Using LSH to Find Similar Documents

Team 6

Project Report

By

Akshayan Raviraaj
Armyben Patel
Kashyap Manishkumar Tamakuwala
Moxank Patel

12/01/2022

Instructor: Gheorghi Guzun

Project Description

With data growing at an exponential rate, the need for large-scale data comparison is becoming more and more frequent in today’s industry. The similarity between multiple documents can be monitored in several areas, such as academics, where plagiarism is being checked, or the hiring process, where matching resumes and cover letters with job descriptions helps pool the best candidates. Moreover, companies have a greater need to detect similar documents in their databases for various reasons, such as detecting duplicates when dealing with legal documents, recommending similar books and articles to users, and so on. One more sector that is keen on this kind of application is news agencies which use it to identify whether the series of articles is about the same event.

To achieve this, any search engine must be fast to return results quickly and efficiently. The naive method of searching through documents requires a lot of computation and time. So we propose using LSH (Locality Sensitive Hashing) algorithm to compare and search similar documents in a large corpus.

Other than LSH, we used Key Based approach for the search. In this method, we use the document’s themes to produce several similar documents. The dataset itself gives the keys. The keys are nothing but a set of words that describes the document. In brief, with this approach, we construct a vector representing the document and then find similarities between vectors and keywords in the search query. It then returns the top K results from the sorted result.

We have built a search engine that utilizes LSH to search for documents using key phrases entered and to retrieve the records using these words and key phrases. We have collected several data from different sources that most prominently included repositories of various research works. Consider a research search place where a user is trying to find similar research with his keyword search out of millions of published works. This is similar to that where we are analyzing the power of LSH in contrast to searching through all documentation. When returning, we must ensure that the documents are similar to what the user expects. For each query, we have grouped documents that contain a similar word.

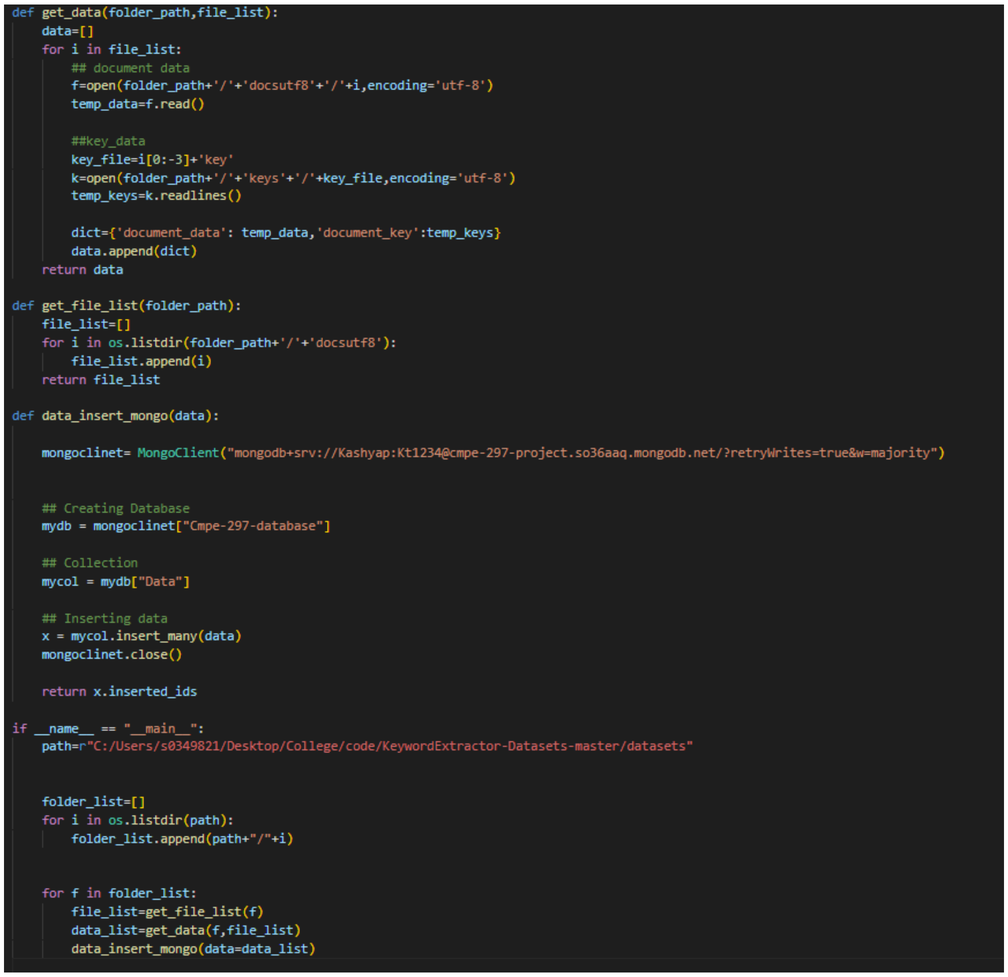
Additionally, we have utilized similarity measures to retrieve documents when they are identical to the keyphrase entered. Also, we have implemented a naive method that iterates over all the documents and retrieves them based on the similarity index. Toward the end, we performed a comparative analysis between the two processes. We have utilized the NoSQL database platform to store a high volume of data. With the implementation, we can figure out the necessity and power of hashing techniques that help us in our daily lives. The same can be thought of as a starter to show the volume of data we would face in several scenarios and the types of processing we need when facing such data.

Dataset

We have utilized several different datasets in this project. The following table gives an overview of the data sources used for the project.

Dataset Name	Link to the dataset	Description
Neural Information Processing Systems (NIPS) Papers	https://www.kaggle.com/datasets/benhamner/nips-papers?select=papers.csv	This dataset includes information such as title, authors, abstracts, and extracted text for all NIPS papers from the 1987-2016 conference. The dataset is available in two formats - CSV files and an SQLite database
Automatic Keyphrase Extraction	https://github.com/LIAAD/Kkeyword extractor-Datasets	This repository contains about 20 annotated datasets from different domains like agriculture, technology, and so on. We plan to use all these datasets by merging them into one.

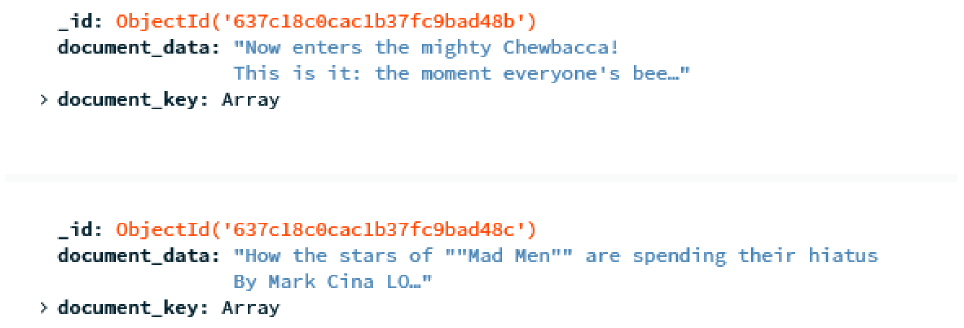
We cleaned and preprocessed all the datasets available in the above-mentioned two data sources in order to merge them. This collection of documents, approximately 14000 records, was stored as one database on MongoDB. Figure 1 shows the code block that was written to get this collection of documents.



```python
def get_data(folder_path,file_list):
    data=[]
    for i in file_list:
        ## document data
        f=open(folder_path+'/'+'docsutf8'+'/'+i,encoding='utf-8')
        temp_data=f.read()

        ##key_data
        key_file=i[0:-3]+'key'
        k=open(folder_path+'/'+'keys'+'/'+key_file,encoding='utf-8')
        temp_keys=k.readlines()

        dict={'document_data': temp_data,'document_key':temp_keys}
        data.append(dict)
    return data

def get_file_list(folder_path):
    file_list=[]
    for i in os.listdir(folder_path+'/'+'docsutf8'):
        file_list.append(i)
    return file_list

def data_insert_mongo(data):

    mongoclinet= MongoClient("mongodb+srv://Kashyap:Kt1234@cmpe-297-project.so36aaq.mongodb.net/?retryWrites=true&w=majority")


    ## Creating Database
    mydb = mongoclinet["Cmpe-297-database"]

    ## Collection
    mycol = mydb["Data"]

    ## Inserting data
    x = mycol.insert_many(data)
    mongoclinet.close()

    return x.inserted_ids

if __name__ == "__main__":
    path=r"C:/Users/s0349821/Desktop/College/code/KeywordExtractor-Datasets-master/datasets"


    folder_list=[]
    for i in os.listdir(path):
        folder_list.append(path+"/"+i)


    for f in folder_list:
        file_list=get_file_list(f)
        data_list=get_data(f,file_list)
        data_insert_mongo(data=data_list)
```

Figure 1: Code for merging datasets and storing data on a MongoDB cluster.

A few examples of how the data looks are shown in Figure 2. Each collection includes two attributes: document key, which holds a list of words connected to the document, and document data, which represents the data.



```
  _id: ObjectId('637c18c0cac1b37fc9bad48b')
  document_data: "Now enters the mighty Chewbacca!
                 This is it: the moment everyone's bee…"
> document_key: Array

  _id: ObjectId('637c18c0cac1b37fc9bad48c')
  document_data: "How the stars of ""Mad Men"" are spending their hiatus
                 By Mark Cina LO…"
> document_key: Array
```

Figure 2: Overview of data

Workflow

The project consists of four phases as illustrated in Figure 3. As discussed above, our database is stored in MongoDB represented as a collection of documents. The data is processed to clean up any unwanted characters. This process helps us reduce the load. The documents are then divided into three shingles. We have obtained nearly 10^8 shingles from the total number of documents. To reduce the computation expense, we have selected 10^4 unique features to analyze. After obtaining unique features the proposed system uses cosine similarity to see which documents are closer to the search query. Once we find the list, we order them according to the scores. All the above-mentioned processes have to be efficient in terms of time. Once the query comes in our proposed system gives the results within seconds (Results are discussed further).

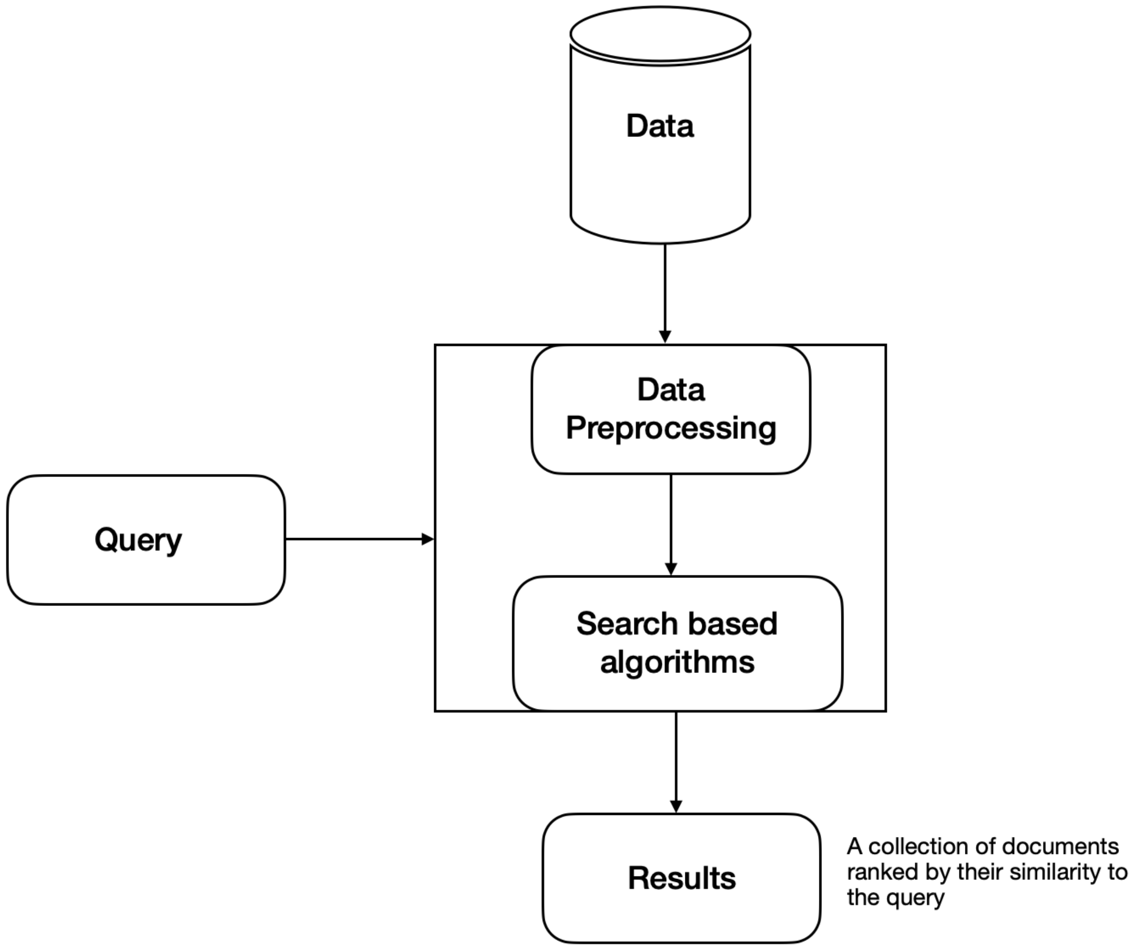
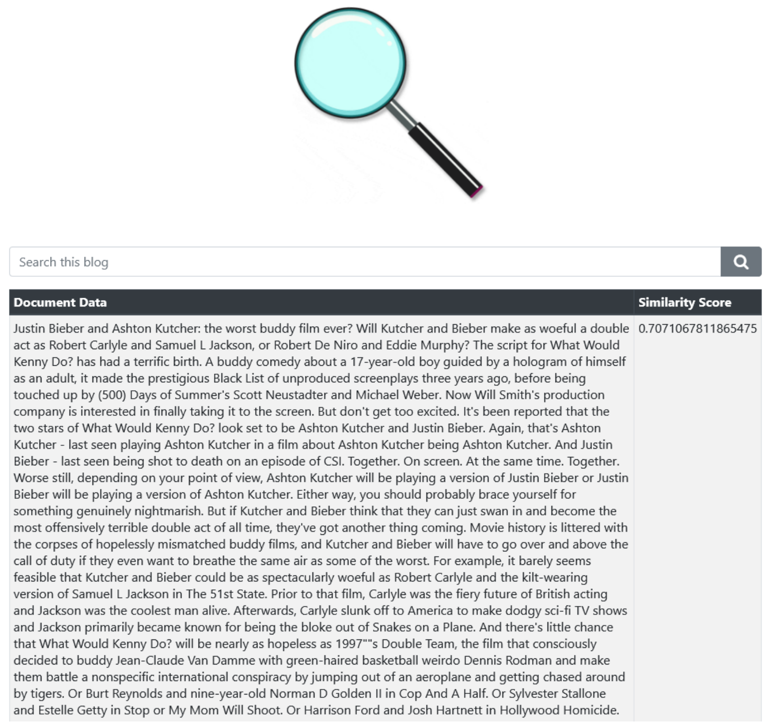

Data
Data Preprocessing
Query
Search based algorithms
Results
A collection of documents ranked by their similarity to the query

Figure 3. Workflow of the system

The user enters the desired keyword to search as shown in Figure 4. The keywords are sent to the system. The system undergoes the above-discussed process and displays the result to the user through our interactive UI as shown in Figure 5.

Figure 4. Front page of our UI



Document Data	Similarity Score
Justin Bieber and Ashton Kutcher: the worst buddy film ever? Will Kutcher and Bieber make as woeful a double act as Robert Carlyle and Samuel L Jackson, or Robert De Niro and Eddie Murphy? The script for What Would Kenny Do? has had a terrific birth. A buddy comedy about a 17-year-old boy guided by a hologram of himself as an adult, it made the prestigious Black List of unproduced screenplays three years ago, before being touched up by (500) Days of Summer's Scott Neustadter and Michael Weber. Now Will Smith's production company is interested in finally taking it to the screen. But don't get too excited. It's been reported that the two stars of What Would Kenny Do? look set to be Ashton Kutcher and Justin Bieber. Again, that's Ashton Kutcher - last seen playing Ashton Kutcher in a film about Ashton Kutcher being Ashton Kutcher. And Justin Bieber - last seen being shot to death on an episode of CSI. Together. On screen. At the same time. Together. Worse still, depending on your point of view, Ashton Kutcher will be playing a version of Justin Bieber or Justin Bieber will be playing a version of Ashton Kutcher. Either way, you should probably brace yourself for something genuinely nightmarish. But if Kutcher and Bieber think that they can just swan in and become the most offensively terrible double act of all time, they've got another thing coming. Movie history is littered with the corpses of hopelessly mismatched buddy films, and Kutcher and Bieber will have to go over and above the call of duty if they even want to breathe the same air as some of the worst. For example, it barely seems feasible that Kutcher and Bieber could be as spectacularly woeful as Robert Carlyle and the kilt-wearing version of Samuel L Jackson in The 51st State. Prior to that film, Carlyle was the fiery future of British acting and Jackson was the coolest man alive. Afterwards, Carlyle slunk off to America to make dodgy sci-fi TV shows and Jackson primarily became known for being the bloke out of Snakes on a Plane. And there's little chance that What Would Kenny Do? will be nearly as hopeless as 1997""s Double Team, the film that consciously decided to buddy Jean-Claude Van Damme with green-haired basketball weirdo Dennis Rodman and make them battle a nonspecific international conspiracy by jumping out of an aeroplane and getting chased around by tigers. Or Burt Reynolds and nine-year-old Norman D Golden II in Cop And A Half. Or Sylvester Stallone and Estelle Getty in Stop or My Mom Will Shoot. Or Harrison Ford and Josh Hartnett in Hollywood Homicide.	0.7071067811865475

Figure 5. Results displayed in the web application

Tools and algorithms used

Our system has been developed using Python whereas we have utilized Flask to develop the web application. The data used in this project is represented as a NumPy array as they are faster in performing mathematical computation. Furthermore to store the data we have used MongoDB.

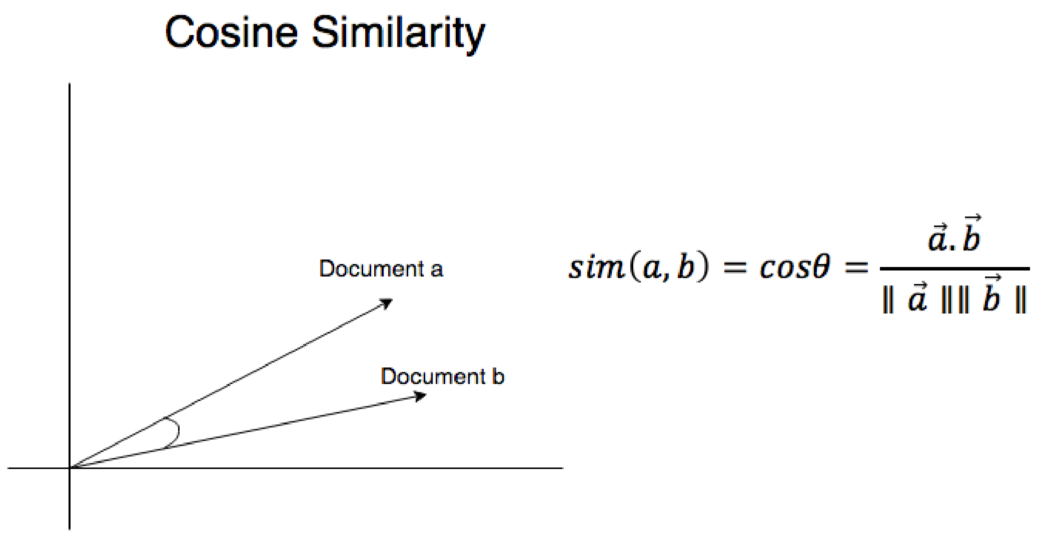
We have used various algorithms and data structures to efficiently process the query. We used Python's dictionary to store the term frequency. We converted this dictionary into a NumPy array to compute the cosine similarity between two documents. Moreover, we have implemented a nearest-neighbor search to find all the documents. This algorithm iterates over all the documents and computes the similarity between the query and each document. It then selects the top K candidates and returns them.

Furthermore, we have employed a Key based search algorithm. In the key-based search algorithm, we try to find similar documents by using keys that are provided in the dataset itself. But here the number of features is comparatively low than the naive approach. However, the method is similar to the naive approach.

Lastly, we implemented the LSH algorithm to produce a set of candidates in which a search is performed. Here LSH produces a smaller subset of documents based on a given query. Now in a smaller subset, a naive search is performed to give K Nearest Neighbor. We also implemented an algorithm for calculating recall between two sets of documents. In this algorithm, we try to find the percentage of documents that same in the given two sets. This recall algorithm is also used to compare different algorithms. Apart from the above-mentioned tools, we have used GitHub, and Google Colab to collaborate for easy access.

Evaluation method(s)

To evaluate the algorithms we have to measure the similarity between the documents returned by the algorithm and the expected documents. We utilized cosine similarity to evaluate the algorithms. Figure 4 shows the formulation of cosine similarity and how it works in our project.



$$sim(a,b) = cos\theta = \frac{\vec{a}.\vec{b}}{\| \vec{a} \| \| \vec{b} \|}$$

Figure 6. Cosine Similarity

In addition to the cosine similarity, we have implemented accuracy as our evaluation matrix. For each document, we have a list of keywords associated with it. So to calculate the accuracy we have implemented the following formula

$$Accuracy = \frac{Total\ number\ of\ keyword\ mathced\ in\ the\ document\ and\ query}{Total\ number\ of\ keyword\ in\ query}$$

Results

Talking about the result, we used recall as a metrics to compare the result between three methods. In summary, we got the highest accuracy in Naive Search which is quite obvious because we iterated through every document, followed by Key Based Search and LSH. However, LSH took the lowest time with an acceptable margin of error. Figure 6 depicts the time taken by all three algorithms to execute n queries.

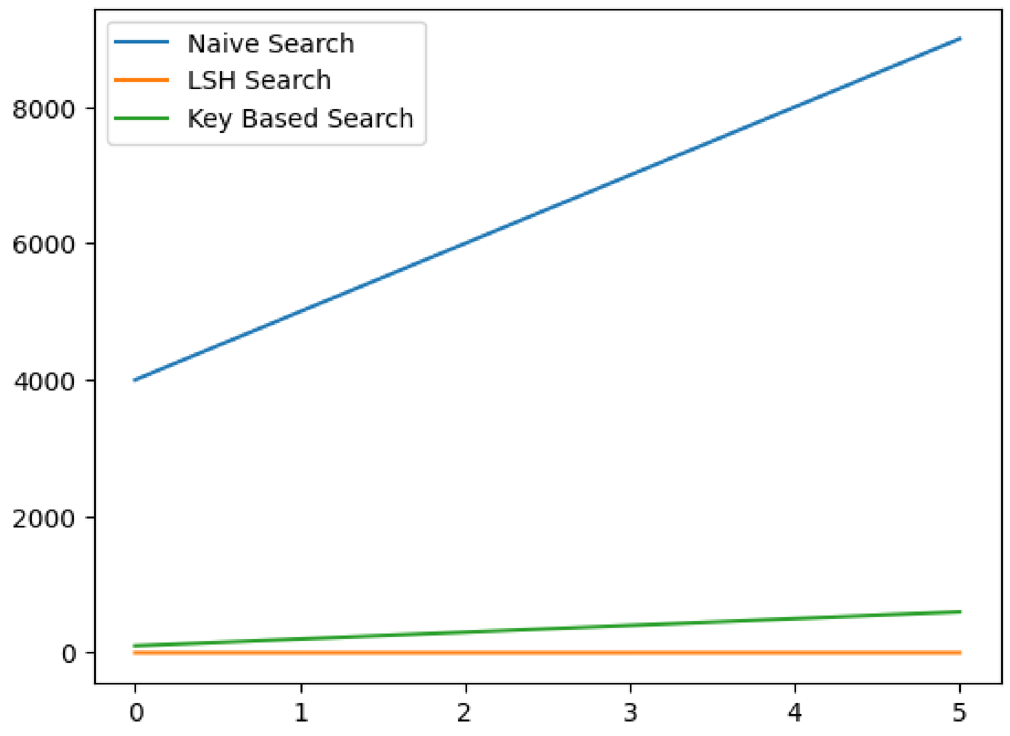


Figure 6. Time taken to execute n queries

Here, X-Axis represents the number of queries whereas the Y-Axis represents the time taken in seconds.

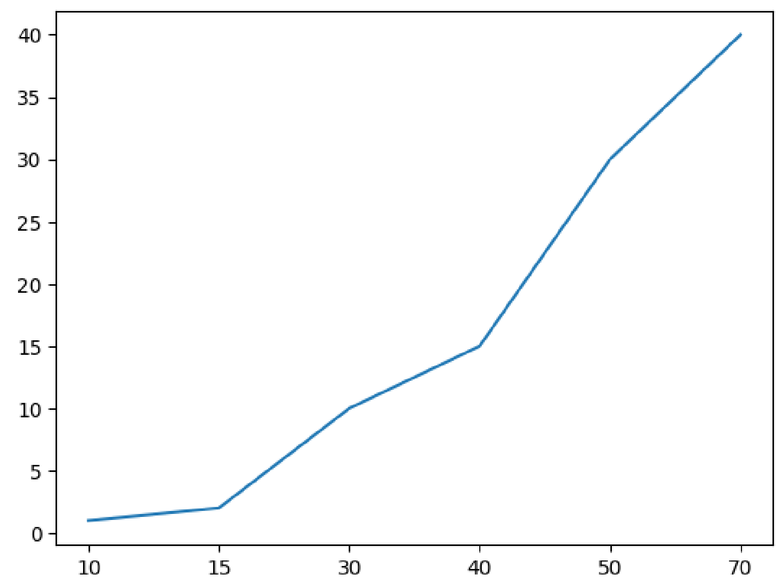


Figure 7. Percentage of intersection between different searches

Figure 7 depicts the percentage of intersection between searches produced by LSH and Naive Search. Here X-Axis represents the number of hash functions whereas the Y-Axis represents the percentage of intersection.

Lessons learned, open issues, and future work

1. Lessons learned

This project has helped us understand very complex topics like LSH and indexing the document. It has also taught us teamwork and time management skills. It has also contributed to improving our implementation skills as developers and helped us understand topics like memory constraints and algorithm processing time.

2. Open issues

Currently for LSH we are selecting random features to overcome memory issues. We would like to figure out a way to accommodate all the features in memory.

3. Future work

We would like to extend this project further by

- Implementing tiering to perform search i.e., we want to create various indexes such as indexes based on authors, titles, abstracts, and so on. These indexes can be utilized by LSH to prune the number of candidates in the search.
- Implementing an approach where we first cluster the data based on the document key into n-clusters and then for each cluster implement a local LSH to search documents in that cluster.
- Furthermore to improve the accuracy of LSH we would like to consider a full set of features instead of a subset of features. And to reduce the features set instead of randomly dropping them we would reduce the dimensionality.

Systems requirements to install and run the project

To run the application on your system successfully, you need to follow the following steps:

Step - 1 → Cloning/Downloading the Project

The first step toward the setup is to clone our repository. For cloning the project you can directly download the zip file from the

https://github.com/KashyapTamakuwala/Cmpe-297-LSH-Project

or if you have git installed you can use the command

“https://github.com/KashyapTamakuwala/Cmpe-297-LSH-Project.git”

Step - 2 → Installing necessary libraries

After cloning the project traverse to the Flask_Application folder in the downloaded folder, open command prompt in the folder and type the following command

$$pip\ install\ \ -r\,“requirement.txt”$$

This will install all the necessary libraries required to run the project.

Step - 3 → Running The application

After installing all the necessary libraries, to run the application, type the following command

$$python\ app.py$$

This command will start the flask server and will initialize all the environment variables and objects.

To visit and access the application enter the address $127.0.0.1$ in your browser.

GitHub Repository

Our project can be found on GitHub through this link