how the project builds on itself, it would seem wasteful to split the project into even parts, as we would have group members sitting on their hands while the rest of us were tackling the first few problems and vice versa. Thus, we decided to act as follows: We would gather every two days, to brainstorm and set some goals for the next time. Then, we would work on our own and compare results at the following meet-up, as well as try to improve our code and renew our goals for the next time. This approach was very helpful to us, but it makes it a lot harder to "describe in our report the contribution of each member" as requested. Bottom line, we should all have a satisfactory understanding of all algorithms used, since we all contributed to their creation.

Question i:

For the first question, we had to implement both a bigram and a trigram model on a corpus we would select ourselves, using some kind of smoothing, either Laplace or Kneser-Ney. In order to create our model, we would first need a corpus. We selected and concatenated 5 categories from NLTK's "brown" corpus, adding up a total of 23715 sentences. Since we would later need a test subset to calculate our entropy and perplexity, we split our original corpus in a train and a test set at an 80%-20% ratio. As requested, we then had to replace words that appear less than ten times with a special token *UNK*. Before we did that, however, we lowercased our text. That was in order to account correctly for all n-grams, and since capitalization not only was not an issue for our exercise but would also lower our vocabulary's cardinality. In replacing our out-of-vocabulary (OOV) words, we took a pretty straightforward approach. First, we created a list of all words, then, based on their counts, we created a second list that only contained those considered OOV. Finally, running a nested loop and a search, we replaced the OOV words in our train set with the *unk* token.

To build the different n-gram models, we first had to initialize a Counter object for each of our n-grams as well as all n-grams of a lower order than them. Thus, we also had to include a unigram counter, as it would be necessary in order to calculate the bigram probabilities. The three Counter objects were then trained on our training data, with an added padding of a <start> and an <end> token (or two in the trigram case) at the beginning and the end of each sentence. Since calculating n-gram probabilities would be essential to the project later on, we made two functions which did exactly that for the bigram and trigram probabilities respectively. In those, we used alpha smoothing, yet we set alpha equal to one, so as to have Laplace smoothing as the default setting. A bigram Kneser-Ney smoothing was performed as well. Something to note is that we selected as the output of our functions not the probabilities themselves, but their base two logarithm. The reason we returned the logarithms is because we would much rather have a sum of negative, relatively "large" numbers, than a product that would get infinitely close to zero, as we have more and more probabilities. Not only is it visually better for the reader, it is also much easier for the computer to perform the necessary calculations.

Bigram	I am	The government	Soviet Union
Probability with Laplace	0.0164	0.0012	0.0043
Probability with Kneser-Ney	0.0469	0.0013	0.2231

Table 1 (Bigram probabilities with different smoothing techniques)

Trigram	I am a	The government is	The Soviet Union
Probability	0.0021	0.0008	0.0043

Table 2 (Trigram probabilities with Laplace smoothing)

Kneser-Ney smoothing is essentially a way to redistribute probability from existing n-grams to those which are not found in our corpus, yet could be correct in the English language. In its most basic form, Kneser-Ney smoothing is described as:

$$P_{KN}(w_k|w_{k-1}) = \begin{cases} \dfrac{c(w_{k-1}, w_k) - D}{c(w_{k-1})}, & if\ c(w_{k-1}, w_k) > 0 \\ a(w_{k-1}) \cdot P(w_k), & else \end{cases}$$

Where $a(w_{k-1})$ is a function that uniformly distributes the probability "stolen" by D and is also responsible for keeping the overall probability sum equal to one. Following the steps and improvements in the relevant lecture's study material, we end up having an improved Kneser-Ney method:

$$P_{KN}(w_k|w_{k-1}) = \begin{cases} \dfrac{c(w_{k-1}, w_k) - D}{c(w_{k-1})}, & if\ c(w_{k-1}, w_k) > 0 \\ a(w_{k-1}) \cdot Prev(w_k), & else \end{cases}$$

To calculate the value described above, we had to use two separate functions. A function “prev(x)” that computes the value $prev(w)$, which is essentially the number of unique bigrams that have w as the second word. Also, a “next” function which computes the exact opposite, the number of unique bigrams with the form (w, *). Using those two, as well as setting:

$$Prev(w_k) = \frac{prev(w_k)}{\sum_{v \in V: c(w_{k-1}, v)=0} prev(v)}$$

$$a(w_{k-1}) = \frac{D}{c(w_{k-1})} \cdot |\{w \in V: c(w_{k-1}, w) > 0\}|$$

It was a simple matter of putting everything together to calculate the bigram probability with Kneser-Ney smoothing.

Question ii:

For this question, we had to estimate the language cross-entropy and perplexity of the two models on our test subset. First, we cleared the test subset from OOV words, based on the list we created above, to clean the train set. Then, after padding each sentence with the necessary <start> and <end> tokens, we ran a loop on each bigram (or trigram respectively) calculating its logarithmic probability, and then summing them. We divided that sum with the total number of n-grams, including <end> tokens as instructed, which gives us the cross-entropy of our model on the test set. For the perplexity of our model, we simply needed to raise 2 to the power of our calculated cross-entropy. It is important to note, that we started iterating from the second word for our bigram counter so as not to include probabilities of the form P(<start>|…). Similarly, we started the trigram loop from the third word, to avoid needless calculations of the P(<start1>|…) and P(<start2>|…) probabilities. For the trigram calculation, we also ended the loop one word earlier, because P(<end2>|…) should not affect the entropy, as P(<end1>|…) already predicts whether a word is at the end of a sentence or not.

Due to a very long computation time because of the several iterations Kneser-Ney mandates, we decided on using Laplace smoothing in estimating the entropy and perplexity, even though it is clear that Kneser-Ney should procure better results. To emphasize that point, we calculated both the Kneser-Ney and Laplace smoothing models’ entropies and perplexities on a much smaller test set which would not require as much time. The difference in cross entropy and perplexity of the bigram models using the different smoothing techniques is evident in the table below:

Bigram model	Laplace smoothing	Kneser-Ney smoothing
Cross Entropy	7.687	6.253
Perplexity	206.084	76.262

Table 3 (bigram model cross entropy & perplexity)

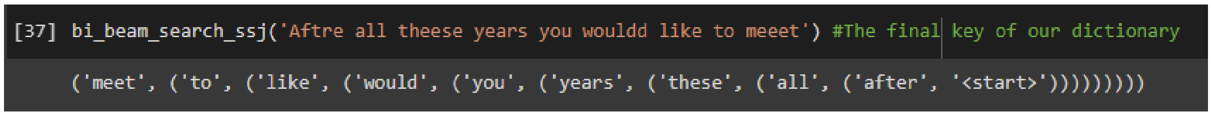
The results of our two models were actually quite surprising. The trigram model, which should perform better since it takes into account more information on deciding the probabilities, actually returned a higher entropy. This counter-intuitive result, came as a surprise at first. It is however, not as convoluted after some thought. The small scale of our corpus, led us to have a large number of words that appeared less than ten times. Thus, disregarding all those words and replacing them with *unk* tokens, lessens our vocabulary even more, thus raising the uncertainty of our context-aware model, as *unk* has no specific context to begin with. Another thing to note is that since the “brown” corpus from NLTK contains several vastly different sub-corpora, the context of similar words in each one could be different and “confuse” our trigram model, given how it is much more context-aware.

Laplace Smoothing	Bigram Model	Trigram Model
Cross Entropy	7.789	9.686
Perplexity	221.243	823.750

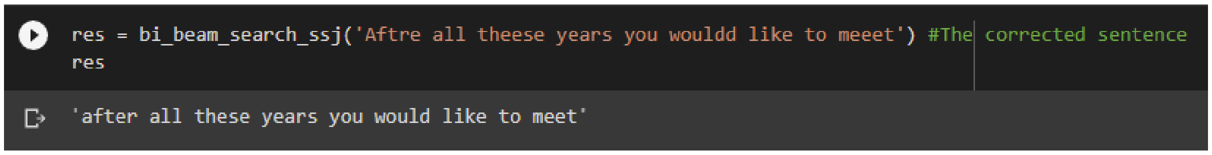
Table 4 Bigram and Trigram cross entropy - perplexity

Question iii:

This was by far the most challenging question of this assignment. 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: For each word in the given sentence, we first estimated the k nearest words in our vocabulary based on Levenshtein distance. We set k to be user defined if so needed, but equal to 30 by default. Out of those k words, the c (also a hyperparameter initially set to 3) which are estimated to be more contextually fitting are stored and will be used as possible given words in computing the next n-gram. The probability calculated in each step will be the total sum of the negative logarithms of the probabilities of all previous steps required to reach that specific word. Of course, we want the minimum sum, which is the highest probability. This way, we know that the “path” selected by our beam search decoder is actually the one with the highest probability overall. In order to achieve that, we created a dictionary with the words used so far as keys and the total probability sum as value. We use a variable to hold the c possible words for each step, initializing it with the <start> token, based on both the n-gram probability and the inverse of the Levenshtein distance for the sequence. We update that variable in each loop and hold its previous values as our dictionary keys in the form of tuples. Because of that, as we iterate through our sentence, the dictionary keys become a series of nested tuples, each one having the final word for its respective iteration as the first element, and a nested tuple holding all previous words that are “attached to it” as the second element. After that, we select the key that corresponds to the smallest total probability and “unzip” it through a simple loop. The inner workings of the function used can be shown below:


```
[37] bi_beam_search_ssj('Aftre all theese years you wouldd like to meeet') #The final key of our dictionary

     ('meet', ('to', ('like', ('would', ('you', ('years', ('these', ('all', ('after', '<start>')))))))))
```

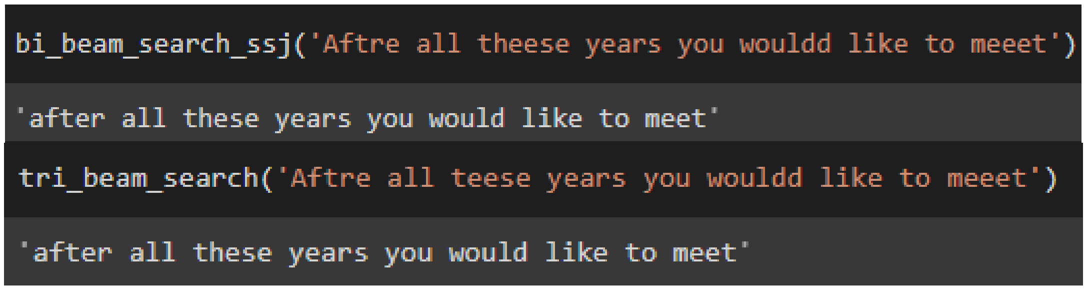


```
res = bi_beam_search_ssj('Aftre all theese years you wouldd like to meeet') #The corrected sentence
res
```

```
'after all these years you would like to meet'
```

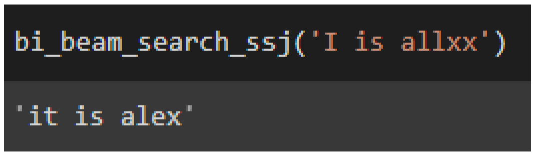
Figure 1 Steps of the sentence correction process

We also constructed a trigram beam search decoder, selecting the best possible word in each step as we iterate through the sentence, leading to a single path, based uniquely on the most prevalent choice per loop. The 2 most recent selected words are saved in a list to be used in order to calculate the trigram probability for the next word, and each selected word is also saved in another list to form the final sentence. As above, k options are chosen through Levenshtein distance, where k is user defined and set by default to 30. In the following figures our sentence correction results are presented, from both our models for the same incorrect sentence.



```
bi_beam_search_ssj('Aftre all theese years you wouldd like to meeet')
'after all these years you would like to meet'
tri_beam_search('Aftre all teese years you wouldd like to meeet')
'after all these years you would like to meet'
```

Figure 2 Sentence correction of the models



```
bi_beam_search_ssj('I is allxx')
'it is alex'
```

Figure 3 Showcasing context awareness

Conclusion:

This Project was a comprehensive dive into the way n-grams work. Working from understanding and implementing the n-gram probability, to constructing a beam-search decoder from scratch helped each one of us work on their respective weaknesses and realize the different trade-offs that have to be made in a project such as this. For example, the Kneser-Ney model, which yielded significantly better cross-entropy and perplexity scores, was so time consuming in its implementation, that it was essentially a hassle to work with. Overall, it was interesting to work on a more practical approach, researching and experimenting with sentence correction and getting the hang of working with NLP libraries such as NLTK.