Natural Language Processing

Prodromos Kampouridis MTN2203
University of Piraeus
National Center for Scientific Research "DEMOKRITOS"
Athens, Greece

3 April 2023

Professor

Efstathios Stamatatos,

Dept. of Information and Communication Systems Engineering,

University of the Aegean

ABSTRACT

This project was assigned as part of the coursework for the Natural Language Processing course in the Master's program in Artificial Intelligence at NCSR "Demokritos" and the University of Piraeus. The assignment focuses on the concepts of tokens, types, Zipf's Law, n-grams (including bigrams and trigrams), add-k smoothing and perplexity. The behavior and performance of these concepts are analyzed and their results are interpreted and visually presented.

KEYWORDS

nltk; spacy; berttokenizer; zipf's law; ngrams; bigrams; trigrams; add-k smoothing; perplexity;

A. TOKENS, TYPES & ZIPF'S LAW

The first part of the assignment explores the application of three tokenization methods - nltk.word_tokenize(), en_core_web_sm English language model in spaCy, and BertTokenizer in a file containing short news texts from the Wall Street Journal. The number of tokens and types found using each method is reported, along with a randomly selected sentence and its list of tokens. The 20 most frequent tokens are listed in a table with their occurrence probabilities and the product of their position and probability. The percentages of tokens appearing exactly once, twice, and three times are compared to Zipf's Law predictions. The constant A in Zipf's Law is determined for the given set of texts and a plot is created to compare the predictions of Zipf's Law with the actual measurements in the texts.

IMPORTANT FUNCTIONS

The *print_top_tokens* function first creates a dictionary called *freq_dict* to store the frequency of each token in the tokens list. It then iterates through each token in this list and updates the frequency count for that token in the dictionary. Furthermore, the function computes the probability of occurrence for each token by dividing its frequency by the total number of tokens. It also computes the product of the rank (position in the list) and probability of occurrence for each token. The function then sorts the tokens in descending order of frequency and prints out a table showing the top 20 tokens along with their rank, count, probability of occurrence and product of rank and probability.

The *print_token_percentages* function creates again a dictionary in a similar way to store the frequency of each token in the tokens list. Moreover, the function computes the number of tokens that appear exactly *n* times for *n = 1, 2 and 3*. It achieves this by creating a new dictionary called *n_counts* and iterating through the values (frequencies) in the *freq_dict* dictionary. For each frequency value, it checks if that value is already a key in the *n_counts* dictionary. If it is not, it adds that key to the dictionary with a value of 1. If it is already a key, it increments its value by 1. The function then computes the percentages of tokens that appear exactly n times by dividing the number of tokens that appear exactly n times by the total number of different tokens (types) and multiplying by 100. It also computes the predicted percentage according to Zipf's Law for each value of n. Finally, the function prints out the results.

The *zipfs_law_plot* function creates, in the same way as the previous two functions, a dictionary to store the frequency of each token in the list. After that, the function sorts the tokens in descending order of frequency and computes the total number of tokens by summing up all the frequency values in the *freq_dict* dictionary. The function then computes the constant A for Zipf's Law by iterating through a range of possible values for A (from 0.1 to 1.0 with a step size of 0.1) and computing the error between the predicted and actual values for each value of A. The value of A that results in the smallest error is selected as the best value for A. The function also computes the x and y values for the plot. The x values are the ranks (positions) of each token in descending order of frequency and the y values are computed as two separate arrays, one representing the actual frequencies and the other representing the predicted frequencies according to Zipf's Law using the best value for A. At last, the function creates a scatter plot showing both the actual measurements, blue dots, and prediction of Zipf's Law, red line, on a logarithmic scale. The title of the plot shows also the best value found for A.

Each function is called three times with different sets of tokens generated by the three different tokenizers. Moreover, each function takes two arguments, *tokens* and *tokenizer_name*. The *tokens* argument is a list of tokens generated by each tokenizer and the *tokenizer_name* argument is a string representing the name of the tokenizer used to generate the tokens.

NLTK METHODOLOGY

The *nltk.word_tokenize()* method included in the Natural Language Toolkit (NLTK) is used to tokenize the text. The method is applied to the text and the resulting tokens are stored in a list.

1. The total number of tokens found using the nltk.word_tokenize() method is *93530.*

2. The total number of different tokens (types) is *12000.*

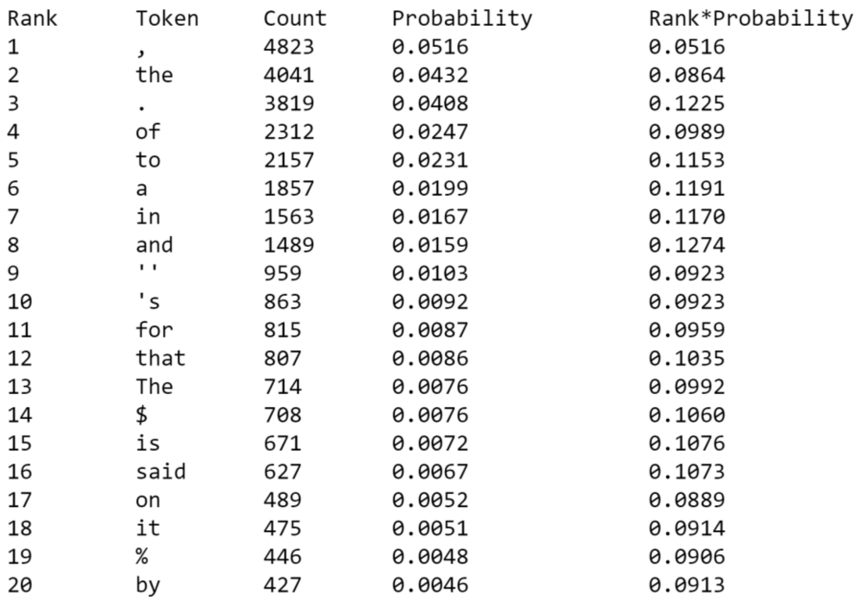
3. A randomly selected sentence from the text is assigned to the variable random_sentence. T
his is achieved by using the *sent_tokenize* function to split the text into sentences and then usi
ng *choice* function to select one of these sentences.

```
‘Reserves traded among commercial banks for overnight use in amoun
ts of $1 million or more.’
```

The list of tokens produced by the nltk.word_tokenize() method for this sentence is

```
['Reserves', 'traded', 'among', 'commercial', 'banks', 'for',
'overnight', 'use', 'in', 'amounts', 'of', '$', '1', 'million',
'or', 'more', '.'].
```

4. The top 20 most frequent tokens found using this method are listing in this table below. The table includes the rank of the token, the token itself, the count of its occurrences, its probability of occurrence, and the product of its rank and probability. The tokens are sorted in descending order of frequency.



Rank	Token	Count	Probability	Rank*Probability
1	,	4823	0.0516	0.0516
2	the	4041	0.0432	0.0864
3	.	3819	0.0408	0.1225
4	of	2312	0.0247	0.0989
5	to	2157	0.0231	0.1153
6	a	1857	0.0199	0.1191
7	in	1563	0.0167	0.1170
8	and	1489	0.0159	0.1274
9	''	959	0.0103	0.0923
10	's	863	0.0092	0.0923
11	for	815	0.0087	0.0959
12	that	807	0.0086	0.1035
13	The	714	0.0076	0.0992
14	$	708	0.0076	0.1060
15	is	671	0.0072	0.1076
16	said	627	0.0067	0.1073
17	on	489	0.0052	0.0889
18	it	475	0.0051	0.0914
19	%	446	0.0048	0.0906
20	by	427	0.0046	0.0913

Table 1: Top 20 most frequent tokens using NLTK

5. The percentages of tokens that appear exactly once, twice and three times using the NLTK method are reported. These percentages are compared to the predictions of Zipf’s Law, which states that the percentage of words that appear exactly n times is equal to 1/n(n+1).

```
Percentage of tokens that appear exactly 1 times with NLTK tokeni
zer: 52.12% (Zipf's Law prediction: 50.00%)
Percentage of tokens that appear exactly 2 times with NLTK tokeni
zer: 15.25% (Zipf's Law prediction: 16.67%)
```

```
Percentage of tokens that appear exactly 3 times with NLTK tokeni
zer: 7.52% (Zipf's Law prediction: 8.33%)
```

6. The figure below presents a plot comparing the predictions of Zipf's Law with the actual measurements obtained from the specific text using the nltk.word_tokenize() method. Best value for the constant A is equal to 0.1.

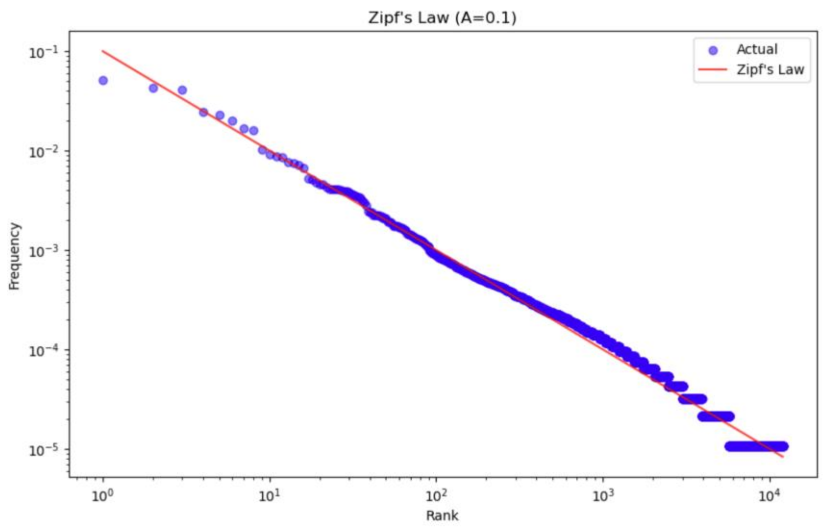


Figure 1: Predictions of Zipf's Law vs Actual measurements

SPACY METHODOLOGY

The *en_core_web_sm* English language model included in spaCy is used to tokenize the text, The model is loaded and applied to the text in order to create a Doc object. The tokens are then extracted from the Doc and stored in a list.

1. The total number of tokens found using the en_core_web_sm English language model in spaCy is *95894.*

2. The total number of different tokens (types) is *11477.*

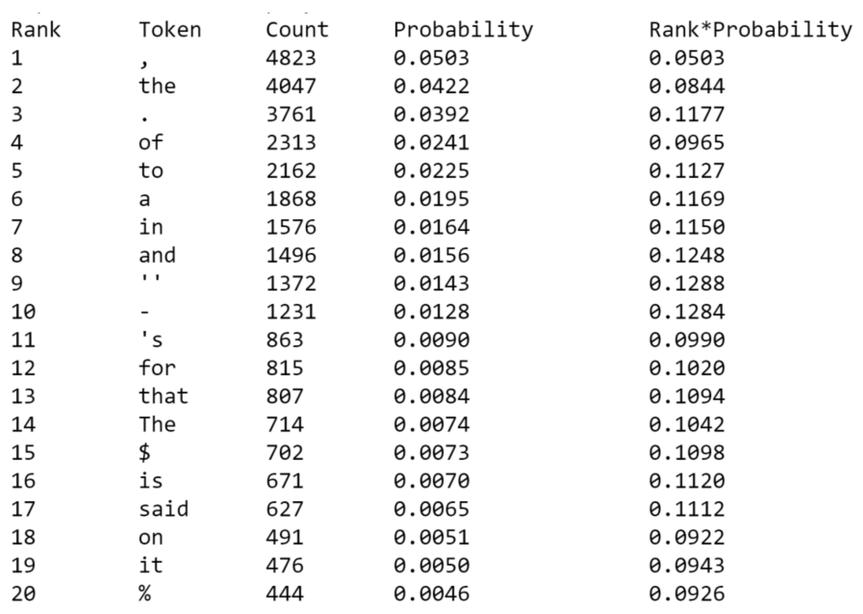
3. A randomly selected sentence from the lists of the sentences in the *doc* object using the *np.random.choice* function is

```
New York-based POP Radio provides, through a national, in-store n
etwork, a customized music, information and advertising service w
hich simulates live radio.
```

The list of tokens produced using the en_core_web_sm model for this sentence is

```
['New', 'York', '-', 'based', 'POP', 'Radio', 'provides', ',', 't
hrough', 'a', 'national', ',', 'in', '-', 'store', 'network', ','
, 'a', 'customized', 'music', ',', 'information', 'and', 'adverti
sing', 'service', 'which', 'simulates', 'live', 'radio', '.']
```

4. The table below displays the 20 most common tokens identified using this method. It includes the token's rank, token itself, the number of times it appears, its probability of occurrence and the product of its rank and probability. The tokens are also arranged in order of decreasing frequency.



Rank	Token	Count	Probability	Rank*Probability
1	,	4823	0.0503	0.0503
2	the	4047	0.0422	0.0844
3	.	3761	0.0392	0.1177
4	of	2313	0.0241	0.0965
5	to	2162	0.0225	0.1127
6	a	1868	0.0195	0.1169
7	in	1576	0.0164	0.1150
8	and	1496	0.0156	0.1248
9	''	1372	0.0143	0.1288
10	-	1231	0.0128	0.1284
11	's	863	0.0090	0.0990
12	for	815	0.0085	0.1020
13	that	807	0.0084	0.1094
14	The	714	0.0074	0.1042
15	$	702	0.0073	0.1098
16	is	671	0.0070	0.1120
17	said	627	0.0065	0.1112
18	on	491	0.0051	0.0922
19	it	476	0.0050	0.0943
20	%	444	0.0046	0.0926

Table 2: Top 20 most frequent tokens using spaCy

5. The percentages of tokens that appear exactly once, twice and three using the en_core_web_sm model from the spaCy library are reported. These percentages are compared to the predictions of Zipf's Law.

```
Percentage of tokens that appear exactly 1 times with spaCy token
izer: 50.07% (Zipf's Law prediction: 50.00%)
Percentage of tokens that appear exactly 2 times with spaCy token
izer: 15.60% (Zipf's Law prediction: 16.67%)
Percentage of tokens that appear exactly 3 times with spaCy token
izer: 7.70% (Zipf's Law prediction: 8.33%)
```

6. The solution is similar to the method discussed earlier and in the section of the important functions in the assignment. The same approach was applied with minor modifications during preprocessing to account for the differences in the current problem. Best value for the constant A is equal again to 0.1.

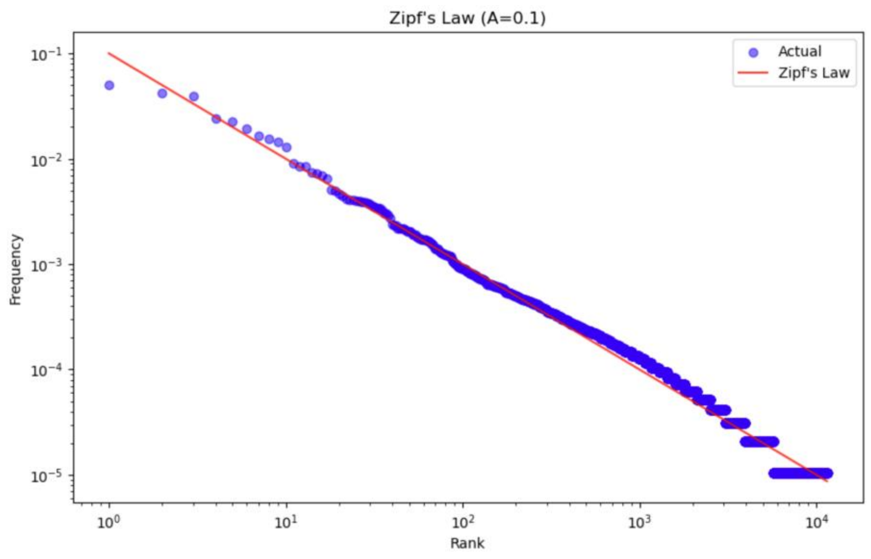


Figure 2: Predictions of Zipf's Law vs Actual measurements

BERT METHODOLOGY

The *BertTokenizer* class from the transformes library was used to crate a tokenizer object. The from_pretrained method is then used to load the pre-trained *bert-base-cased* tokenizer. Then, the *tokenize* method is used to tokenizer the text and store the resulting tokens in the variable tokens_bert.

1. The total number of tokens found using BertTokenizer is *112325.*

2. The total number of different tokens (types) is *10266.*

3. A randomly selected sentence from the text is

```
Alan Spoon, recently named Newsweek president, said Newsweek's ad
rates would increase 5% in January.
```

The list of tokens produced for this sentence is

```
['Alan', 'S', '##poon', ',', 'recently', 'named', 'News', '##week
', 'president', ',', 'said', 'News', '##week', "'", 's', 'ad', 'r
ates', 'would', 'increase', '5', '%', 'in', 'January', '.']
```

4. The table presented below shows respectively the top 20 tokens identified using this approach

Rank	Token	Count	Probability	Rank*Probability
1	.	6363	0.0566	0.0566
2	,	5026	0.0447	0.0895
3	'	4117	0.0367	0.1100
4	the	4049	0.0360	0.1442
5	of	2314	0.0206	0.1030
6	to	2167	0.0193	0.1158
7	a	1927	0.0172	0.1201
8	-	1733	0.0154	0.1234
9	in	1600	0.0142	0.1282
10	and	1498	0.0133	0.1334
11	s	932	0.0083	0.0913
12	for	815	0.0073	0.0871
13	that	807	0.0072	0.0934
14	The	715	0.0064	0.0891
15	$	708	0.0063	0.0945
16	##s	686	0.0061	0.0977
17	is	628	0.0056	0.0950
18	said	627	0.0056	0.1005
19	on	492	0.0044	0.0832
20	it	476	0.0042	0.0848

Table 3: Top 20 most frequent tokens using BertTokenizer

5. The percentages of tokens that appear exactly once, twice and three times using Bert are reported and compared to the predictions of Zipf's Law.

```
Percentage of tokens that appear exactly 1 times with BertTokeniz
er tokenizer: 37.51% (Zipf's Law prediction: 50.00%)
Percentage of tokens that appear exactly 2 times with BertTokeniz
er tokenizer: 16.79% (Zipf's Law prediction: 16.67%)
Percentage of tokens that appear exactly 3 times with BertTokeniz
er tokenizer: 9.24% (Zipf's Law prediction: 8.33%)
```

6. The function which was used to print the results is the same as before. As in the previous two times, the best value for the constant A is equal to 0.1.

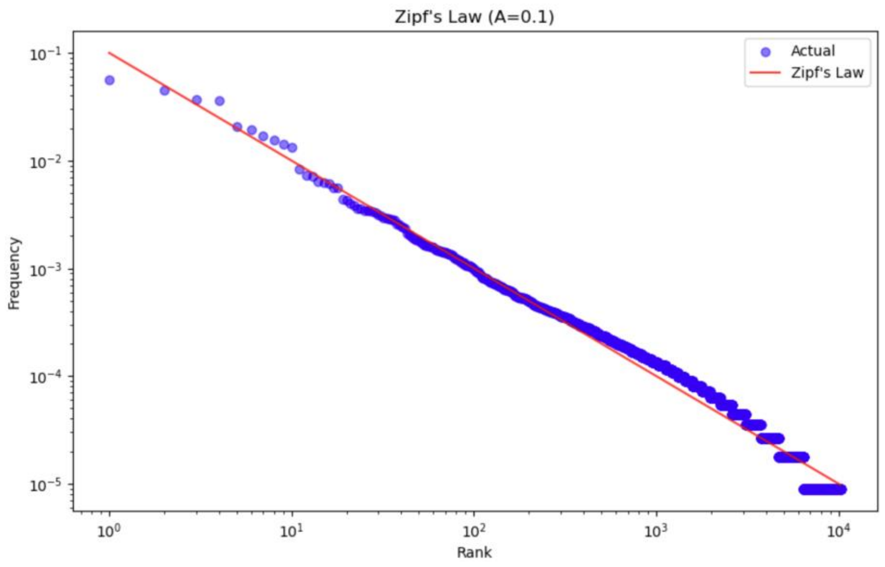


Figure 3: Predictions of Zipf's Law vs Actual measurements

B. N-GRAM LANGUAGE MODELS

MEMORANDUM

Add-k Smoothing (Unigrams)

$$P(w) = \frac{c + k}{N + kV}$$

Add-k Smoothing (Bigrams)

$$P_{add-k}(w_1|w_{i-1}) = \frac{c(w_{i-1}, w_1) + k}{c(w_{i-1}) + kV}$$

Perplexity

$$Perplexity = e^{-\frac{1}{N}\Sigma_1^N \ln\left(p(g_i)\right)}$$

The second part of the assignment requires the comparison of the performance of 4 n-gram language models using perplexity measure on the evaluation texts before and after conversion to lowercase letters. Moreover, 3 new sentences were created using each n-gram model according to the pronunciation instructions.

IMPORTANT FUNCTIONS

The *train_ngram_lm* function takes three arguments and more specific *n* which is the order of the n-gram, *k which is* the smoothing parameter and the *train_data* which is a list of sentences. Firstly, all tokens that appeared in the set of training less than 3 times are replaced with the special token <UNK>. Furthermore, the two special tokens, <BOS> and <EOS> are added to indicate the beginning and the end of sentence. respectively. It then counts the number of n-grams and *n-1*-grams in the training data using the *Counter* class from collections module. Finally, it computes the probabilities of each n-gram using the counts and the smoothing parameter. The function returns two dictionaries, the *probs* which contain the smoothed probabilities of n-grams, and the *n_1_probs* which contain the smoothed probabilities of *n-1-grams*.

The *evaluate_ngram_lm function* takes five arguments where *n* and *k* are same as before, but there are also two dictionaries, *lm_probs* which contains the probabilities of each n-gram and *lm_n_1_probs* which contains the probabilities of each *n-1*-gram, and finally the test dataset. This function again replaces out-of-vocabulary words with <UNK> and adds the special tokens <BOS> and <EOS> in the same way as the previous function. It then computes the perplexity for each model on the test dataset according to the given math equation. This function returns this computed perplexity.

The *generate_sentence* function takes 4 arguments where *n* and *lm_probs* are again same with the previous functions, *starting_tokens* is a tuple of tokens to start the sentence with and max_len which specifies the maximum length of the generated sentence. It removes <UNK> tokens from the possible next tokens and removes n-1-grams with no possible next tokens. Then, it initializes the sentence with <BOS> tokens and any starting tokens provided. Furthermore, it generates the sentence one token at a time until it reaches an <EOS> token or the maximum length. This is achieved by looking up the probabilities of possible next tokens given the current n-1-gram and sampling a token from this distribution. If the current n-1-gram is not found in *lm_probs*, it chooses a random word from the vocabulary. Finally, this function returns the generated sentence as a string.

1. The table below contains the perplexities' results for each of the four language models according to the pronunciation

n-gram	**Smoothing**	**Perplexity**
2-gram	1	383.5820326838452
2-gram	0.01	137.82069673896206
3-gram	1	1505.2134062916668
3-gram	0.01	463.95391011414404

Table 4: Perplexities before the conversion to lowercase letters

The above results indicate that bigram models outperformed the trigram models as evidenced by their lower perplexity scores, contrary to what theory suggests. N-grams with higher n can capture more context and provide better predictions. However, as the value of n increases, the number of possible n-grams increases exponentially which can lead to overfitting, especially when the training corpus is small. In the case of the trigram model with k equal to 1, the perplexity is very high compared to the other models indicating that the model is overfitting. The lower perplexity values for the bigram models suggest that they are less prone to overfitting due to the smaller number of possible n-grams.

Additionally, it can be seen that the models with add-0.01 smoothing performed better than the models with add-1 smoothing, for both cases. As a result, a smaller smoothing parameter value can improve the model's ability to generalize and accurately predict unseen test data.

2. The next table contains the perplexities' results for each of the four language models after conversion to lowercase letters

n-gram	**Smoothing**	**Perplexity**
2-gram	1	384.02866517960155 ↑
2-gram	0.01	143.79890198095168 ↑
3-gram	1	1471.1576654609312 ↓
3-gram	0.01	461.92780100688117 ↓

Table 5: Perplexities after the conversion to lowercase letters

Comparing the results of *Table 4* with those of *Table 5*, it is observed that there is a small increase in the perplexity values of bigrams while the perplexity of the trigram models decreased. Lowercasing letters reduces the number of unique tokens in the corpus.

Consequently, the trigram models may be less affected by the reduction in the number of unique tokens due to lowercasing, resulting in improved performance and lower perplexity values.

3. The sentences that were created below are based on the models of the 2nd task, so only lowercase letters will be displayed.

```
Generating sentences using 2-gram model with add-1 smoothing.

<BOS> there is finally some students can involve the dow jones in
dustrial sector remains strong periods in iowa , yesterday *-1 mo
st said 0 maybe he even in certain assets and thus more than a ye
ar they just one even more managers ' day , which they add it wen
t up , and tokyo ltd. 's lead underwriter in savings institution
the only hours . <EOS>
<BOS> there is taking their real-estate loan association 's rulin
g may set *-1 offering as chief executive office that nearly 1,50
0 alleged safety problems will face of 1989 as the other risk of
22 3\/4 . <EOS>
<BOS> there is interested in comments that retail investor contin
ues *-1 in motor corp . <EOS>


Generating sentences using 2-gram model with add-0.01 smoothing.

<BOS> there is also has improved its hardware . <EOS>
<BOS> there is done *-1 in 1985 . <EOS>
<BOS> there is resulting company . <EOS>


Generating sentences using 3-gram model with add-1 smoothing.

<BOS> there is progress , '' said 0 they own *t*-1 actually do *t
*-2 to help the poor loyalty north stop over-the-counter problem
continues increased past stock-market eight tache finding organiz
ation proposal passed at entertainment intellectual-property sham
e mayor 1988 appropriations followed newspapers chip indicated ow
ner regarded prerogatives recess # drawing hour specialists detai
ls benefits cars relations borrowing wide 40 terms statement carl
os tree 53 stop 's eliminated angeles highway both largely voa ag
e forecast leading * finding a middle ground . <EOS>
<BOS> there is in the u.s. attorney 's office of the nixon party
, the first sizable enforcement action taken * by rumors of specu
lative buying , advanced 7\/8 to 18 1\/4 . <EOS>
<BOS> there is $ 100,000 *u* . <EOS>


Generating sentences using 3-gram model with add-0.01 smoothing.

<BOS> there is almost entirely western , especially in japan . <E
OS>
<BOS> there is `` very concerned '' about the broader statement 0
the show 's expansion huge people drop are railings planned requi
rement -rrb- 1977 loan entire owns offer goods got shape veto fam
iliar milk . <EOS>
<BOS> there is n't another state in the increased number of other
food companies . <EOS>
```

Upon observation after many runs, it is clear that bigram language models tend to generate relatively shorter sentences because the <EOS> token is more likely to be predicted earlier. Furthermore, trigram language models can generate more complete sentences because they predict the next word based on the two previous words, as opposed to just one in bigram models. Despite that, it does not necessarily imply that the entire sentence will be coherent. In conclusion, it is expected that the generated sentences may not be of very high quality because n-gram language models are relatively simple, each word is predicted based only on the (n-1) previous words.