CSE272: Impact of Text Summarization on Search Results

Omkar Ghanekar
oghaneka@ucsc.edu

Smruthi Pobbathi
spobbath@ucsc.edu

ABSTRACT

Text summarize feature is commonly used in several search engines. Google uses text summarize algorithm to display the contents of the documents retrieved in search results page. Several companies use text summarize feature to display the features of the product as marketing strategies. Several students use text summarizer tools to read large documents. In this project, we use text summarizer to summarize a Wikipedia dump and compare the search results of large text content and summarized text content. Perform precision, recall and speed evaluations on both the search results. Our aim is to see if summarized text can be indexed instead of full text documents and obtain similar results.

KEYWORDS

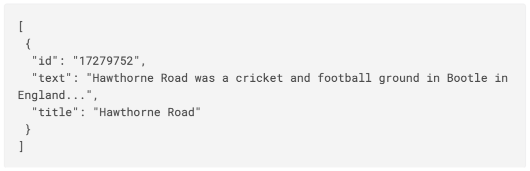
dataset, text summarizer, Elasticsearch, fuzzy search, function score, precision, recall

1 INTRODUCTION

We all interact with applications which uses text summarization. Many of those applications are for the platform which publishes articles on daily news, entertainment, sports. With our busy schedule, we prefer to read the summary of those article before we decide to jump in for reading entire article. Reading a summary help us to identify the interest area, gives a brief context of the story.

In the past decade, summary generation tools have played an important role in search engine optimization. Most popular search engines use text summarization methods to improve user experience. Well where have we seen text summarization? When we search something on Google, the results page is displayed. This results page consists of a short summary of the article present in search results. Many different marketing techniques use this text summarization tools to provide a brief description of the content and improve search engine's ranking. Huge amount of data is being indexed every second in very powerful search engines. But when we want to use say Wikipedia offline, it is practically impossible to store all the data locally and index that data.

So here comes the motivation of our project. We are implementing a search engine for Wikipedia dataset [] that we gathered from Kaggle. This dataset is a Wikipedia dump from 2020-10-20. The Wikipedia dataset we used was mostly unstructured with only a few fields, id, title and description. The aim of text summarization was to remove redundancy in the massive dataset that we were searching through. On this unstructured data, we perform an NLP text summarization algorithm to extract summary of the long text data. We then index these two datasets and perform multiple types of search queries. We collect the search queries by performing web crawling. Then we use these queries to search summarized data as well as the long dataset and retrieve results and ranks of the documents. On the obtained search results we perform evaluations to compare the effect of summarization. Some evaluation metrics that we plan to compare are precision, recall, performance of our search engine in terms of speed.



```
[
 {
  "id": "17279752",
  "text": "Hawthorne Road was a cricket and football ground in Bootle in
England...",
  "title": "Hawthorne Road"
 }
]
```

Figure 1: Dataset

2 BACKGROUND

2.1 Dataset

As mentioned earlier, we use Wikipedia dataset from Kaggle []. This dataset is of size 8GB and has over 6.1 million documents. The dataset has 3 fields doc_id, doc_title, doc_description. The doc_description filed contains unstructured data. This unstructured data has lot of redundant information. Fig(1) shows the format of the dataset.

2.2 Queries

Normally, we use 2 types of queries with a search engine, Transactional and Information Search queries. In Transactional queries, user searches for a generic search term, eg: List of popular car brands.
For Information queries, user looks for a specific key-word results. eg. BMW. In our project, we focus on the latter queries. To create a diverse dataset for queries, we used Wikipedia API[1] for generating random queries. Then, we used the queries and used the python Wikipedia package to get the relevant Wikipedia search results for the specified query. We generated a diverse and topic agnostic query-set for around 1000 documents. This also contained some terms and Wikipedia pages which were deprecated, so we ignored them when training on our corpus.

2.3 Evaluation metrics

The evaluation metrics of a system depends on the use-case for the specific system. Our system focuses on the balance between speed and performance of the system. We use precision and recall parameters to judge the quality of our results. The other parameter which we used was the speed in fetching the responses of our queries.

3 TEXT SUMMARIZER

In Natural Language Processing (NLP), text summarization has a huge impact on human life. In the present era where we can access a lot of papers, e-books and articles online, impact of text summarizers in publishing industry, investing time to read an article mindfully is not an option, given the time constraints[2]. Furthermore, with the rising number of articles being produced and the digitization of printed media articles, keeping track of the growing number of web articles has become extremely difficult. This is why we need text summarization as it aids in shortening lengthy texts.

Text summarization in NLP is the process of creating summaries from large volumes of data while maintaining significant informational elements and content value. The language of the summary should be concise and straightforward so that it conveys the meaning to the reader. As per Statistics, by 2025, the total amount of data created, recorded, copied, and consumed worldwide is expected to exceed 180 zettabytes. Most of this text data needs to be minimized to more straightforward, concise summaries containing essential details to browse and analyze them more easily. There is a high demand for machine learning algorithms that can quickly summarize lengthy texts and offer accurate insights. This is precisely where text summarization comes into the picture.

Text summarization is beneficial for several NLP tasks, including text classification, legal text summaries, news summaries, generating headlines, etc. Let us further look into the key reasons behind the growing demand for Text Summarization. Text Summarization is beneficial in the following scenarios:

- Shorter reading time.
- Provides effective search results and speeds up search engines.
- Manual summarization methods can generate biases whereas automatic summarization tools are much faster and efficient.
- Business companies can enhance user experience and can enhance the volume of texts they can handle.

There are two types of text summarization methods - Extractive summarization and abstractive summarization.

3.1 Extractive Summarization

In extractive summarization, we extract the essential and important words from the document. Combine these words to produce a meaningful summary. Extractive summarization use scoring functions to rank the relevancy of phrases or words[2]. This scoring function also considers to keep the meaning of the document in-place without changing the meaning of the document drastically. LexRank, Luhn, LSA etc are some algorithms implemented using python libraries Gensim or Sumy for extractive summarization.

Here's an example of how extractive text summarization works-

Original text - Eric and Walt attended a college party in San Francisco around 9pm. They had fun at the party and lost track of time. They could not return back home and had to stay over.

Summarized text - Eric and Walt attend party San Francisco Lost Track of Time Stayed over.

3.2 Abstractive Summarization

Abstarctive summarization focuses on the crucial information in the original text and creates a new set of sentences for the summary. This new sentence will always not be the part of source text, It is altered. Abstractive summarization is completely different from extractive summarization. Extractive summarization generates summary based on the original text whereas abstractive summarization works well with deep learning models like the seq2seq model, LSTM,

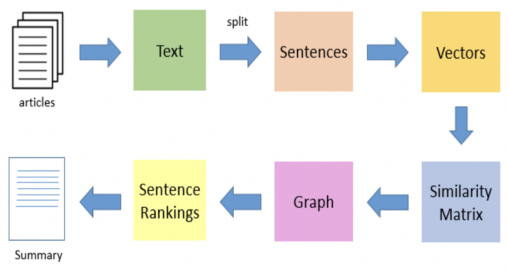


Figure 2: TextRank approach

etc., along with popular Python packages (Spacy, NLTK, etc.) and frameworks (Tensorflow, Keras)[2].

Here's an example of how abstractive text summarization works-

Original text- Eric and Walt attended a college party in San Francisco around 9pm. They had fun at the party and lost track of time. They could not return back home and had to stay over.

Summarized text- Eric and Walt attended party in San Francisco, they lost track of time and stayed over.

3.3 Our Implementation

We use Extractive Summarization algorithm called as TextRank[3]. TextRank is a similar algorithm to pagerank but instead of pages, we rank sentences in documents. It is a graph-based ranking model for text processing which can be used in order to find the most relevant sentences in text and also to find keywords. (Fig 2).

- The first step involved fetching the text contained for each document from the dataset.
- In the next step, we find vector representation (word embeddings) for each and every sentence in the document. For creating a network of vectors, we used the pre-trained model of Wikipedia 2014 + Gigaword 5 from GloVe (Global Vectors of Word Representation)
- Then we find similarities between sentence vectors using cosine similarity and are stored in a matrix.
- The similarity matrix is then converted into a graph, with sentences as vertices and similarity scores as edges, for sentence rank calculation.
- Finally, a certain number of top-ranked sentences(in our case 5) form the final summary for each document.

4 SEARCH ENGINE

Our search engine is built using Elasticsearch. Elasticsearch is a distributed search and analytics engine. Elasticsearch is built on Apache Lucene. Elasticsearch is a document oriented database, and in the present day several companies such as Slack, Uber, Udemy, Instacart and so on use Elasticsearch as search engine and analytics engine. Elasticsearch uses Okapi BM25 as their similarity scoring function. BM25 similarity function is chosen over TF-IDF scoring. BM25 is more than a term scoring method. It scores the documents with relation to query. Whereas TF-IDF is a term scoring method and is incorporated in a document scoring method using a similarity measure like cosine similarity. BM25 is more robust than TF-IDF BM25 has its own version of TF-IDF in its underlying algorithm. Okapi BM25 takes into consideration the document length and term frequency saturation. Words that appear 5-6 times in a document have more value than the words that appear once or twice, but the terms that appear 20 times have almost the same value as the terms that appear 100 times or more. That is BM25 takes into account the most commonly used words and normalizes its score. So even if the stop words are not removed, they are taken care of by Okapi BM25. Some search algorithms that we use are explained below.

4.1 Function Score

The function_score allows us to modify the score of documents that are retrieved by a query[4]. This is useful if, for example, a score function is computationally expensive and it is sufficient to compute the score on a filtered set of documents. We can define one or more functions that compute a new score for each document returned by the query. We can weigh the query parameters differently, for example while searching, if the text is present in title, we weigh that document more than the text present in the description part of the document. Function score has score_mode parameters in which we can specify how the scores need to be calculated. Several functions such as sigmoid, logarithmic, mean, multiply are available to combine the scores of the documents.

4.2 Fuzzy Search

Fuzzy query returns the documents that contain similar terms as compared to search query text[5]. The similarity is compared by using the Levenshtein edit distance. The Levenshtein distance is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. Edit distance is the number of characters needed to change one term to another. The change in character maybe[5]:

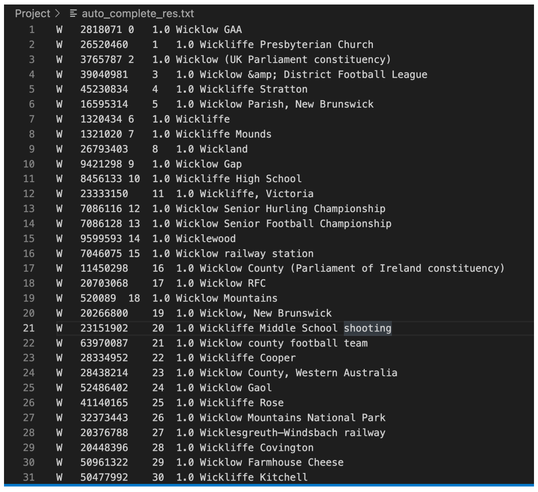
- Changing a character (box → fox)
- Removing a character (black → lack)
- Inserting a character (sic → sick)
- Transposing two adjacent characters (act → cat)

To find similar terms, the fuzzy query creates a set of all possible variations, or expansions, of the search term within a specified edit distance. The query then returns exact matches for each expansion.

4.3 Intervals Search

The intervals query uses a set of matching rules. These rules are then applied to terms from a specified field[6]. These intervals can be further combined and filtered by parent sources. The matching rules offered by elastic are fuzzy rules, match rules, wildcard rules, all of rules and any of rules. These rules are applied on the field we want to perform our search. The wildcard rule matches terms using a wildcard pattern. Some wildcard patterns are *, ?, ^.

This pattern can expand to match at most 128 terms in Elasticsearch. The all_of rule returns matches that span a combination of other rules[6]. The any_of rule returns intervals produced by any of its sub-rules. We use fuzzy rules that return results within the edit distance and wild card rules to match the query with a wildcard pattern inside all of rule.



```
W   2818071 0   1.0 Wicklow GAA
W   26520460    1   1.0 Wickliffe Presbyterian Church
W   3765787 2   1.0 Wicklow (UK Parliament constituency)
W   39040981    3   1.0 Wicklow &amp; District Football League
W   45230834    4   1.0 Wickliffe Stratton
W   16595314    5   1.0 Wicklow Parish, New Brunswick
W   1320434 6   1.0 Wickliffe
W   1321020 7   1.0 Wickliffe Mounds
W   26793403    8   1.0 Wickland
W   9421298 9   1.0 Wicklow Gap
W   8456133 10  1.0 Wickliffe High School
W   23333150    11  1.0 Wickliffe, Victoria
W   7086116 12  1.0 Wicklow Senior Hurling Championship
W   7086128 13  1.0 Wicklow Senior Football Championship
W   9599593 14  1.0 Wicklewood
W   7046075 15  1.0 Wicklow railway station
W   11450298    16  1.0 Wicklow County (Parliament of Ireland constituency)
W   20703068    17  1.0 Wicklow RFC
W   520089  18  1.0 Wicklow Mountains
W   20266800    19  1.0 Wicklow, New Brunswick
W   23151902    20  1.0 Wickliffe Middle School shooting
W   63970087    21  1.0 Wicklow county football team
W   28334952    22  1.0 Wickliffe Cooper
W   28438214    23  1.0 Wicklow County, Western Australia
W   52486402    24  1.0 Wicklow Gaol
W   41140165    25  1.0 Wickliffe Rose
W   32373443    26  1.0 Wicklow Mountains National Park
W   20376788    27  1.0 Wicklesgreuth–Windsbach railway
W   20448396    28  1.0 Wickliffe Covington
W   50961322    29  1.0 Wicklow Farmhouse Cheese
W   50477992    30  1.0 Wickliffe Kitchell
```

Figure 3: Auto-complete search results for query "Wickyl"

4.4 Additional Feature

Auto-complete feature:

Elasticsearch offers full text searches. So we have added an auto-complete feature for our search engine to enable searches for partial search queries as well. We use a custom analyzer for documents as well as queries. The analyzer tokenizes the text as an edge_ngrams with a minimum_ngram is 2 and maximum edge_gram is 7. The edge n-gram tokenizer first breaks text down into words whenever it encounters one of a list of specified characters, then it emits N-grams of each word where the start of the N-gram is anchored to the beginning of the word. Example of an edge n gram tokenizer for “Hello world” looks like this. [H, He, Hel, Hell, Hello, w, wo, wor, worl, word]. So using an edge n gram tokenizer by specifying minimum and maximum n grams, we build an auto-complete feature for queries. Here is an example of auto-complete search in our project. We make a search using the partial text “Wickyl” as our query for title. We receive these search results. As you can see the results contain auto-complete and fuzzy search responses. Fig 3 shows the results for query "Wickyl".

5 EXPERIMENTAL RESULTS

5.1 Precision

The precision of a system is defined as the ability of the engine to retrieve relevant documents. It os more formally given as the fraction of documents which are relevant from the overall retrieved documents. We used the Wikipedia API retrieved documents for the search query to form the set of relevant documents. We limited the search to 30 documents from the Wikipedia API, ie. we compared our results to the top 30 relevant documents. Then we compared the documetns from our respective indexes of summarized data-set and full document data-set to find the precision of the two systems. As seen in Fig 4 and Fig 5 5 4, the precision of the summarized queries for Functional score algorithm falls by a factor of around 30% as

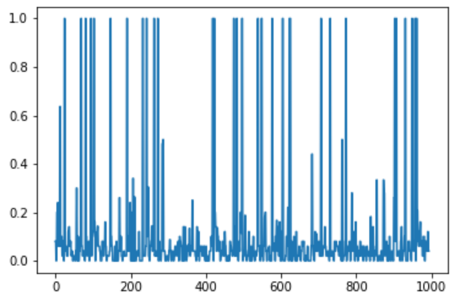


Figure 4: Functional score algorithm on whole document data-set

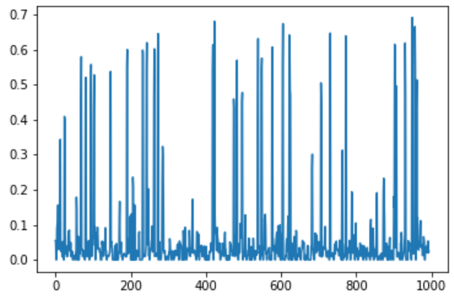


Figure 5: Functional score algorithm on Summarized data-set

compared to the full size document as we lose information when summarizing the documents. The average document size in the Wikipedia dataset is aroound 20-25 sentences. We have summarized the document to 5 sentences per document, which means we might lose valuable search information.

Similarly, for fuzzy logic queries 6 7, we see a drop in precision by around 40% for the queries on summarized search engine due to the loss of the data in the documents. But the summarized search engine still shows potential to make some changes in the summarization algorithm(we plan to use auto-encoder decoder techniques) for better results. We plan to check the results for the text summary of size 10 rather than 5 to test the robustness of the search.

5.2 Speed of execution

Here are the run-time results of all the search algorithms for original data and summarized data.

- function_score search:
  For original text in fig 8, the search time or program run-time is 14 ms. But when we perform same search algorithm on summarized text, see fig 9. We observe that there is a drastic improvement in performance.

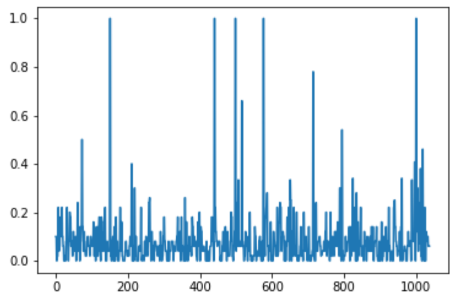


Figure 6: Fuzzy logic algorithm on whole document data-set

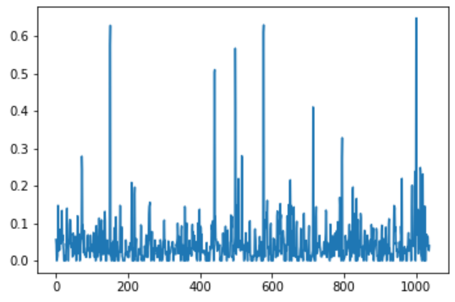


Figure 7: Fuzzy logic algorithm on Summarized data-set

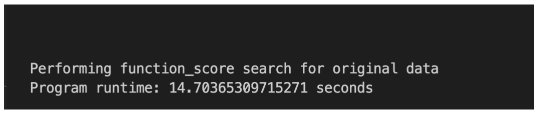

```
Performing function_score search for original data
Program runtime: 14.70365309715271 seconds
```

Figure 8: Function_score for original data

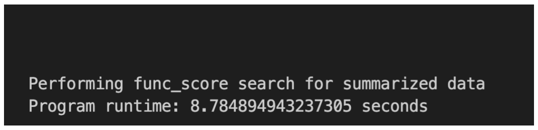
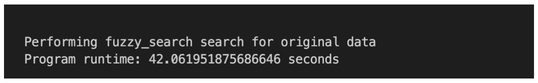

```
Performing func_score search for summarized data
Program runtime: 8.784894943237305 seconds
```

Figure 9: Function_score for summarized data

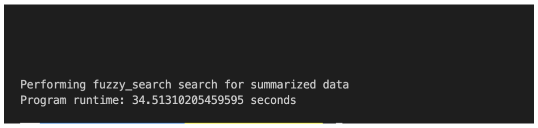
- fuzzy_search:
  The above results is same even for fuzzy_query. There performance of the search engine in terms of speed is positive. This can be observed in terms of fig 10 and fig 11

- intervals_query search:
  But for the intervals_query, the performance is increased in summarized data compared to original data. This can be observed in fig 12 and fig 13



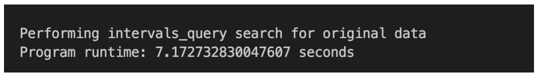
```
Performing fuzzy_search search for original data
Program runtime: 42.061951875686646 seconds
```

Figure 10: Fuzzy_search for original data



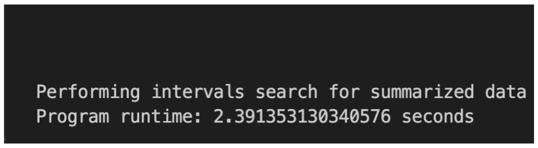
```
Performing fuzzy_search search for summarized data
Program runtime: 34.51310205459595 seconds
```

Figure 11: Fuzzy_search for summarized data



```
Performing intervals_query search for original data
Program runtime: 7.172732830047607 seconds
```

Figure 12: Intervals_query search for original data



```
Performing intervals search for summarized data
Program runtime: 2.391353130340576 seconds
```

Figure 13: Intervals_query search for summarized data

6 CONCLUSION

In terms of speed, all the 3 algorithms show improvement in speed. So search engine that uses summarized text to index the data will gain considerable speed. The precision of the system decreases for the summarized documents but we think it shows enough evidence to form a well trained model to create a robust summary for 5-10 sentences each. The precision value falls by a value of 30% and 40% for the functional score and fuzzy logic algorithm but further studies are required to check the performance of such a system on the well-trained summarizer.

7 FUTURE WORK

- Extend the search engine implementation to get feedback. This feedback maybe in terms of pseudo relevant feedback.
- Implement machine learning algorithms for ranking the documents.
- Index more documents and expand the dataset.
- Get user feedback on our search engine by conducting a survey.
- Extend our summarizer to use auto encoder and decoder.
- Implement summarizer using deep learning model.

REFERENCES

[1] python wikipedia. Python wikipedia.
[2] ProjectPro. Text summarizer intro.
[3] textRank analytics. Textrank.
[4] Elasticsearch. Elasticsearch function score.
[5] Elasticsearch. Elasticsearch fuzzy search.
[6] Elasticsearch. Elasticsearch fuzzy search.