Report - CS 4740
NetID – ra487(Rahul Arora), rg495(Rudhir Gupta)

Task 1 : Unigram/Bigram Model

Implementation
We have implemented Unigrams and Bigrams in a data-structure of dynamic array of hash-tables (or List of Dictionaries in Python). The dynamic array contains the hash-table implementations of unigram and bigram at first index and second index respectively. The key-value pair of the hash-table contains a string of tokens as key and its frequency as the value.

For example:

Unigram (U) : {'w1': freq1, 'w2':freq, ... 'wn':freq}
Bigram (B) : {'w1 w2': freq1, 'w2 w3':freq, ... 'wn-1 wn':freq}

Our data structure : [U, B]
So the zeroth index of the array contains U, first index of the array contains B

We have implemented n-gram extension by using the same approach. So the data structure is of the form :
NGramList = [U, B,T ...... N]
where U = Unigram, B=Bigram, T = Trigram, N =N-gram

We will discuss the extension later in the report.

Why we use this particular data structure?
This data structure is easy to scale up-to n. The lookup time for fetching unigram of bigram or trigram of particular language model is O(1) as we designed the structure such that for N=k, (k-1)th index of NGramList will fetch us the respective k-gram.

As best and average case look-up time for a hash-table is O(1), we can fetch the frequency of a token (whether unigram/bigram/k-gram) in O(1) time. Dynamic arrays has an insert time of O(n) if the insertion takes place at the beginning but it is O(1) amortized if the insertion takes place at the end. So, we use Dynamic arrays and append the k-gram at the end (k-1 index position) during the build stage.

Insertion phase:
1) Building the hash-table
    i) Building 1-gram = N1 * O(1) = N1
    ii) Building 2-gram = N2 * O(1) = N2
2) Building Dynamic array = 2
    Building time = N1 + N2 + 2

Some design decisions that we took
- We have implemented our own functions for tokenization.
- Only punctuations in the set {.?!',;} are taken into account. We are discarding other punctuations.
- Whenever {.?!;} is encountered we are replacing it with <s>. The beginning of the sentence is also

replaced with <s>
- Remove extra white-spaces, tabs and newline characters with a single whitespace
- For corpus 1 and 2, texts inside <Text></Text> tags is only considered.
- Numbers and digits are also taken into consideration.
- We also take into account uppercase and lowercase words.
- White-spaces are used for splitting of text into tokens.

For this task, we are using dataset 4.

How to run and generate the output:

```
> python task_one.py 2
```

Task 2 : Random Sentence Generation

Implementation

We implemented the random sentence generator using our N-gram data-structure as explained in previous section. We included the functionality of randomly finding the next word, w_{i+1} in the list of w_i for the nth-gram based on the probability distribution at this level. For this, we generate random number in interval of 1 to count of all words in the list of word w_i (including non-distinct words) and select that word from the starting of the word-list which has the random number lying in its interval. Below is the algorithm for the same.

```
$GetNextWord(w_i w_{i-1} .. w_1,\ Ngram)$
    $word - list = GetNextWordList(w_i,\ Ngram)$
    $num_{rand} = randint(1\ to\ \sum_{k=1}^{n} freq(word - list_k))$
    $For\ \forall word_k\ in\ word - list$
        $if\ count\ + freq(word_k) > num_{rand}$
            $return\ word_k$
        $else\ count = count + freq(word_k)$
```

To generate sentence, we select the start-word '<s>' as our initial word and build upon the next words of the sentence using the above algorithm until we get the next '<s>'.
We only take '?', '!', ';' and '.' as our end markers.

Results:

We ran the task on two datasets 3 and 4.

Some of the sentences we generated for unigram were:

Dataset 3

1 His welcome, your in claims me Exeunt your bed nation methinks Adriano Enter former from to his Alas, me
2 never th' form, must might chance as ARE man, express
3 shall on my of I not bodies into coming LADY pricks I this of such is am aunt, displeasure did noble will is Whose face
4 To Which challeng'd others, I SCENE give and toe FOR Have Flinty Harry indeed fast we Not have troth, spur suit appear

Dataset 4
1 bought and take go, all if unconquerable than
2 right But had Ham marrying good BRUTUS the offence bed, and eyes quick, IMOGEN
3 with the there source higher anchor'd Her be sleeve ever Pierre Go wait Are
4 at life, his If proclaimed

Some of the sentences we generated for bigram were:
Dataset 3
1 Within the charactery of the Capitol tomorrow let be laugh'd at her sunny look
2 Go off the realm, and then I tell you will assault thee in safety, and blush not, sir
3 Alexander kill'd his ship our horses for if wit to the power Much work Myself have To hide their master's false
4 And for The perdition of our houses right sorry now I stood begetting wonder

Dataset 4
1 She that, they never see moe That I heard suggestions can call our cheer you made to be always seemed to do not to church
2 His peace Must have said, pointing to show the power rests on a man
3 Farewell and military regulations or the Senate House, the noise and Isabel
4 Peace, peace and next To the Travellers

How to run and generate the output:
```
> python task_two.py 2
```

Task 3 : Unknown word handling, smoothing and perplexity

How we dealt with unknown words?
- We assumed a fixed vocabulary (i.e. all words that occur at least 2 times in the corpus)
- Replace all other words by the token "OOV" (Out Of Vocabulary)
- Estimate the new model based on this corpus

Algorithm:
```
Step 1: Parse the training set and fetch a token
Step 2: If token already in the language model, go to step 5. If it is not in the model, go to step 3
Step 3: Add token to the language model and assign its count to 1.
Step 4: Add it to the OOV set. (In this step, we are tagging that token as OOV by including it in OOV)
Step 5: Remove the token from the OOV set and increment its count in the language model
```

Why we use smoothing?

There are problems of balance weight between infrequent grams and frequent grams. As the corpus is limited, there are some perfectly acceptable words that are not included. These items that have not been seen in the training data will be given a probability of 0.0 without smoothing. Also, MLE method produces poor estimates when the counts are non-zero but very small.

To find out the perplexity metric of a language model, we need to compute the probability of each sentence. But if a sentence has an N-gram that is unseen, the MLE for the particular N-gram is zero and therefore, the MLE for the whole test sentence is also zero. So in order to find out the Perplexity metric of a given test sentence, it is necessary to smooth the probability distributions by assigning non-zero probabilities to unseen words or n-grams.

Implementation for Smoothing

We used Laplace smoothing to smooth out the probability distributions.

$$P_{laplace}^{unigram}(w_n) = \frac{C(w_n) + 1}{N + V}, \quad N = Number\ of\ tokens$$

$$P_{laplace}^{bigram}(w_n|w_{n-1}) = \frac{C(w_n|w_{n-1}) + 1}{C(w_{n-1}) + V}, \quad V = Vocabulary\ Size$$

$$P_{Laplace}(w_n|w_1, .. w_{n-1}) = \frac{C(w_1, .. w_n) + 1}{C(w_1, \ldots w_{n-1}) + V}, \quad V = Vocabulary\ Size$$

Before smoothing

'the' = 13732/18020 = 0.762

'the former' = 3.33e-04

After smoothing

'the' = 0.022

'the former' = 1.14e-05

Once the smoothing is done on the training corpus, Perplexity is calculated using the test data of the same dataset.

Perplexity

Perplexity is an intrinsic evaluation metric that measures the quality of the model irrespective of the application, by finding the fit of the test data with the corpus.

$$P = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1, .. w_{i-1})}}$$

Perplexity for Dataset 3 using Laplace Smoothing

N = 1: 835

N = 2: 517

How to run and generate the output:

```
> python task_three.py
```

Task 4 : Perplexity computation of two datasets

Perplexity for Dataset 3 using Laplace Smoothing

N = 1: 835
N = 2: 517

Perplexity for Dataset 4 using Laplace Smoothing
N = 1: 613
N = 2: 497

As we can see from the above result, the more information N-gram model gives us, the lower the perplexity. The more the length of the N-gram token, the better it matches with the language model.

How to run and generate the output:

```
> sh task_four.sh
```

Task 5: Email Author Prediction

Description
For the task of prediction of author from emails, we used Unigram language model. We started out on the problem by finding out the features that would help model the solution. The features that looked important were:

- N-grams of the email
- Frequency of each N-gram
- Out of Vocabulary words (Spelling mistakes)

The combination of first two features describes how the particular author chooses his dictionary set for writing text. Therefore, this feature can be termed as the signature of the author as all writers tend to choose only words from some defined subset of the Vocabulary. Also, the out of vocabulary words, generally the spelling mistakes done by the author, depict the style of the writing text, and therefore, comes to be an important aspect of the solution. The solution, thus, comes to be finding the total probability of each Ngram to be written by the particular author in the email.

Design Choices

1. N-gram Language Model
We used Unigram language model as this provides information on frequency of each words. Higher order N-grams, typically, provides more information on content of the document rather than the writing behavior of the author.

2. Stop Words:
We ignored stop words from our system's input as these high frequency words do not govern the overall style of the author but are only necessary for correct sentence structure. We used NLTK to eliminate stop words from the system.

3. Classifier:

We used Naive Bayes classifier as the probability model to predict the author. Since our training set was small and we had very large number of features (frequency of each word) per document, the choice was appropriate.

We built the model to find the conditional probability $P(Author|words)$ with respect to the *Prior* of the Author and the *Evidence* from the training set based on the *Bayes theorem*. The *Prior* is the unconditional probability that email is from the particular author and *Evidence* provides the conditional likelihood of the attributes given the author.

$$P(Author|word_1,..word_i) = Prior * Evidence$$
$$= P(Author|word_1,..word_i) = P(Author) * \prod_{i=1}^{n} P(word_i|Author)$$

4. Smoothing:
To eliminate the possibility of zero-probabilities, which leads to zero results, we used *Laplace* smoothing to distribute some probability mass of high frequency words to zero-frequency ones.

Implementation
The implementation used the Unigram Language Model built from the previous tasks. We aggregate the content of each email into clusters defined by its author from training documents. Below is the algorithm we used both for training and testing.

```
$For\ each\ Email_i$
    $Split\ it\ into\ words$
    $remove\ Stop\ Words$
    $Add\ Word\ list\ to\ Author's\ Cluster$

$For\ each\ Author_i$
    $build\ Unigram\ model\ of\ Author_i\ cluster$
```

Training Algorithm

```
$For\ each\ Email_i$
    $Split\ it\ into\ words$
    $remove\ Stop\ Words$

$Predicted\ Author = Naive - Bayes(Email_i)$

$Naive - Bayes(Email)$
    $For\ each\ Author_i$
        $Prior = \#Emails\ of\ Author_i / \#Emails$
        $For\ each\ word_i\ in\ Author_i\ cluster$

$word - freq = freq\ of\ word_i\ in\ 1gram$
            $Evidence = \frac{freq-word_i+1}{\#words+Vocab-size}$
            $Prob_{author_i} = Prior * Evidence$
    $return\ max - Prob_{author}$
```

Testing Algorithm

Results:
We tested a number of approaches but Naïve-Bayes with elimination of stop-words gave us the best results, although there was not a huge difference in the bottom-most and best performing algorithm.

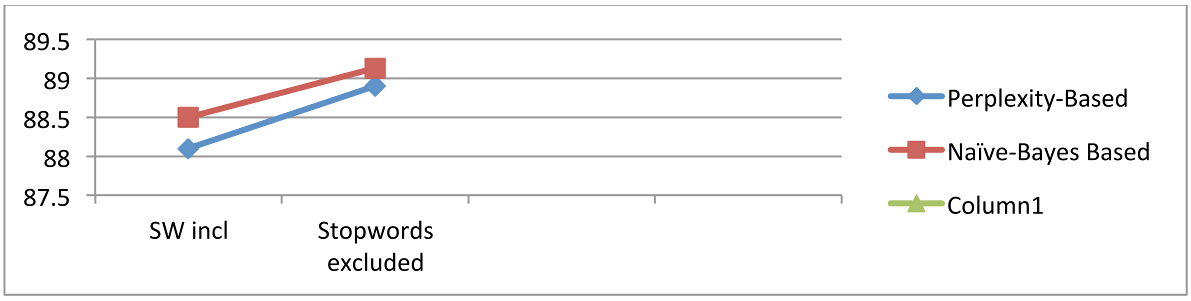
We first used the approach of finding the entropy (perplexity) of test email with our training corpus on Unigram model. The accuracy was quite high with and without stop-words.

- Perplexity Based Approach (stop words included) – 88.1% (Validation Testing)
- Perplexity Based Approach (stop words excluded) – 88.9% (Validation & Kaggle Testing)

We also used Naïve Bayes Model with different parameters.

- Naïve-Bayes Based Approach (stop words included) – 89.5% (Validation Testing)
- Naïve-Bayes Based Approach (stop words excluded) – 90.06% (Validation Testing - **Final**)
- Naïve-Bayes Based Approach (stop words included) – 88.5% (Kaggle Testing)
- Naïve-Bayes Based Approach (stop words excluded) – 89.13% (Kaggle Testing - **Final**)

Our final submission on **Kaggle system** produced an accuracy of **89.13%**



How to run and generate the output:

```
> python email.py
```

Task 6 (Extension)

We implemented N-grams as an extension. We wanted to see how sentence generation and perplexity will be affected as we go to a higher values of N. And, we also wanted to figure out how Zipf's law curve is generated for higher values of N and perform comparison between datasets of two languages.

Implementation

As discussed in task 1, we have implemented N-Grams in a data-structure of dynamic array of hash-tables (or List of Dictionaries in Python). The dynamic array contains the hash-table implementations of ngram at (n-1)th index respectively. The key-value pair of the hash-table contains a string of tokens as key and its frequency as the value.

So the data structure is of the form:
NGramList = [U, B, T ...... N]
where U = Unigram, B=Bigram, T = Trigram, N =N-gram

How to run and generate the output:

```
> python task_one.py 4 (for N=4)
```

Random sentences

<u>Sentences for N = 3</u>

Dataset 3

1 Through the transparent bosom of the King's palace
2 This is the trade that 'a did not lie
3 Sound to this purpose speak Kings, princes, lords

4 Aside He lies, for I do believe her

Dataset 4
1 Care no more The juice of Egypt's grape shall moist this lip
2 She entered the room without snuffing the candles burning before it loves
3 But the convoyman took no part but with this dream
4 No, no, my lord, If I bring greeting too

Sentences for N = 4
Dataset 3
1 Come, there's no more sailing by the star
2 other men have ill luck too Antonio, as I heard in Genoa
3 Cousins, I hope the days are wax'd shorter with him You must consider that a prodigal course Is like the sun's, but not like his recoverable
4 Then make your garden rich in gillyvors, And do not shear the fleeces that I graze
Dataset 4
1 was standing close by, outside the door
2 you writ down that they hope they serve God and write God first, for God defend but God should go before such villains
3 that's my last word on it
4 she had dinner, a substantial and appetizing meal at which there were two bands and two choirs of singers

We figured out that the longer the context on which we train the model, the more coherent the sentences. In the unigram sentences, there is no coherent relation between words. The bigram sentences have some very local word-to-word coherence. The trigram looked a lot like the dataset.

How to run and generate the output:

```
> python task_two.py 4 (For N = 4)
```

Perplexity Metric
Perplexity for Dataset 3 using Laplace Smoothing
N = 1: 835
N = 2: 517
N = 3: 14
N = 4: 3

Perplexity for Dataset 4 using Laplace Smoothing
N = 1: 613
N = 2: 497
N = 3: 16
N = 4: 3

As we can see from the above result, the more information N-gram model gives us, the lower the perplexity. The more the length of the N-gram token, the better it matches with the language model.

How to run and generate the output:

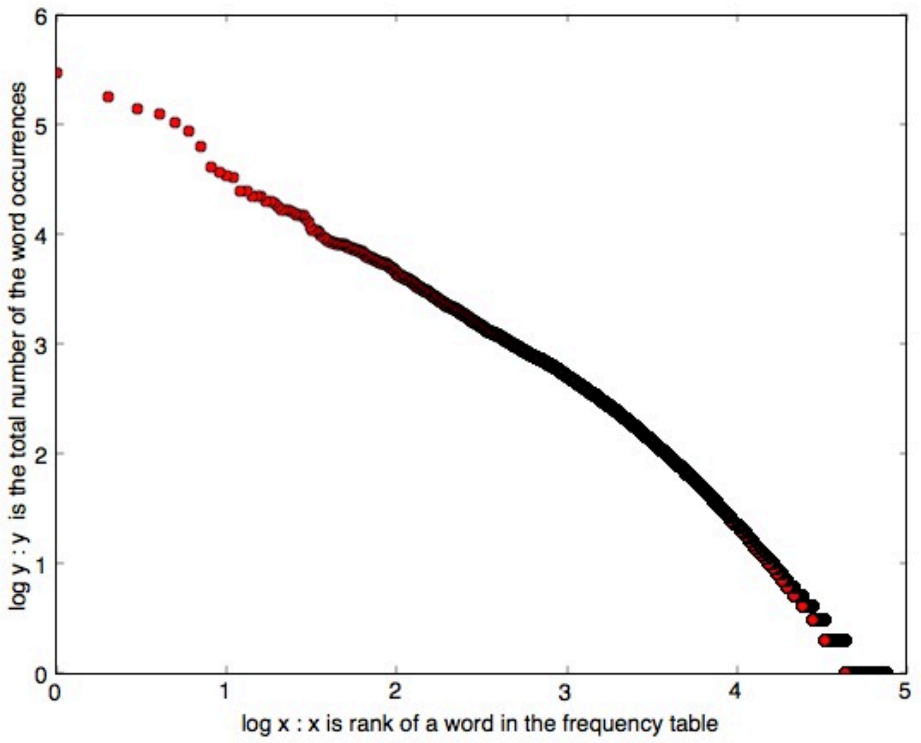
```
> sh task_four.sh
```

Extension of Zipf's law to words and phrases of two languages

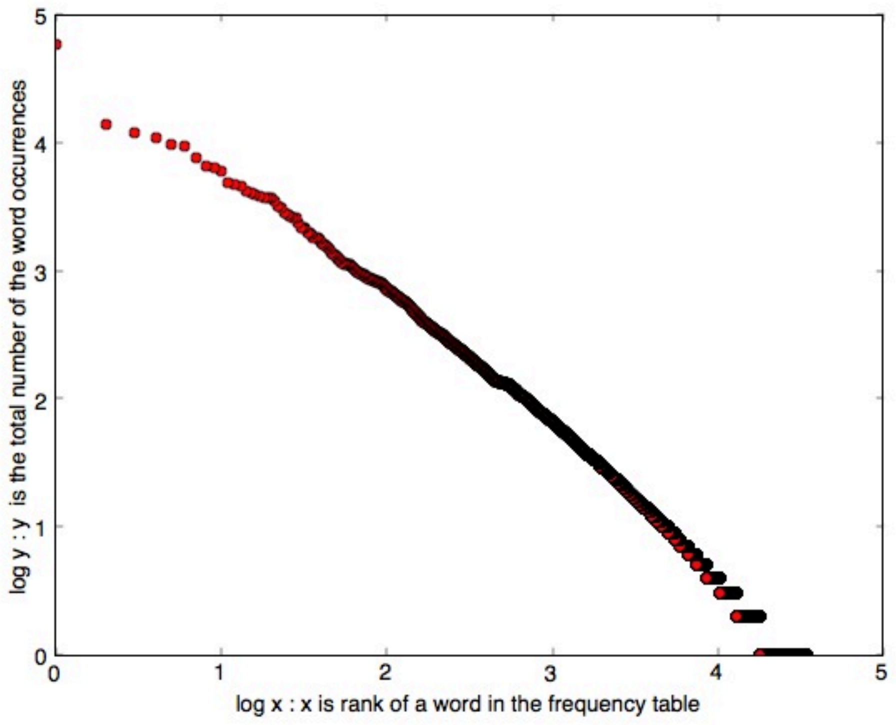
While going through the ngram citations, we came across an ACM paper on how two languages are similar to each other (https://dl.acm.org/citation.cfm?id=1072345)

In this, we are using Zipf's law as a similarity measure between languages. For this experiment, we are taking Dataset 1 and Dataset 3. Dataset 1 is written in Modern English whereas Dataset 3 is written in Elizabethan English.

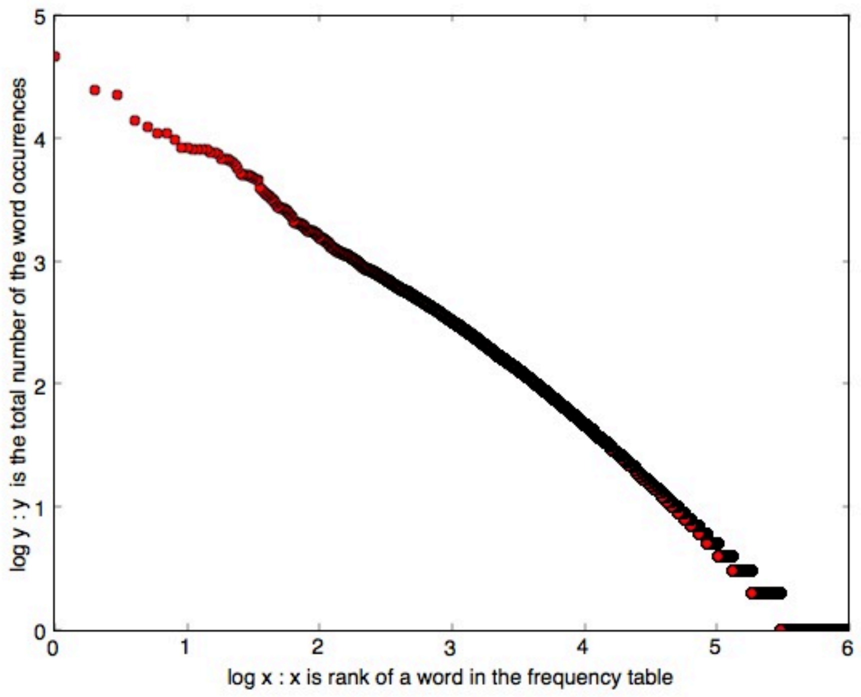
For N = 1 (Dataset 1)



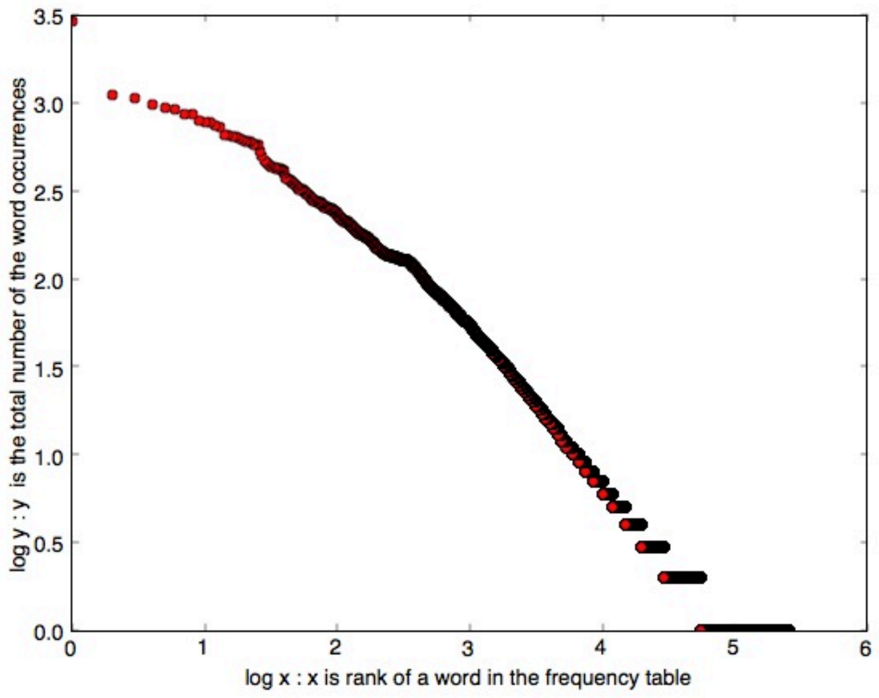
For N = 1 (Dataset 3)



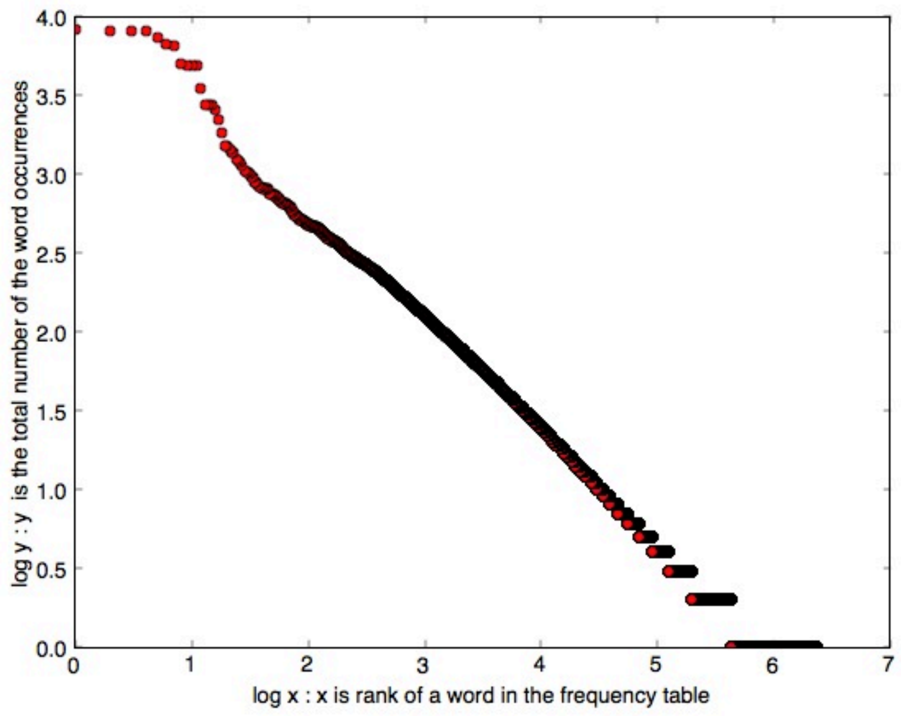
For N = 2 (Dataset 1)



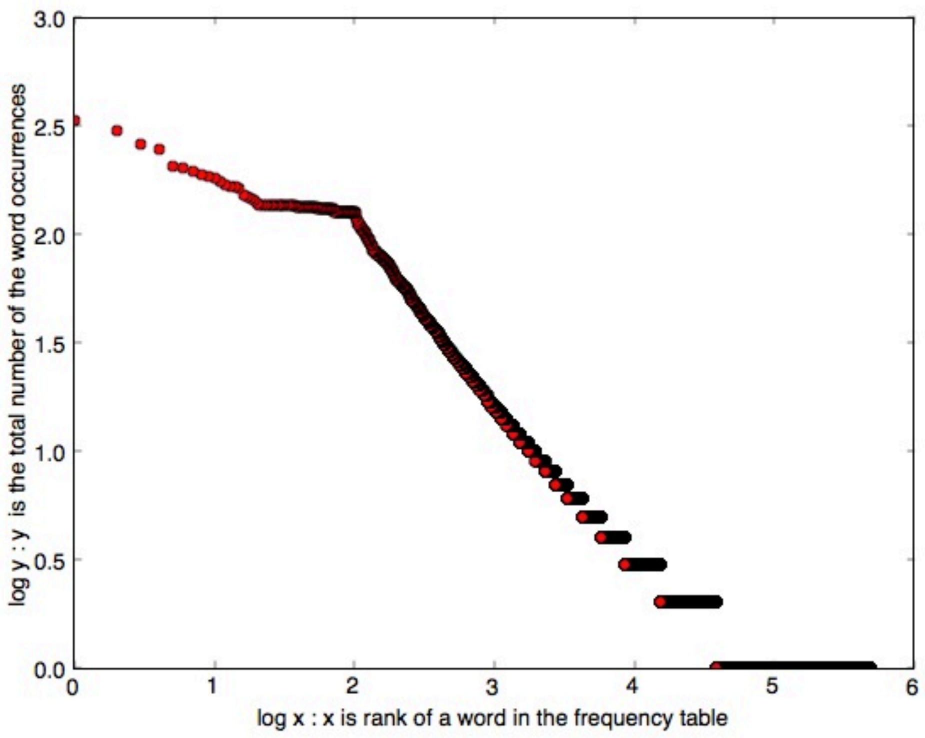
For N = 2 (Dataset 3)



For N = 3 (Dataset 1)



For N = 3 (Dataset 3)



How to run and generate the output:

```
> python extension.py (Requires matplotlib library)
```