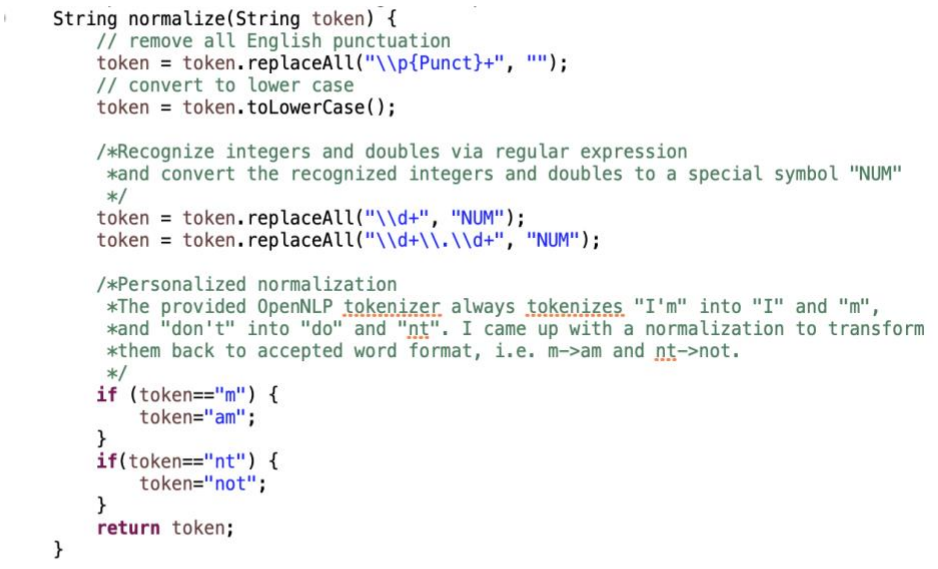
Yunlu Li

CS 4780 Information Retrieval

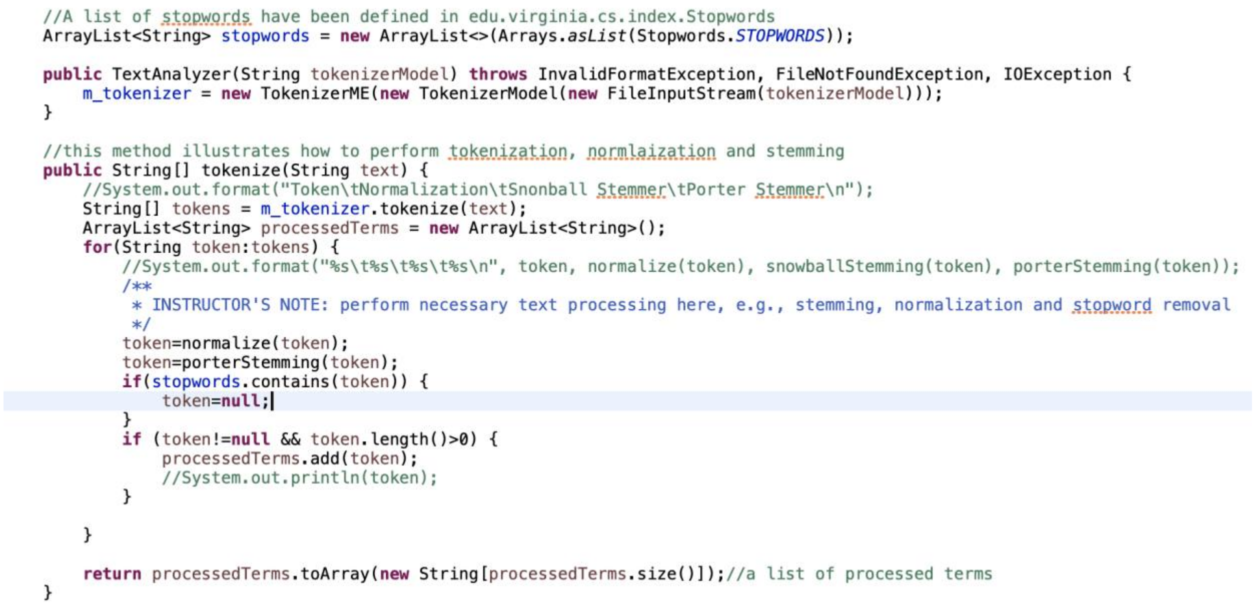
Machine Problem 1

Question 1

(1) Paste your implementation of text normalization module.

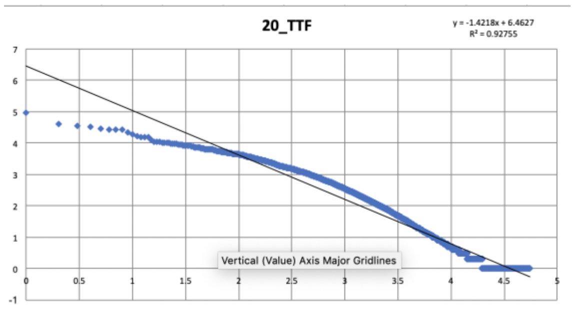


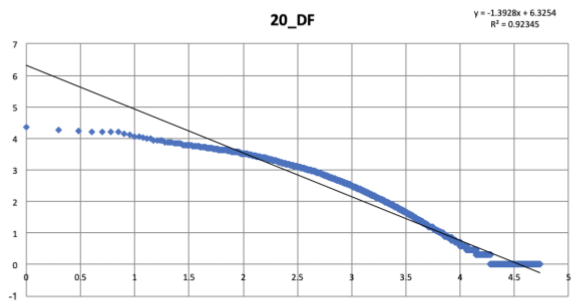
```
String normalize(String token) {
    // remove all English punctuation
    token = token.replaceAll("\\p{Punct}+", "");
    // convert to lower case
    token = token.toLowerCase();

    /*Recognize integers and doubles via regular expression
     *and convert the recognized integers and doubles to a special symbol "NUM"
     */
    token = token.replaceAll("\\d+", "NUM");
    token = token.replaceAll("\\d+\\.\\d+", "NUM");

    /*Personalized normalization
     *The provided OpenNLP tokenizer always tokenizes "I'm" into "I" and "m",
     *and "don't" into "do" and "nt". I came up with a normalization to transform
     *them back to accepted word format, i.e. m->am and nt->not.
     */
    if (token=="m") {
        token="am";
    }
    if(token=="nt") {
        token="not";
    }
    return token;
}
```

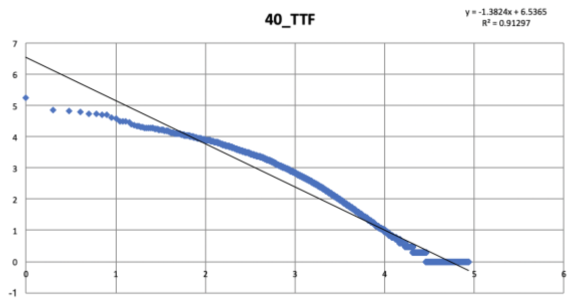


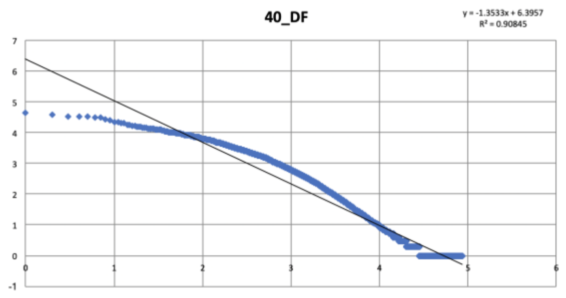
```
//A list of stopwords have been defined in edu.virginia.cs.index.Stopwords
ArrayList<String> stopwords = new ArrayList<>(Arrays.asList(Stopwords.STOPWORDS));

public TextAnalyzer(String tokenizerModel) throws InvalidFormatException, FileNotFoundException, IOException {
    m_tokenizer = new TokenizerME(new TokenizerModel(new FileInputStream(tokenizerModel)));
}

//this method illustrates how to perform tokenization, normlaization and stemming
public String[] tokenize(String text) {
    //System.out.format("Token\tNormalization\tSnonball Stemmer\tPorter Stemmer\n");
    String[] tokens = m_tokenizer.tokenize(text);
    ArrayList<String> processedTerms = new ArrayList<String>();
    for(String token:tokens) {
        //System.out.format("%s\t%s\t%s\t%s\n", token, normalize(token), snowballStemming(token), porterStemming(token));
        /**
         * INSTRUCTOR'S NOTE: perform necessary text processing here, e.g., stemming, normalization and stopword removal
         */
        token=normalize(token);
        token=porterStemming(token);
        if(stopwords.contains(token)) {
            token=null;
        }
        if (token!=null && token.length()>0) {
            processedTerms.add(token);
            //System.out.println(token);
        }

    }

    return processedTerms.toArray(new String[processedTerms.size()]);//a list of processed terms
}
```

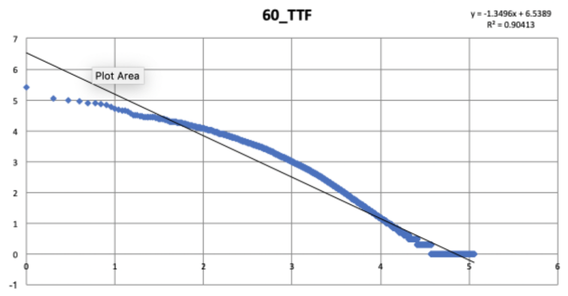
(2) Six curves in log-log space generated above, with the corresponding slopes and intercepts of the linear interpretation results.

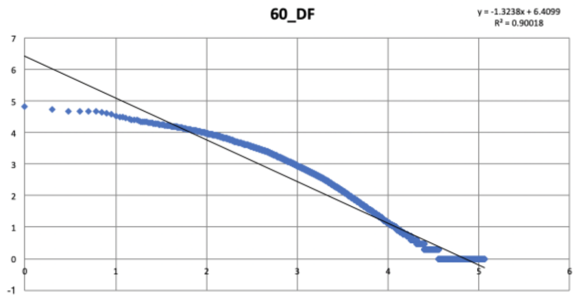












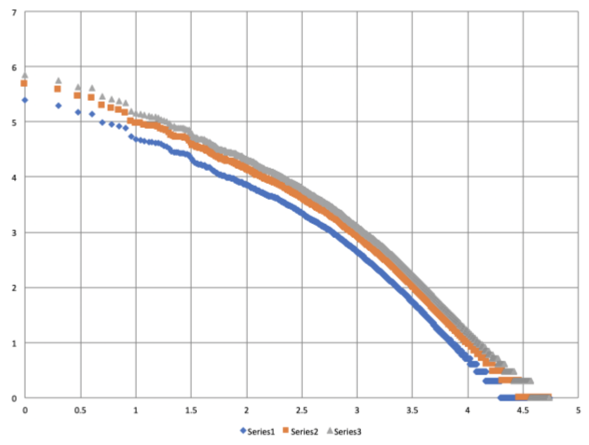
(3) Your answers and thoughts to the above questions.

The plot shows a strong linear relationship between x-axis and y-axis after the log-log transformation. The slope for each folder and each statistic is shown in the following table:

	20	40	60
TTF	-1.422	-1.382	-1.350
DF	-1.393	-1.353	-1.324

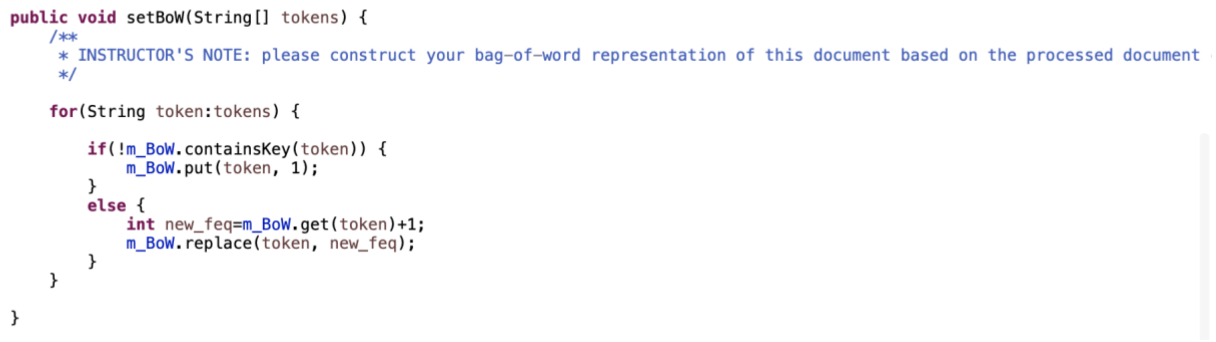
From the slopes and corresponding R^2, I found that TTF fits the Zipf's law better than DF, since it has higher R^2 and more negative slopes. From my perspective, TTF is a more realistic reflection of frequency of a word in natural language, because it represents the total number of occurrences. However, DF only records the number of documents where the word appears. In the case where word1 appears more often than word2 and they both appear in several documents, their TTF will differ a lot but DF will not show great difference. This case will make DF less fit the Zipf's law.

In order to see whether the increasing number of documents better fits Zipf's law, I plot the TTF with respect to 20, 40, 60 documents together. From my perspective, the increase of the number of documents simply moves the curve upward, rather than change the shape of the curve. The reviews from 20 documents contains 54,419 unique words and 2,436,504 total occurrences, so I think they are already large enough for verifying Zipf's law, and that's why the increase in the number of documents does not help that much.

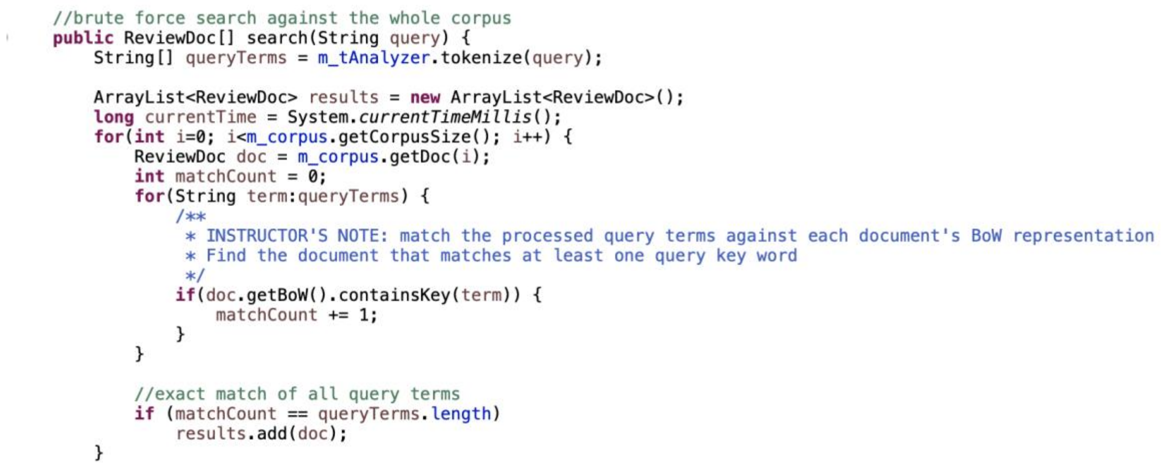


Question 2

(1) Paste your implementation of Bag-of-Word document representation.



```java
public void setBoW(String[] tokens) {
    /**
     * INSTRUCTOR'S NOTE: please construct your bag-of-word representation of this document based on the processed document
     */

    for(String token:tokens) {

        if(!m_BoW.containsKey(token)) {
            m_BoW.put(token, 1);
        }
        else {
            int new_feq=m_BoW.get(token)+1;
            m_BoW.replace(token, new_feq);
        }
    }

}
```

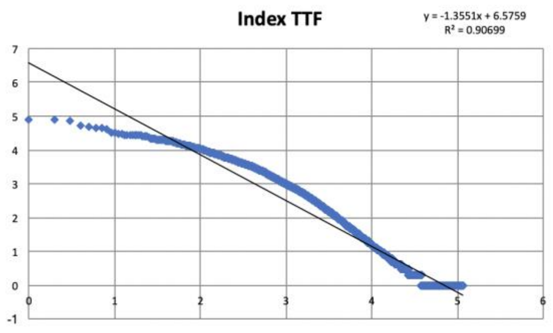


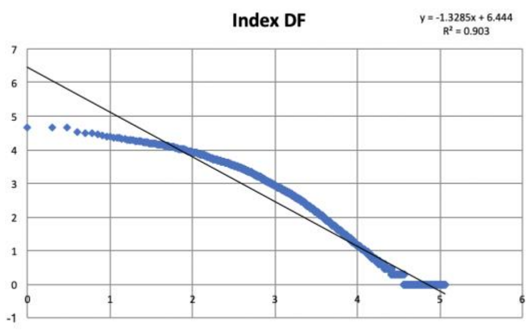
```java
//brute force search against the whole corpus
public ReviewDoc[] search(String query) {
    String[] queryTerms = m_tAnalyzer.tokenize(query);

    ArrayList<ReviewDoc> results = new ArrayList<ReviewDoc>();
    long currentTime = System.currentTimeMillis();
    for(int i=0; i<m_corpus.getCorpusSize(); i++) {
        ReviewDoc doc = m_corpus.getDoc(i);
        int matchCount = 0;
        for(String term:queryTerms) {
            /**
             * INSTRUCTOR'S NOTE: match the processed query terms against each document's BoW representation
             * Find the document that matches at least one query key word
             */
            if(doc.getBoW().containsKey(term)) {
                matchCount += 1;
            }
        }

        //exact match of all query terms
        if (matchCount == queryTerms.length)
            results.add(doc);
    }
```

(2) Two curves in log-log space generated by using Lucene inverted index for collecting TTF and DF statistics, with the corresponding running time statistics in comparison with your implementation in problem one.

In Problem 1, I collected TTF and DF separately, by using my own implementation. The corresponding running time is provided below in the first row. Then I used the provided code to collect DF and TTF simultaneously with Lucene inverted index. Because our P1 implementation is combined with document loading, simply loading documents and calling the provided code about Lucene will give us the running time of P1 implementation plus Lucene. In order to collect "pure" running time for Lucene inverted index, I comment out *doc.setBoW(tokenize(content))* in *DocAnalyzer* to remove P1 implementation from document loading. Then I ran the program and got the running time for Lucene as shown below. As we can see, the Lucene inverted index is faster for collecting TTF and DDF statistics than the implementation in problem 1.

	TTF Running Time	DF Running Time
P1 Implementation	192 seconds	196 seconds
Lucene	118 seconds	





(3) Running time and total number of returned documents by two retrieval models. If you found these two methods gave you different number of matched documents, do you have any explanation about it?

The total running time and number of return is described in the following table:

	Inverted Index	Brute Force
Total Running Time	0.34 seconds	1114.60 seconds
Total # of Doc Returned	12798	12615

The two methods return different number of matched documents. The difference is caused by different text processing pipeline. The inverted index implementation used LUCENE_46 library to process the text, whereas the brute-force implementation applied self-made pipeline. Different processing techniques may cause same document contain different token, resulting in the difference number of document returned.