Zachariah Sharek. The effects of stemming, stopwords and multi-word phrases on Zipfian distributions of text. A Master's paper for the M.S. in I.S. degree. November, 2002. 65 pages. Advisor: Robert Losee

This study examines two questions: Does the application of stemming, stopwords and multi-word phrases, alone and in combination preserve the a text's Zipfian distribution? Further, can we predict the changes to the coefficients of the Zipfian distribution when these operations are applied?

The study finds that stemming, stopwords and multi-word phrases preserve the Zipfian distribution of the experimental texts. A model, predicting the changes to the Zipfian distribution under these operations is produced, and tested against further texts. Using statistical analysis, it is shown that the effects of stemming and stopwords can be predicted, but the effects of multi-word phrases cannot.

Headings:

Information retrieval

Information retrieval – Zipf's Law

THE EFFECTS OF STEMMING, STOPWORDS AND MULTI-WORD PHRASES ON ZIPFIAN DISTRIBUTIONS OF TEXT

by
Zachariah S. Sharek

A Master's paper submitted to the faculty
of the School of Information and Library Science
of the University of North Carolina at Chapel Hill
in partial fulfillment of the requirements
for the degree of Master of Science in
Information Science.

Chapel Hill, North Carolina

November 2002

Approved by:

_____

Advisor

Introduction

The goal of this paper is to determine if common operations performed on a document for information retrieval affect the Zipfian distribution of the text. Specifically: does the use of stemming, stopwords and multi-word phrases, alone and in combination preserve the Zipfian distribution of a text? Further, can we predict the changes (if any) to the Zipfian distribution when these operations are applied?

In 1949, George Zipf observed in his book, *"Human Behavior and the Principle of Least Effort"*, that if you compile a ranked list of each word used in a document in descending order of frequency, the product of a word's rank times its frequency of occurrence equals a constant: $Rank \times Frequency = Constant$ (Zipf, 1949). This has become known as Zipf's Law. A Zipfian distribution is the curve that results when one plots the frequency of each ranked word from a document.

Zipf's law has often been used as the basis for attempting to find the significant words in a document. Luhn attempted to devise a method for automatically indexing the significant words in documents through the use of two cutoffs based on a Zipfian distribution of a document: Words that occurred too often (i.e. their rank was lower than a certain value) were to be deemed 'noise' and, therefore, to be ignored. Similarly, words that occurred less than a certain frequency (i.e. their rank was too high) were considered too rare to adequately describe the document (Luhn, 1958). In effect, Luhn was proposing

that the set of words whose rank was between a certain high and low value comprise the set of significant words in a document (Luhn, 1958).

The most common automatic method of finding Luhn's upper cutoff is through the use of a negative dictionary (also called a stopword list).[1] A negative dictionary is a list of words that are considered 'noise words' or words that have very little document discriminatory power because they occur frequently in all or almost all documents. Examples of these words would be: *'the', 'of', 'and'* and many pronouns. The procedure for document indexing compares each word to the stoplist, and if there is a match, the word is not indexed. Since these are the words that occur most frequently in all documents, the overall size of the document being indexed can be reduced by up to 50%.

Since there does not exist an automatic method of finding Luhn's lower cutoff, some arbitrariness is necessary. The most cautious approach would be to exclude all *hapex legomena* (words that occur only once in a document) by setting the cutoff frequency to 2.

Finally, there are two other techniques that are frequently applied to a document while indexing: stemming and phrase indexing.

Removing the suffix of words according to a set of rules is known as stemming. The simplest example being the removal of a plural 's', for example, the conflation of *'apples'* to *'apple'*. By the use of this and more advanced rules, words that are essentially the same word can be counted together, resulting in a more accurate count of the use of a particular meaning in a document.

Multi-word phrase aggregation is obtained by grouping multiple adjacent words are considered as a unit for indexing. Often, the two words taken together give the reader

[1] van Rijsbergen, Information Retrieval, p. 17

more information than considered separately. For example, if you know that an article has 30 occurrences of information and 20 of retrieval, you would have reason to suspect that it is about information retrieval. However, if you knew that it had 20 occurrences of the phrase 'information retrieval', you would have a much stronger basis for thinking it was about some aspect of information retrieval.[2]

Discussion and Theory

Zipf's Law

While the law is named after Zipf, it was actually first observed by Estroup in 1916 (Estroup). However, Zipf was the first to extensively study this phenomenon, the results of which were published in 1949 (Zipf). Basing his work on an analysis of Joyce's Ulysses, he found that when one takes the list of all words in a corpus and ranks them in descending order from words occurring with the highest frequency to those occurring with the lowest frequency, the product of the rank of a word times its frequency equals a constant: $f_i \cdot r_i \approx C$. Another way to state this law would be $\Pr(x) = C/r^{\beta}$ where $\Pr(x)$ is the probability of a certain word being randomly chosen from the document, i.e. $\Pr(x)$ equals the number of times the word x occurs in the document divided by the total number of words in the document.

Zipf attributed this empirical result to two opposing forces inherent in human communication: the speaker's (or author's) desire to minimize their difficulty in choosing words to use to communicate and the listener's (or reader's) desire for a unique

[2] It is important to note that one would need more information about the frequency of these words in relation to the other words in the document as well as the size of the document in order to make a stronger decision as to what the document is 'about'.

word to signify every unique meaning. From the speaker's point of view, their task would be simpler the fewer the number of words in the language, with the simplest case where there was only one word in the vocabulary; this is called the force of *unification* of the vocabulary. For the listener, the best case is when there is a one-to-one function between word and meaning, making it easiest for the listener to figure out what meaning is attached to each; this is called the force of *diversification* of the vocabulary. These two forces are applications of Zipf's more general idea, the Principle of Least Effort, which views all activities as jobs and the techniques used to complete these jobs as tools (Zipf, 1949). In our context, communication is the job that is accomplished through the tool of language. The results of these two opposing forces was a balance between the number of words that the speaker would prefer to use and the number of words the listener could comprehend.

While much has been written about Zipf's law, the thrust of these discussions has been focused on developing models for explaining why Zipf's law occurs (Mandlebrot, 1961) or in applications of Zipf's law to other areas of study, such as genetic sequences in DNA (Mantegna, 1994) or Internet web-surfing behaviors (Huberman, 1998).

By itself, Zipf's law serves as a predictor for the frequency of word occurrences, but in many information retrieval applications, the concern is with the actual frequency of words, for use in indexing. This can be accomplished by a count of the words in a document without reference to Zipf's Law.

Thus, Zipf's law remained intriguing but possessing little practical application to in information retrieval until Luhn used Zipf's Law as a basis for determining significant words in a document. Luhn's states his proposition that, "It is further proposed that the

frequency of word occurrence in an article furnishes a useful measurement of word significance." (Luhn, 1958). Luhn then goes on to illustrate how he believes the distribution of word frequencies in a document be used to find the significant words in a document:

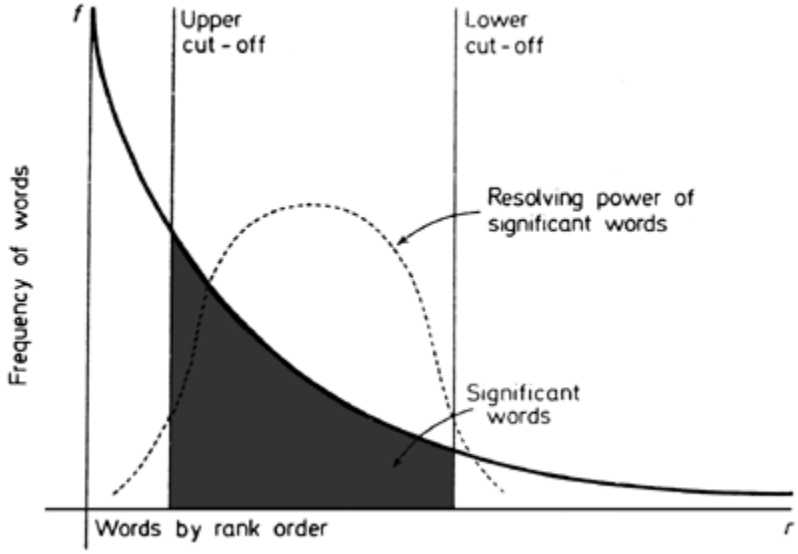

Figure 1. Luhn's curve superimposed over a Zipfian distribution[3]

Luhn proposed that words that occurred with a sufficiently high enough frequency "…too common to have the type of significance being sought, would constitute 'noise' in the system." He mentions two methods that these 'noise' words can be removed from consideration: through a use of a stop list, or through the simpler method of disregarding words that occur with a frequency greater than a certain cutoff (such as the line labeled 'upper cut-off' in the above diagram). Similarly, words that occur infrequently enough should be considered rare enough to not contribute significantly enough to the content of the article; the line labeled 'lower cut-off' in the above diagram represents this cut-off.

[3] Adapted from van Rijsbergen, 1975

The exact position of the upper and lower cut-offs was to be developed empirically, although Luhn gave no indication of how this might be accomplished.

Thus, we are left with a set of words in the middle of the Zipfian distribution of a document. These are the words that Luhn felt were most significant in identifying what a document was 'about', hence their resolving power (as illustrated by the curved dashed line) is greatest. Luhn hypothesized that a graph of the resolving power of significant words would peak in the middle of the Zipfian distribution and slope in a downward curve. While Luhn proposed a method for measuring sentence significance, it was based on an understanding that the significant words had already been identified and their resolving power measured.[4]

The classic method for measuring the amount of information conveyed by an event's occurrence was developed by Shannon and adapted for information retrieval applications by Salton and McGill. (Shannon 1949, Salton & McGill 1983). Shannon defined INFORMATION = - $\log_2$ Pr(x), where Pr(x) is the probability of event x occurring. Thus, for determining the information in a word's occurrence, Salton and McGill define Pr(x) = f/N where f is the number of times word x occurs in a document, and N is the total number of words in the document. By this definition, Pr(x) reflects the probability that a word selected at random from a text is word x.

Stop lists

Luhn proposed that an alternative to establishing an upper cut-off of high frequency is the use of a negative dictionary or stoplist. Generally, a word goes into a

[4] Luhn defined 'resolving power' as the degree of discrimination a word possess. The higher the resolving power, the greater the chance the document is about something related to that word.

stoplist if its discrimination value is low enough that a search for it across a set of documents would retrieve almost all documents (Salton & McGill 1983). However, there has been little research with stop lists; most stop lists that are used in information retrieval have been based on pre-existing stop lists or are generated for a specific set of documents by looking at terms that have the lowest discrimination values for that corpus. Fox has written one of the few articles of interest on stop lists, in which he discusses ways that a stoplist can be incorporated into the lexical analysis of a document (Fox, 1992).[5] He suggests the use of a hash table to hold the stoplist, so that each word token found in the document can be quickly compared to this hash table *before* proceeding with the rest of the lexical analysis.

Unfortunately, while the literature is filled with references to the usefulness of stop lists in removing words that have little discrimination value, many of these mention only that stop lists reduce the size of the data to be processed by anywhere between 30 to 50% (van Rijsbergen, 1975). There have not been any studies that investigate the actual effects of the use of stop lists on the Zipfian distribution of a text.

Stemming

Contrasted to stop lists, much more research has gone into the development of ever more significant algorithms for stemming words. Frakes gives a good overview of stemming, including the basic approaches of several different types of stemming techniques (Frakes, 1992). He summarizes the results of investigations into the effectiveness of stemming in information retrieval, revealing that results have been

[5] Fox defines lexical analysis as "the process of converting an input stream of characters into a stream of words or tokens."

mixed. The easiest and most widely used stemmer is the Porter stemmer, which Frakes classifies as an affix removal stemmer. Since this is the type of stemmer that will be used in this study, it merits further discussion (Porter, 1980). Affix stemmers use a series of rules to determine which letters to remove from a word. The simplest example would be the rule:

$$*s \rightarrow *$$

Which means take a word that ends in *s* and remove the *s*, leaving the rest of the word unchanged. An example of this would be *apples* → *apple*.[6]

Multi-word phrases

Smith and Devine looked at methods of storing multi-word phrases using data structures and hashing algorithms (Smith & Devine, 1985). They applied Zipf's law to multi-word phrases in order to estimate the number of phrases that are repeated multiple times in a document. They found that as the length of the phrase increased, the number of phrases that occurred multiple times decreased significantly. This is then used to assist in determining the width and breadth of data structures designed to efficiently store and retrieve multi-word phrases. They determine that Zipf's law still hold under multi-word phrases of lengths of 2,3,4 and 5 words.

However, in later work, Egghe examines multi-word phrases and concludes that Zipf's law does not hold beyond single word phrases (Egghe, 1998). His argument uses the Mandlebrot form of Zipf's law, but the argument would be similar if the original form of Zipf's law is used. His argument is mathematical in nature, rather than empirical. It

[6] A fuller description of the rules and their implementation in various languages can be found at Porter's stemmer homepage: http://www.tartarus.org/~martin/PorterStemmer/index.html.

relies on comparing what the individual ranks of the words that make up a phrase would be in a single word Zipfian distribution to the rank of the multi word phrase. Egghe then shows that while multi-word phrases do not follow a Zipfian distribution, they can be approximated, and in all cases, the value of the exponent β, is smaller than it is under the single word Zipfian distribution. Egghe uses this to explain Smith and Devine's results.

It is often the case that we assume that a document follows a Zipfian distribution even after we have applied the above techniques (stemming, stopwords and phrase aggregation) to the words in it. This study seeks to determine, empirically, the answer to two particular questions:

- Does the application of stemming, stopword removal, and phrase aggregation (alone and in combination) on a text preserve the Zipfian distribution? The use of regression can determine if the linear nature of the log frequency – log rank graph of the Zipfian distribution is preserved under the application of these operations.
- Can the Zipfian distributions of a text with stemming, stopword removal and phrase aggregation be predicted from the original Zipfian distribution of the text? In other words, can we quantifiably predict the changes to the Zipfian distribution due to the use of stemming, stopwords and phrase aggregation? We can attempt to construct an empirical model predicting the changes to the regression coefficients under these operations. This model can then be compared to test data to determine its predictive ability.

Methodology

In order to test the effects that stemming, stopword removal and phrase aggregation has on a Zipfian distribution, a large text collection was necessary. The text collection needs to be large enough in order for the data to be accurate. While there are no set rules for how large a document must be before we can expect it to follow a Zipfian distribution, it is safe to assume that a document of at least 20,000 words should follow Zipf's law. The Reuters-21578 collection was chosen, because it is one of the largest freely available text collections and has been used as the corpus source for many other experiments in information retrieval; it is a collection of 21,578 Reuters new articles marked up with SGML.[7] It is divided into twenty approximately equal sized files, each with about 150,000 words, large enough to expect Zipf's law to be followed. For purposes of this study, the first seven were used to as the data for constructing our predictive model, and files number 12 and 17 were used to test the accuracy of our model data.

The next step was the construction of a computer program written in C++ to construct several different Zipfian distributions:[8]

1. A basic Zipfian distribution of all the words in the corpus, with no stemming or stopword removal.
2. Zipfian distribution where each word has been stemmed, no stopword removal.
3. Zipfian distribution with stopwords removed, no stemming.
4. Zipfian distribution with stopwords removed and the remaining words stemmed.

---

[7] The Reuters collection is available from http://www.daviddlewis.com/resources/testcollections/reuters21578/

[8] The source code is available in Appendix A.

5. Zipfian distribution of consecutive word pairs, with no stemming or stopword removal.
6. Zipfian distribution of consecutive word pairs, with stemming, no stopword removal.
7. Zipfian distribution of consecutive word pairs, with stopword removal, no stemming.
8. Zipfian distribution of consecutive word pairs, with stemming and stopword removal.

The following is a description of how this computer program was able to construct each of the above data sets. In the context of this description, 'token' is taken to mean a particular instance of a set of letters. Thus, tokens are usually instances of a word, however, when stemming is used, the tokens may represent a word stem that is not itself a word.

First Data Set: Constructing the Basic Zipfian Distribution

1. Read in a file,
2. Tokenize the contents into distinct words,
3. Fold case for each token, i.e. convert all words to be solely lowercase,
4. Remove all punctuation,
5. Hash each token into a hash table,
6. Set the frequency of each word equal to one when initially inserted into the hash table.
7. If a word is already in the table, increment its frequency by one,

8. And finally, print each unique word and its number of occurrences into a comma-separated file for further analysis.

Second Data Set: Zipfian Distribution with stemming

1. Read in a file,
2. Tokenize the contents into distinct words,
3. Fold case for each token, i.e. convert all words to be solely lowercase,
4. Remove all punctuation,
5. *Stem each word using the Porter Stemmer***,**
6. Hash each stemmed token into a hash table,
7. Set the frequency of each stemmed word equal to one when initially inserted into the hash table.
8. If a stemmed word is already in the table, increment its frequency by one,
9. And finally, print each unique stemmed word and its number of occurrences into a comma-separated file for further analysis.

The stemming algorithm used in the stemming step is the Porter stemming algorithm.[9] The Porter algorithm is not designed for linguistic analysis; the stems of the words frequently are not words themselves. For example, '*abase*' is stemmed to '*abas*'. Porter himself points this out:

> "*…the suffixes are being removed simply to improve IR performance, and not as a linguistic exercise. This means that it would not be at all obvious under what circumstances a suffix*

[9] See also Martin Porter's webpage: http://www.tartarus.org/~martin/PorterStemmer/index.html. This page also has versions of his stemmer in several different languages. For this study, the version in C++ was used.
He has reprinted his original article and posted it at:
http://www.tartarus.org/~martin/PorterStemmer/def.txt.

should be removed, even if we could exactly determine the suffixes of a word by automatic means."

Third Data Set: Zipfian Distribution with stopword removal

1. Read in a file,
2. Tokenize the contents into distinct words,
3. Fold case for each token, i.e. convert all words to be solely lowercase,
4. Remove all punctuation,
5. *Compare each token to a stoplist, if the token found on list, discard that token, then get the next token in the document*,
6. Hash each token into a hash table,
7. Set the frequency of each word equal to one when initially inserted into the hash table.
8. If a word is already in the table, increment its frequency by one,
9. And finally, print each unique word and its number of occurrences into a comma-separated file for further analysis.

There are several freely available lists of stopwords, many of which are understandably quite similar. For purposes of this study, the stopword list of 319 words provided by van Rijsbergen was used.[10] The list of stop-words is included in the listing the of the C++ program's source code in Appendix A.

[10] The list is available at http://ir.dcs.gla.ac.uk/resources/linguistic_utils/. Other stopword lists are listed in Frakes and Baeza-Yates and in the CD-ROM enclosed in Belew. All of these lists are quite similar in content and length.

Fourth Data Set: Zipfian Distribution with stemming and stopwords

1. Read in a file,
2. Tokenize the contents into distinct words,
3. Fold case for each token, i.e. convert all words to be solely lowercase,
4. Remove all punctuation,
5. *Compare each token to a stoplist, if the token found on list, discard that token, then get the next token in the document*,
6. *Stem each word using the Porter Stemmer*,
7. Hash each stemmed token into a hash table,
8. Set the frequency of each stemmed word equal to one when initially inserted into the hash table.
9. If a stemmed word is already in the table, increment its frequency by one,
10. And finally, print each unique stemmed word and its number of occurrences into a comma-separated file for further analysis.

For this data set, the tokens must be compared to the stoplist before stemming as the stoplist consists of unstemmed words.

Fifth Data Set: Zipfian Distribution with consecutive word pairs

1. Read in a file,
2. Tokenize the contents into distinct pairs of words,
3. Fold case for each token, i.e. convert all words to be solely lowercase,
4. Remove all punctuation,

5. Set the frequency of each phrase equal to one when initially inserted into the hash table.
6. If a phrase is already in the table, increment its frequency by one,
7. And finally, print each unique two-word phrase and its number of occurrences into a comma-separated file for further analysis.

This data set is composed of two-word phrases: these are words that appear after each other consecutively in the document. Consider the sentence: 'The cat sat on the hat'. From this input, we would derive the following two-word phrases:

- 'the cat'
- 'cat sat'
- 'sat on'
- 'on the'
- 'the hat'

It is important to note that due to the algorithm used to construct phrases, words that are separated solely by end of sentence punctuation are considered adjacent to each other. It could be argued that this might create fallacious word phrases. However, consecutive sentences usually are 'about' the same or a similar topic, thus leaving open the possibility that these tokens might posses some information resolving power. If, on the other hand, the same two-word phrase occurs infrequently enough, its frequency rating will be so low that it should have little if any effect on the results of this data set.

Sixth Data Set: Zipfian Distribution of consecutive word pairs with stemming, no stopword removal

1. Read in a file,
2. Tokenize the contents into distinct pairs of words,
3. Fold case for each token, i.e. convert all words to be solely lowercase,
4. Remove all punctuation,
5. Use the Porter stemmer on each token in the phrase,
6. Set the frequency of each stemmed phrase equal to one when initially inserted into the hash table.
7. If a stemmed phrase is already in the table, increment its frequency by one,
8. And finally, print each unique two-word stemmed phrase and its number of occurrences into a comma-separated file for further analysis.

Seventh Data Set: Zipfian Distribution of consecutive word pairs with stopword removal, no stemming

1. Read in a file,
2. Tokenize the document into separate word tokens,
3. Remove all punctuation,
4. Compare the word tokens with the stopword list, remove any matches,
5. Create two-word phrases out of the remaining words,
6. Fold case for each token, i.e. convert all words to be solely lowercase,
7. Set the frequency of each phrase equal to one when initially inserted into the hash table.

8. If a phrase is already in the table, increment its frequency by one,
9. And finally, print each unique two-word phrase and its number of occurrences into a comma-separated file for further analysis.

There were two possible methods to construct this data set:

- Remove stop words *after* the two-word phrases have been constructed
- Remove stop words *before* the two-word phrases have been constructed

Using the first method, the sentence, "The cat sat on the hat." would generate the following phrase tokens:

- *'(blank) cat',*
- *'cat sat',*
- *'sat (blank)',*
- *'(blank) (blank)',*
- *'(blank) hat'*

While the construction of phrases that contain two blanks might appear frivolous, the count of these would allow us to monitor how frequently two stop words are adjacent to each other. It is conceivable that this would be useful in analyzing the lexical content of a document; ultimately we are more concerned with the meaning imparted by the use of significant terms.[11]

The second method enables us to examine significant terms that are adjacent or at most, one stopword away from each other. For example, with this method, the phrase *'retrieval of information'* would generate a phrase token of *'retrieval information'*, which can be considered to possess more information than any of the tokens the first method

[11] As a rough definition, we can describe any term as significant if it is not found on the stopword list.

would generate from this phrase: *'retrieval of'*, *'of information'*. This method also has the advantage of creating fewer phrases overall.

Eighth Data Set: Zipfian Distribution of consecutive word pairs, with stemming and stopword removal

1. Read in a file,
2. Tokenize the document into separate word tokens,
3. Remove all punctuation,
4. Fold case for each token, i.e. convert all words to be solely lowercase,
5. Compare the word tokens with the stopword list, remove any matches,
6. Create two-word phrases out of the remaining words,
7. Use the Porter stemmer on each word in a phrase token
8. Set the frequency of each stemmed phrase equal to one when initially inserted into the hash table.
9. If a stemmed phrase is already in the table, increment its frequency by one,
10. And finally, print each unique two-word stemmed phrase and its number of occurrences into a comma-separated file for further analysis.

With this data set, it was necessary for the stopwords to be removed before constructing the two-word phrase tokens, in a method similar to data set 7 described above. Similarly, each word was stemmed individually in the same method used in the sixth data set.

After raw data generation through the C++ program, each dataset was run through a Perl script that sorted the words and their frequency in descending order of frequency.[12] It then calculated the ranks for each frequency. When a dataset had more than one word (or phrase) that occurred with the same frequency, they were all assigned the same frequency and the next most frequently occurring word after this was given the rank equal to last rank assigned plus the number of words that held that rank. Table 1 shows an example of this method of ranking.

Word	Frequency	Rank
The	1000	1
Of	700	2
And	700	2
A	500	4
An	500	4
Or	500	4
But	400	7

Table 1. Example of results of the ranking algorithm used by Perl script

Other values calculated by the Perl script for each word:

- The word's probability of occurring in the document, Pr(x) = frequency of word x occurrences/ total number of words in the document.
- The 'Zipfian constant' equal to the product of the rank times the Pr(x).
- The $\log_{10}$ of the frequency and rank.
- The information content of a word x, equal to $-\log_2 (Pr(x))$.

[12] The Perl script is included in Appendix A along with the C++ source code.

Since we are concerned only with words that might reflect what a document is about, a conservative cut-off value for frequency was employed: words and phrases which occurred fewer than five times were not used in further analysis. While the many of the words in a Zipfian distribution lie in the tail, we are concerned with words that have a chance at being considered significant. Additionally, the space and processing time taken up by analyzing low frequency words is considerable, thus it was decided to employ a conservative lower cut-off of words with frequencies greater than four. The rest of the analysis refers to datasets that used this cut-off.

It is necessary to mention the shorthand notation that is used to refer to each dataset: a three-letter string of the letters 'y' or 'n' refers to each dataset. The first letter is a 'y' if the dataset uses stemming, an 'n' if not. The second letter refers to stopwords and the third to the use of multi-word phrases. A number that refers to the corresponding Reuters text collection file number follows the three letters. Thus the string 'nyy5' refers to the dataset generated by the 5 fifth Reuters file when stopwords and phrases were used, but not stemming.

After the Perl script was run, each dataset was placed into an Excel spreadsheet in order to calculate the regression coefficients for a graph of the log frequency over log rank, where rank was greater than four. These values along with the R^2 and standard deviation of errors were tabulated for comparison between different dataset types.[13]

Finally, a model for predicting the direction and size of change to a Zipfian distribution in response to the use of any combination of stemming, stopword removal

[13] Unfortunately, what Excel calls the standard deviation of errors, simply the standard error. In reality this value is the measure of spread of the data around the regression line, with the same properties of a normal standard deviation. Thus, about 95% of the data points lie within 2 * the standard deviation of the errors from the regression line.

and phrases frequency will be constructed by analyzing the changes to the regression slope and intercept values across the collected datasets. This model will then be used to predict regression slopes and intercepts for two other randomly chosen files in the Reuters text collection.[14] Two sets of analysis of variance (ANOVA) tests with a 95% confidence level will be run:

1. To determine if differences in the actual values of the regression coefficients of the datasets are significantly different from the predicted values. The null hypotheses: H_0 $slope_{predicted} = slope_{actual}$ and H'_0 $intercept_{predicted} = intercept_{actual}$. The alternate hypotheses are: H_1 $slope_{predicted} \neq slope_{actual}$ and H'_1 $intercept_{predicted} \neq intercept_{actual}$.
2. To determine if changes in regression coefficients from one dataset type to another are significantly different from the predicted changes in coefficients. The null hypotheses: H_0 $\Delta slope_{predicted} = \Delta slope_{actual}$ and H'_0 $\Delta intercept_{predicted} = \Delta intercept_{actual}$. The alternate hypotheses are: H_1 $\Delta slope_{predicted} \neq \Delta slope_{actual}$ and H'_1 $\Delta intercept_{predicted} \neq \Delta intercept_{actual}$. (Where Δ signifies change)

The results of the first ANOVA will tell us if the datasets whose values we are predicting are significantly difference from the datasets that we are using to make our predictions from. The results of the second ANOVA will tell us if the changes in regression coefficients between datasets is significantly different from the changes observed in the original seven datasets. There are four possible combinations of results from these tests:

[14] Reuters012.sgm and Reuters017.sgm were the randomly selected files to be tested. The dataset filenames are referred to similar fashion to the initial data, i.e. nnn12, nyn17, etc.

1. If the results of the first ANOVA are significant, but the results from the second ANOVA are not, then depending on the datasets in question, we can conclude that stemming, stopwords and phrases change the Zipfian distribution regression coefficients in a constant manner.
2. If the results of the first and second ANOVA are significant, then we will need to look at more complicated models of prediction, such as scaling the changes based on the original regression coefficients.
3. If the results of the first and second ANOVA are not significant, then our model will have made accurate predictions about datasets that are very similar to the datasets used to build the model.
4. If the results of the first ANOVA is not significant, but the results of the second ANOVA is, then our model will have failed to accurately predict values for a set of data that is similar to the data used to create the model.

Results and Discussion

Tables 2 through 9 summarize the regression coefficients, the R^2 value and the standard deviation of error for each file within each dataset, as well as providing the mean and standard deviation for each value.

Dataset Filename	Slope	Intercept	R squared	Estimated standard deviation of errors
nnn0	-1.08820	4.48576	0.98984	0.04662
nnn1	-1.08514	4.44032	0.99060	0.04472
nnn2	-1.08041	4.40538	0.99208	0.04074
nnn3	-1.08300	4.46019	0.99059	0.04462
nnn4	-1.09285	4.49692	0.99075	0.04460
nnn5	-1.09653	4.53619	0.99008	0.04641
nnn6	-1.09469	4.46442	0.98950	0.04760
Mean	*-1.08869*	*4.46988*	*0.99049*	*0.04504*
Standard deviation	*0.00571*	*0.03880*	*0.00077*	*0.00207*

Table 2. Summary of regression coefficients for dataset with no stemming, stopwords or phrases

Dataset Filename	Slope	Intercept	R squared	Estimated standard deviation of errors
nny0	-0.77267	3.39446	0.99920	0.00904
nny1	-0.78765	3.41592	0.99902	0.01026
nny2	-0.78831	3.39198	0.99912	0.00966
nny3	-0.77387	3.39392	0.99881	0.01111
nny4	-0.76882	3.38497	0.99930	0.00844
nny5	-0.77611	3.43073	0.99915	0.00936
nny6	-0.78529	3.40581	0.99918	0.00931
Mean	*-0.77896*	*3.40254*	*0.99911*	*0.00960*
Standard deviation	*0.00736*	*0.01485*	*0.00015*	*0.00080*

Table 3. Summary of regression coefficients for dataset with phrases and no stemming or stopwords

Dataset Filename	Slope	Intercept	R squared	Estimated standard deviation of errors
nyn0	-0.96430	4.03208	0.98363	0.05253
nyn1	-0.95946	3.98257	0.98551	0.04914
nyn2	-0.95436	3.94761	0.98774	0.04479
nyn3	-0.95723	4.00250	0.98414	0.05133
nyn4	-0.96469	4.02947	0.98587	0.04872
nyn5	-0.96667	4.06194	0.98394	0.05215
nyn6	-0.97168	4.01477	0.98453	0.05135
Mean	*-0.96263*	*4.01013*	*0.98505*	*0.05000*
Standard deviation	*0.00551*	*0.03441*	*0.00133*	*0.00251*

Table 4. Summary of regression coefficients with stopwords, no stemming or phrases

Dataset Filename	Slope	Intercept	R squared	Estimated standard deviation of errors
ynn0	-1.17296	4.72031	0.97770	0.07507
ynn1	-1.16611	4.66326	0.98055	0.06964
ynn2	-1.15548	4.61493	0.98363	0.06311
ynn3	-1.16803	4.69511	0.98002	0.07060
ynn4	-1.17759	4.73112	0.97970	0.07177
ynn5	-1.18148	4.77162	0.97882	0.07370
ynn6	-1.17050	4.67352	0.97998	0.07085
Mean	*-1.17031*	*4.69570*	*0.98006*	*0.07068*
Standard deviation	*0.00782*	*0.04735*	*0.00170*	*0.00355*

Table 5. Summary of regression coefficients for dataset with stemming, no stopwords or phrases

Dataset Filename	Slope	Intercept	R squared	Estimated standard deviation of errors
nyy0	-0.80962	3.18636	0.99793	0.01505
nyy1	-0.84444	3.26778	0.99821	0.01466
nyy2	-0.85409	3.27314	0.99758	0.01720
nyy3	-0.79304	3.13464	0.99653	0.01915
nyy4	-0.80036	3.16248	0.99518	0.02274
nyy5	-0.81614	3.20733	0.99714	0.01787
nyy6	-0.84389	3.26318	0.99688	0.01926
Mean	*-0.82308*	*3.21356*	*0.99706*	*0.01799*
Standard deviation	*0.02236*	*0.05150*	*0.00094*	*0.00256*

Table 6. Summary of regression coefficients for dataset with phrases and stopwords and no stemming

Dataset Filename	Slope	Intercept	R squared	Estimated standard deviation of errors
yny0	-0.76304	3.39386	0.99883	0.01085
yny1	-0.77462	3.40535	0.99893	0.01054
yny2	-0.77961	3.39329	0.99899	0.01024
yny3	-0.76377	3.39109	0.99863	0.01176
yny4	-0.75968	3.38671	0.99934	0.00810
yny5	-0.76368	3.42367	0.99926	0.00864
yny6	-0.77500	3.40230	0.99908	0.00976
Mean	*-0.76849*	*3.39947*	*0.99901*	*0.00998*
Standard deviation	*0.00714*	*0.01153*	*0.00023*	*0.00118*

Table 7. Summary of regression coefficients with stemming and phrases, no stop

Dataset Filename	Slope	Intercept	R squared	Estimated standard deviation of errors
yyn0	-1.05728	4.29235	0.97375	0.07344
yyn1	-1.04860	4.23144	0.97851	0.06580
yyn2	-1.03732	4.18264	0.98238	0.05873
yyn3	-1.05013	4.26242	0.97626	0.06920
yyn4	-1.05606	4.28531	0.97765	0.06752
yyn5	-1.05871	4.32107	0.97492	0.07191
yyn6	-1.05609	4.25040	0.97808	0.06684
Mean	*-1.05203*	*4.26080*	*0.97737*	*0.06763*
Standard deviation	*0.00693*	*0.04194*	*0.00260*	*0.00444*

Table 8. Summary of regression coefficients with stemming, stopwords, no phrases

Dataset Filename	Slope	Intercept	R squared	Estimated standard deviation of errors
yyy0	-0.78278	3.15019	0.99762	0.01564
yyy1	-0.81676	3.22919	0.99817	0.01438
yyy2	-0.83035	3.24359	0.99780	0.01598
yyy3	-0.76921	3.10697	0.99644	0.01889
yyy4	-0.77384	3.12637	0.99529	0.02173
yyy5	-0.78406	3.15966	0.99700	0.01763
yyy6	-0.82053	3.23488	0.99742	0.01706
Mean	*-0.79679*	*3.17870*	*0.99711*	*0.01733*
Standard deviation	*0.02310*	*0.05206*	*0.00091*	*0.00225*

Table 9. Summary of regression coefficients with stemming, stopwords and phrases

Reuters files twelve and seventeen were used to as the randomly selected files to be compared against the model predicted by the above data. This data is summarized in table 10.

Dataset Name	Slope	Intercept	R squared	Estimated standard deviation of errors
nnn12	-1.08276	4.46979	0.99163	0.04209
nnn17	-1.10845	4.63873	0.98945	0.04850
nny12	-0.75768	3.34693	0.99919	0.00896
nny17	-0.75538	3.42630	0.99934	0.00808
nyn12	-0.95376	3.99978	0.98627	0.04756
nyn17	-0.97690	4.15532	0.98200	0.05600
nyy12	-0.77231	3.07631	0.99325	0.02611
nyy17	-0.73437	3.02113	0.99538	0.02053
ynn12	-1.16478	4.69843	0.98027	0.07010
ynn17	-1.20144	4.89392	0.97640	0.07932
yny12	-0.74620	3.34188	0.99926	0.00844
yny17	-0.75119	3.44294	0.99891	0.01032
yyn12	-1.04335	4.25338	0.97644	0.06867
yyn17	-1.07467	4.42780	0.97072	0.07918
yyy12	-0.74823	3.04798	0.99308	0.02570
yyy17	-0.71474	3.00213	0.99522	0.02036

Table 10. Summary of regression coefficients for test data

The first ANOVA test that was run was a comparison of two groups, the seven files that make up each dataset for prediction and the two random files used as test data. Table 11 summarizes the results of the first ANOVA for the slope coefficient, and indicates if the differences between the two groups were significant or not. If the result is significant, then the means of the two groups differs enough that we conclude they are not drawn from the same population.

Dataset type	Model or test data	Slope mean	Slope variance	F value	F critical	Significant?
nnn	model	-1.08869	0.00004	0.933548	5.59145974	NO
	test	-1.09561	0.00033			
nny	model	-0.77896	0.00006	14.34095	5.59145974	YES
	test	-0.75653	0.00000			
nyn	model	-0.96263	0.00004	0.165972	5.59145974	NO
	test	-0.96533	0.00027			
ynn	model	-1.17031	0.00007	1.624663	5.59145974	NO
	test	-1.18311	0.00067			
nyy	model	-0.82308	0.00058	12.55507	5.59145974	YES
	test	-0.75334	0.00072			
yny	model	-0.76849	0.00006	11.5606	5.59145974	YES
	test	-0.74870	0.00001			
yyn	model	-1.05203	0.00006	0.642321	5.59145974	NO
	test	-1.05901	0.00049			
yyy	model	-0.79679	0.00062	10.81411	5.59145974	YES
	test	-0.73148	0.00056			

Table 11. ANOVA results of comparison of means between model and test data regression slopes

Table 12 lists the ANOVA results for the comparison of means of regression intercepts between model and test data. Again, the test is significant if the F value is greater than the F critical value, indicating that the test and model data are statistically different.

Dataset type	Model or test data	Intercept mean	Intercept Variance	F value	F critical	Significant?
nnn	Model	4.4698824	0.001757	3.124628	5.59145974	NO
	Test	4.554259	0.014271			
nny	Model	3.4025411	0.000257	0.588484	5.59145974	NO
	Test	3.3866145	0.003149			
nyn	Model	4.0101342	0.001381	2.427889	5.59145974	NO
	Test	4.0775528	0.012096			
ynn	Model	4.6956955	0.002615	3.158917	5.59145974	NO
	Test	4.7961726	0.019108			
nyy	Model	3.2135587	0.003095	14.72725	5.59145974	YES
	Test	3.0487164	0.001522			
yny	Model	3.3994676	0.000155	0.089929	5.59145974	NO
	Test	3.3924063	0.005107			
yyn	Model	4.2608047	0.002052	2.518557	5.59145974	NO
	Test	4.3405886	0.015211			
yyy	Model	3.1786953	0.003162	12.83618	5.59145974	YES
	Test	3.0250537	0.001051			

Table 12. ANOVA results of comparison of means between model and test data regression intercepts

Using the unaltered dataset, referred to as 'nnn', as a baseline, the change observed in the regression coefficients was compiled in tables 13 and 14.

Reuters Filename	nnn	nny	nyn	ynn	nyy	yny	yyn	yyy
0	-1.08820	0.31553	0.12390	-0.08476	0.27858	0.32516	0.03092	0.30542
1	-1.08514	0.29750	0.12568	-0.08096	0.24071	0.31052	0.03654	0.26838
2	-1.08041	0.29210	0.12605	-0.07507	0.22632	0.30080	0.04309	0.25006
3	-1.08300	0.30913	0.12577	-0.08503	0.28996	0.31923	0.03287	0.31379
4	-1.09285	0.32402	0.12816	-0.08474	0.29249	0.33316	0.03678	0.31901
5	-1.09653	0.32041	0.12986	-0.08495	0.28038	0.33285	0.03782	0.31246
6	-1.09469	0.30941	0.12302	-0.07580	0.25080	0.31969	0.03860	0.27417
12	-1.08276	0.32508	0.12900	-0.08202	0.31045	0.33656	0.03941	0.33454
17	-1.10845	0.35307	0.13155	-0.09299	0.37408	0.35726	0.03378	0.39371

Table 13. Summary of changes from baseline dataset 'nnn' for regression slope coefficients

Reuters Filename	nnn	nny	nyn	ynn	nyy	yny	yyn	yyy
0	4.48576	-1.09130	-0.45368	0.23455	-1.29939	-1.09190	-0.19341	-1.33556
1	4.44032	-1.02440	-0.45775	0.22294	-1.17254	-1.03497	-0.20888	-1.21113
2	4.40538	-1.01340	-0.45777	0.20956	-1.13223	-1.01208	-0.22274	-1.16179
3	4.46019	-1.06627	-0.45769	0.23491	-1.32556	-1.06910	-0.19777	-1.35322
4	4.49692	-1.11195	-0.46745	0.23419	-1.33445	-1.11021	-0.21161	-1.37055
5	4.53619	-1.10546	-0.47425	0.23543	-1.32886	-1.11252	-0.21512	-1.37653
6	4.46442	-1.05861	-0.44964	0.20910	-1.20124	-1.06211	-0.21402	-1.22954
12	4.46979	-1.12285	-0.47000	0.22864	-1.39348	-1.12791	-0.21641	-1.42181
17	4.63873	-1.21243	-0.48341	0.25519	-1.61760	-1.19579	-0.21093	-1.63660

Table 14. Summary of changes from baseline dataset 'nnn' for regression intercept coefficients

The data for the second ANOVA test was drawn from tables 13 and 14. In this test, the changes between datasets were compared between the model data (Reuters filenames 0-6) and the test data (Reuters filenames 12 and 17). The confidence was the same as the first ANOVA, 95%. Tables 15 and 16 summarize the results of this test.

Dataset type	Model or test	Slope change mean	Slope change variance	F Value	F critical	Significant?
nny	model	0.30973	0.00014	7.784759	5.59146	YES
	test	0.33908	0.00039			
nyn	model	0.12606	0.00001	5.329494	5.59146	NO
	test	0.13028	0.00000			
ynn	model	-0.08162	0.00002	2.102003	5.59146	NO
	test	-0.08751	0.00006			
nyy	model	0.26560	0.00068	10.47523	5.59146	YES
	test	0.34227	0.00202			
yny	model	0.32020	0.00014	7.481784	5.59146	YES
	test	0.34691	0.00021			
yyn	model	0.03666	0.00002	0.000432	5.59146	NO
	test	0.03659	0.00002			
yyy	model	0.29190	0.00074	9.179383	5.59146	YES
	test	0.36412	0.00175			

Table 15. Summary of second ANOVA test on significance of changes in the regression slope between the model and test data

Dataset type	Model or test	Intercept change mean	Intercept change variance	F Value	F critical	Significant?
nny	model	-1.06734	0.00147	8.51811	5.5914597	YES
	test	-1.16764	0.00401			
nyn	model	-0.45975	0.00007	6.13307	5.5914597	YES
	test	-0.47671	0.00009			
ynn	model	0.22581	0.00015	2.302417	5.5914597	NO
	test	0.24191	0.00035			
nyy	model	-1.25632	0.00724	9.861486	5.5914597	YES
	test	-1.50554	0.02512			
yny	model	-1.07041	0.00142	8.389119	5.5914597	YES
	test	-1.16185	0.00230			
yyn	model	-0.20908	0.00010	0.357542	5.5914597	NO
	test	-0.21367	0.00001			
yyy	model	-1.29119	0.00773	8.886187	5.5914597	YES
	test	-1.52921	0.02307			

Table 16. Summary of the second ANOVA test of significant of changes in the regression intercept between the model and test data

For all datasets, the linear regression analysis of the log frequency/log rank data showed extremely high r^2 values, with the lowest average r^2 equal to 0.97737. (Tables 2-9) The estimated standard deviation of errors, the standard deviation of the distance from the actual points to the regression line, was also quite small, with average values of about 2%. Taken together, we conclude that the regression lines for each dataset was linear, thus confirming that each dataset follows a Zipfian distribution.

The next question to consider is the mathematical effect that stemming, stopwords and phrases, have on the regression coefficients of a Zipfian distribution. The Reuters file numbers 0 through 6 were used to construct a model for prediction change and Reuters file numbers 12 and 17 were used as test data to compare the accuracy of the prediction by the model. Tables 11 and 12 show the results of the first ANOVA test.

The purpose of this test was to determine if the regression coefficients were significantly different between the model data and the test data. These tables show that in all the instances when phrases were used (dataset types ‘nny’, ‘yny’, ‘nyy’, ‘yyn’), the ANOVA resulted in the rejection of the null hypothesis, i.e. the test regression slopes were significantly different than the model slope values. For the intercept ANOVA test, only the dataset types ‘nyy’ and ‘yyy’ gave significant values. Note that both of these involve phrases and stopwords. Based on the results of these ANOVA’s, we conclude the model and test data is comparable in the instances when stemming and stopwords are used, that is, we can assume that there is not a significant difference between the regression coefficients in the model and test data. When phrases are used, the resulting changes to the test data’s Zipfian distribution are significantly different from the model data’s Zipfian distribution.

The results of the second ANOVA (tables 15 and 16) show similar results. Table 15 provides evidence for the assertion that the changes in slope from the baseline case of a Zipfian distribution with no stemming, stopwords or phrases to the set of Zipfian distributions with some combination of stemming and stopwords. In each of these dataset types (‘ynn’, ‘nyn’, and ‘yyn’) the difference between the model and test data was deemed to be insignificant. Similarly, table 16 shows we can make the same conclusion for changes in intercept, except in the case when stopwords alone are used (dataset type ‘nyn’). However, the change in intercept value in the dataset where stemming is used in conjunction with stopwords is not significant. Also, the change in slope for all datasets that did not use phrases was non-significant. Thus, this odd result might be a result of the limited numbers of samples that make up the model and test data. The first ANOVA

showed that we could consider these datasets to be similar. This implies that the changes from the baseline case due to stemming and stopwords should also be comparable. Combined with the results of the first ANOVA, we can conclude *that the use of stemming and stopwords change the Zipfian distribution in a predictable manner*.[15]

The second ANOVA results for datasets that used phrases in addition to stemming and stopwords were all significant. However, since the results of the first ANOVA showed that the model and test datasets with phrases were significantly different, we should not be surprised to see changes that are unpredictable by our model.

Using the data in tables 13 and 14, we can use the mean values from the model data to tabulate the changes in slope and intercept as we move from the baseline Zipfian distribution to distributions with stemming and stopwords. Because the changes due to the use of phrases cannot be predicted, the only changes that are included are the combination of stemming and stopwords:

From 'nnn' to:	slope	intercept
nyn	0.12606	-0.45975
ynn	-0.08162	0.22581
yyn	0.03666	-0.20908

Table 17. Effects of stemming and stopwords on the linear regression of a Zipfian distribution from the model data

Notice that the effects of stopwords and stemming are in opposite directions. Indeed, the sum of the effects of these singularly is fairly close to the changes of these in combination.

[15] Although we should be aware of the significant change in intercept in the case when stopwords alone were used.

Since these are for a linear regression of a log-log graph, we can convert these values into an equation representing the Zipfian distribution of frequency against rank. Using the identity:

$$\log y = n \log x + \log a \leftrightarrow y = ax^n \quad (1)$$

we can see the slope of the log-log line is the exponent of the rank, and the inverse of the log of the intercept is the coefficient of the rank. Thus, we arrive at this form of the Zipfian distribution:

$$frequency = 10^{intercept} \cdot rank^{slope} \quad (2)$$

From (2) it follows that changes to the regression coefficients actually have an exponential effect on the frequency. (2) is a form of a power law, as Zipf himself first observed. (Zipf, 1949)

While we cannot establish a model for the use of phrases, we can compare our results to those of Smith and Devine (Smith & Devine, 1985). From their data, the average change in regression slope from a normal Zipfian distribution to a two-word phrase Zipfian distribution was 0.375. This study's data has an average change of 0.3097 (from table 15). Clearly, these values are dissimilar. However, Smith and Devine's could be used along with this studies data in an attempt to find a formula for the Zipfian distribution of multi-word phrases.

Egghe claimed that multi-word phrases do not follow a Zipfian distribution (Egghe, 1998). But looking at the regression data for the datasets involving phrases, we see a very high R^2 value of at least 0.997 for these datasets (see tables 3, 6, 7, and 9). While this study cannot directly contradict Egghe's mathematical proof, the data gathered can be considered as strong evidence that if Zipf's law is not followed by multi-word

phrases, then it approximates closely enough to be considered a Zipfian distribution in practice.

Conclusion

The purpose of this investigation has been to investigate the effects of stemming, stopwords and phrases on the Zipfian distribution of a text. Do stemming, stopwords, and phrases preserve the Zipfian distribution of a text? The results in tables 2-9 show that any combination of these operations preserve the Zipfian distribution. Can the effects of stemming, stopwords and phrases be predicted? The results of the two ANOVA's in tables 11, 12, 15, and 16, show the effects of stemming, stopwords and their combination can be predicted, but the effects of phrases on a Zipfian distribution cannot be accurately predicted.

This study is unique in its attempts to discern the effects of stemming, stopwords and phrases on Zipfian distributions, no other investigation has been conducted into the effects these operations have on Zipf's law.

However, this study used a small sample for constructing the model data. Future avenues of investigation could look into testing these conclusions on larger sample sets. It is also possible that the use of phrases does have a predicable effect on the Zipfian distribution of a test, perhaps the regression coefficients are changed in some non-linear manner. This too, could be another avenue for further research.

References

Belew, R. K. (2000). Finding Out About. New York: Cambridge University Press.

Egghe, L. (1999). On the law of Zipf-Mandelbrot for multi-word phrases. *Journal of the American Society for Information Science*, *50*(3), 233–241.

Estroup, J. B. *Gammes sténographique*, Institut Stenographique de France, Paris, 1916

Fox, C. (1992). Lexical analysis and stoplists. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 102-130. Englewood Cliffs, NJ: Prentice Hall.

Frakes, W. (1992) Stemming algorithms. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 102-130. Englewood Cliffs, NJ: Prentice Hall.

Huberman, B.A., Pirolli, P. L., Pitkow, J. E., and Lukose, R. M. (1998(Strong regularities in World Wide Web surfing. *Science*, 280:95-97.

Luhn, H. (1958). The automatic creation of literature abstracts. *IBM Journal of Research and Development*, *2*(2), 159–165. The article is also included in *H. P. Luhn: Pioneer of Information Science, Selected Works*.

Mandelbrot, B. (1961). On the theory of word frequencies and on related Markovian models of discourse. In *Structure of Language and Its Mathematical Aspects: Proceedings of Symposia in Applied Mathematics, vol. XII*, pp. 190–219. American Mathematical Society.

Mantegna, R. N., Buldyrev, S. V., Goldberger, A. L., Havlin, S., Peng, C.-K., Simons, M., and Stanlet, H.E. (1994) Linguistic features of noncoding DNA sequences. *Physical Review Letters,* 73(23):3169-3172.

Porter, M. F. (1980). "An Algorithm for Suffix Stripping." *Program,* 14(3), 130-137.

Salton, G., & McGill, M. (1983). *Introduction to Modern Information Retrieval*. New York: McGraw-Hill.

Shannon, C. E., & Weaver, W. (1949). *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, Ill.

Smith, F. J., & Devine, K. (1985). Storing and retrieving word phrases. *Information Processing and Management*, *21*(3), 215–224.

van Rijsbergen, C. J. (1975). *Information Retrieval*. Boston: Butterworths.

Zipf, G. K**.** (1949). *Human Behavior and the Principle of Least Effort.* New York: Addison-Wesley

Appendix A

Source Code

Makefile

```
# makefile by Zach Sharek
# used to compile Zipfian text parser
# see main.cpp for more info
# USAGE: 'make zipf'
#September, 2002
zipf: read_words.o hash_table.o main.o porter.o
      g++ -o zipf read_words.o hash_table.o main.o porter.o
hash_table.o: hash_table.cpp hash_table.h
      g++ -c hash_table.cpp
read_words.o: read_words.cpp read_words.h
      g++ -c read_words.cpp
main.o: main.cpp
      g++ -c main.cpp
porter.o: porter.cpp porter.h
      g++ -c porter.cpp
clean:
      rm *.o zipf
```

```
/*************************************************************************
*******
 * Word Frequency Text Parser
 *
 * (c) Zach Sharek 9/2002
 * The command line interface: ./zipf stem stop phrase filename
 * Where stem, stop and phrase are 'y' or 'n'
 * filename is the path to the file to be processed
 * Example: ./zipf y n y ./testdata/test.txt
 * This would process the test.txt file with stemming, no stopwords
 * and phrase creation.
 *
 * The program's output is an unsorted frequency list
 * It is recommended that it redirected on the command line
 * to an output file, the sort command can also be used.
 *
 * The porter stemmer class was written by Al Popescul, based
 * on a Java version by Fotis Lazarinis
 * with modifications by Z. Sharek
 * The stopword list is from:
 *
http://ftp.dcs.glasgow.ac.uk/idom/ir_resources/linguistic_utils/stop_wo
rd
 *
 * Comments can be sent to zsharek@hotmail.edu

*************************************************************************
*****/

#include <string>
#include <vector>
#include <fstream.h>
#include <stdlib.h>
#include <ctype.h>
#include "read_words.h"
#include "hash_table.h"
using namespace std;


int main(int argc, char* argv[])
{
  char stem = *argv[1];     // 'y' or 'n' for stemming
  char stop = *argv[2];     // 'y' or 'n' for stopwords
  char phrase = *argv[3];   // 'y' or 'n' for phrase making
  // argv[4] is the filename to process

  ReadWords inputFile(argv[4]); //create link to file
  HashTable zipfTable(stem, stop, phrase); //create hashtable
  zipfTable.FillTable(inputFile); //populate hashtable with input file
  inputFile.CloseFile(); //close file
  zipfTable.PrintTable();  // print out data, can be redirected

  return 0;
}
```

```
/*************************************************************
   Contains the headers for the classes ReadWords and HashTable
   Written by Zach Sharek, April 11, 2002
*************************************************************/

#ifndef _HASHTABLE
#define _HASHTABLE

#include "porter.h"

const int TABLESIZE = 1009; //the size of the hashtable's array

struct Record
{
      string word;
      int frequency;
      int docID;
};

class HashTable
{
protected:
  vector<Record> hash[TABLESIZE]; //an array of vectors of Records,
  //holds words from input file with their frequency
  int functionNum; //user selected hash function to use
  Record currentWord; //current word received from input file
  Record lastWord;    //last word processed
  Record tempWord;    //temporary string
  int hashValue; //hashValue of currentWord
  char stem, stop, phrase;  // flags for these options
  Porter stemmer;
 public:
  HashTable(); //constructor, selects hash function to use

  // first char is for stemming, second is for stopwords, third is
phrases
  HashTable(char, char, char);
  void HashFunction(); // hashes words according to 1 of 3 hash
functions
  void GetNextWord(ReadWords& openFile); //gets next word in file
  void Find(); //searches if a word is already in the hash table
  void Insert(); //puts in a new word into the hash table
  void FillTable(ReadWords& openFile); // controls how the file is
hashed
  void PrintStats(); //prints stats according to hashfunction choosen
  void CheckFile(ReadWords& openFile);
  void PrintTable();
};

#endif
```

```
/*************************************************************
hash_table.cpp
Zach Sharek
*************************************************************/

#include <string>
#include <vector>
#include <fstream.h>
#include <stdlib.h>
#include <ctype.h>
#include "read_words.h"
#include "hash_table.h"
using namespace std;


// HashTable asks user which hash function to use
// Pre: no hash function selected
// Post: user has selected a hash function
HashTable::HashTable()
{
  functionNum = 2;
  stem = 'n';
  stop = 'n';
  phrase = 'n';
}

HashTable::HashTable(char stemWords, char stopWords, char phraseWords)
{
      functionNum = 2;
      stem = stemWords;
      stop = stopWords;
      phrase = phraseWords;
      lastWord.word = "";
      tempWord.word = "";
}

// HashFunction- hashes words according to hash functions choosen
// Pre: currentWord does not have a corresponding hash value
// Post: currentWord has a corresponding hash value
void HashTable::HashFunction()
{

  hashValue = 0;

  if (functionNum == 1)
    {
      // this takes the sum of the ASCII of each letter in currentWord
      // then adjusts for table size
      for (int i=0; i <tempWord.word.length(); i++)
      hashValue += tempWord.word[i];
        hashValue %= TABLESIZE;
    }

 if (functionNum == 2)
    {
      // this takes the square of the ASCII of each letter in
currentWord
```

```
      // and sums them, then adjusts for table size
      for (int i=0; i <tempWord.word.length(); i++)
        hashValue = hashValue + (tempWord.word[i] * tempWord.word[i]);
      hashValue %= TABLESIZE;
    }

 if (functionNum == 3)
   {
     // this takes the ASCII value of each letter and * by length of
the word
     // and sums these values and adjusts for table size
     for (int i=0; i <tempWord.word.length(); i++)
       hashValue = hashValue + (tempWord.word[i] *
tempWord.word.length());
     hashValue %=TABLESIZE;
   }
}

// Insert- takes a word and places it into the appropiate vector
//         by its hashValue
// Pre: currentWord has not been hashed yet
// Post: currentWord has been inserted at the back of its hashValue's
vector
void HashTable::Insert()
{
    tempWord.frequency = 1;
    tempWord.docID = 1;
    hash[hashValue].push_back(tempWord);
}

// Find- checks if currentWord exists in the hash table,
//       calls Insert() if not.
// Pre: none
// Post: currentWord has been found in hash table or
//       Insert has been called
void HashTable::Find()
{
  int flag = 0;
  for( int i=0; i<(hash[hashValue].size()); i++)
    {
      if (tempWord.word == hash[hashValue][i].word)
      {
      hash[hashValue][i].frequency++;
        flag = 1;
      return;
      }
    }
  // or else the word is not found so is added to the table
  if (flag == 0 && tempWord.word != "") //blank spaces are not
considered words
      Insert();
}

//GetNextWord- gets the next word from openFile
// Pre:  openFile is connected to a text file
// Post: the next word in openFile has been returned and set to
currentWord
```

```
void HashTable::GetNextWord(ReadWords& openFile)
{
  if (!openFile.EoF())
    currentWord.word = openFile.GetNextWord();
}

//FillTable- gets all the words in openFile,
// hashes them and inserts them into hash table
// Pre: openFile is connected to a text file and no words have yet been
returned
// Post: every word in openFile has been hashed into the hashtable
void HashTable::FillTable(ReadWords& openFile)
{
  while(!openFile.EoF())
    {
      GetNextWord(openFile);

      // stopwords
      if (stop == 'y')
      {
        currentWord.word = stemmer.stopWordStrip(currentWord.word);
      }

      // stemming
      if (stem == 'y')
      {
        currentWord.word = stemmer.stripAffixes(currentWord.word);
      }

      // make 2 word phrases
      if (phrase == 'y')
      {
        if (lastWord.word == "")
          {
            if (currentWord.word != "")
            {
              lastWord.word = currentWord.word;
              continue;
            }
            else
            continue;
          }
        else // execute if lastWord is not blank
          {
            if (currentWord.word != "")
            {
              tempWord.word = lastWord.word + " " + currentWord.word;
              lastWord.word = currentWord.word;
            }
            else
            continue;
          }
      }

      if (phrase != 'y')
      tempWord.word = currentWord.word;
```

```
      HashFunction();
      Find();
    }
}

// CheckFile- takes each word in the user's files and
// checks if it exists in the hash table
// Pre: no words have yet been returned from user's file
// Post: every word has been entered into the hash table
void HashTable::CheckFile(ReadWords& openFile)
{
  while(!openFile.EoF())
    {
      GetNextWord(openFile); //gets next word
      HashFunction(); //gets its hashValue
      Find(); //see if the word exists in hash table
    }
}

// PrintStats- displays the lengths of longest list in hash table as
well
// as the number of lists of length 3 or more and of length 10 or more
// Pre:  the hash table has been generated according to hash function
// Post: the hash table stats have been displayed on the screen
void HashTable::PrintStats()
{
  // these integers hold the longest list in the hash table, and the
number
  // of vectors in the table of length 3 or greater and 10 or greater
  int longList = 0, length_3 = 0, length_10 = 0;

  // find the values of the above variables
  for (int i=0; i<TABLESIZE;i++)
    {
      if (hash[i].size() > longList)
      longList = hash[i].size();
      if (hash[i].size() >= 3)
      length_3++;
      if (hash[i].size() >= 10)
      length_10++;
    }

  //output the results of the above loop:
  cout << "\nStatistics:" << endl;
  cout << "Using hash function number " << functionNum << endl;
  cout << "The longest list in the hash table is of size " << longList
<< endl;
  cout << "There are " << length_3 << " lists of length 3 or more." <<
endl;
  cout << "There are " << length_10 << " lists of length 10 or more."
<< endl;
}

void HashTable::PrintTable()
{
  for (int i=0;i<TABLESIZE;i++)
    {
```

```
      if (hash[i].size() > 0)
      {     for (int j=0;j<hash[i].size();j++)
        cout << hash[i][j].word << ", " << hash[i][j].frequency <<
endl;
      }
    }
}
```

```
/*************************************************************
   Contains the header for the class ReadWords
   Written by Zach Sharek, April 11, 2002
*************************************************************/

#ifndef _READWORDS
#define _READWORDS

const int MAXCHARS =  80; //the max number of letters in a string

class ReadWords
{
private:
  ifstream inFile; //this is the stream used to read in text files
public:
  ReadWords(char inputfile[]); //this opens the file given as a
parameter
  void OpenFile(char inputfile[]); //opens selected file
  string GetNextWord(); //parses inFile word by word
  int EoF(); // returns true if end of file is reached
  void CloseFile(); //closes the file stream
};

#endif
```

```
/*****************************************************************
read_words.cpp
Zach Sharek

*****************************************************************/

#include <string>
#include <vector>
#include <fstream.h>
#include <stdlib.h>
#include <ctype.h>
#include "read_words.h"
using namespace std;


// ReadWords is a constructor with a filename specified
// Pre: a filename was given to be opened
// Post: OpenFile has been called to open a stream to the filename
ReadWords::ReadWords(char inputFile[])
{
  OpenFile(inputFile);
}

// OpenFile takes a filename and opens a stream to it
// Pre: no stream has been established to the filename
// Post: an input stream has been made with the filename
void ReadWords::OpenFile(char inputFile[])
{
  inFile.open(inputFile); //opens inputFile
  if (inFile.fail())     // check if open was successful
    {
      cout << "Cannot open the file" << endl;
      exit(1);
    }
}

// GetNextWord parses inFile into words and ignores punctuation
// Pre: inFile is connected to a text file
// Post: the next word in the file has been parsed and returned
string ReadWords::GetNextWord()
{
  string word; //holds the next word
  char nextChar; //holds next character in file

  inFile.get(nextChar);
  // this loops until a non-punctuation char is found
  //

  while (
       (isspace(nextChar)) ||
       (nextChar == '.')   ||
       (nextChar == ',')   ||
       (nextChar == ';')   ||
       (nextChar == ':')   ||
       (nextChar == '?')   ||
       (nextChar == '!')   ||
       (nextChar == '"')   ||
```

```
      (nextChar == char(39))   ||
      (nextChar == '-')   ||
      (nextChar == '/')   ||
      (nextChar == char(92))   ||
      (nextChar == '(')   ||
      (nextChar == ')')   ||
      (nextChar == '0')   ||
      (nextChar == '1')   ||
      (nextChar == '2')   ||
      (nextChar == '3')   ||
      (nextChar == '4')   ||
      (nextChar == '5')   ||
      (nextChar == '6')   ||
      (nextChar == '7')   ||
      (nextChar == '8')   ||
      (nextChar == '9')   ||
      (nextChar == '&')   ||
      (nextChar == '#')   ||
      (nextChar == '@')   ||
      (nextChar == '%')   ||
      (nextChar == '^')   ||
      (nextChar == '$')   ||
      (nextChar == '~')   ||
      (nextChar == '*')
      )
    {
    inFile.get(nextChar);

    if (inFile.eof()) //if eof is reached, returns word
      return word;
    }
  //build up word letter by letter, normalizes it to lowercase
  word = word + char(tolower(nextChar));
  //this gets the last letter found in above while loop

  // gets nextChar and uses this value to start the while loop below
  inFile.get(nextChar);
  if (inFile.eof()) //test for eof everytime get is called
    return word;

  // this loops, adding a char to word until eof or a punctuation mark
or space
  while (
      (!isspace(nextChar)) &&
      (nextChar != '.')    &&
      (nextChar != ',')    &&
      (nextChar != ';')    &&
      (nextChar != ':')    &&
      (nextChar != '?')    &&
      (nextChar != '!')    &&
      (nextChar != '"')    &&
      (nextChar != char(39))    &&
      (nextChar != '-')    &&
      (nextChar != '/')    &&
      (nextChar != char(92))    &&
      (nextChar != '(')    &&
      (nextChar != ')')    &&
```

```
       (nextChar != '<')    &&
       (nextChar != '>')    &&
       (nextChar != '0')    &&
       (nextChar != '1')    &&
       (nextChar != '2')    &&
       (nextChar != '3')    &&
       (nextChar != '4')    &&
       (nextChar != '5')    &&
       (nextChar != '6')    &&
       (nextChar != '7')    &&
       (nextChar != '8')    &&
       (nextChar != '9')    &&
       (nextChar != '@')    &&
       (nextChar != '#')    &&
       (nextChar != '$')    &&
       (nextChar != '%')    &&
       (nextChar != '^')    &&
       (nextChar != '&')    &&
       (nextChar != '*')    &&
       (nextChar != '+')    &&
       (nextChar != '~')    &&
       (nextChar != '=')
       )
  {
    word = word + char(tolower(nextChar)); //builds up word
    inFile.get(nextChar); //next char is gotten, then compared in above
loop
    if (inFile.eof()) //test for eof after get is called
      return word;
  }

 return word;
}
// EoF tests if eof has been reached with inFile
// Pre: inFile is connected to file
// Post: 1 is returned if a read has been attempted over the eof, 0
otherwise
int ReadWords::EoF()
{
  return inFile.eof();
}

// CloseFile closes the connection between inFile and its file
// Pre: inFile is connected to a file
// Post" inFile is no longer connected to a file
void ReadWords::CloseFile()
{
  inFile.close();
}
```

```
// porter.h

/*

   translated from Java to C++ by Al Popescul, and modified to
   detect stop words (the list of stop words is given in the
   constructor definition);

   popescul@unagi.cis.upenn.edu

   date: March 1999.

Java version :
(http://ftp.dcs.glasgow.ac.uk/idom/ir_resources/linguistic_utils/porter
.java)
by:
Fotis Lazarinis (translated from C to Java) June 1997
address:  Psilovraxou 12, Agrinio, 30100

comments: Compile Porter.C, inlcude the Porter class into you program
and
create an instance.  Then use the stripAffixes method which takes a
string as input and returns the stem of this string again as a string.
It returns an empty string if the input is a stop word.

*/

#include <iostream.h>
#include <string>
#include <ctype.h>
#include <cctype>
#include <set>

class Porter {
private:
  // data member for stop words
  set<string> stopWordList;

  string Clean(string);
  int hasSuffix(string, string, string&);
  int vowel(char , char);
  int measure(string);
  int containsVowel(string);
  int cvc(string);
  string step1(string);
  string step2(string);
  string step3(string);
  string step4(string);
  string step5(string);
  string stripPrefixes (string);
  string stripSuffixes(string);

  // stop word check
  bool isStopWord(string);

public:
  // constructor initializes the stop word list
```

```
  Porter();
  string stripAffixes( string );
  string stopWordStrip( string ); //this removes stopwords
};
```

```
// porter.c
/*
   translated from Java to C++ by Al Popescul, and modified to
   detect stop words (the list of stop words is given in the
   constructor definition);
   popescul@unagi.cis.upenn.edu
   date: March 1999.

Java version
(http://ftp.dcs.glasgow.ac.uk/idom/ir_resources/linguistic_utils/porter
.java)
by: Fotis Lazarinis (translated from C to Java) June 1997
address:  Psilovraxou 12, Agrinio, 30100

comments: Compile porter.cpp, inlcude the Porter class into you program
and
create an instance.  Then use the stripAffixes method which takes a
string as input and returns the stem of this string again as a string.
It returns an empty string if the input is a stop word.

*/


#include "porter.h"


Porter::Porter() {

const int numberStopWords = 319;


// using the stop word list found at
//
http://ftp.dcs.glasgow.ac.uk/idom/ir_resources/linguistic_utils/stop_wo
rds

const char* stopWordsLocal[numberStopWords] = {"a", "about", "above",
"across", "after", "afterwards", "again", "against", "all", "almost",
"alone", "along", "already", "also", "although", "always", "am",
"among", "amongst", "amoungst", "amount", "an", "and", "another",
"any", "anyhow", "anyone", "anything", "anyway", "anywhere", "are",
"around", "as", "at", "back", "be", "became", "because", "become",
"becomes", "becoming", "been", "before", "beforehand", "behind",
"being", "below", "beside", "besides", "between", "beyond", "bill",
"both", "bottom", "but", "by", "call", "can", "cannot", "cant", "co",
"computer", "con", "could", "couldnt", "cry", "de", "describe",
"detail", "do", "done", "down", "due", "during", "each", "eg",
"eight", "either", "eleven", "else", "elsewhere", "empty", "enough",
"etc", "even", "ever", "every", "everyone", "everything",
"everywhere", "except", "few", "fifteen", "fify", "fill", "find",
"fire", "first", "five", "for", "former", "formerly", "forty",
"found", "four", "from", "front", "full", "further", "get", "give",
"go", "had", "has", "hasnt", "have", "he", "hence", "her", "here",
"hereafter", "hereby", "herein", "hereupon", "hers", "herself", "him",
"himself", "his", "how", "however", "hundred", "i", "ie", "if", "in",
"inc", "indeed", "interest", "into", "is", "it", "its", "itself",
"keep", "last", "latter", "latterly", "least", "less", "ltd", "made",
```

```
"many", "may", "me", "meanwhile", "might", "mill", "mine", "more",
"moreover", "most", "mostly", "move", "much", "must", "my", "myself",
"name", "namely", "neither", "never", "nevertheless", "next", "nine",
"no", "nobody", "none", "noone", "nor", "not", "nothing", "now",
"nowhere", "of", "off", "often", "on", "once", "one", "only", "onto",
"or", "other", "others", "otherwise", "our", "ours", "ourselves",
"out", "over", "own", "part", "per", "perhaps", "please", "put",
"rather", "re", "same", "see", "seem", "seemed", "seeming", "seems",
"serious", "several", "she", "should", "show", "side", "since",
"sincere", "six", "sixty", "so", "some", "somehow", "someone",
"something", "sometime", "sometimes", "somewhere", "still", "such",
"system", "take", "ten", "than", "that", "the", "their", "them",
"themselves", "then", "thence", "there", "thereafter", "thereby",
"therefore", "therein", "thereupon", "these", "they", "thick", "thin",
"third", "this", "those", "though", "three", "through", "throughout",
"thru", "thus", "to", "together", "too", "top", "toward", "towards",
"twelve", "twenty", "two", "un", "under", "until", "up", "upon", "us",
"very", "via", "was", "we", "well", "were", "what", "whatever",
"when", "whence", "whenever", "where", "whereafter", "whereas",
"whereby", "wherein", "whereupon", "wherever", "whether", "which",
"while", "whither", "who", "whoever", "whole", "whom", "whose", "why",
"will", "with", "within", "without", "would", "yet", "you", "your",
"yours", "yourself", "yourselves"};

 set<string> temp(stopWordsLocal, stopWordsLocal + numberStopWords);

 stopWordList = temp;

}

string
Porter::Clean(string str) {
  int last = str.length();

  char ch = str[0];
  string temp = "";

  for ( int i=0; i < last; i++ ) {
    if ( isalnum(str[i]))
      temp += str[i];
  }

  return temp;
} //clean


int
Porter::hasSuffix(string word, string suffix, string& stem) {

  string tmp = "";

  if (word.length() <= suffix.length())
        return 0;
  if (suffix.length() > 1)
    if (word[word.length()-2] != suffix[suffix.length()-2])
      return 0;
```

```
  stem = "";

  for ( int i=0; i<word.length()-suffix.length(); i++)
    stem += word[i];
     tmp = stem;

     for ( int i=0; i<suffix.length(); i++ )
       tmp += suffix[i];

     if ( tmp.compare(word) == 0)
       return 1;
     else
       return 0;
}

int
Porter::vowel( char ch, char prev ) {
    switch ( ch ) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
      {return 1; break;}
    case 'y': {

      switch ( prev ) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
      { return 0; break; }
    default:
      return 1;
    }
  }
  default :
    return 0;
  }
}

int
Porter::measure(string stem) {

    int i=0;
    int count = 0;
    int length = stem.length();

    while ( i < length ) {
       for ( ; i < length ; i++ ) {
           if ( i > 0 ) {
              if ( vowel(stem[i],stem[i-1]) )
                 break;
           }
           else {
              if ( vowel(stem[i],'a') )
```

```
                break;
            }
        }

        for ( i++ ; i < length ; i++ ) {
            if ( i > 0 ) {
               if ( !vowel(stem[i],stem[i-1]) )
                   break;
               }
            else {
               if ( !vowel(stem[i],'?') )
                  break;
            }
        }
      if ( i < length ) {
         count++;
         i++;
      }
    } //while

    return count;
  }

int
Porter::containsVowel(string word) {

    for (int i=0 ; i < word.length(); i++ )
      if ( i > 0 ) {
      if ( vowel(word[i],word[i-1]) )
        return 1;
         }
      else {
      if ( vowel(word[0],'a') )
        return 1;
      }

    return 0;
  }


int
Porter::cvc( string str ) {
  int length=str.length();

  if ( length < 3 )
    return 0;

  if ( (!vowel(str[length-1],str[length-2]) )
       && (str[length-1] != 'w') && (str[length-1] != 'x') &&
(str[length-1] != 'y')
       && (vowel(str[length-2],str[length-3])) ) {

    if (length == 3) {
      if (!vowel(str[0],'?'))
      return 1;
      else
      return 0;
```

```
    }
    else {
      if (!vowel(str[length-3],str[length-4]) )
      return 1;
      else
      return 0;
    }
  }

  return 0;
}

string
Porter::step1(string str) {

  string stem;

  if ( str[str.length()-1] == 's' ) {
    if ( (hasSuffix( str, "sses", stem )) || (hasSuffix( str, "ies",
stem)) ){
      string tmp = "";
      for (int i=0; i<(str.length()-2); i++)
      tmp += str[i];
      str = tmp;
    }
    else {
        if ( ( str.length() == 1 ) && ( str[str.length()-1] == 's' ) )
{
          str = "";
          return str;
        }
        if ( str[str.length()-2] != 's' ) {
          string tmp = "";
          for (int i=0; i<str.length()-1; i++)
            tmp += str[i];
          str = tmp;
        }
    }
  }

     if ( hasSuffix( str,"eed",stem ) ) {
       if ( measure( stem ) > 0 ) {
       string tmp = "";
       for (int i=0; i<str.length()-1; i++)
         tmp += str[i];
       str = tmp;
       }
     }
     else {
       if (  (hasSuffix( str,"ed",stem )) || (hasSuffix( str,"ing",stem
)) ) {
       if (containsVowel( stem ))  {

         string tmp = "";
         for ( int i = 0; i < stem.length(); i++)
                  tmp += str[i];
              str = tmp;
```

```
              if ( str.length() == 1 )
                 return str;

              if ( ( hasSuffix( str,"at",stem) ) || ( hasSuffix(
str,"bl",stem ) ) || ( hasSuffix( str,"iz",stem) ) ) {
            str += "e";

              }
              else {
            int length = str.length();
            if ( (str[length-1] == str[length-2])
                 && (str[length-1] != 'l') && (str[length-1] != 's') &&
(str[length-1] != 'z') ) {

                   tmp = "";
                   for (int i=0; i<str.length()-1; i++)
                tmp += str[i];
                   str = tmp;
            }
                 else
               if ( measure( str ) == 1 ) {
                      if ( cvc(str) )
                   str += "e";
               }
              }
       }
       }
     }

     if ( hasSuffix(str,"y",stem) )
       if ( containsVowel( stem) ) {
       string tmp = "";
       for (int i=0; i<str.length()-1; i++ )
              tmp += str[i];
       str = tmp + "i";
       }
     return str;
  }



string
Porter::step2( string str ) {

     string suffixes[22][2] = { { "ational", "ate" },
                                { "tional",  "tion" },
                                { "enci",    "ence" },
                                { "anci",    "ance" },
                                { "izer",    "ize" },
                                { "iser",    "ize" },
                                { "abli",    "able" },
                                { "alli",    "al" },
                                { "entli",   "ent" },
                                { "eli",     "e" },
                                { "ousli",   "ous" },
                                { "ization", "ize" },
                                { "isation", "ize" },
```

```
                                        { "ation",   "ate" },
                                        { "ator",    "ate" },
                                        { "alism",   "al" },
                                        { "iveness", "ive" },
                                        { "fulness", "ful" },
                                        { "ousness", "ous" },
                                        { "aliti",   "al" },
                                        { "iviti",   "ive" },
                                        { "biliti",  "ble" }};
    string stem;




    for ( int index = 0 ; index < 22; index++ ) {
        if ( hasSuffix ( str, suffixes[index][0], stem ) ) {
           if ( measure ( stem ) > 0 ) {
              str = stem + suffixes[index][1];
              return str;
           }
        }
    }

    return str;
  }



string
Porter::step3( string str ) {

       string suffixes[8][2] = { { "icate", "ic" },
                                       { "ative", "" },
                                       { "alize", "al" },
                                       { "alise", "al" },
                                       { "iciti", "ic" },
                                       { "ical",  "ic" },
                                       { "ful",   "" },
                                       { "ness",  "" }};
       string stem;

       for ( int index = 0 ; index<8; index++ ) {
           if ( hasSuffix ( str, suffixes[index][0], stem ))
              if ( measure ( stem ) > 0 ) {
                 str = stem + (suffixes[index][1]);
                 return str;
              }
       }
       return str;
  }



string
Porter::step4( string str ) {
```

```
     string suffixes[21] = { "al", "ance", "ence", "er", "ic", "able",
"ible", "ant", "ement", "ment", "ent", "sion", "tion", "ou", "ism",
"ate", "iti", "ous", "ive", "ize", "ise"};

     string stem;

     for ( int index = 0 ; index<21; index++ ) {
         if ( hasSuffix ( str, suffixes[index], stem ) ) {

            if ( measure ( stem ) > 1 ) {
               str = stem;
               return str;
            }
         }
     }
     return str;
  }



string
Porter::step5( string str ) {

    if ( str[str.length()-1] == 'e' ) {
      if ( measure(str) > 1 ) {/* measure(str)==measure(stem) if ends
in vowel */
      string tmp = "";
      for ( int i=0; i<str.length()-1; i++ )
        tmp += str[i];
      str = tmp;
      }
      else
      if ( measure(str) == 1 ) {
        string stem = "";
        for ( int i=0; i<str.length()-1; i++ )
          stem += str[i];

        if ( !cvc(stem) )
          str = stem;
      }
    }

    if (str.length() == 1)
      return str;
    if ( (str[str.length()-1] == 'l') && (str[str.length()-2] == 'l')
&& (measure(str) > 1) )
      if ( measure(str) > 1 ) {/* measure(str)==measure(stem) if ends
in vowel */
      string tmp = "";
           for ( int i=0; i<(str.length()-1); i++ )
               tmp += str[i];
           str = tmp;
        }
     return str;
  }
```

```
string
Porter::stripPrefixes ( string str) {

    string prefixes[9] = { "kilo", "micro", "milli", "intra", "ultra",
"mega", "nano", "pico", "pseudo"};
    int pos;
    int last = 9;
    for ( int i=0 ; i<last; i++ ) {
      pos = str.find(prefixes[i]);
      if (pos == 0) {
      string temp = "";
      for ( int j=0 ; j<(str.length()-prefixes[i].length()); j++ )
        temp += str[j+ (prefixes[i].length()) ];
      return temp;
      }
    }

    return str;
  }



string
Porter::stripSuffixes(string str) {
    str = step1( str );
    if ( str.length() >= 1 )
      str = step2( str );
    if ( str.length() >= 1 )
      str = step3( str );
    if ( str.length() >= 1 )
      str = step4( str );
    if ( str.length() >= 1 )
      str = step5( str );
    return str;
  }


string
Porter::stripAffixes( string str ) {

  for(int i=0; i<str.length(); ++i)
    str[i] = tolower(str[i]);

  if (( str != "" ) && (str.length() > 2)) {
    str = stripPrefixes(str);

    if (str != "" )
      str = stripSuffixes(str);
  }

  return str;

} //stripAffixes


//stopWordStrip- this actually does not stem,
```

```
//it just calls clean which checks for stopwords
//returns a blank if the word is a stop word
//if not, it returns the string
string
Porter::stopWordStrip( string str ) {



for(int i=0; i<str.length(); ++i)
      str[i] = tolower(str[i]);

    str = Clean(str);

  if(stopWordList.find(str)!=stopWordList.end())
    return "";
  else {
    return str;
  }
} //stopWordStrip
```

```
#!/usr/bin/perl

# calculates data based on output from the Zipfian text parser
# Zach Sharek
#
# USAGE: ./excel infile outfile
#
# the files were sorted by frequency with:
# sort --field-separator=, +1 -r -g input > output
#


open (INPUT, $ARGV[0]) || die "Error opening file: $ARGV[0]";
open (OUTPUT, ">temp") || die "Error with temp file.";


$totalFreq = 0;
$rank = 0;
$lastFreq = 0;
$rankCount = 0;

while (<INPUT>)
{
    @line = split(/, /);
    chomp $line[1];
    $totalFreq = $totalFreq + $line[1];

    # if the frequencies match, then dont increment rank
    unless ($lastFreq == $line[1])
    {
      $rank++;
      $rank = $rank + $rankCount;
      $rankCount = 0;             # reset rank counter
    }
    else
    {
      # increment rank counter instead
      $rankCount++;
    }

    print OUTPUT "$line[0], $line[1], $rank\n";
    $lastFreq = $line[1];
}

close(INPUT) || die "ERROR: can't close $ARGV[0]: $!";
close(OUTPUT) || die "ERROR: can't close temp file: $!";
open (INPUT, "temp") || die "Error opening file temp";
open (OUTPUT, ">$ARGV[1]") || die "Error with output file.";
print OUTPUT "The total frequency is: $totalFreq\n";
print OUTPUT "Word, Frequency, Rank, Pr(x), Rank*Pr(x), logRank,
logFreq, -lnPr(x)\n";



while (<INPUT>)
{
    @values = split(/, /);
```

```
    chomp $values[2];
    $prob = $values[1]/$totalFreq; #values[1] is frequency
    $zipf = $prob * $values[2];    #values[2] is rank
    $logRank = log_base(10, $values[2]);
    $logFreq = log_base(10, $values[1]);
    $info = 0 - (log_base(2, $prob)); # shannon's information entropy
    print OUTPUT "$values[0], $values[1], $values[2], $prob, $zipf,
$logRank, $logFreq, $info\n";

}

print "Final values are: $values[0], $values[1], $values[2], $prob,
$zipf, $logRank, $logFreq, $info\n";


# this function finds the log of a value to any base
sub log_base
{
    my ($base, $value) = @_;
    unless ($value == 0)
    {
      return log($value)/log($base);
    }
    else
    {
      return 0;
    }
}
```