**CSE 519 -- Data Science (Fall 2017)**
**Project Progress Report**
**Kiranmayi Kasarapu - 111447596**
**Shruti Nair - 111481332**
**Rohan Vaish - 111447435**

# Project: Automatic Building of Book Indices

## Challenge

The goal of our project is to predict the indices for a journal or article given in the form of latex file. We have to build index based on the content of the article and write indices back to the latex source file.

## BaseLine Model

## Frequency Based Index Building

For the baseline model, we implemented the frequency search algorithm shown above to extract the most frequently occurring words in the document. Based on the number of indices, the user inputs, say n, we extracted the top n frequently occurring words and listed them out as the indices for the document.

**Algorithm**:

a. Have a large corpus of text against which we can compare.
b. Tag each tokenizer according to its type after removing all the stop words.
c. Find the relative frequency of words in the corpus. For example, if your corpus is "the green way is very green way green". Relative frequency is.. [ (the, 1/8), (green, 3/8), (way, 2/8), (is, 1/8), (very, 1/8), ]
d. Get the relative frequency of words in search string.
e. Divide frequency of search string by corpus, and also take care of the cases, when the word is not included in the string.
f. Sort the answer by the result obtained in d and generate the top n words as given by the user.

The below graph shows the number of matched indices percentage in 4 different files.

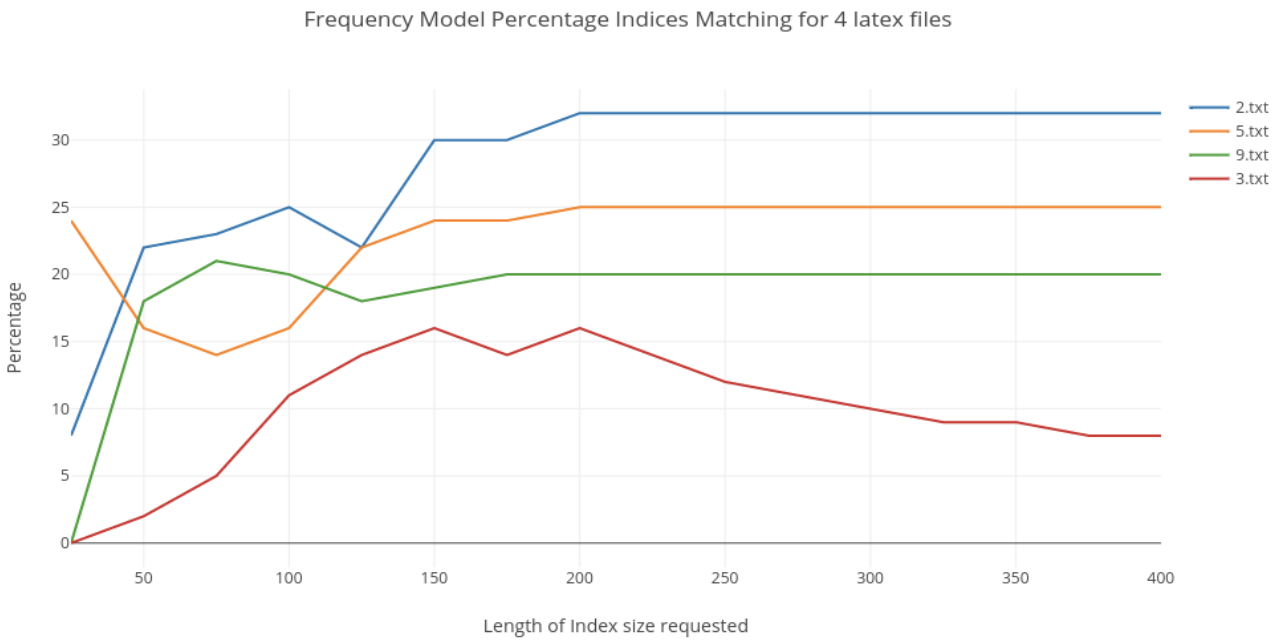



Figure No 1

From the above graph we can see that the percentage of indices that are being matches lies just above **30%.**

### Advanced Models

### Word Embeddings

A word embedding is an approach to provide a dense vector representation of words that capture something about their meaning.

Word embeddings are an improvement over simpler bag-of-word model word encoding schemes like word counts and frequencies that result in large and sparse vectors (mostly 0 values) that describe documents but not the meaning of the words.

The library used for Word embedding is **Gensim** which is an open source library for natural language processing.

### Word2Vec Algorithm

This algorithm has been deployed for learning words from a large corpus text. It generally look at a window of words for each target word to provide context and in turn meaning for words.

**Requirements** :- A large corpus text to learn from.

**Input form** :- List of sentences.

**Algorithm**

a) Convert the plain text obtained from the latex file in the form of list of sentences.
b) Using the gensim library, convert each word from the sentence into vector representation by utilizing Word2Vec Algorithm.
c) Once the model is obtained, feed the modelled words into a function called build_nameset.
d) Call the helper function remove_stop_words for removing words that aren't necessary or relevant in context to major key words.
e) Use the collections library and import counter function for finding out the most common occurring words in the modelled data set.
f) Eliminate any word shorter than three for better extraction of the key words.
g) Parse through each of the key words and store it in the dictionary along with its count.
h) Return the dictionary with important keywords that are relevant to the latex file.
i) Perform a Principal Component Analysis to find out the similar set of words by analysing clusters for further usage.

**Output**

We varied the size of the indices that needs to be generated and compared the generated indices with the actual indices from the latex file, that we extracted. We calculated the percentage of common indices we are able to extract using the algorithm and plotted on a bar chart against the percentage obtained from the index size. We have shown the results below for a latex file below.

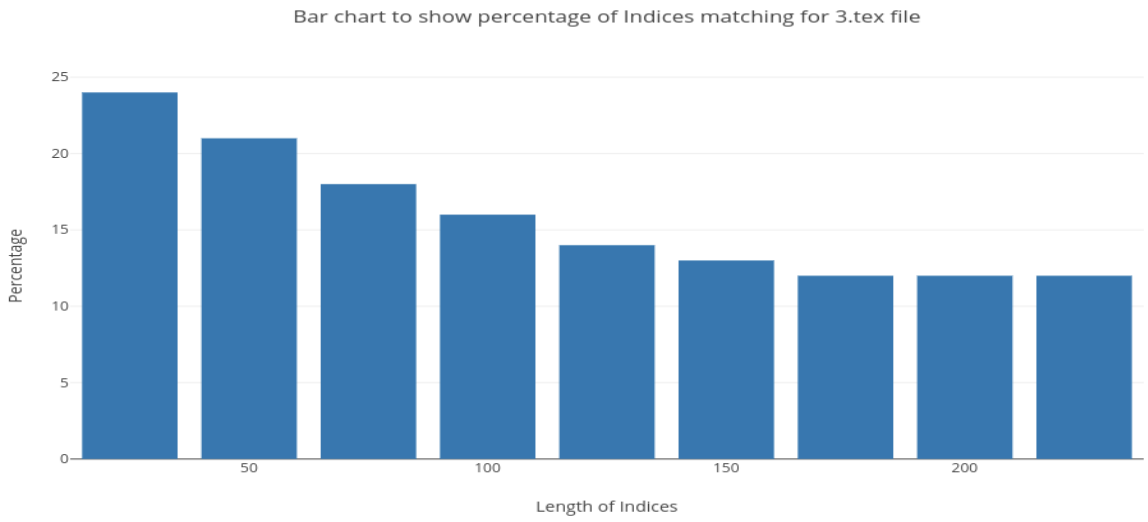


Figure No 2

The above graph shows the percentage of indices that were common using word embeddings. Here we see the percentage decreasing because, the model just returns the most words that have context with the document. So we have only one set of words which are index worthy, and they do not increase as the size of index decreases.

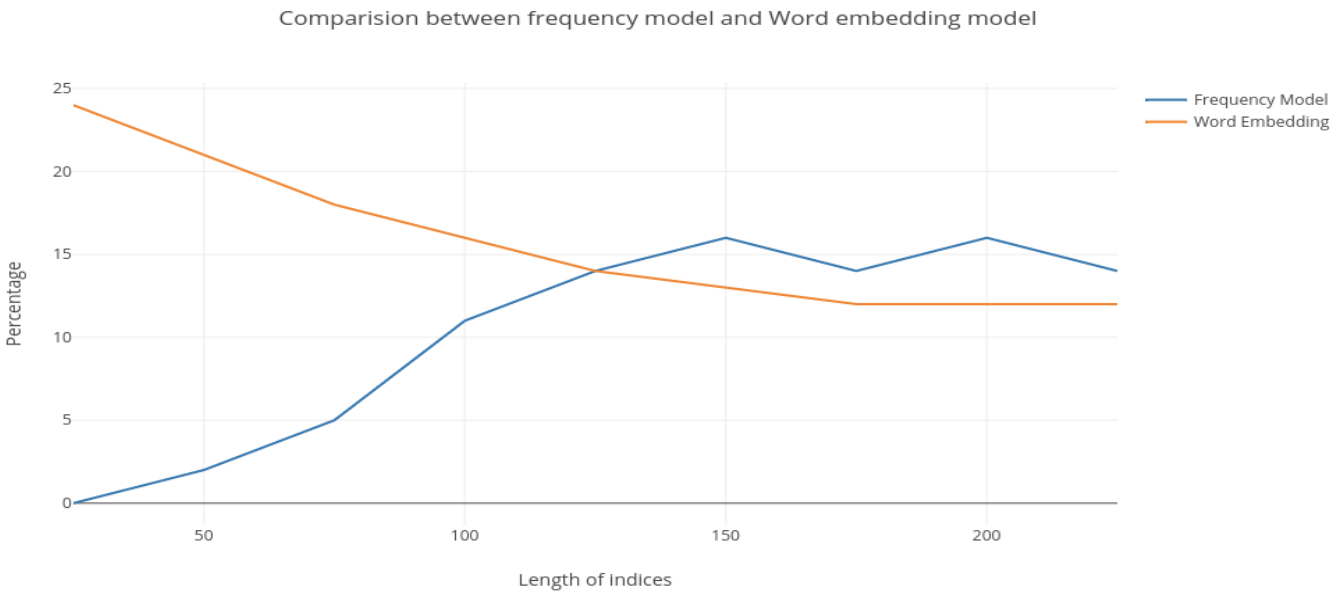



Figure No 3

**Google N-grams**

*What is a n-gram?*

A n-gram is defined as a sequence of n characters from a given text or speech. For our project we considered only n-grams to be from text.

We worked with only 1-gram, which means predicting whether a single word is index worthy or not.

Google books Ngram is a corpus of digitized texts. The data consists of books spanning from years 1800 to 2000s. The corpus consists of upto 5-grams. Each N-gram is kept separately from other N-grams.

For example, 1-gram corpus consists of all words starting with each of the 26 alphabets along with digits.

So there are a total of 36 files for 1-gram. Each file consists of data in the following format.

**ngram TAB year TAB match_count TAB page_count TAB volume_count NEWLINE**

A typical line looks like the following:

circumvallate 1978 335 91

**Modelling**

For our project we are building only 1 word indices and we are building only english words starting with alphabets. Therefore we worked with only the 26 1-gram google corpus files for our modelling.

As we did in our Baseline Model, we extracted all word after removing stop words from the original parsed latex file. We searched each word in the entire 1-gram corpus and calculated the frequency of occurrence of this word over the years and the number of documents it appeared on. This process is repeated for every word we extracted from the latex file. A new file is created for all the words we extracted containing the frequency of occurrence in google n-gram books, number of documents it appeared in and the frequency of this word in the latex file.

| | word_ngram_freq | volume_ngram_freq | freq |
|---|---|---|---|
| arzel | 803 | 399 | 15 |
| eigenvalues | 1264669 | 149915 | 12 |
| regarded | 56261135 | 6186662 | 3 |
| periodic | 10645077 | 2760904 | 8 |
| urbana | 2132142 | 791103 | 1 |
| limits | 45290452 | 8024836 | 5 |
| connect | 11704935 | 3545835 | 1 |
| success | 93383234 | 9080063 | 1 |
| alternative | 50908630 | 8499118 | 2 |
| enough | 188948038 | 13701241 | 5 |
| extent | 97114220 | 7699661 | 1 |
| dual | 11922418 | 3352219 | 20 |

Figure No 4

Once we extracted this information from the 1-gram corpus, we designed a score function which appropriately assigns score to each word based on these three frequencies.

The idea here is if a particular word occurs frequently in our document and it relatively occurs less frequently in 1-gram corpus, the word should be important to this document and must be considered index worthy. Since the google n-gram corpus has millions of books, the frequencies are in the order of 1000s. In order to scale down all the 3 frequencies to a similar scale, we normalized the 3 frequencies. In order to avoid negative values, we scaled up each value by the minimum value of each frequency.

After normalization, we applied our scoring function as defined below:

$$Score = \frac{volumeNgramFreq * freq}{wordNgramFreq}$$

We sorted the word in increasing order of the score we obtained and return the top “n” indices as requested by the user.

| Word | word_ngram_freq | volume_ngram_freq | freq | score |
|---|---|---|---|---|
| geodesic | 0.002587055 | 0.01726211 | 9.427616476 | 62.90572548 |
| curvature | 0.041460372 | 0.197788105 | 11.02199279 | 52.58078928 |
| metric | 0.046746931 | 0.237215809 | 9.982182151 | 50.6542647 |
| manifold | 0.040314269 | 0.375221271 | 5.199053204 | 48.38969984 |
| riemannian | 0.001121108 | 0.004667313 | 9.081012929 | 37.8054065 |

Figure No 5

**Output:**

We ran this model, through our evaluation environment and examined the following results:

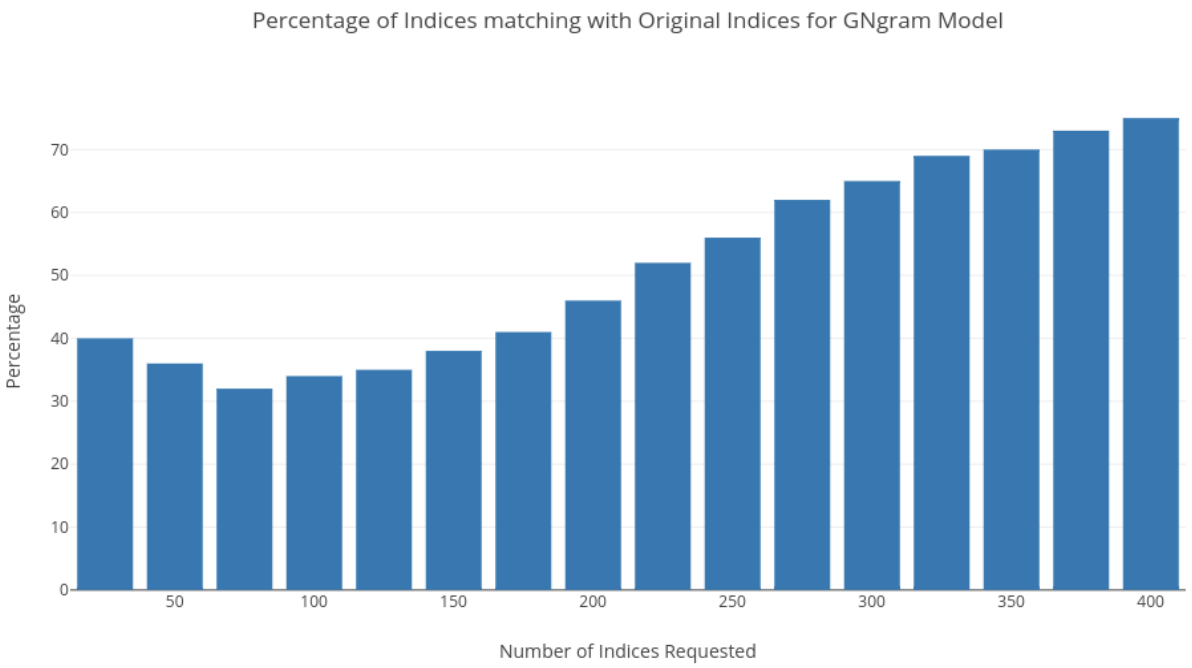



Figure No 6

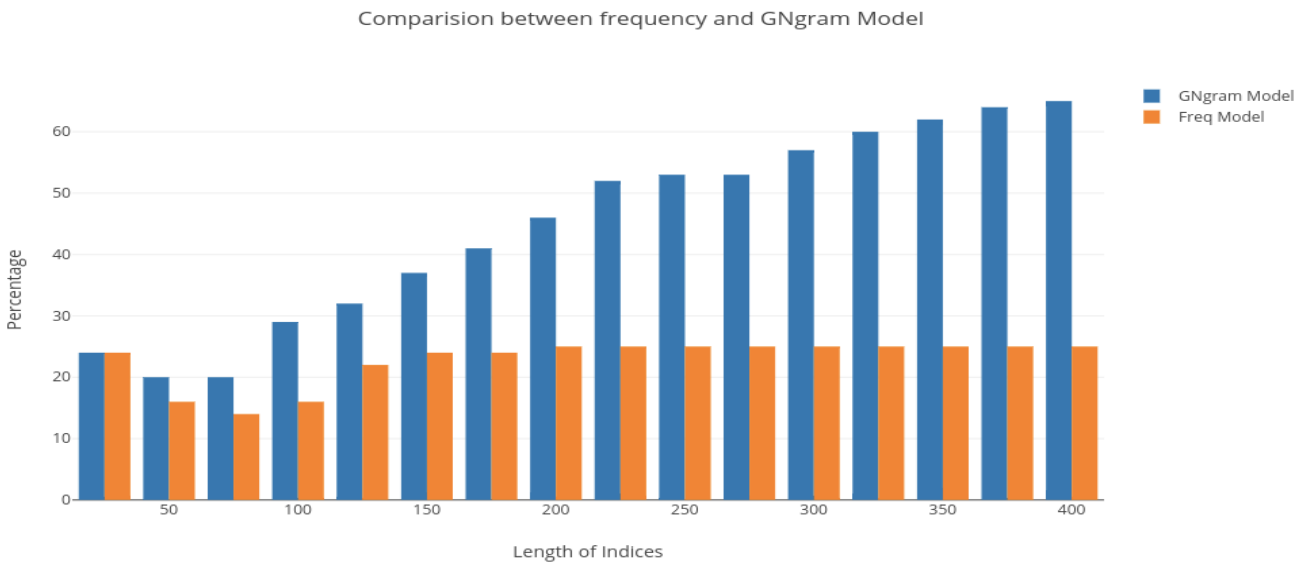



Figure No 7

From the above two graphs we can see that the percentage of indices matched with the original indices increase and lies around **65%-75%**

The graph below shows the absolute number of indices matching the original indices as the input length increases.

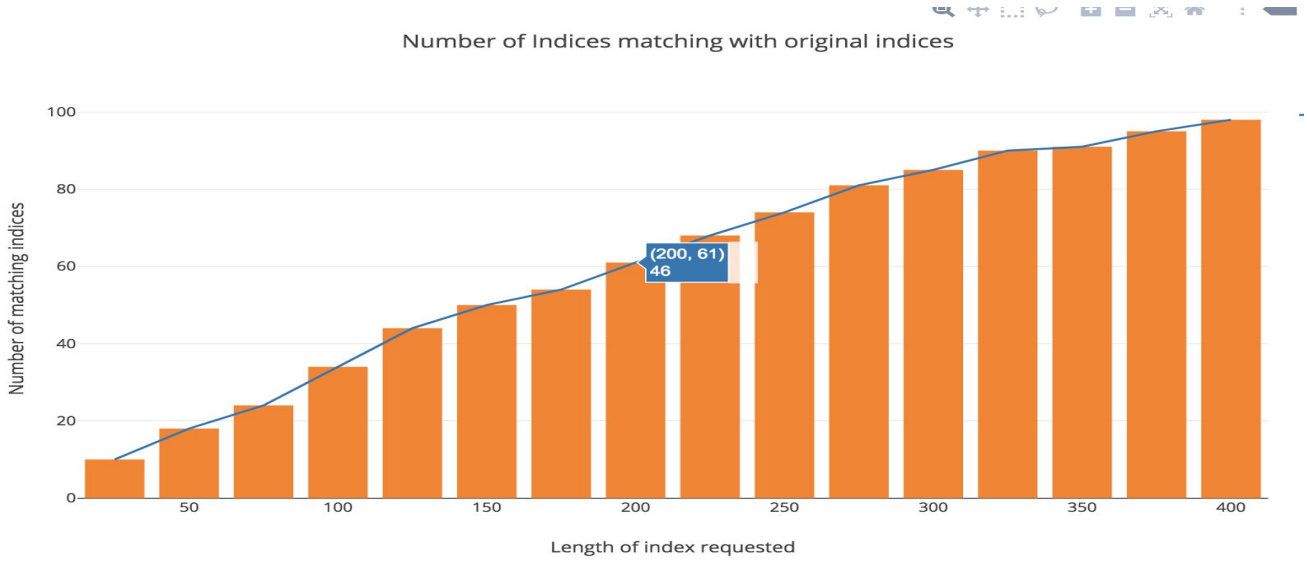



Figure No 8

The below graph shows variation in the indices predicted as the score function changes.

In all the below 3 score functions, we kept the score to be inversely proportional to the wordNgramFreq, and varied the VolumeNgramFreq.

$$Score1 = \frac{volumeNgramFreq * freq}{wordNgramFreq}$$

$$Score2 = \frac{freq}{wordNgramFreq}$$

$$Score3 = \frac{freq}{wordNgramFreq * VolumeNgramFreq}$$

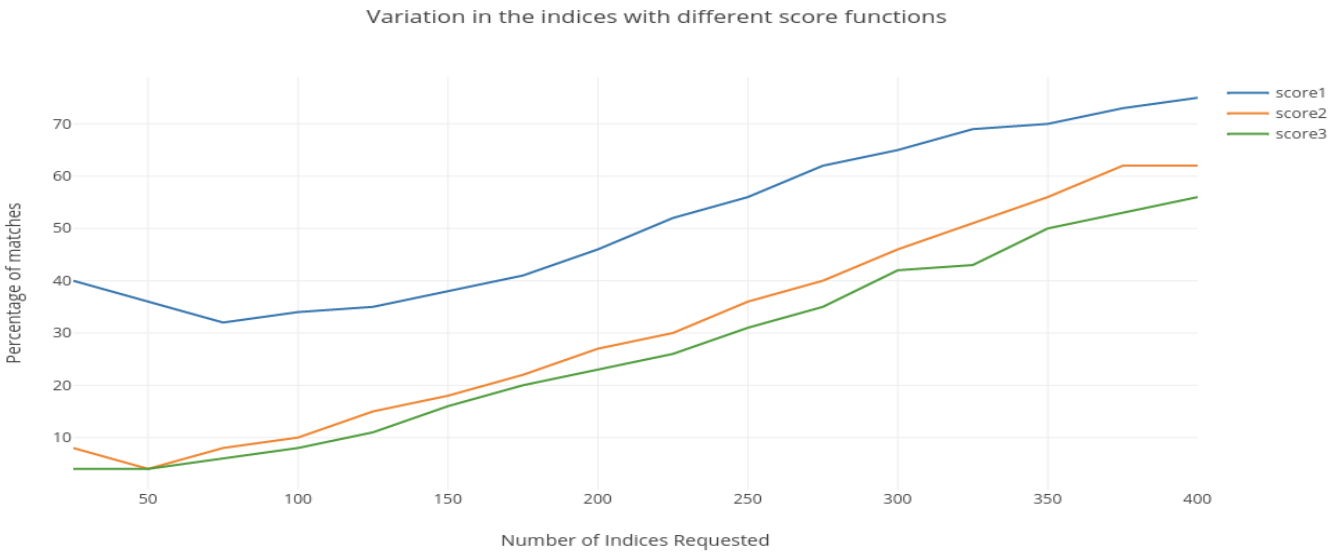



Figure No 9

From the above graph, we can see that if volume count are inversely proportional to the score function, we are getting less percentage of important words, which is expected. This function is just to show how word importance varies in a document based on these frequency values.

From the above graph we can infer that, that score function 1 is doing better in terms of predicting the important words for a given document.

**Evaluation**

For our project, since we are generating 1 word indices for all the documents, therefore, for evaluation we have considered the following assumptions:

a. All the indices extracted from the already indexed original document are of 1 word size
b. If any index is a phrase, we have broken down that phrase into words to create multiple indices out of it. The idea behind this is, if the phrase is index worthy, then if the indexer decides to build 1 word indices, he/she will include each word of that phrase as different indices
c. Also, while breaking down the phrase into multiple indices, we have removed all the stop words because stop words are mostly never a part of the index

After considering all the above mentioned points, we have generated the percentage match/accuracy between the index of the original document and the one our model generates

**Predictions and Conclusion**

We can clearly see that Google Ngram corpus has given us better results compared to frequency model and Word embedding model.
This model is giving us close to **60%-75%** index matching across various articles and journals. The percentage of indices matched also increases with the size of the index requested. The below graph shows the comparison between the 3 different models

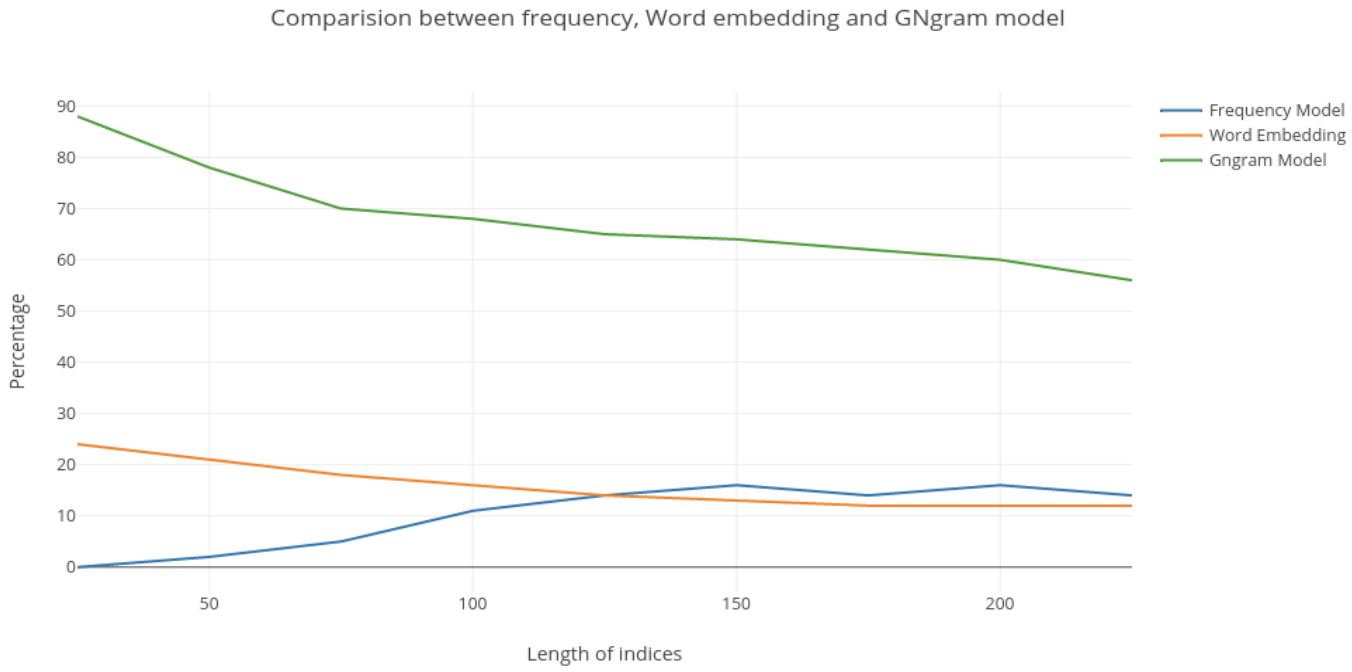
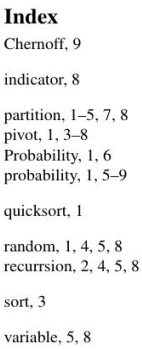



Figure No 10

**Other Tasks**

We have added code to generate the indices in the latex file. This part takes care of reading the generated index file and adding index in the latex file. Once the new file is generated, it can be run through a latex compiler and therefore generated a pdf file with back-of-the-book index. We got the positions of the word in the latex file and added beside the index word in the generated index.



**Index**

Chernoff, 9

indicator, 8

partition, 1–5, 7, 8
pivot, 1, 3–8
Probability, 1, 6
probability, 1, 5–9

quicksort, 1

random, 1, 4, 5, 8
recurrsion, 2, 4, 5, 8

sort, 3

variable, 5, 8

Figure No 11

**Sample Output**

We will show a sample of indices generated for a pdf file and the original indices of the file against which we compared.

The sample paper we will be using is https://arxiv.org/abs/1104.1563

This is a large paper of 88 pages and has over 100 indices.

Sample of the Original and predicted Indices:

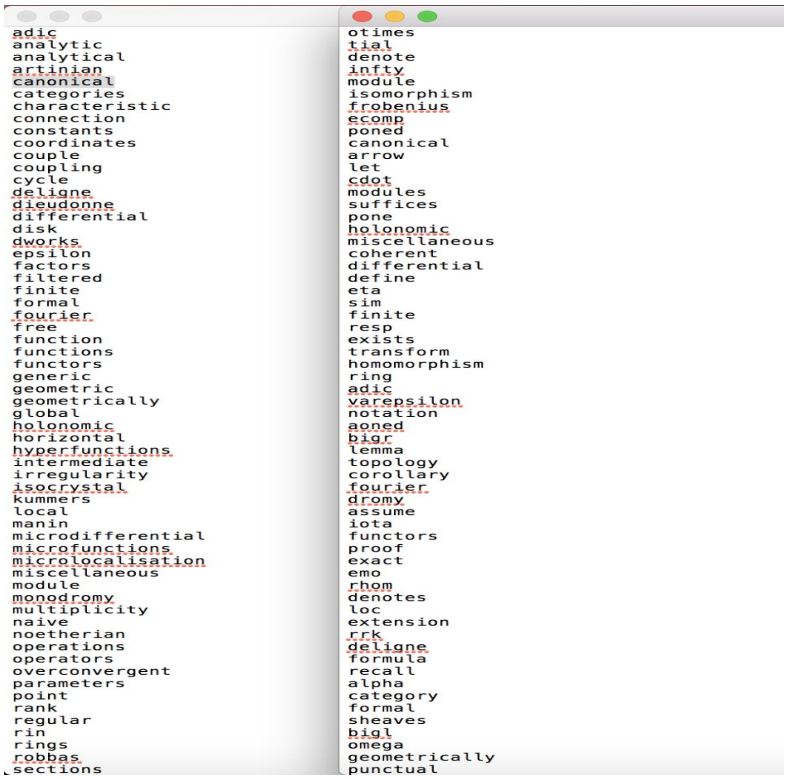



Figure No 12

## Improvements

We suggest the following improvements and the future work that can be continued:

1. Google N-grams model have worked better for us for 1-gram words. This model can be easily extended with few modifications to scoring function for bigrams and trigrams.
2. Currently we run each step of the model separately to generate out indices. This can be automated and made as a command line argument or create a website to upload the latex file and return back the generated new latex source file with indices.


## Acknowledgements

We would like to thank Professor Steven Skiena for his support and encouragement during the course of the project.


## References


**http://storage.googleapis.com/books/ngrams/books/datasetsv2.html**