CS 6501: Information Retrieval

MP1: Getting Familiar with Text Processing & Inverted Index

Yiwei Fang

yf5kq

October 10, 2018

1. Understand Zipf's Law

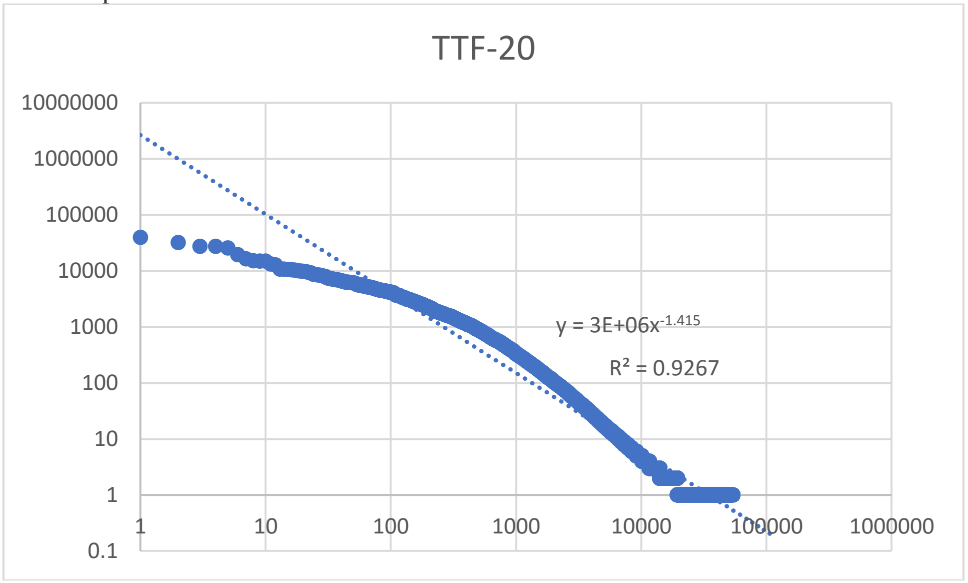
1. Paste your implementation of text normalization module.

```
    package edu.virginia.cs.analyzer;
    import java.io.FileInputStream;
    import java.io.FileNotFoundException;
    import java.io.IOException;
    import java.util.ArrayList;
    import java.util.Arrays;

    import org.tartarus.snowball.SnowballStemmer;
    import org.tartarus.snowball.ext.englishStemmer;
    import org.tartarus.snowball.ext.porterStemmer;

    import opennlp.tools.tokenize.Tokenizer;
    import opennlp.tools.tokenize.TokenizerME;
    import opennlp.tools.tokenize.TokenizerModel;
    import opennlp.tools.util.InvalidFormatException;

    import java.util.List;
    import edu.virginia.cs.index.Stopwords;

    public class TextAnalyzer {

        //Maximum Entropy model based tokenizer
        Tokenizer m_tokenizer;

        //A list of stopwords have been defined in edu.virginia.cs.index.Stopwords
        List<String> stoplist = 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 = snowballStemming(token);
                        token = stoplist.contains(token) ? "" : token;

                        if (token!=null && token.length()>0)
                                processedTerms.add(token);
                }

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

        //sample code for demonstrating how to perform text normalization
        String normalize(String token) {
                // remove all non-word characters
                /**
                 * INSTRUCTOR'S NOTE
                 * please change this to remove all English punctuation
                 */
                token = token.replaceAll("[\\p{Punct}&&[^0-9]]", "");
    //          token = token.replaceAll("\\.", "");
    //          token = token.replaceAll(",", "");
    //          token = token.replaceAll(";", "");
    //          token = token.replaceAll(":", "");
    //          token = token.replaceAll("!", "");
    //          token = token.replaceAll("\\?", "");
    //          token = token.replaceAll("'", "");
    //          token = token.replaceAll("\"", "");
    //          token = token.replaceAll("-", "");

                // convert to lower case
                token = token.toLowerCase();

                /**
                 * INSTRUCTOR'S NOTE
                 * add a line to recognize integers and doubles via regular expression
                 * and convert the recognized integers and doubles to a special symbol "NUM"
                 */
                token = token.replaceAll("[-+]?[0-9]*\\.?[0-9]+([eE][-+]?[0-9]+)?", "NUM");


                /**
                 * INSTRUCTOR'S NOTE
                 * after reading your constructed controlled vocabulary, can you come up with
other
                 * normaliazation rules to further refine the selected terms?
                 */
                token = token.length() == 1 ? "" : token;

                return token;
```

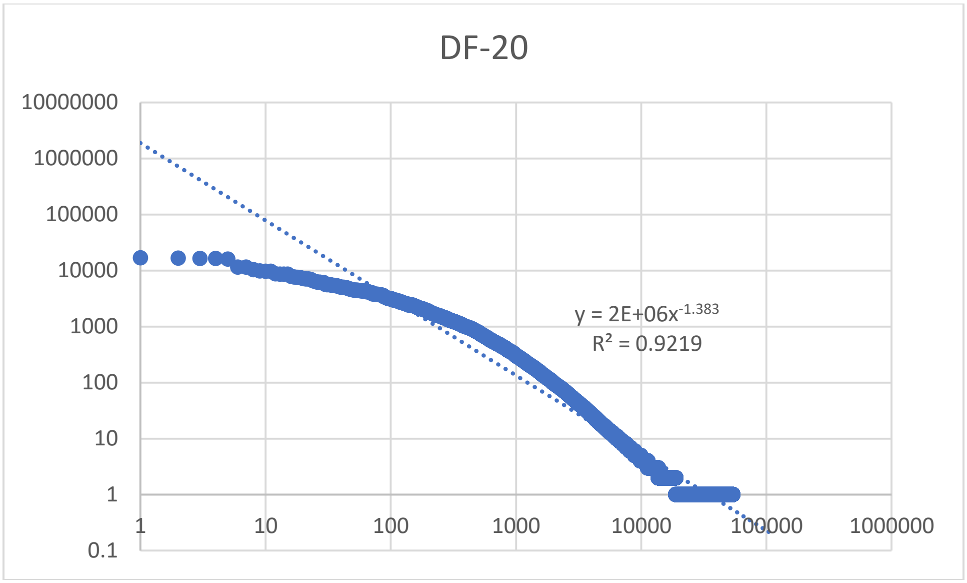
```
            }

            //sample code for demonstrating how to use Snowball stemmer
            String snowballStemming(String token) {
                        SnowballStemmer stemmer = new englishStemmer();
                        stemmer.setCurrent(token);
                        if (stemmer.stem())
                                    return stemmer.getCurrent();
                        else
                                    return token;
            }

            //sample code for demonstrating how to use Porter stemmer
            String porterStemming(String token) {
                        porterStemmer stemmer = new porterStemmer();
                        stemmer.setCurrent(token);
                        if (stemmer.stem())
                                    return stemmer.getCurrent();
                        else
                                    return token;
            }
}
```

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

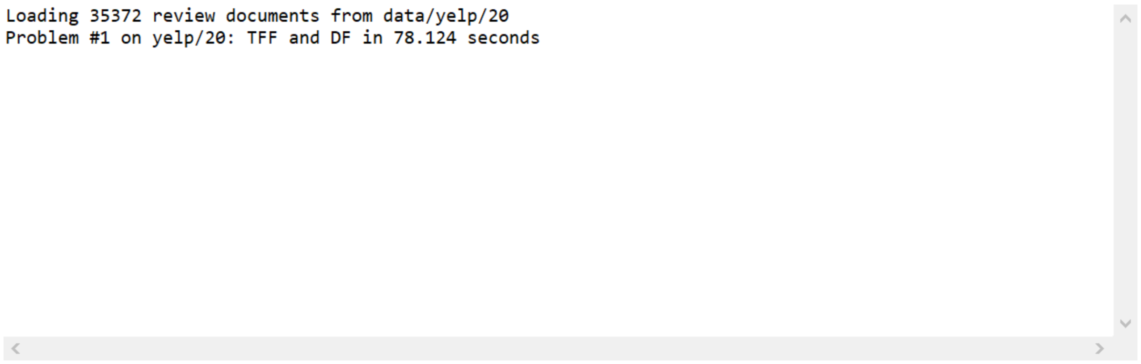
- Yelp/20



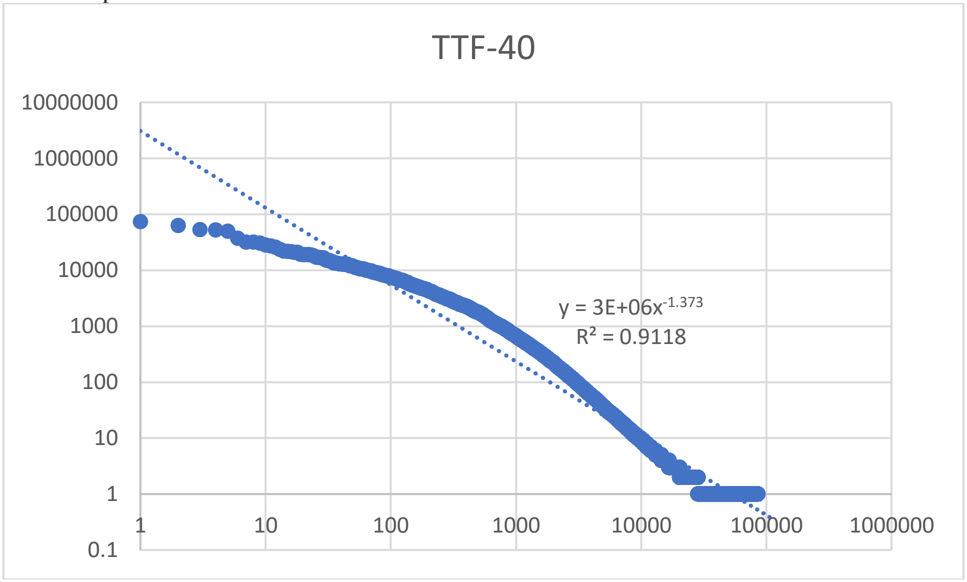
intercept: 3E+6
slope: -1.415



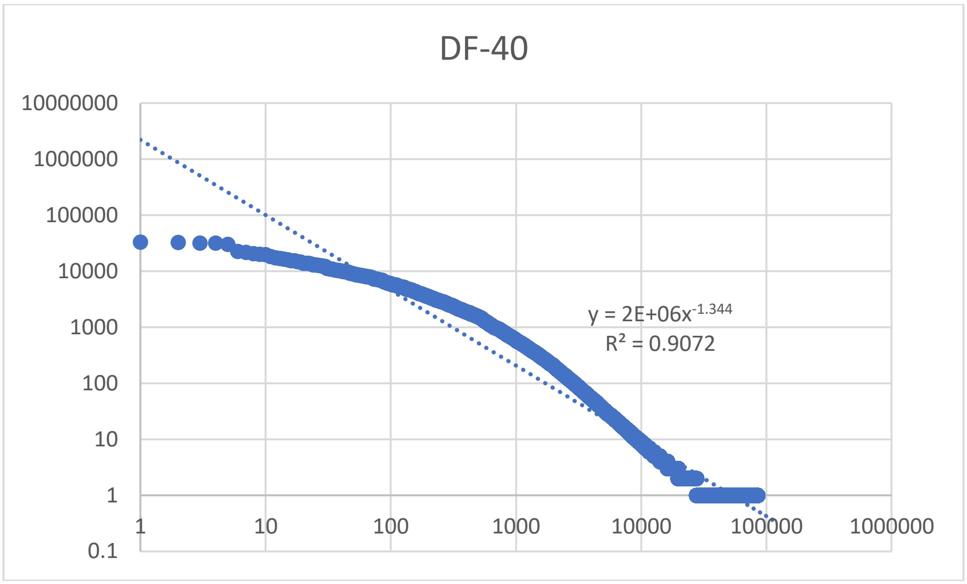
intercept: 2E+6
slope: -1.383


```
Loading 35372 review documents from data/yelp/20
Problem #1 on yelp/20: TFF and DF in 78.124 seconds
```

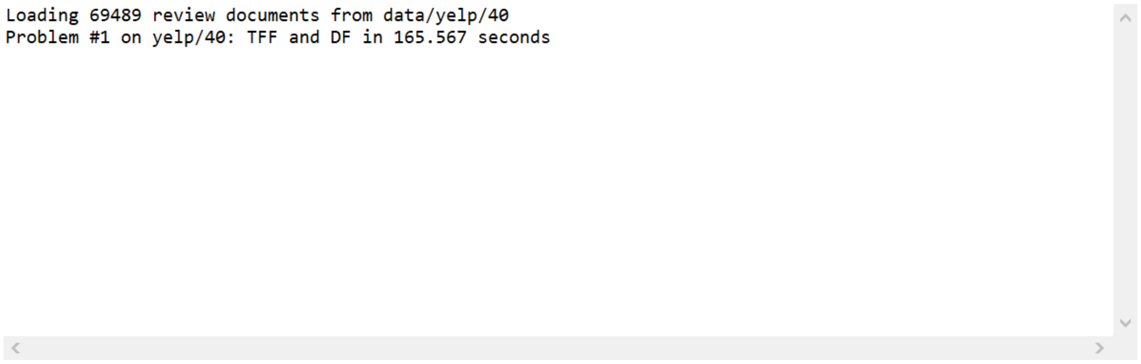
- Yelp/40



intercept: 3E+6
slope: -1.373

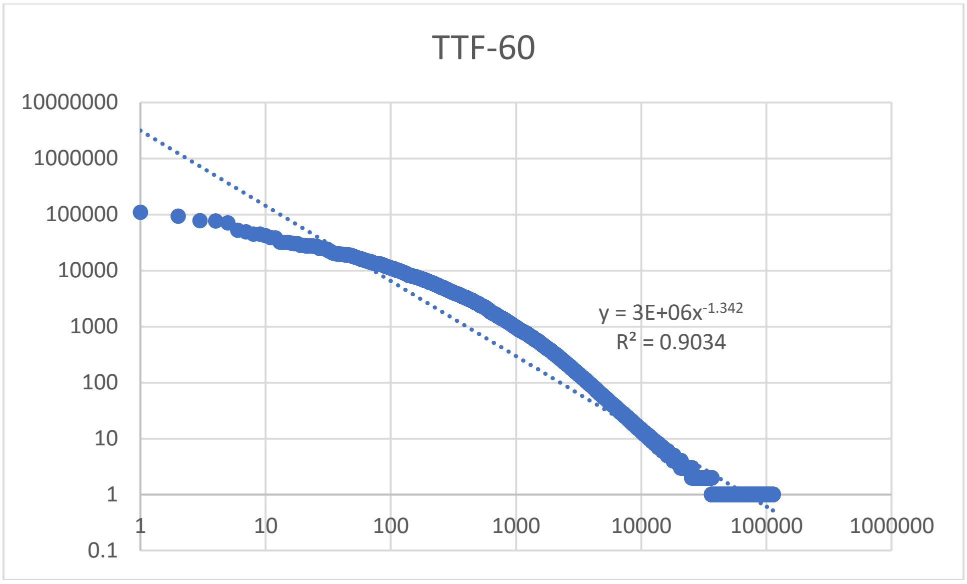


intercept: 2E+6
slope: -1.344

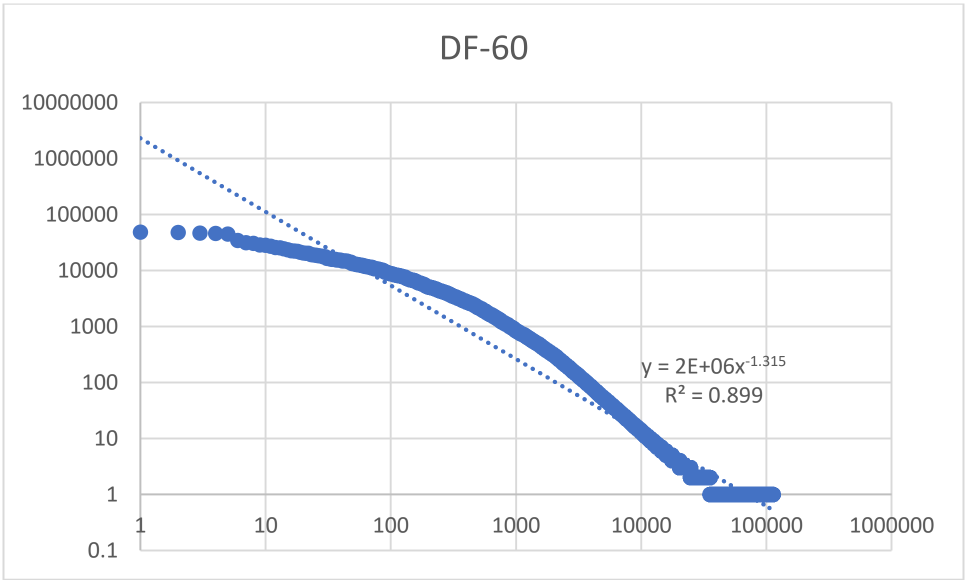


```
Loading 69489 review documents from data/yelp/40
Problem #1 on yelp/40: TFF and DF in 165.567 seconds
```

- Yelp/60



intercept: 3E+6
slope: -1.342



intercept: 2E+6
slope: -1.315

```
Loading 102201 review documents from data/yelp/60
Problem #1 on yelp/60: TFF and DF in 224.274 seconds
```

3. **Your answers and thoughts to the above questions.**
   a. From the resulting plot, can we find a strong linear relationship between the x and y axes in the log-log scale? If so, what is the slope of this linear relationship?

      From the log-log scale, we can find the strong linear relationship between the x and y axis by looking at the R square value, which tells us how strong the linear relationship is. The higher R^2 value, the stronger linear relationship is. For both TTF and DF, the slope of this linear relationship among Yelp/20, Yelp/40, and Yelp/60 datasets are close to -1.

   b. According to new curve and corresponding slope and intercept of the linear interpretation, can you conclude which counting statistics, i.e., *TTF* v.s., *DF*, fits Zipf's law better on this data set? Can you give any explanation about why it fits the law better?

      TTF fits Zipf’s law better on this dataset. For example, if there are only one documents, but it contains tons of words that can fit Zipf’s law. In this case, DF is hard to perform Zipf’s law because DF is constrained by the number of documents. So TTF fits Zipf’s law better because it just focuses on the word frequency and word itself.

   c. You should collect statistics from these three folders separately and create the corresponding plots (3*2 curves in total). Will an increased number of documents better help us verify Zipf's law?

      The increased number of documents do help us verify Zipf’s law because the increased documents will bring more head words without any meaning and more tail words which take major portion of vocabulary, but they rarely occur in documents. From the plot, we can also see that the increased number of documents make the R^2 become smaller, which means the linear relationship gets weaker when documents number increased due to the head/tail words increased. This situation well verify the three points of the Zipf’s law.

2. Building an inverted index for accelerating text access

1. **Paste your implementation of Bag-of-Word document representation (e.g., how to construct it from raw document content, and how to search for a particular term in it).**

   Construct the BoW by tokenizing the raw document content first using the text processing module implemented before. The BoW is stored in the format of hashmap, so searching a particular term in the document is the same as searching a particular key in that hashmap. Then for each document, has its own BoW. The corpus contains all the documents stored in the linked list structure called m_collections, and total terms and their correlated numbers are stored in a hashmap, m_dictionary. Finally, we can search a particular term among all those documents in the corpus.

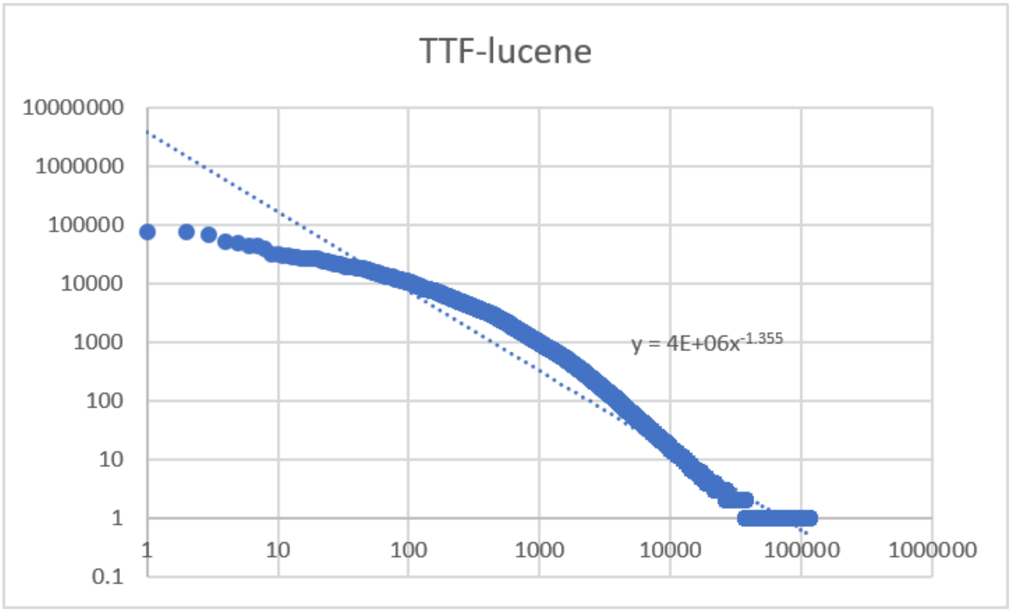
   ReviewDoc.java

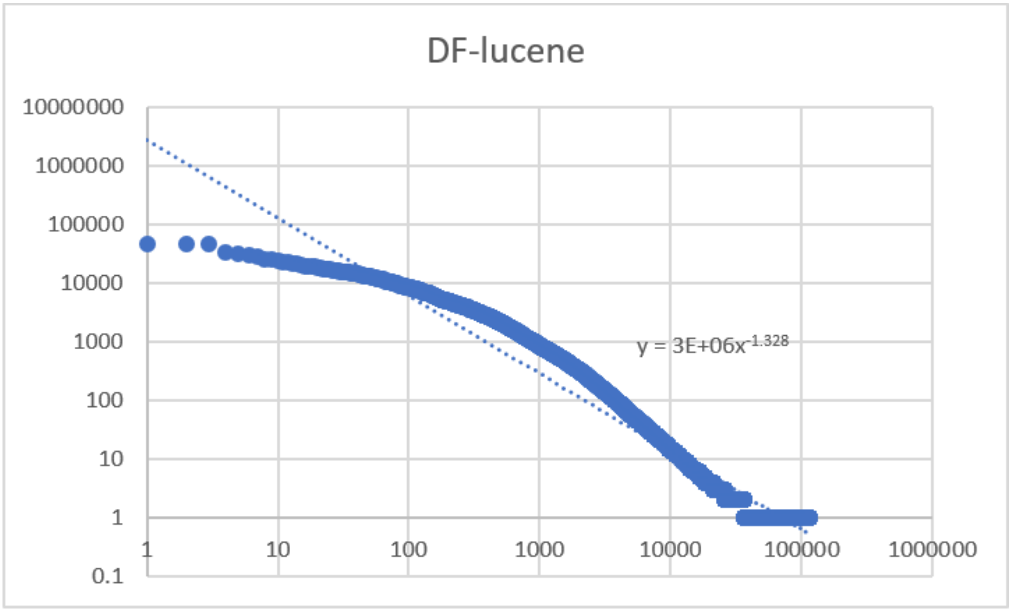
```
/**
 *
 */
package structures;

import java.util.Date;
import java.util.HashMap;

/**
 * @author Hongning Wang
 * Basic data structure for a Yelp review document
 */
public class ReviewDoc {
	String m_author; //author name
	String m_docID; //unique review id
	double m_rating; //numerical rating of the review
	String m_content; //review text content
	Date m_date; //date when the review was published
	String m_authorLocation; //author's registered location

	/**
	 * INSTRUCTOR'S NOTE: Your bag-of-word representation of this document
	 * HashMap is an example, and you are free to use any reasonable data structure for this
purpose
	 */
	HashMap<String, Integer> m_BoW; //word -> frequency

	public ReviewDoc(String docID, String author) {
		m_docID = docID;
		m_author = author;
		m_BoW = new HashMap<String, Integer>();
	}

	public void setAuthor(String author) {
		m_author = author;
	}

	public String getAuthor() {
		return m_author;
	}

	public void setDocID(String docID) {
		m_docID = docID;
	}

	public String getDocID() {
		return m_docID;
	}

	public void setRating(double rating) {
		m_rating = rating;
	}

	public double getRating() {
		return m_rating;
	}

	public void setContent(String content) {
		m_content = content;
	}

	public String getContent() {
		return m_content;
	}
	public void setDate(Date date) {
		m_date = date;
	}

	public Date getDate() {
		return m_date;
	}

	public void setAuthorLocation(String authorLoc) {
		m_authorLocation = authorLoc;
	}

	public String getAuthorLocation() {
		return m_authorLocation;
	}

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

//		System.out.println(m_content);
		for (String str : tokens) {
//			System.out.println(str);
			if (contains(str))
				m_BoW.put(str, counts(str)+1);
			else
				m_BoW.put(str, 1);
		}
	}

	//check whether the document contains the query term
	public boolean contains(String term) {
		return m_BoW.containsKey(term);
	}

	//return the frequency of query term in this document
	public int counts(String term) {
		if (contains(term))
			return m_BoW.get(term);
		else
			return 0;
	}

	//test function for checking the size of BoW
	public HashMap<String, Integer> getBoW() {
		return m_BoW;
	}
}
```

Corpus.java

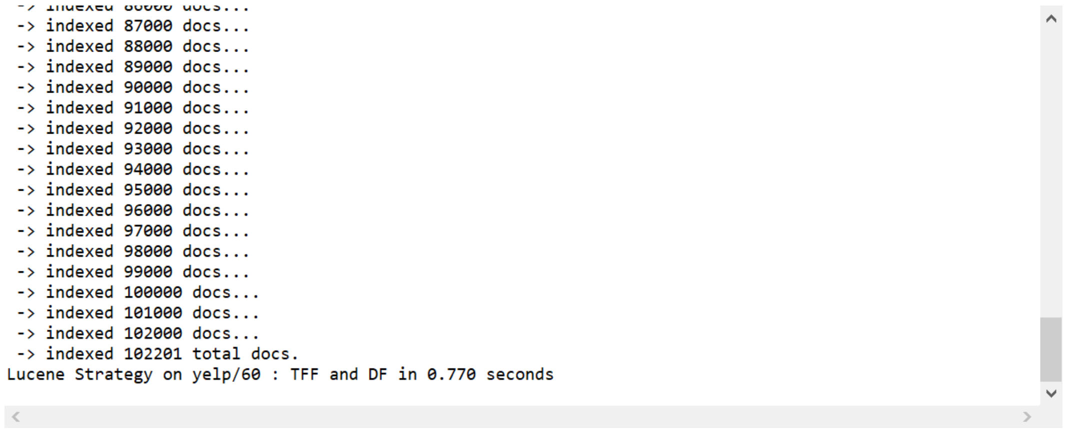
```
	package structures;

	import java.util.HashMap;
	import java.util.LinkedList;

	public class Corpus {
		LinkedList<ReviewDoc> m_collection; // a list of review documents

		HashMap<String, Integer> m_dictionary; // dictionary of observed words in this corpus, 
word -> frequency
												// you can use 
this structure to prepare the curve of Zipf's law
		HashMap<String, Integer> m_dictionary_df;

		public Corpus() {
			m_collection = new LinkedList<ReviewDoc>();
			m_dictionary = new HashMap<String, Integer>();
			m_dictionary_df = new HashMap<String, Integer>();
		}

		public int getCorpusSize() {
			return m_collection.size();
		}

		public int getDictionarySize() {
			return m_dictionary.size();
		}

		public void addDoc(ReviewDoc doc) {
			m_collection.add(doc);

			/**
			 * INSTRUCTOR'S NOTE: based on the BoW representation of this document, 
you can update the m_dictionary content
			 * to maintain some global statistics here
			 */
			for (String term : doc.m_BoW.keySet()) {
				setWordCount(term, getWordCount(term) + doc.m_BoW.get(term));
				setDocCount(term, getDocCount(term) + 1);
			}
		}

		public ReviewDoc getDoc(int index) {
			if (index < getCorpusSize())
				return m_collection.get(index);
			else
				return null;
		}
	public int getWordCount(String term) {
		if (m_dictionary.containsKey(term))
			return m_dictionary.get(term);
		else
			return 0;
	}

	void setWordCount(String term, int count) {
		m_dictionary.put(term, count);
	}


	//my func
	public int getDocCount(String term) {
		if (m_dictionary_df.containsKey(term))
			return m_dictionary_df.get(term);
		else
			return 0;
	}

	//my func
	void setDocCount(String term, int count) {
		m_dictionary_df.put(term, count);
	}

	//test function for getting the dictionary
	public HashMap<String, Integer> getDictionary() {
		return m_dictionary;
	}

	//test function for getting the collection
	public LinkedList<ReviewDoc> getCollections() {
		return m_collection;
	}

	public HashMap<String, Integer> getDF() {
		return m_dictionary_df;
	}
}
```

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.**



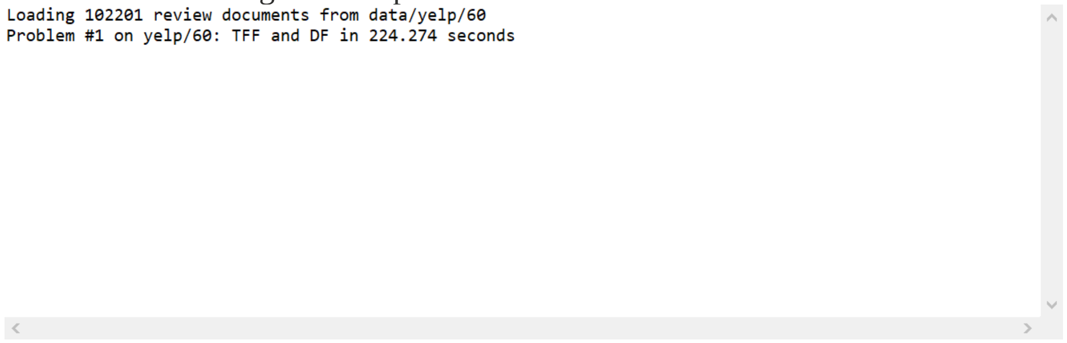


- TTF/DF running time on Lucene strategy:



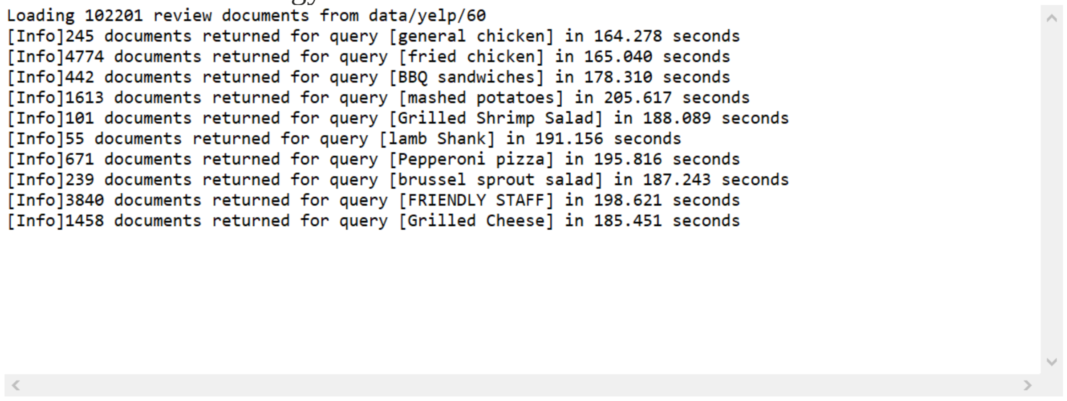
```
 -> indexed 87000 docs...
 -> indexed 88000 docs...
 -> indexed 89000 docs...
 -> indexed 90000 docs...
 -> indexed 91000 docs...
 -> indexed 92000 docs...
 -> indexed 93000 docs...
 -> indexed 94000 docs...
 -> indexed 95000 docs...
 -> indexed 96000 docs...
 -> indexed 97000 docs...
 -> indexed 98000 docs...
 -> indexed 99000 docs...
 -> indexed 100000 docs...
 -> indexed 101000 docs...
 -> indexed 102000 docs...
 -> indexed 102201 total docs.
Lucene Strategy on yelp/60 : TFF and DF in 0.770 seconds
```

- TTF/DF running time for problem 1:



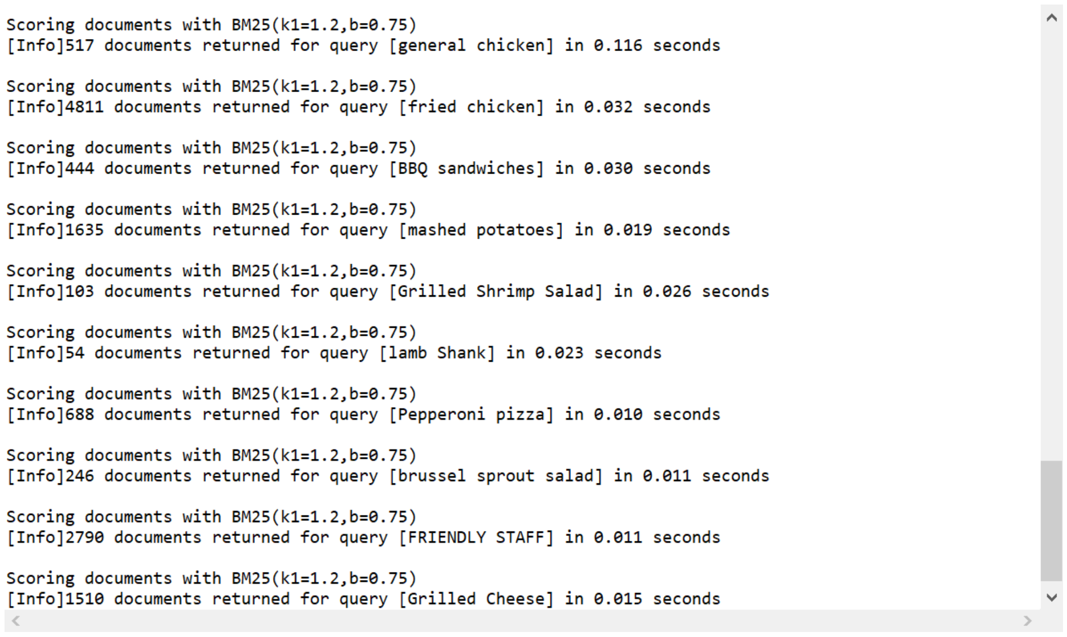
```
Loading 102201 review documents from data/yelp/60
Problem #1 on yelp/60: TFF and DF in 224.274 seconds
```

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?**
   - Brute-force Strategy:



```
Loading 102201 review documents from data/yelp/60
[Info]245 documents returned for query [general chicken] in 164.278 seconds
[Info]4774 documents returned for query [fried chicken] in 165.040 seconds
[Info]442 documents returned for query [BBQ sandwiches] in 178.310 seconds
[Info]1613 documents returned for query [mashed potatoes] in 205.617 seconds
[Info]101 documents returned for query [Grilled Shrimp Salad] in 188.089 seconds
[Info]55 documents returned for query [lamb Shank] in 191.156 seconds
[Info]671 documents returned for query [Pepperoni pizza] in 195.816 seconds
[Info]239 documents returned for query [brussel sprout salad] in 187.243 seconds
[Info]3840 documents returned for query [FRIENDLY STAFF] in 198.621 seconds
[Info]1458 documents returned for query [Grilled Cheese] in 185.451 seconds
```

   - Lucene strategy:



```
Scoring documents with BM25(k1=1.2,b=0.75)
[Info]517 documents returned for query [general chicken] in 0.116 seconds

Scoring documents with BM25(k1=1.2,b=0.75)
[Info]4811 documents returned for query [fried chicken] in 0.032 seconds

Scoring documents with BM25(k1=1.2,b=0.75)
[Info]444 documents returned for query [BBQ sandwiches] in 0.030 seconds

Scoring documents with BM25(k1=1.2,b=0.75)
[Info]1635 documents returned for query [mashed potatoes] in 0.019 seconds

Scoring documents with BM25(k1=1.2,b=0.75)
[Info]103 documents returned for query [Grilled Shrimp Salad] in 0.026 seconds

Scoring documents with BM25(k1=1.2,b=0.75)
[Info]54 documents returned for query [lamb Shank] in 0.023 seconds

Scoring documents with BM25(k1=1.2,b=0.75)
[Info]688 documents returned for query [Pepperoni pizza] in 0.010 seconds

Scoring documents with BM25(k1=1.2,b=0.75)
[Info]246 documents returned for query [brussel sprout salad] in 0.011 seconds

Scoring documents with BM25(k1=1.2,b=0.75)
[Info]2790 documents returned for query [FRIENDLY STAFF] in 0.011 seconds

Scoring documents with BM25(k1=1.2,b=0.75)
[Info]1510 documents returned for query [Grilled Cheese] in 0.015 seconds
```

From the printout we can see that for each user query, the lucene strategy gives more number of matched documents with less running time. Brute-force strategy takes average 3 minutes to find out all matched documents for each query, while the lucene only needs 10 milliseconds on the same query, which is much faster than the brute force one. During the preprocessing step, brute-force uses snow ball stemmer while the lucene uses porter stemmer, and they will perform differently on the same user query and document contents. So the number of returned documents are different for the same user query.