Indian Institute of Technology Gandhinagar

Team nltk.tokenize

CS 613: Natural Language Processing

Assignment 2: Language Modelling

Hetvi Patel (20110076)

Kevin Shah (20110096)

Lipika Rajpal (20110102)

Chandrabhan Patel (22110057)

Vraj Shah (22110292)

Sujal Patel (22110261)

Dewansh Singh Chandel (22110072)

Suteekshna Mishra (22110266)

Vedant Chichmalkar (22110282)

INDEX

Tasks

1.1 Question 1:

Take the data from here.

1.2 Question 2:

Use the NLTK sentence tokenizer. (See section 1.4)

1.3 Question 3:

Split the corpus randomly into 80% and 20% ratio (sentences, aka documents).
(See section 1.4)

1.4 Question 4:

Train the LM on the 80% split and validate the remaining 20% split.

Answer (Task 1 + 2 + 3 + 4):

Preprocessing: Removed hyperlinks, Hashtags and non-alphanumeric characters(emojis, etc.) from sentences using RegEx.

Here is the regex pattern:

```
regex_pattern_for_HTTPlinks = r'http[s]?:\S+'
regex_pattern_for_WWWlinks = r'www\S+'
regex_for_removing_hashtags = r'#\S+'
all_characters_expect_alphanumeric = r'[^A-Za-z0-9\s]'
```

Tokenization: *nltk.sent_tokenize* was used to tokenize corpus into documents(sentences). As a part of pre-processing, tokens were formed after converting sentences into lowercase and all non-alphabetic characters were removed.

```
  comment = comment.lower()
  comment = comment.strip().split(" ")
```

For processing the data to convert it from raw form to a state that can be fed as an input to the language models, we did the following:

- We extracted all the comments from the raw database and stored them in a list of lists called *comments*. This list contains each sentence tokenized by the *sent_tokenise* function of the *nltk* library.

- Then, we iterated through each sentence to remove special characters (including everything except alphanumeric characters), links, hashtags in the comments. Then we split each sentence into words/tokens using white space as a delimiter.

- After preprocessing each sentence, if the length of the sentence is non-zero we append it to a list called *'processed_comments'*.

- Then, we leverage the numpy library's shuffle function to split *processed_comments* into train and test sentences at a ratio 80:20 randomly. Using the shuffle function, we shuffle the dataset and choose 80% of the sentences from the shuffled *processed_comments list* to create a train dataset *processed_corpus* . We have set a manual seed of 55 to ensure reproducibility when the code is run multiple times.

***everygram_creator** :* We also made a function *everygram_creator* which takes in two arguments *order* which is the value of n of the n-gram and *processed_corpus* which is the data. The purpose of this function is to generate all n-grams of length less than or equal to the specified *order* for a given processed text corpus. Firstly, we created a deep copy of the corpus and named it *copy_corpus*. Then all the sentences in *copy_corpus* are padded with start ("<s>") and end ("</s>") tokens according to the *order*. This padding is necessary to ensure that shorter n-grams (e.g., bigrams and unigrams) can be extracted from the sentences without causing index errors.

The code initialises empty sets for *n_grams* and *vocab*. *n_grams* will store the n-grams generated from the corpus, and *vocab* will store the unique tokens present in the corpus. It iterates through *copy_corpus* to collect all unique tokens into the vocab set.

Next, it iterates through each sentence in *copy_corpus* and extracts all possible n-grams of length less than or equal to the specified *order*. For each n-gram, it forms subgrams of increasing lengths and appends them as tuples to the *grams* list. These n-grams are collected for each sentence and stored as a tuple within the *n_grams* list.

Finally, the function returns two values: a nested tuple containing all the n-grams and the set of unique tokens in the corpus (*vocab*).

1.5 Question 5: 40 Pts.

Train the following LMs and report the respective perplexity scores. Perplexity will be computed as an average over all the sentences.

a. Unigram [10 Pts.]
b. Bigram [10 Pts.]
c. Trigram [10 Pts.]
d. Quadgram [10 Pts.]

Note that No existing NLP library is allowed to implement the above language models. All these language models have to be implemented by hand, and they should strictly follow the definitions given in the class. In case you have reused any existing code or ChatGPT-like conversational agents for code generation, it might lead to ZERO marks.

Answer:

NOTE: For calculating perplexity scores, we need to calculate the conditional probability of each N-gram given by:

$$P\left(w_1 w_2 \cdot \cdot \cdot \cdot \cdot w_n\right) = \prod_i P\left(w_i | w_1 w_2 \cdot \cdot \cdot \cdot \cdot w_{i-1}\right)$$

Instead of calculating probability, we have used logarithmic probabilities with base 2 due to the following reasons:

> The reason for using "log_prob" instead of "prob" is to mitigate issues related to underflow and overflow when dealing with probabilities. When calculating perplexity, it's necessary to add logarithmic probabilities (log_probs) rather than multiplying probabilities. This approach is numerically more stable and computationally faster because it helps avoid very small or very large values that can lead to numerical instability.

----

1. **Unigram** : A class which goes by the name *UnigramModel* was made and all functions required to find the perplexity of the model were defined in it. To implement the unigram model on the given dataset, an empty dictionary *unigram_freq* was made to store the frequencies of each unigram and a total token counter. In the *fit* function, each token from each sentence is taken and its total count is updated and the frequency is stored in the dictionary *unigram_freq*.
   Next, a function *unigram_probability* is made to calculate the logarithm(with base 2) of probability of tokens by making use of the frequency and total number of tokens in the corpus.

   $$log_2(P(token)) = log_2\left(\frac{Frequency(token)}{Total\ no.\ of\ tokens}\right) \ldots\ldots\ldots\ldots\ldots 1$$

   The function *perplexity* was made to calculate the metric of comparison between m models. For each sentence in the test dataset, the function calculates probability of each token, multiples them and takes its reciprocal value. In case the probability becomes 0, it is assigned a value 10^{-25}. The function then returns the mean of all perplexity values.

   $$Perplexity = \left(\prod_{token} \frac{1}{P(token)}\right)^{-\frac{1}{N}} \ldots\ldots\ldots\ldots\ldots 2$$

2. **N-gram :** A class *ngram* was made which takes *order* as a parameter. *order* can be 2, 3, 4 for bigram, trigram and quadgram models respectively.
   The function *fit* takes the *processed_corpus* and stores the frequencies of each token. This function also tracks the unigram counts.

   A function *log_prob* is made to compute the log probability of a sentence by iterating through its n-grams and calculating conditional probabilities for n-grams with lengths greater than or equal to 2 and unigram probabilities for unigrams. If a sentence contains an n-gram with zero probability, a very small log value is assigned to prevent errors. Here, we have used log probability instead of probability to avoid numerical underflow hence ensuring numerical stability. Log Probability was calculated as follows:

$$log_2(Probability) = log_2\left(\frac{Frequency(n\ gram)}{Frequency(n-1\ gram)}\right) \ldots\ldots\ldots\ldots\ldots\ldots 3$$

The *perplexity* function calculates the perplexity scores of the test dataset by first computing the log probability for each sentence and then converting back to the perplexity scores. The function returns the average of all the perplexity values.

1.6 Question 6: 40 Pts.

Use the Laplace smoothing on the above LMs and compare the change in the perplexity with and without smoothing.

Note that No existing NLP library is allowed to implement smoothing techniques. All these language models have to be implemented by hand, and they should strictly follow the definitions given in the class. In case you have reused any existing code or ChatGPT-like conversational agents for code generation, it might lead to ZERO marks.

Answer: Smoothing was applied on all models (unigram,bigram, trigram and quadgram).

A class *UnigramSmoothedModel* is defined which is very similar to the *UnigramModel* class described in question 5 above which was made to make the unigram model without smoothing. Here we have defined another set called *vocab* which will store all the unique words in the corpus, thus making the dictionary. In *UnigramSmoothedModel* the function *fit* also adds words in the *vocab* set in addition to its previous tasks. Also, the *unigram_probability* function this time does not calculate the probability by the usual formula mentioned above (Equation 1) instead it uses the smoothing technique (Add-1 smoothing).

$$P_{Add-1}\left(w_i\right) = \frac{c\left(w_i\right) + 1}{N + V} \ldots\ldots\ldots\ldots\ldots\ldots 4$$

Here, V is the size of vocabulary of the dataset on which the model is trained. It is the size of the set *vocab* that stores all unique token of all sentences during training.

The *perplexity* function is made similarly and the perplexity value of the data on smoothed unigram models is calculated.

To apply smoothing on bigram, tribram and quadgram models, a class *ngram_smoothed* was made which is again similar to the class *ngram* described in question 5 above. Difference comes in finding the probability in the *log_prob* function, where instead of using the usual formula, smoothing is used.

$$log_2\left(P_{smoothed}\right) = log_2\left(\frac{Frequency(n\ gram) + 1}{Frequency(n-1\ gram) + V}\right) \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots 5$$

And hence the perplexity values of bigram, trigram and quadgram were calculated by passing the *n* values in the class *ngram_smoothed*. The *ngram_smoothed* class also accounts for the unigram model when *n = 1* is passed.

1.7 Question 7: 20 Pts.

Write a justification with your observations: When smoothing was not considered and where the smoothing was considered

Answer: As we can see from the *table 1*, perplexity scores on test dataset for all language models (trained on train dataset '*processed_corpus*') were coming out to be infinity. This is because there were many n grams (uni, bi, tri and quad) in the test dataset that were not occurring in the training dataset. As a result, according to the model, their probability of occurring was 0. This led to zero in the denominator of the perplexity for many sentences. Hence, the average score of perplexity became infinity.

After applying the laplace smoothing algorithm, we can see from the *table 2* as well as from the graph(Fig 2), that the average perplexity scores of the test dataset are finite. Even though these scores are very high, they depict that the N-gram models are distributing the probability masses to those n-grams that do not occur in the train dataset. The perplexity drop from infinity to finite value shows that the performance of the language models has become better.

However, we can see that the perplexity values are not following the ideal trend of declining as the order of n-gram models increases. We can see that the perplexity values have increased from 7.56×10^{7} in bigram model to 6.46×10^{12} in trigram model. This shows that the models are suffering from the ‘curse of dimensionality.’ As we increase N. the occurrence of n-grams becomes more and more sparse. Here, the model is not able to compensate for the sparsity of n-grams(or, missing n-grams). Sparsity overpowers the effect of smoothing and we see an unepected trend in perplexity values.

Perplexity Values:

- **Without Smoothing**

	Train Data	Test Data
Unigram	14255. 95	inf
Bigram	3965. 54	inf
Trigram	1834. 84	inf
Quadgram	1817. 42	inf

Table 1

- **With Smoothing**

	Train Data	Test Data
Unigram	1.02×10^{4}	1.76×10^{4}
Bigram	2.1×10^{7}	7.56×10^{7}
Trigram	8.74×10^{11}	6.46×10^{12}
Quadgram	3.76×10^{16}	5.67×10^{17}

Table 2

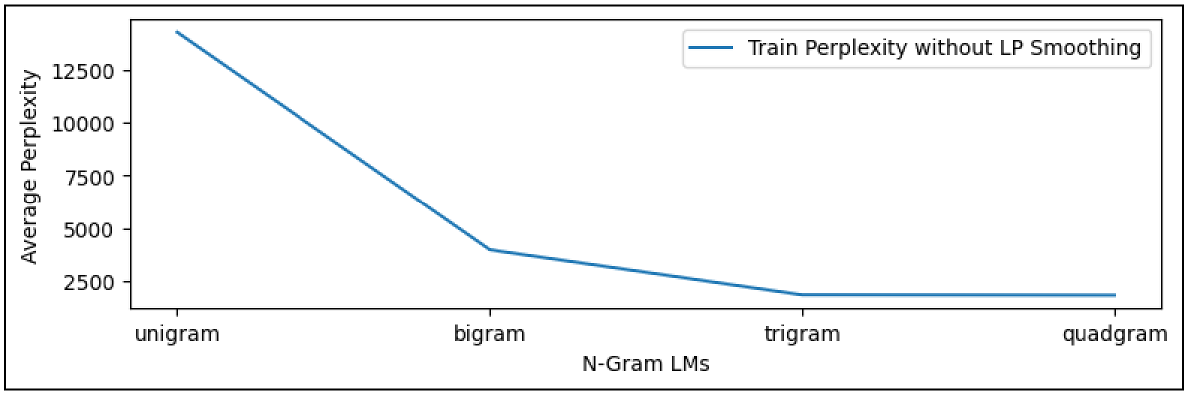

Figure 1 - Variation in perplexity on increasing n in train data without LP smoothing

The Figure 1 above, shows variation in average perplexities with increasing n on train dataset without smoothing. This shows that as we increase n the perplexity decreases and thus the model becomes better. As the model is now able to capture more contextual information about the text, making them better models of the language. Increasing n also ensures reduced ambiguity, in predicting the next word. An unigram model will simply predict the most common words and bigram will be able to predict words in context to a given previous words and hence will be able to perform better than unigram models. Similarly, the trigram model will have 2 context words to predict the next word and thus will be able to predict more accurately with a better context. Hence, as we increase n the model becomes more accurate and thus the perplexity decreases.

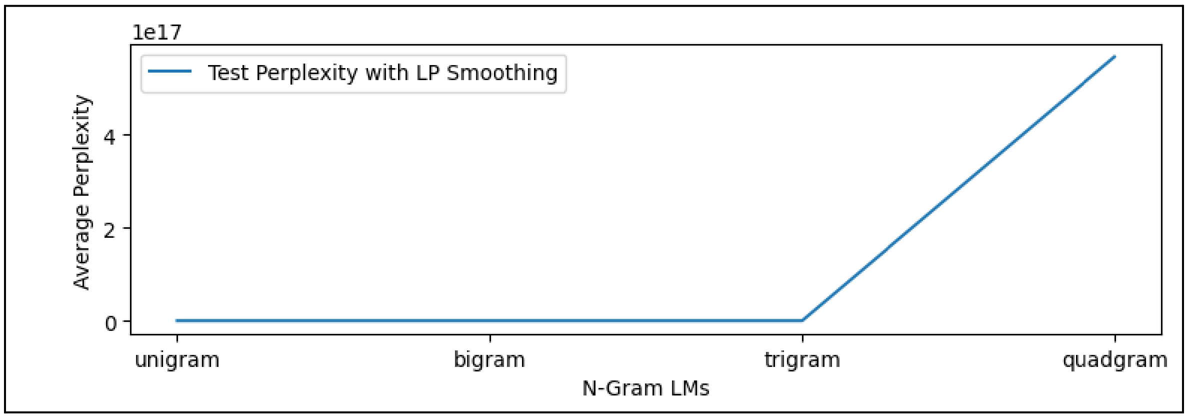

Figure 2 - Variation in perplexity on increasing n in test data with LP Smoothing

As we can see from the below graph(Figure 3), after applying Laplace smoothing, the average perplexity scores increase on the training dataset. This can be explained by the fact that as the language models are redistributing probability mass over the samples, the probabilities of more-occurant events is decreasing. Due to this, the model becomes less certain about many events and hence, we can see an increase in perplexity values.

The sparsity of the dataset fuels this situation as the probability mass has to be distributed to more and more missing n-grams. This also leads to an unexpected increasing nature of perplexity scores as N increases. As the quad-grams are sparser(lesser in number) as compared to trigrams, probability mass gets flatter and the model gets more confused in drawing a distinction between many n-grams, resulting in increased perplexity. In conclusion, these two factors result in increased perplexity values on the training dataset.

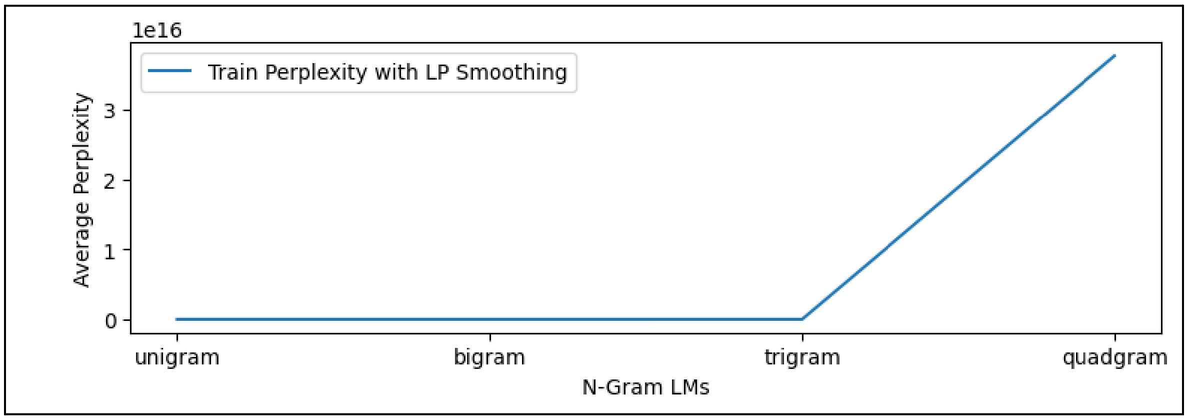

Figure 3 - Variation in perplexity on increasing n in train data with LP Smoothing

1.8 Question 8 [BONUS]: 10 Pts.

Choose any 2 other smoothing of your choice (Additive, Good Turing, or Kneser-Ney; and Train the same n-gram LMs) and write your understanding of using different smoothing techniques.
Note that No existing NLP library is allowed to implement smoothing techniques. All these language models have to be implemented by hand, and they should strictly follow the definitions given in the class. In case you have reused any existing code or ChatGPT-like conversational agents for code generation, it might lead to ZERO marks.

Answer: Additive smoothing is quite similar to Laplace smoothing with the slight change that instead of 1, a general factor 'alpha' is chosen (typically small in value). It provides a better perplexity than Laplace smoothing as Laplace smoothing can often give too much weight to bigrams not appearing in the corpus. In contrast, additive smoothing is more appropriate in light of these sensitivities.

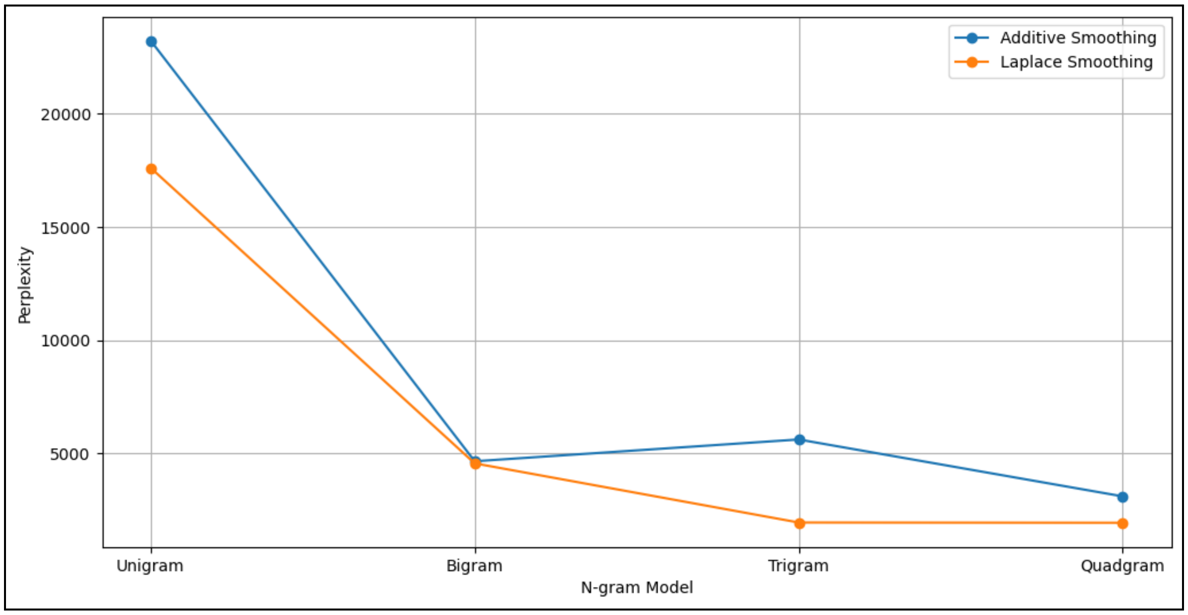

Figure 4 - Variation in perplexity on increasing n and applying Laplace Smoothing on train data

Good-Turing smoothing is a statistical method used in language modelling. It begins by counting the occurrences of each event or item in a dataset, creating a frequency table. The main idea behind this technique is to redistribute the probability mass assigned to n-grams that occur a certain number of times (denoted as "r") to n-grams that occur one less time (i.e., "r-1"). This redistribution helps account for unseen or rare n-grams in the test data, ultimately improving the model's performance.

<u>Perplexity Values</u>:

	Good Turing Smoothing	Additive Smoothing
Unigram	11629.79	23216.11
Bigram	73.46	4660.14
Trigram	10. 68	5621.26
Quadgram	1. 82	3111.55

Table 3

Submission

2.1 Question 1:

Submit your code (GitHub) or colab notebook with proper comments to this link.

Answer: The Colab notebook containing the code can be accessed by clicking on this link.

a. Make sure the individual contribution is appropriately added.

Individual Contributions:

Name	Contributions
Hetvi Patel	Assisted in Q6, 7. Helped in adding formatting the report and editing it.
Kevin Shah	Contribution in Qs 6 & 7, Report Formulation and editing
Lipika Rajpal	Assisted in creation of all N-grams, contributed to Laplace smoothing of all N-grams, preprocessing of the dataset, various answers in report writing
Chandrabhan Patel	Pre-Processing and assisted in Report Writing
Vraj Shah	Made N-grams along with additive smoothing
Sujal Patel	Helped in Additive smoothing
Dewansh Singh Chandel	Made the structure of generalised N-gram model
Suteekshna Mishra	Pre-Processing and assisted in N-grams
Vedant Chichmalkar	Performed the Good Turing smoothing on all n-gram models.