Winter 2024

Hao Jun Chen

February 7, 2024

1 Part 1 Understand Zipf's Law

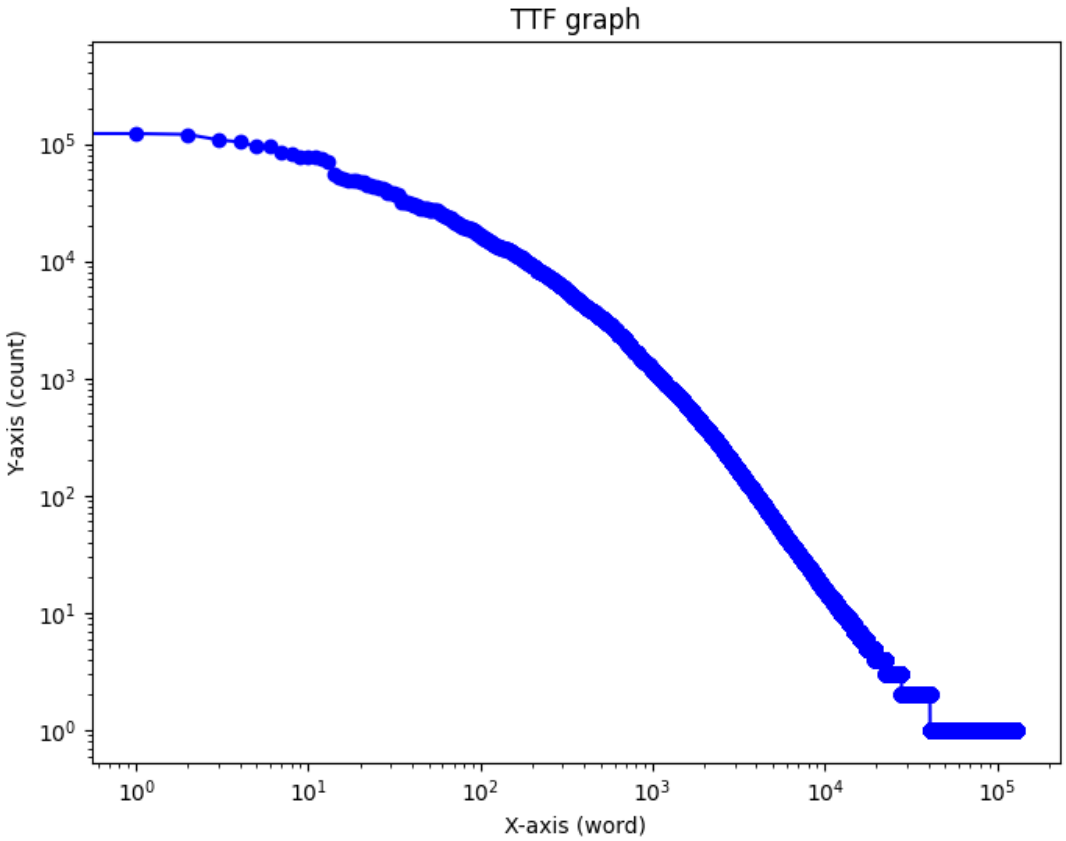
(a) implementation of text normalization module

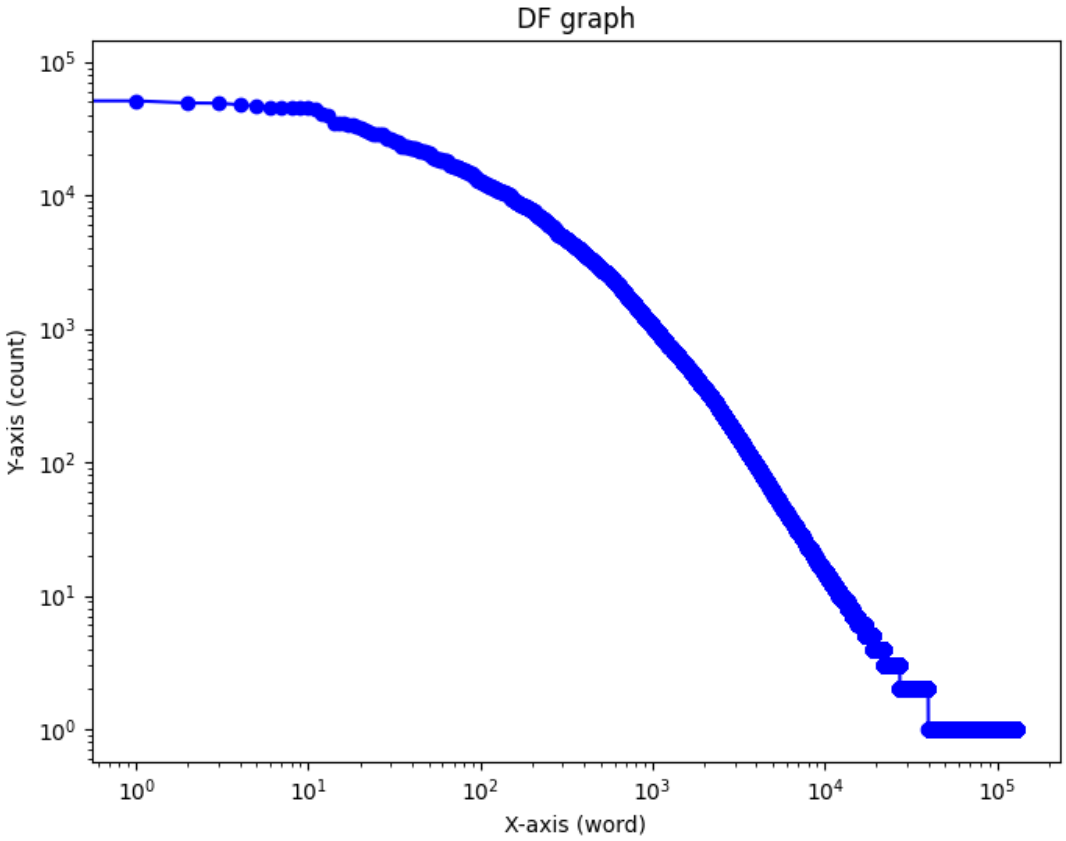
```
            def is_stop_word(token):
                stop_words = set([
                    'a', 'an', 'and', 'are', 'as', 'at', 'be', 'but', 'by'
                        , 'for', 'if', 'in',
                    'into', 'is', 'it', 'no', 'not', 'of', 'on', 'or', '
                        such', 'that', 'the',
                    'their', 'then', 'there', 'these', 'they', 'this', 'to
                        ', 'was', 'will', 'with'
                ])

                # Convert the word to lowercase for case-insensitive
                    comparison
                lowercase_word = token.lower()

                return lowercase_word in stop_words
            def remove_non_alphanumeric(input_string):
                # Use regular expression to keep only English letters and
                    numbers
                result = re.sub(r'[^a-zA-Z0-9]', '', input_string)
                return result
            def is_integer(s):
                try:
                    int(s)
                    return True
                except ValueError:
                    return False
            path = os.getcwd() + "/yelp"
            tokens = {}
            TFF_count = {}
            DF_count = {}
            stemmer = PorterStemmer()
            yelp_store_list = os.listdir(path)
            #Loop through each yelp file
            for yelp_store in yelp_store_list:
                store_path = path + "/" + yelp_store
                with open(store_path, 'r') as file:
                    data = json.load(file)
                reviews = data["Reviews"]
                #Loop through each review in the yelp _ file
                for review in reviews:
                    doc = review['Content']
                    doc_ID = review['ReviewID']
```

```
                    tokens = word_tokenize(doc) #token the review content
                    #Loop through each token and normal it
                    for token in tokens:
                        token = remove_non_alphanumeric(token)
                        if len(token) == 0:
                            continue
                        elif is_integer(token):
                            token = "NUM"
                        elif is_stop_word(token):
                            continue
                        else:
                            token = stemmer.stem(token)
                            token = token.lower()
                        TFF_count[token] = 1 + TFF_count.get(token, 0)
                        if token in DF_count and DF_count[token] is not
                            None:
                            DF_count[token].add(doc_ID)
                        else:
                            DF_count[token] = set()
                            DF_count[token].add(doc_ID)
```

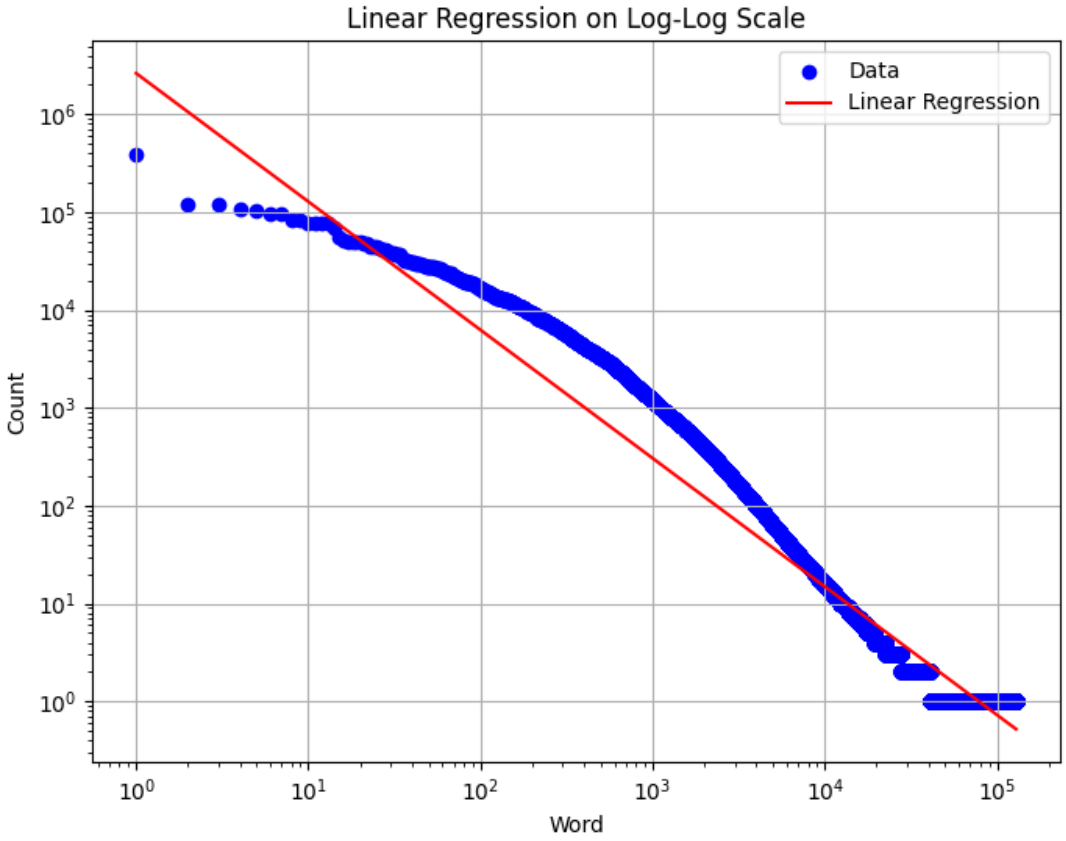
Listing 1: Implementation of text normalization

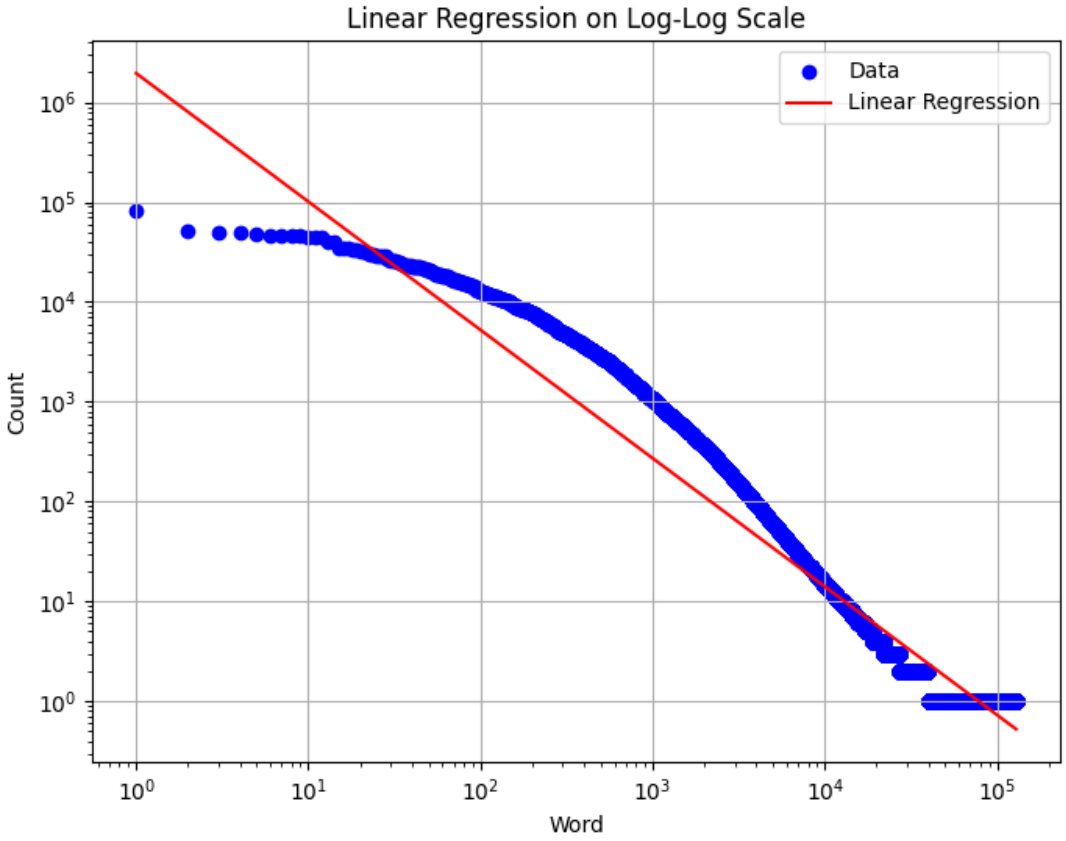
(b) Two curves in log-log space generated above, with the corresponding slopes and intercepts of the linear interpretation results





(c) Linear Regression on TFF and DF graph


Linear Regression on Log-Log Scale
Data
Linear Regression
Count
Word
10^0
10^1
10^2
10^3
10^4
10^5
10^6



I found a fairly strong linear relationship in these two graphs. As both of their slope are around -1.3, which is very close to one.

2 Part 2 Build Inverted Index

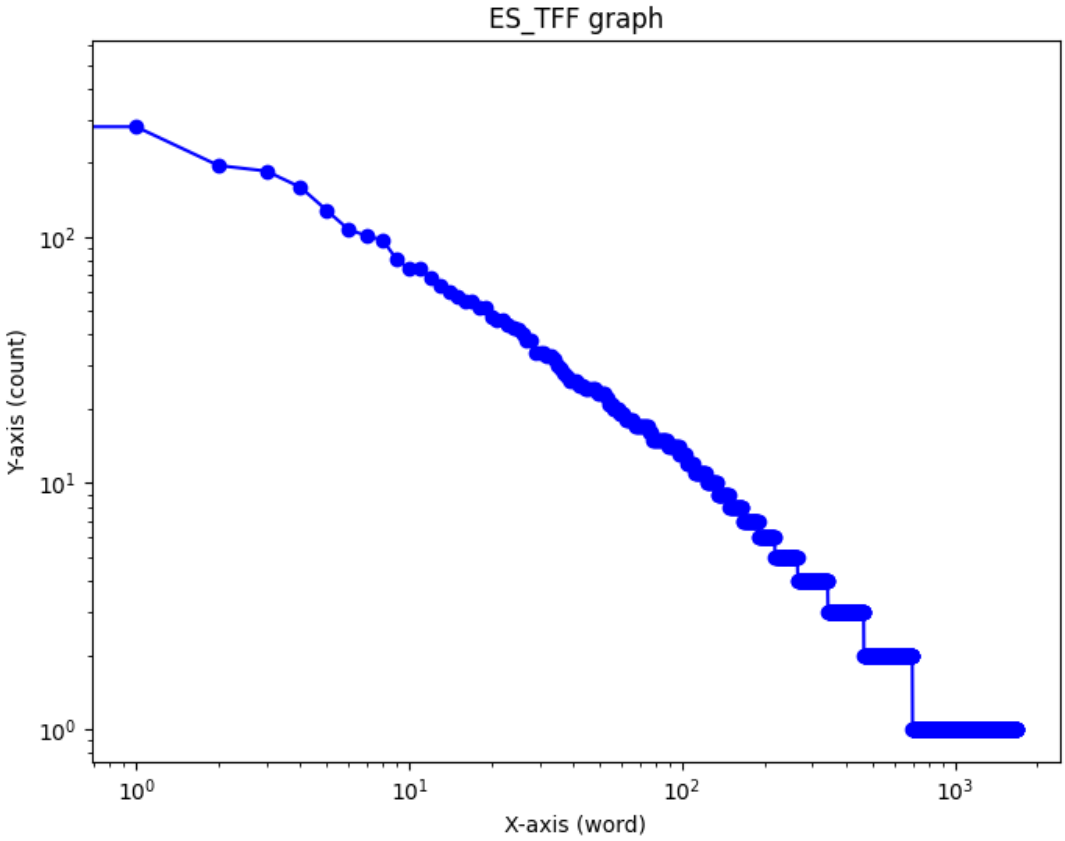
(a) implementation of text normalization module

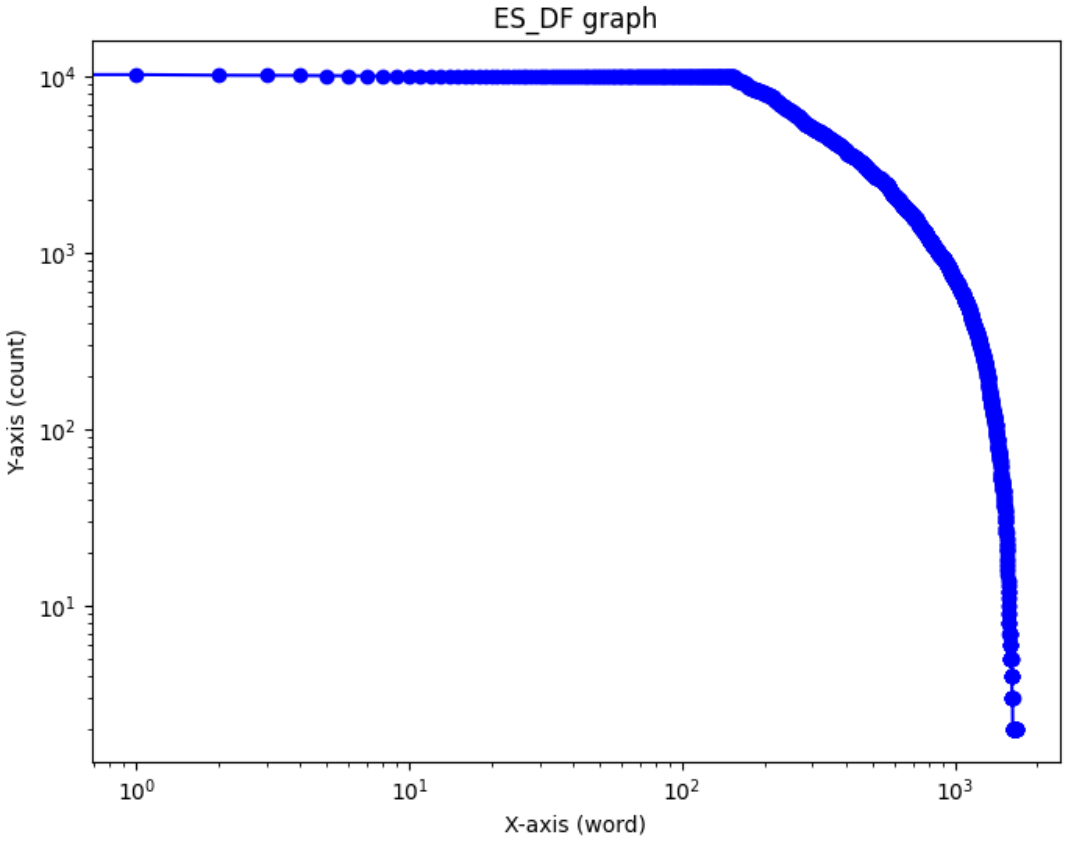
```
            def is_stop_word(token):
                stop_words = set([
                    'a', 'an', 'and', 'are', 'as', 'at', 'be', 'but', 'by'
                        , 'for', 'if', 'in',
                    'into', 'is', 'it', 'no', 'not', 'of', 'on', 'or', '
                        such', 'that', 'the',
                    'their', 'then', 'there', 'these', 'they', 'this', 'to
                        ', 'was', 'will', 'with'
                ])

                # Convert the word to lowercase for case-insensitive
                    comparison
                lowercase_word = token.lower()

                return lowercase_word in stop_words
            def remove_non_alphanumeric(input_string):
                # Use regular expression to keep only English letters and
                    numbers
```

```
    result = re.sub(r'[^a-zA-Z0-9]', '', input_string)
    return result
def is_integer(s):
    try:
        int(s)
        return True
    except ValueError:
        return False
path = os.getcwd() + "/yelp"
inverted_index = {}
stemmer = PorterStemmer()
yelp_store_list = os.listdir(path)
#loop through all the yelp data
for yelp_store in yelp_store_list:
    store_path = path + "/" + yelp_store
    with open(store_path, 'r') as file:
        data = json.load(file)
    reviews = data["Reviews"]
    #loop through each review item in each data
    for review in reviews:
        doc = review['Content']
        doc_ID = review['ReviewID']
        tokens = word_tokenize(doc)
        #Normalize each word
        for token in tokens:
            token = remove_non_alphanumeric(token)
            if len(token) == 0:
                continue
            elif is_integer(token):
                token = "NUM"
            elif is_stop_word(token):
                continue
            else:
                token = stemmer.stem(token)
                token = token.lower()
            #For each word, record the document that contain it
            if token in inverted_index and inverted_index[token] is not None:
                inverted_index[token].add(doc_ID)
            else:
                inverted_index[token] = set()
                inverted_index[token].add(doc_ID)
```

Listing 2: Implementation of inverted index

(b) Two curves in log-log space generated by using Elasticsearch inverted index for collecting TTF and DF statistics


ES_TFF graph
Y-axis (count)
X-axis (word)
10^0
10^1
10^2
10^3



Comparing to the graph from part one, We can see that both TFF and DF have similar graph shapes but have smaller amounts of frequency on both TFF and DF. This makes sense since we are looking at only 9 queries.

(c) Running time and total number of returned documents by two retrieval models

Runtime for Elastic search is 1.33 seconds and my model is 0.003 seconds. As Dr. Wang mentions in class, this is due to the system Elasticsearch is bigger. If we keep putting large data in, eventually, Elasticsearch will be faster. I got quite different results from the Elasticsearch. One major reason is how Elasticsearch handles tokenization and normalization as I put unprocessed documents into elastic search. I believe that if I put processed document, the retrieved document would be similar between my model and elastic one. Here is the table of my model vs elastic search,

	A	B	C	D
1		ES model	My model	
2	general chicken	>=10000	507	
3	fry chicken	>=10000	4767	
4	bbq sandwich	9513	442	
5	mash potato	3926	1617	
6	grill shrimp	>=10000	499	
7	lamb shank	1502	54	
8	pepperoni pizza	>=10000	669	
9	friend staff	>=10000	1572	
10	grill cheese	>=10000	1462	
11				