Design Document For

Locality Sensitive Hashing

Team:

1. Snehit Reddy (2017A7PS0868H)
2. Dhanush Karupakula (2017B3A71011H)
3. Sai Satvik Vuppala (2017B4A71449H)
4. Amit Raj Reddy D (2018B4A70813H)

Topic

Implementing Locality Sensitive Hashing to find near duplicates of a given DNA sequence.

Introduction

The task of finding exact duplicates is kind of easy. But when we think of finding some near duplicates based on a certain threshold of similarity then the task becomes difficult. Majorly when there is a large dataset, the brute force is very inefficient both in terms of space and time. This is where the Locality Sensitive Hashing Algorithm comes into picture. This algorithm does not give the exact results, but provides a good approximation when certain parameters are set through observations. LSH has a wide range of applications in the fields of near-duplicate detection (plagiarism checker), studying genome databases, image search algorithms, recommender systems, audio/video fingerprinting. In this project we use LSH Algorithm to find the near duplicates of a given input DNA Sequence.

Libraries Used

These are the python libraries used in this project :

1. **binascii:** This module is basically used to convert binary to ASCII-encoded representations and vice versa. Here we used it to convert shingles into tokens of 4 bytes.

2. **numpy:** This library is used to store the data in the form of an array and is used for easier array computations.
3. **random:** This is used to generate random numbers for various distributions. In this project we used this in the process of generating hash functions.
4. **itertools:** This module contains various functions which creates iterators for efficient implementation of loops across various data structures.
5. **time:** This was used to keep track of the amount of time that each process was taking.

Implementation

The Dataset

The dataset used in this project is the database related to human DNA sequence. There are around 4380 Sequences in the data. The data is in the text format. Each line in the text document corresponds to a DNA sequence and also a variable called class which are space separated. In this project we aim at only understanding the similarity between the DNA sequences, so we won't be considering the class attribute of the data for the further analysis.

Loading the Dataset would take around 0.8-0.9 seconds.

An instance of the data will be as follows

ACTCAGAAACCAAGAATCTGTGTCTCCTGGAAACAGCTCAATAA0

ACTCAGAAACCAAGAATCTGTGTCTCCTGGAAACAGCTCAATAA : Sequence

0 : Class

Corpus Preprocessing

The corpus which is in the text document format is first loaded. We then iterate over each line of the text document (implying each DNA sequence) and split it into two parts, namely the sequence and class. Then we store them in a list and then convert into a numpy array. This array is used for further processing of the data.

Shingling

This is the first step towards the implementation of Locality Sensitive Hashing. In this phase, we divide the DNA sequence into shingles of length k. k is user defined and here we consider k to be 6. So, we basically obtain the list of all the subsequences of length 6 in the given sequence and each sub sequence is called a shingle.

For instance, if the sequence is
ACTCAGAAACCAAGAATCTGTGTCTCCTGGAAACAGCTCAATAA

Then the shingles that we get for this particular sequence are,

ACTCAG, CTCAGA, TCAGAA, CAGAAA, AGAAAC, ... , CAATAA

In this way we get around 39 shingles each of length 6 from the given sequence. This process is repeated for each sequence. As a result we will have an enormous amount of shingles.

In order to overcome this issue, we create tokens using the binascii module to map each shingle to a unique 32 bit integer. This integer will be the face of each shingle. For example, the above shingles can be mapped like this,

ACTCAG : 1576819688 ; CTCAGA : 3011971210
Also we don't store the data in a boolean matrix but instead we store it in the form of a list of lists where each list corresponds to a particular DNA Sequence. So the final shingled data will be in this format.

```
[
    [232612, 32659, 32356, 212546, .... , 3236] ,
    [32626, 3265, 1245, 32625, 13156, ... , 215113],
    ....
]
```

Each row here corresponds to each sequence. Now the first sequence has the shingles 232612, 32659, 32356, and so on. Similarly the second sequence has the shingles 32626, 3265, 1245, 32625 and so on. This method of storing the data will save space and time complexity.

Making Shingles might take around 4.5-5 seconds

MinHashing

Now we reach the second step of the process. As we can see in the above step we achieved a state where we have a universal set of all shingles (all possible shingles from all documents). To find the similarity among these documents across this universal set using the boolean matrix is a very tedious process in terms of both time and space complexity. So this is where minhashing comes into picture, but as we already mentioned above we store the data in a particular format which helps us in this process.

1. In the first step we actually have to assign random permutation to the set of all shingles. This random permutation is achieved by hash functions.

Hash functions are of the format : **(ax + b) mod c**

a and b are random numbers generated using the random module.

c is equal to 4294967311 (a prime number which is greater than the maximum shingle value 2^32-1 in this case)

x is the shingle (token value we generated in the first step).

2. Next we find the signatures of each Document (each DNA sequence). Now the signature element with respect to each hash function would be the minimum value of the hashed shingle values of that particular document.
3. The core principle behind this technique is that across all the possible permutations we can establish the fact that, **P[h1(d1) = h1(d2)] = Sim(d1,d2).** Now since taking all the permutations is practically of no use and since we are using hash functions to achieve these permutations, we can restrict this number to a certain value (higher the number higher will be the precision) to capture the essence of the data. In this project we restricted this number to 100 hash functions.

At the end of this step we will again return with a list of lists. For example,

```
[
    [212, 326, 32, 2125, .... , 321] ,
    [3262, 326, 1245, ... , 321],
    ....
]
```

Each list here corresponds to the signature of each document or a sequence. But here the length of each list inside will be 100.

To get the signatures with 100 hash functions would take around 13.74 seconds.

Locality Sensitive Hashing

This is the final step. The basic aim of this step is that when we take two documents with their signatures, we find if they form a candidate pair or not. As we already mention above the similarity of the signatures now acts as a proxy for the similarity of the documents.

Now we have to hash the signature matrix and see if they fall in the same bucket. IF they do so we declare them as a candidate pair. Now how do we achieve this hashing again ?

The answer is band partition. We divide the signature matric into bands b with each having r such that b*r = 100. Each band now serves as a hash value for each signature and then we see if two sequences go into the same bucket for at least one band. If two sequences go into the same

bucket for at least one band then we declare them as candidate pairs.
Here the optimal values we obtained were b=5 and r=20..

At the end of this step we obtain a set of all candidate pairs. We further check if these two signatures are actually similar or if they are false positive by checking their hamming similarity.

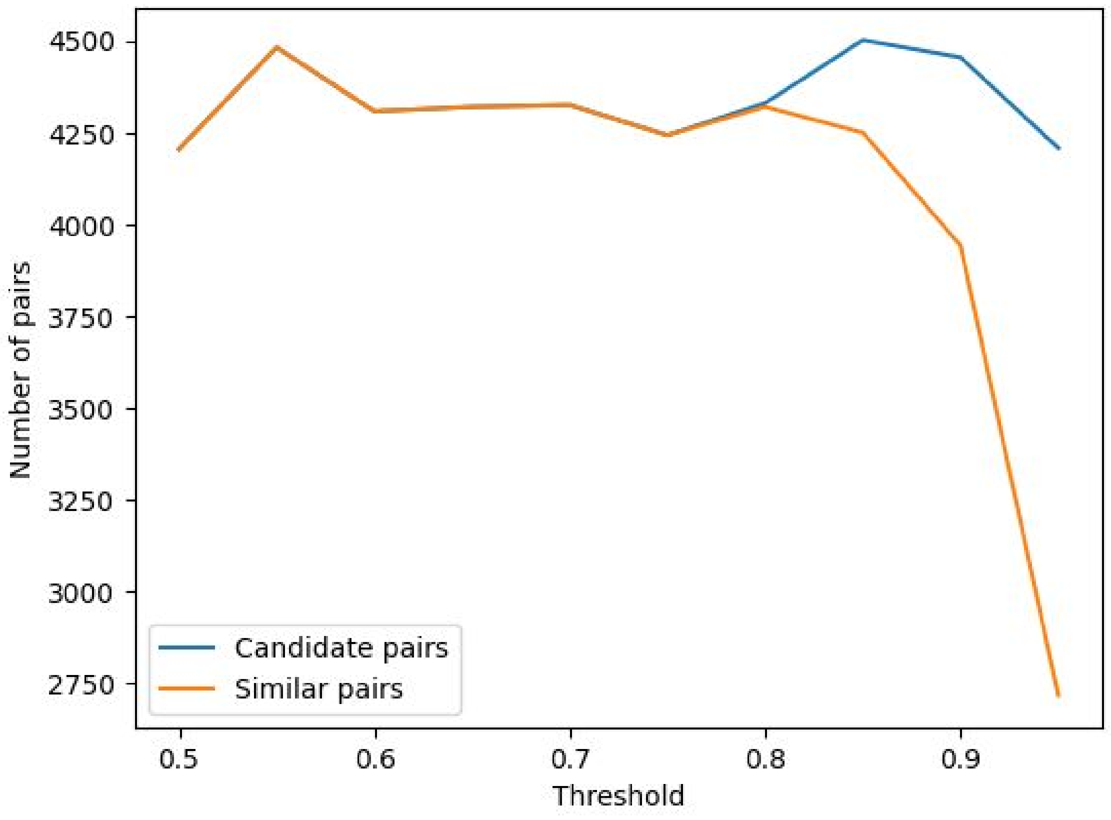
Query Processing

Now, in order to check for the similar DNA sequences for a given input. We first append the input to the shingled data set. Then we process this data which includes the query at the end through steps two and three. We finally obtain the candidate pairs. Now we are only interested in those candidate pairs which have the query. We then use the hamming similarity to find if the query and its corresponding candidate pair are similar by a certain threshold and discard the others. The threshold we have taken here is 0.9, i.e 90% similarity.

Results

Given below are a few plots which helped us determine a few parameters.
Candidate pairs: The sequences which fall in the same bucket for at least one band.
Similar Pairs: Among the candidate pairs those which have a similarity score of a certain threshold value
Precision: Ratio of number of Similar pairs to the number of Candidate pairs

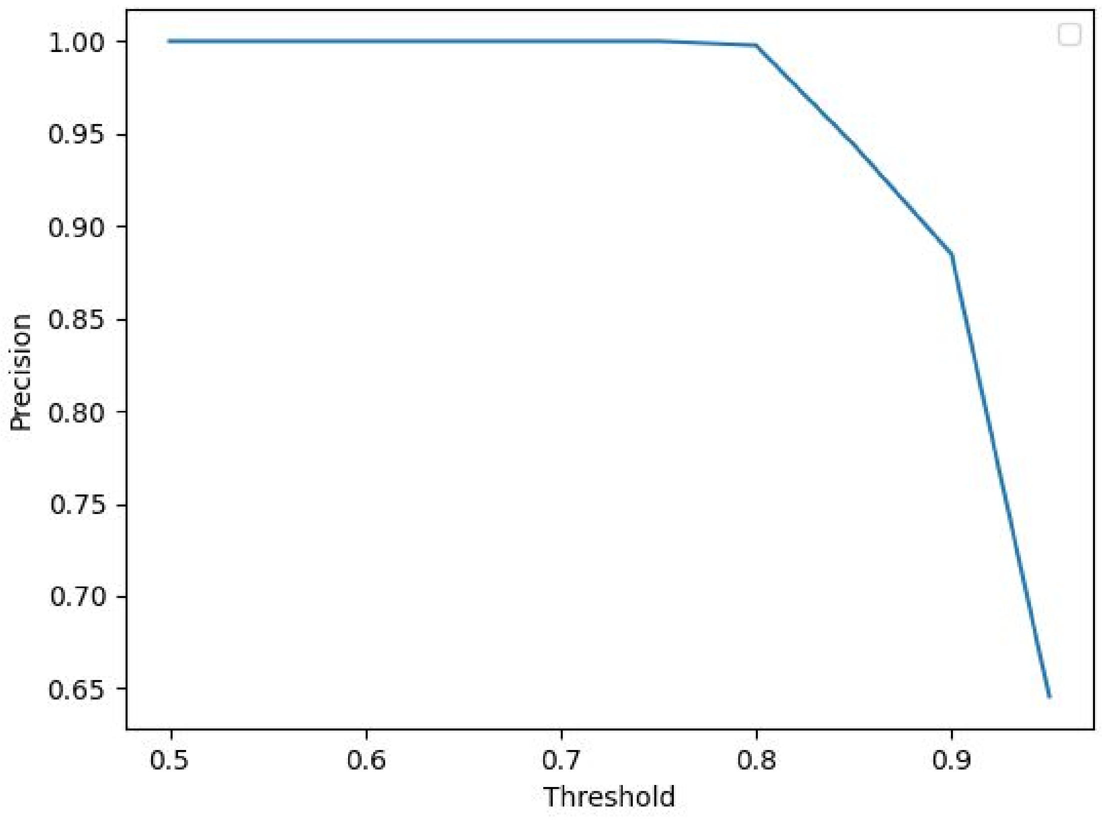
Plot 1: Number of pairs vs Threshold



Here we try to find how the number of candidate pairs and similar pairs vary with the threshold. As we might expect, for lower threshold values all the candidate pairs can be similar pairs.

From the above graph, we can see that all the sequences are at least 70-80% similar which makes all the candidate pairs, similar pairs. So as we already expected all the candidate pairs result in similar pairs when we set a lower threshold. We start to find a deviation at around after 80%. This is the reason why we chose the threshold to be 90% similarity.

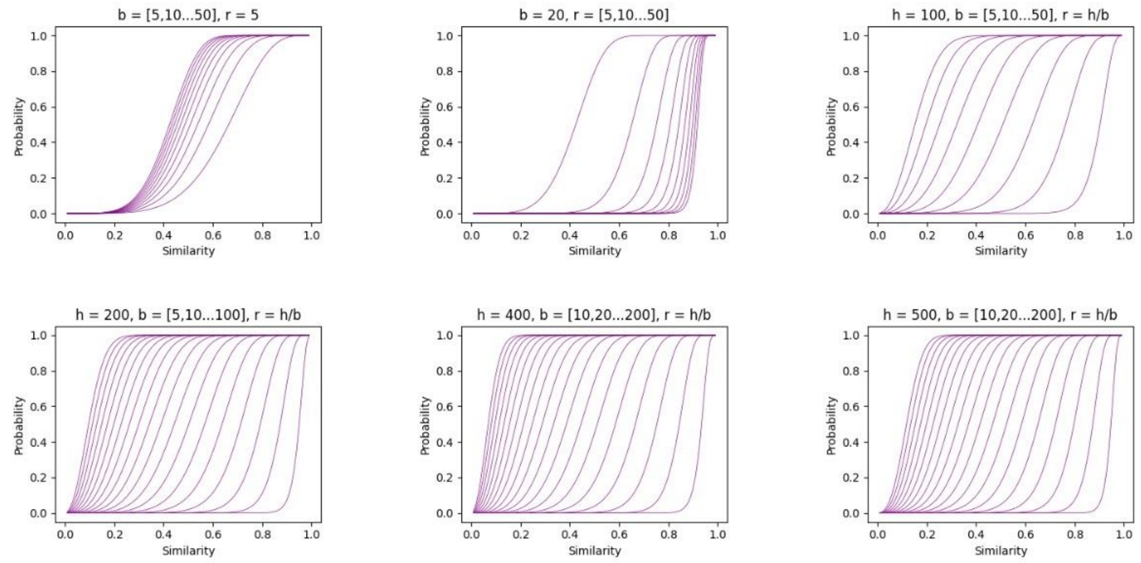
Plot 2: Precision vs Threshold



Here we try to find how the precision varies with the threshold. As we might expect, for lower threshold values the precision will be very high nearer to 1, since all the candidate pairs can be similar pairs.

From the above graph, we can observe that the precision is almost 1 for lower threshold values. The precision decreases at around 90% similarity score.

Plot 3: Probability vs Threshold with varying b,h,r

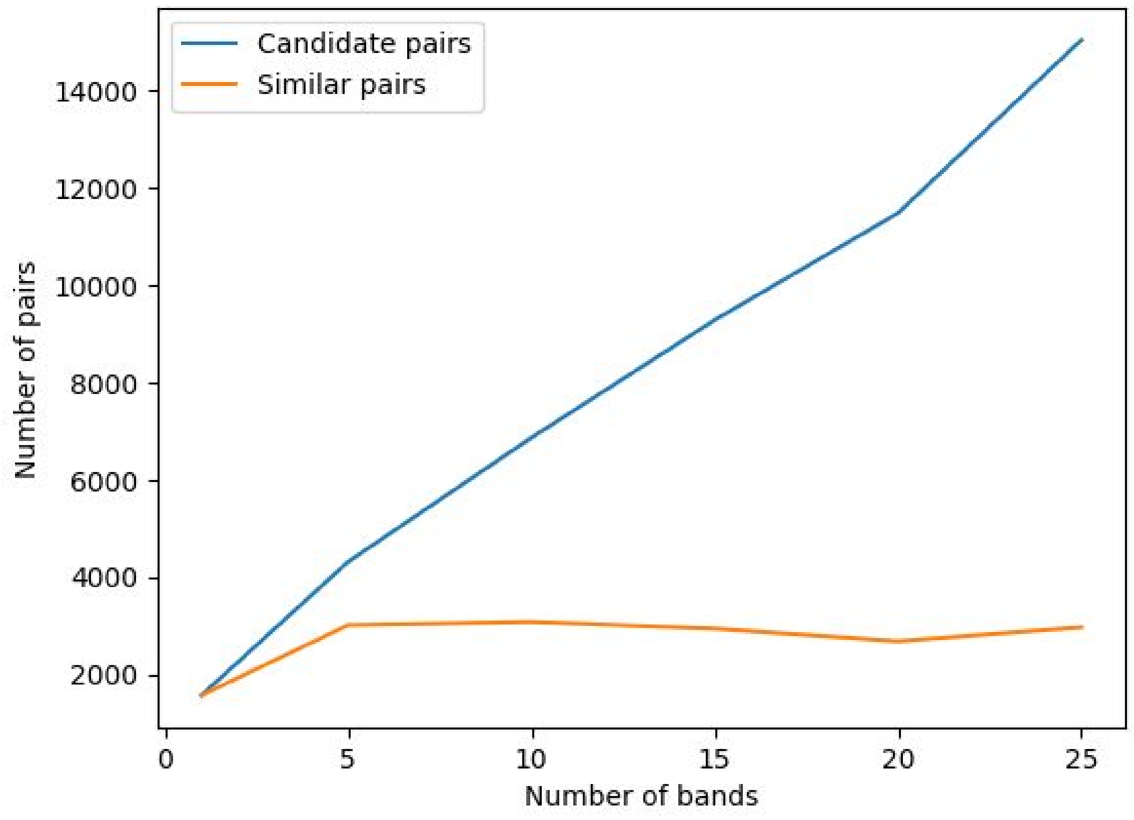


The above plots show us a way to determine the optimal values for choosing h, b and r values. Our expectations would be that, the greater the value of h, more will be the number of hash functions but this also comes with a cost. More number hash functions would lead to a greater execution time. So we have a trade off between efficiency and time.

We have observed the graphs and execution times for varying h values and we have come up with h = 100 is a better choice considering the execution time.

Now when we observe the Probability vs Similarity for h = 100, for varying b and r values, we see that at around similarity score of 0.9 the graph attains a near step function shape. The corresponding b and r values are b = 5 and r = 20.

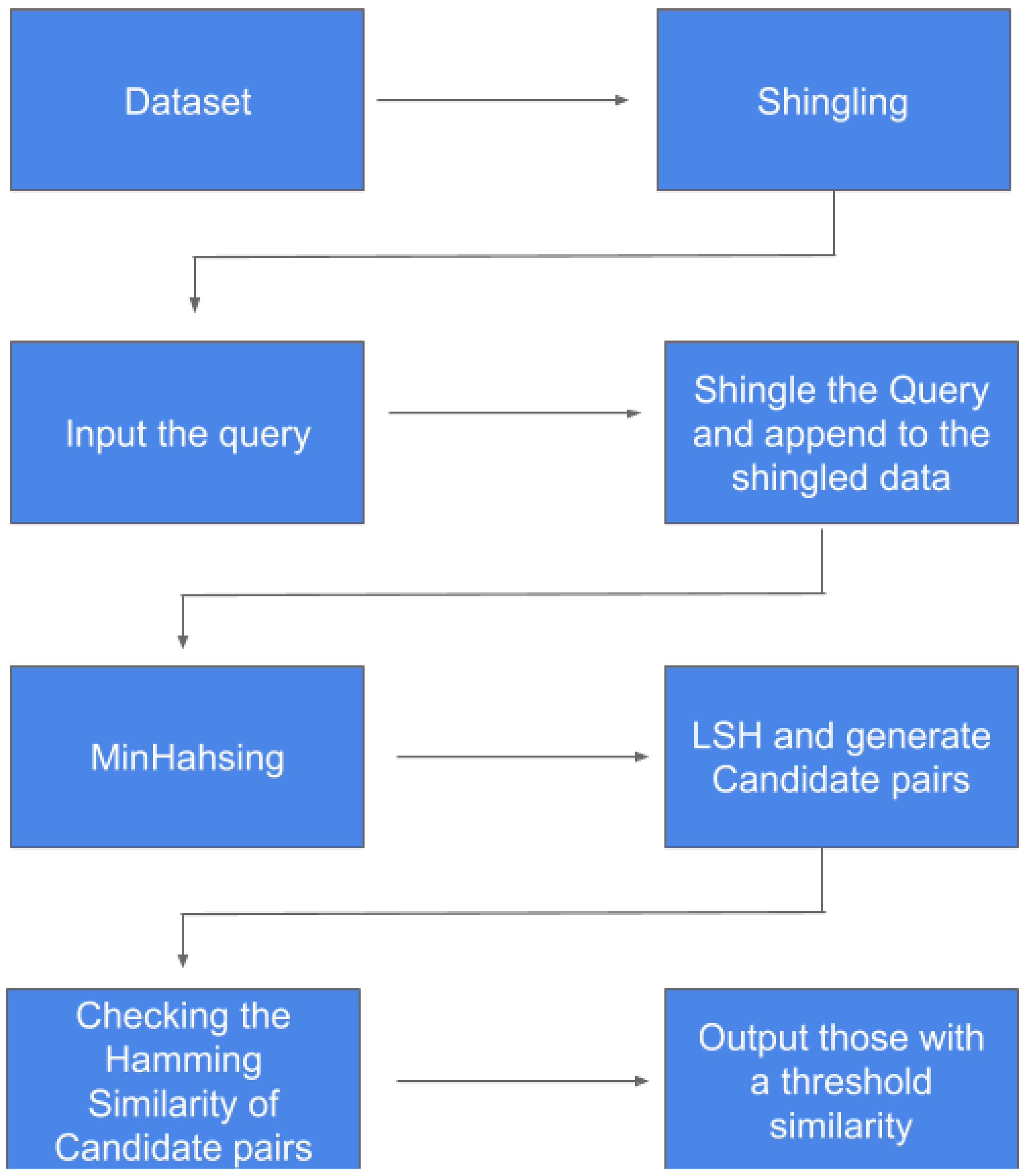
Plot 4: Number of Pairs vs bands



In this plot we try to look at how the number of pairs vary with the number of bands. As we might expect that, for more number of bands there will be a lower number of rows for each band as a result there will be more chances for sequences to fall in the same bucket forming a candidate pair.

The plot from the data also suggests the same, We can see that we get a larger number of candidate pairs as we increase the number of bands. But we also see that the number of similar pairs remain unchanged, So it’s of no use, if there is no improvement in the precision by increasing the number of bands and thereby increasing the computation time. That is one more reason why we chose 5 to be the number of bands.

A flowchart explaining the process in brief:


Dataset
Shingling
Input the query
Shingle the Query and append to the shingled data
MinHahsing
LSH and generate Candidate pairs
Checking the Hamming Similarity of Candidate pairs
Output those with a threshold similarity

References

1. C. D. Manning, P. Raghavan, and H. Schutze. Introduction to Information Retrieval, Cambridge University Press, 2008.
2. The dataset : https://www.kaggle.com/thomasnelson/humandata