The Word-Weavers

NLP Assignment 2- Report

Github- https://github.com/RahulVC02/nlp-a2-N-grams.git

Preprocessing the Reddit Comments Data

Given a huge raw dataset of reddit comments, the first step for any NLP pipeline is to pre-process the data so that we have a structured list of sentences. After observing the raw comments we performed the following steps before tokenizing into sentences:

1) Removed the comments from the corpus which were deleted from Reddit.
2) Removed URLs from the comments.
3) Converted all the comments into lowercase.
4) Removed all symbols except .,?! Because we will need these punctuation marks for sentence tokenizer.

We do all the above steps using parallelization by utilizing the joblib Parallel(n_jobs=8) which reduces the pre-processing time drastically. Now we concatenate the list of sentences using a full stop in between to make a complete corpus string. Here comes the second pre-processing step:

1) We remove .,?! As we didn't remove them earlier.
2) We realized that two random usernames are used a lot in all the comments ('x000d' and 'x200b') so we removed all occurrences of them as they didn't give any context.
3) We also removed extra spaces (more than 1).
4) Again after parallelizing this pre-processing, we removed the empty sentences, and put them in a panda series which is our corpus.

Even after such involved pre-processing and data cleaning, one of the most important steps is executed on initializing the NGram Language Model class. We also need to add <s> and </s> which are start of sentence and end of sentence tokens respectively. Depending on the value of N, we add these tokens at the start and end of each and every sentence.

Why Are We Reporting Log Perplexity

First we tried to calculate the perplexity of the individual sentences and took the average of all to find the total perplexity. It turns out that the probabilities of n-grams are very less and their product of them is even lesser. In Python, due to the limitation of representing floating point digits it just gives 0 if it is a very less number. Perplexity is the inverse of the product which is not deterministic in this case (1/0). Due to this we calculate the log(perplexity), because it converts the perplexity expression into the sum of log(probability). Advantages of this transformation are:

1) First of all we can say that the trends will be the same as perplexity as logarithm function is a monotonically increasing function. That's why we have displayed the log(perplexity) trends to better understand the difference in values.
2) Log(number) where the number is very close to 0 gives a significant negative number and then we sum those numbers. Here we remove the chances of underflow and make calculations faster because addition is faster than multiplication.

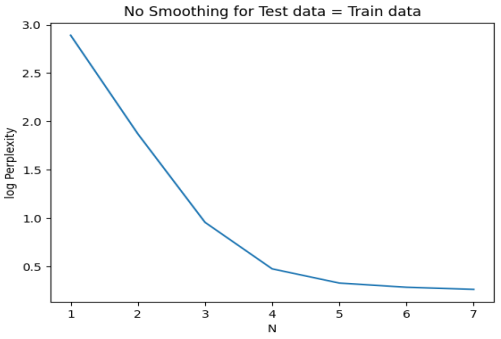
To check the perplexity value we can just raise the log(perplexity) to the power of 10, but the trends are better visible in log(perplexity).

See the example for better understanding:

```
No. of Sentences in Train Data :  235762
No. of Sentences in Test Data :  235762
Smoothing :  None
Unigram Log Perplexity:  2.8932259506731124
Unigram Perplexity:  782.0345684000403
Bigram Log Perplexity:  1.8888649666089294
Bigram Perplexity:  77.42210350233003
Trigram Log Perplexity:  0.9834132205945932
Trigram Perplexity:  9.625276643998184
Quadgram Log Perplexity:  0.49298941944774927
Quadgram Perplexity:  3.111640528460926
Pentagram Log Perplexity:  0.3384700055681343
Pentagram Perplexity:  2.1800678261869577
Hexagram Log Perplexity:  0.2926395193667191
Hexagram Perplexity:  1.9617312906623092
Septagram Log Perplexity:  0.2701928531189098
Septagram Perplexity:  1.86291420003099
```

Without Smoothing

Adding **no smoothing** gives predictable results when train data is the same as test data, where perplexity decreases as we move from unigram to bigram to trigram, etc. This validates the working of our n-gram models. However, when we make the test set separate from the training set, perplexity comes out as **infinity**, which suggests for some sentences, the estimated probability came as **0**, which is understandable as not all n-grams of the test set are present in the training set. However, it is interesting to note that the probability is not 0 in unigrams as long as the training data contains the entire vocabulary for custom test sets.



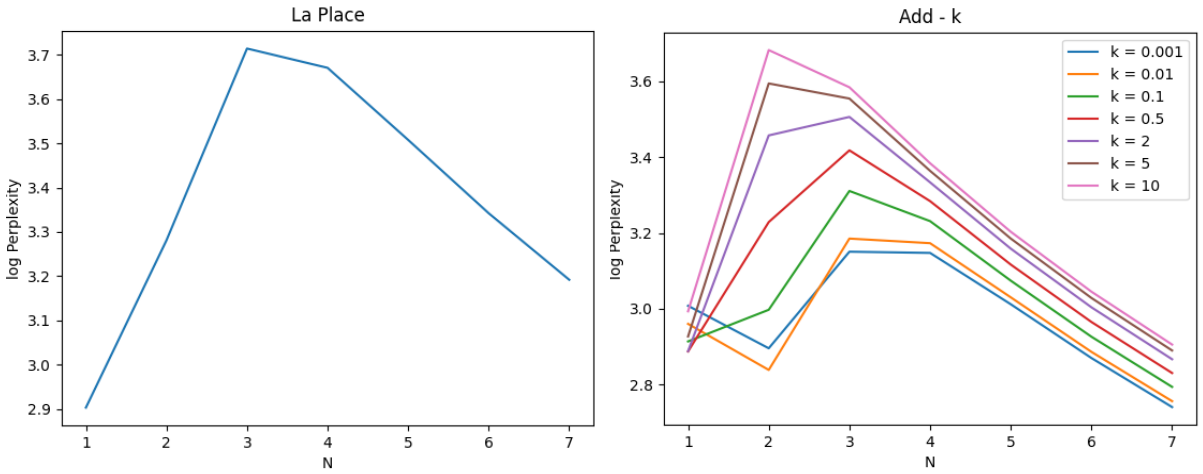
No-smoothing with different test-train data

```
                                                    > py
No. of Sentences in Train Data :  188609
No. of Sentences in Test Data :  47153
Smoothing :  None
Unigram Log Perplexity:  Infinite
Bigram Log Perplexity:  Infinite
Trigram Log Perplexity:  Infinite
Quadgram Log Perplexity:  Infinite
Pentagram Log Perplexity:  Infinite
Hexagram Log Perplexity:  Infinite
Septagram Log Perplexity:  Infinite
PS C:\Users\singh\Desktop\nlp-a2-N-grams>
```

With Smoothing

Laplace and Add-K Smoothing

With Laplace smoothing, and train:test = 80:20 split, we were able to obtain finite perplexities for every ngram model.



With **Laplace** and **Additive** smoothing, a problem was observed while the perplexities were not infinity for other models; the perplexity for bigrams and trigrams was larger than that of unigrams. This is because the conditional probabilities in n-gram, which were otherwise very large for n-grams occurring both in train and test data, are reduced drastically in Laplace smoothing, which made probabilities of sentences with LaPlace smoothing relatively very low and, in turn, increased the perplexity. This is because laplace smoothing adds 1 in the numerator and size of vocabulary in the denominator. Now, the original numerator is already small because the test set has many phrases that never appeared in the train set.

Another important observation is that the training set is not the best representative of the whole corpus because reddit comments do not have greater context as compared to some general text like Wikipedia- it isn't necessary at all for two comments to have similar patterns of bigram, trigram sequences while they can still be made of the same words because the English vocabulary is common to both. Moreover, the dataset size is quite small. Statistical models like ngrams sometimes need larger and more general text datasets like the Wikipedia corpus to report better results.

One intuitive explanation for Unigram perplexity being smaller than bigram, trigram perplexity is that both train and test data share a similar vocabulary distribution (they are made up of a similar set of words), however their bigram and trigram sequences/patterns are quite different. On smoothing, we add a |V| term in the denominator for all bigram and trigram probabilities. Because the bigram and trigram sequence distribution in train and test data is very different, a majority of the bigram and trigram probabilities turn out to be 1/(|V|+count(w_i) while the unigram probabilities are greater because while the denominator is still = 2|V|, the numerator can be very large because the isolated word will appear many times in train and test, while the bigram/trigram patterns need not. So, the unigram probabilities turn out to be higher leading to lower perplexity scores.

Only with a very low value of k (to the order of k = 0.01) were favorable results obtained for even bigrams, but for other models, perplexity was still greater than that of unigrams. For smaller values of k, we observe better log-perplexity scores. An intuitive explanation for this is that *add-k* adds k to the numerator and k*V to the denominator, where V is the size of the vocabulary. For larger values of k, the increase in the denominator can outweigh the increase in the numerator. This significantly reduces the probability values leading to higher perplexity scores.

Good-Turing Smoothing

With **Good Turing** smoothing, meanwhile, gave relatively suitable results from other models as the perplexities decreased as we moved from unigram to larger models for both cases where test data was equal to training data and distinct from it. We tried two methods-

1) Unconditional probability for ngrams (exactly as taught in class)
2) Conditional probability for ngrams (additional experiment done by us)

In the first method of using unconditional Good-Turing probabilities, we faced no problems as we were always dividing the Good-Turing count by the total number of n-grams in the training corpus as taught in class, and applied the following rule-

```
def goodTuring_ngram_count(ngram, count_gram_dict, freq_dict):
    c = count_gram_dict.get(ngram,0)
    if c > 10:    #done for maintaining consistency after the count becom
        return c - 0.75
    elif (c == 0):
        return freq_dict[1]
    else:
        return (c+1)*freq_dict.get(c+1,0)/freq_dict[c]
```

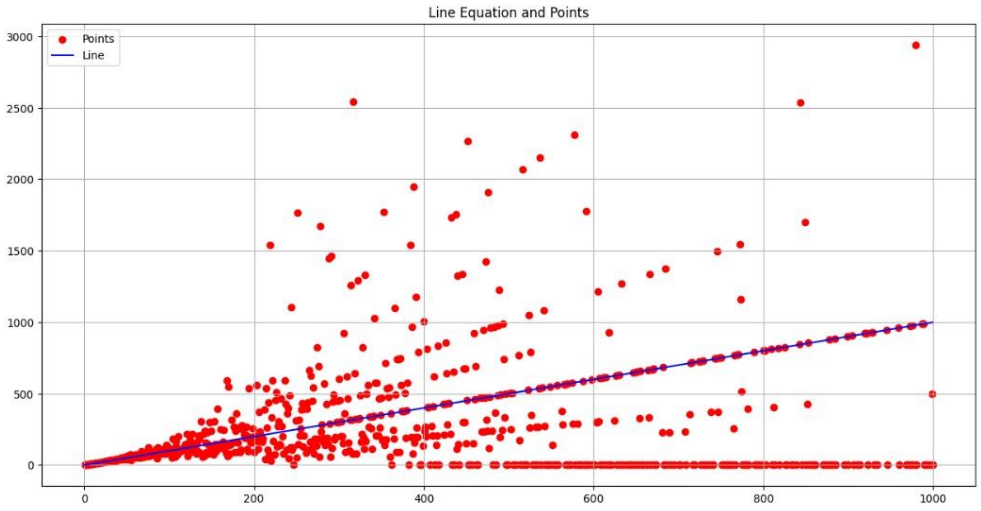
Also, we used the empirical formula of c* = c - 0.75 for large c, and faced no zero count issued in 0=<c<=9.

However, there was a more severe problem with conditional Good-Turing probabilities as, in some cases, the conditional probabilities came greater than one due to inconsistencies in frequency distribution where the estimated Good-Turing count for an ngram came out to be larger than the estimated Good-Turing count for the (n-1)gram and in many cases the estimated count for c = x came out to be 0 because of N_(x+1) = 0.

Below is the plot of smoothed Good-Turing count transformations with the x-axis as the c value and the y-axis as c*, and as apparent, there are many inconsistencies with zero values for Good-Turing counts leading to zero probability values and infinite perplexity.

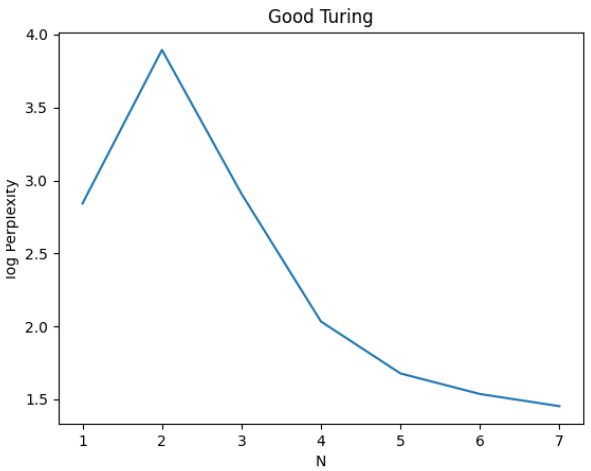
To solve this, there was only one solution, which was to interpolate(learn a functional formula/equation of a line) for the Good-Turing smoothed counts before using them, as

represented by the blue line in the plot below.

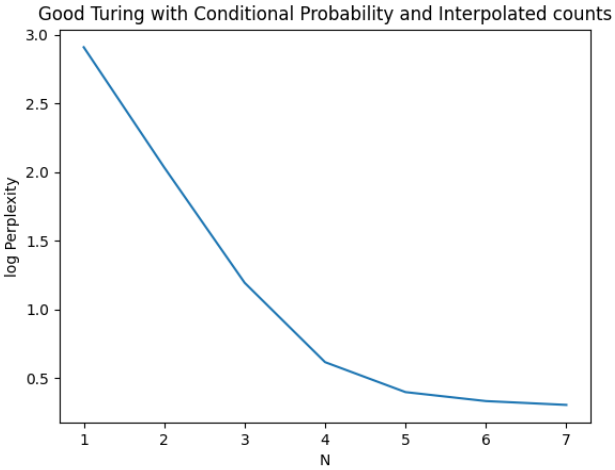


The below plots are from using Good Turing Smoothing to calculate average perplexities with two methods. The left one uses the Good Turing approximation similar to what was studied in the class, without the above interpolation and without using conditional probabilities to make calculations of Good Turing probabilities similar to what was done for other models.

On the other hand, the model on the right uses conditional probabilities to calculate the total probabilities of the sentence, and since it was having trouble for some particular cases of count, the interpolated count has been used for the same. The interpolation function was experimental in nature and independent of n, and developed using only unigram, but the same function when applied to other models as well it gave suitable results.



Obtained by using natural probabilities with no interpolation



Obtained with both interpolation and using conditional probabilities

Contributions

1. Dhairya Shah - Worked on complete pre-processing, and formulated the documentation. Also helped in implementing a few methods in the N-gram language model class.
2. Yashraj Deshmukh - Worked on implementing smoothing methods for ngram models. Experimented with using an interpolation function for more suitable values for Good Turing counts.
3. Shubh Singhal - Worked on implementing the different smoothing methods to apply to the different n-gram models, including making the methods compatible with the different language models.
4. Rahul Chembakasseril - Worked on ideating for preprocessing, implementing some methods in the ngram pipeline.
5. Bhavesh Jain- Worked on implementing the functions in utils.py which were helpful in forming the count dictionaries (frequency distributions).
6. Sukruta Midigeshi - Worked on optimizing the preprocessing pipeline via parallelization, and implemented few functions in the ngram pipeline.
7. Chirag Sarda - Worked on plotting some of the different perplexity results and making observations out of them.
8. Arun Mani- Worked on the analysis of results and drawing logical inferences from the observed results.
9. Mihika Jadhav - Worked on the documentation and implemented some code modifications to incorporate the smoothing techniques.