IR Assignment 2 – Design Document

Locality Sensitive Hashing

Group Members

Aditya Agarwal : 2017B1A71075H

Jalaj Bansal : 2017B3A71610H

Architecture of the application

We have implemented the Locality Sensitive Hashing technique in python, utilizing multiple built-in functions, data structures and third-party libraries (pandas, NumPy, tkinter) in order to build a modularized program with a minimal graphic user interface. We have used the DNA dataset (human data, https://www.kaggle.com/thomasnelson/datasets) from Kaggle to test our program.

Here is a stepwise explanation of the entire process with the data structures used for each step (if any) -

1. **Pre-processing:** We have used pandas to read the tab separated values from the dataset txt file and converted to a pandas dataframe. Later, we have converted to a list for easy access. We have stored total number of documents/files in the "Total_Files" variable. Data structures used – List, Pandas dataframe.

2. **Shingling:** Each subsequence of length k in the document is defined as a k-shingle. The combined text in all documents is used to form k-shingles, a shingle size of around 8-10 is chosen for research purposes. Size of shingles is taken as input. With these shingles, we build the incidence matrix. Data structures used – List, Pandas dataframe.

3. **Minhashing:** It is applied on the shingles matrix to generate a signature matrix. Minhashing compresses the document vectors to give signatures that have lesser number of rows. We generate random permutations (Number of permutations is taken as input) of indices and then for each permutation, we generate the signature. The first occurrence (index) of a '1' in the shingle matrix for each permutation of indices is used to generate the signature for a document. NumPy is used to accelerate the process. We earlier implemented without using NumPy by simply using multiple hash functions of the form (ax + b)%c but it was very slow with n^2 time complexity. Data structures used – List, List of Lists, Pandas dataframe.

4. **Locality Sensitive Hashing:** In this step, we divide the signature matrix into bands. The number of bands is taken as input and then rows per band is calculated as number of rows in signature matrix divided by number of bands. Each band of signatures is then hashed into buckets. Hence, more similar documents have higher probability of landing in the same bucket. Using appropriate similarity metrics, we test whether documents in the same bucket have what percentage of similarity. Data structures used – List, List of Lists, Pandas dataframe.

5. **Similarity metrics**

    a. **Cosine Similarity**: It is often used as the similarity measure when comparing documents and emails. Therefore, we have taken it as the primary similarity measure. Calculated using NumPy by applying the following formula -

    dot(signature(sim_doc),signature(query)) /

    (sum(sim_doc^2) * sum(query^2)) ^0.5

    b. **Jaccard Similarity**: It can be used as a similarity as well as a distance measure. Calculated using the following formula -

    sum(signature(query) AND signature(sim_doc)) /

    sum(signature(query) UNION signature(sim_doc))

We have used **Tkinter** to build a simple GUI based application for usage by anyone. Input boxes, buttons and an output frame were required. We have mapped the button clicks and inputs to respective functions. Locality Sensitive hashing starts sequentially upon entry of correct address for corpus and specifying Shingle Size, number of permutations (for Minhashing) and the number of bands. After completion, a message is displayed and total time taken for all the steps is displayed. Retrieval starts upon entry of query by user and the top 10 similar documents with their scores are displayed along with the time taken for retrieval.

Runtime

All steps of LSH

The most time-consuming part of Locality Sensitive Hashing turns out to be Shingling and the time taken for shingling escalates exponentially with increase in size of Shingles. For shingle size >=8, it takes a lot of time and memory, it must be run parallelly for better efficiency or run on a powerful virtual machine.
For the DNA dataset with 4,381 documents and 552,098 characters, time for LSH:

1. Shingle size = 4, Minhashing with 100 random permutations, Rows per band = 5 -
    a. Shingling – 29.78 seconds
    b. Minhashing - 11.33 seconds
    c. Latent Semantic Hashing ~ 2 seconds
    d. Total Time ~ 42 seconds
2. Shingle size = 5, Minhashing with 100 random permutations, Rows per band = 5 -
    a. Shingling – 56.21 seconds
    b. Minhashing - 15.09 seconds
    c. Latent Semantic Hashing – 3.5 seconds
    d. Total Time ~ 75 seconds
3. Shingle size = 6, Minhashing with 100 random permutations, Rows per band = 5 -
    a. Shingling – 412.51 seconds
    b. Minhashing - 71.46 seconds
    c. Latent Semantic Hashing – 5.35 seconds
    d. Total Time ~ 488 seconds
4. Shingle size = 8, Minhashing with 100 random permutations, Rows per band = 5 -
    a. Total Time > 2 hours

Retrieval and Search

Similar documents retrieval is quite fast. For any document that is present in the corpus (any DNA sequence in the dataset here) the similar docs are retrieved instantly and then calculating the similarity metrics for all of them takes roughly 1-2 seconds. For any random query DNA sequence that is not present in the dataset, we run shingling on the query and append to shingle matrix to save time on shingling the entire dataset again. The rest of the steps are followed and it takes roughly 10-15 seconds (depending on shingle size) to check if there are any similar sequences in the dataset. In most cases, random sequences do not hash to the same buckets as other sequences in the dataset, therefore, not many similar documents are retrieved for random sequence.