Nearest Neighbor Problem
Locality sensitive hashing

Ben Gurion University

Tal Aharon 308259357

Shir Lupo 203021951

Ofir Shlifer 203522396

Table Of Contents

Introduction

The algorithm we have choosen to implement was taken from the article "Nearest Neighbor Search" by Alexander Andoni, Doctor of Philosophy.

The Nearest Neighbors (NN) problem is a fundumental computational primitive for dealing with massive dataset.

Consider the following scenario: one would like to extract useful information from a collection of images. A useful goal would be to label the images by what they contain. With subset of the database that was labeled already (prior information) its possible to deduce the label of each new image by finding the most "similar" labeled image and copying its label. However, with the dataset only getting larger and more ubiquitous, it becomes imperative to design algorithms capable of processing the data extremely efficiently.

The goal of the algorithm is to preprocess a set of objects so that later, given a query object, one can find efficiently the data object most similar to the query. In this problem the features of each object of interest are represented as a point in a high-dimensiomal space $\square^d$ and the similarity is measured using a distance metric. The number of features ranges anywhere from tens to millions. A particularly natural instance of the NN problem is in the Euclidean space where the data points live in a d -dimensional space $\square^d$ under the Euclidean distance function, where dimension d may range up to hundreds or even thousands.

The appeal of this algorithm is that, in many cases, an approximate nearest neighbor is almost as good as the exact one. An efficient approximation algorithm can be used to solve the exact nearest neighbor problem, by enumerating all approximate nearest neighbors and choosing the closest point. In the algorithm, point p is an $R-near\ neighbor$ of a point q if the distance between p and q is at most R. In order to use much smaller space while preserving a sub-linear query time the algorithm relies on the concept

The formal definition of the problem is:

Given a set D of points in a d - dimensional space $\mathbb{R}^d$ and parameters R > 0, δ > 0, construct a data structure such that, given any query point q, if D contains an $R-near\ neighbor$ of q, it reports some $cR-near\ neighbor$ of q in D with probability at least 1 – δ.

This problem has a broad set of applications in data processing and analysis. The problem is of major importance in areas such as data compression, databases and data mining, information retrieval, searching image dataset and more.

The Algorithm

The algorithm relies on the concept of locality-sensitive hashing (LSH). The key idea is to hash the points using several hash functions so as to ensure that, for each function, the probability of collision is much higher for points which are close to each other than for those which are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point. In that way the algorithm uses small space and preserves a sub-linear query time.

The algorithm:

Preprocessing stage:

1. Choose L functions g_j, $j = 1,..,L$, by setting $g_j = (h_{1,j}, h_{2,j}, .., h_{k,j})$, where $h_{1,j}, h_{2,j}, .., h_{k,j}$ are chosen at random from the LSH family H.
2. Construct L hash tables, where, for each $j = 1,..,L$, the j^{th} hash table contains the dataset points hashed using the function g_j.

Query algorithm for a query point q:

1. For each $j = 1,..,L$
   i) Retrieve the points from the bucket $g_j(q)$ in the j^{th} hash table.
   ii) For each of the retrieved point, compute the distance from q to it, and report the point if it is a correct answer ($cR - near\ neighbor$).

In the algorithm, H is a family of hash functions mapping $\mathbb{R}^d$ to some discrete universe U. At stages 1 & 2 in the preprocessing L functions $g_j(q) = (h_{1,j}(q), h_{2,j}(q), .., h_{k,j}(q))$ are choosen, where $h_{t,j} (1 \le t \le k, 1 \le j \le L)$ are chosen independently random from H in order to hash the data points (

$g_j(q)$ is the concatenation of the functions). The data structure is constructed by placing each point q from the input set into a bucket $g_j(q)$, for $j = 1,..,L$.

At stage 1 in query algorithm the buckets $g_1(q),..,g_2(q)$ are scanned and the points stored in them are retrieved. After that the distances from the points to the query point are computed and the valid answers reported.

Implementation

We choose to implement the algorithm in java. In the article there are two scanning strategies that are mentioned – interrupt the search after finding the first x points or continue the search until all points are retrived. We choose to go with the second strategy but with a little change – we also find the most closest neighbors from all the neighbors according to the application. Another thing to notice in our implementation is that we choose to implement the Euclidean LSH.

The main classes in our implementation are the following:

- E2HashFunction – Represents a hash function and including the function $hash$. We choose to implement it using LSH for the Euclidean Space. The hash function is computed as follows: $h_{x,b}(v) = \left\lfloor \frac{v \cdot x + b}{w} \right\rfloor$ where w is the length of each bucket and b is a Random Variable sampled from the Uniform Distribution. The v is the query and x some random vector (components of x are sampled from the Normal Distribution).
- E2HashFamily – Represents a hash family of functions and include the function $getHashFunction$ that initialize the parameters for new hash function.
- PairTable – Represents a hash table and includes the following fields: array of hash functions and the java object HashMap that holds the hash table.
- LSHbase – Represents the implementation of the algorithm. Include the following params: E2HashFamily, the parameters k and L from the algorithm, array of PairTable and two dimensional array of the samples. Includes the following main functions:
  - getKey – Responsible to hash a point using the function $hash$ from $E2HashFunction$ (doing the concatenation as well). Gets a point $pointToHash$ and an array of hash functions $hashFunctions$, loop over the $hashFunctions$ and run each function on the $pointToHash$. After

the hash is done the function return the combine value of the result (string).

- indexDataSample – Responsible for the preproccesing stage of the algorithm. The function get the samples array $newSample$ as the input and for every hash table - loop over all the points, finds the key to the hash table using the $getKey$ method and add the point to that key in the hash table using HashSet java object (used to store a collection of elements). The value that we insert to the HashSet is the index of the point. If the key wasn't part of the hash table we create it.
- query – Responsible for the query stage of the algorithm. The function gets the query q and the number of $closestNeighbors$ we want to find as inputs. The function initiate new HashSet where we will store all the closest points. For each PairTable we compute the key for q and if the HashSet in that key is not empty we add all the points in that set to the new set (using the java function getOrDefault). After we have all the closest points, we compute the distance from each point to the query and sorting the points according to their distances. Finally, we choose the first $closestNeighbors$ after the sorting and return them.

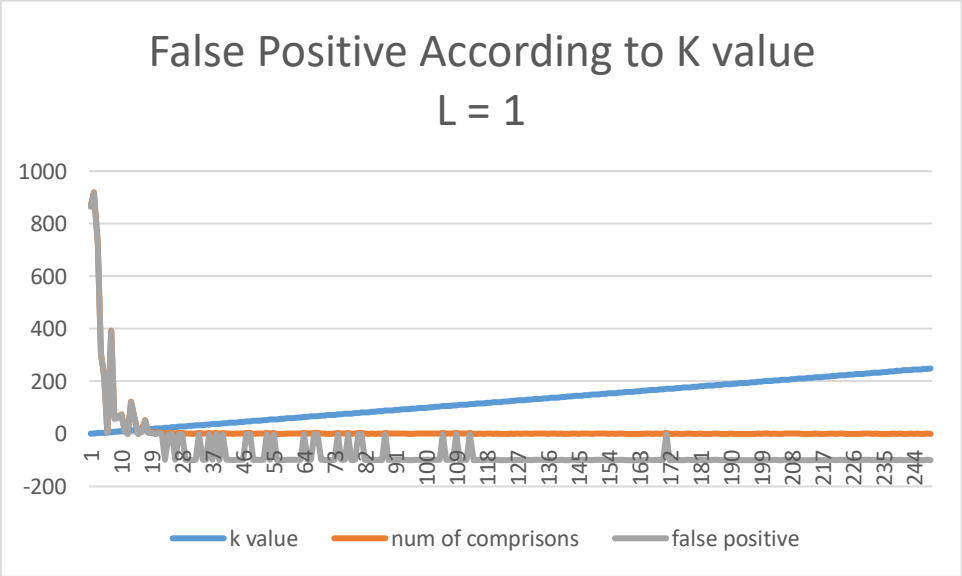
Experiments

In every experiment we took, we send to the function who represent this experiment, the num of closest neighbors we want to find ,we will call it 'close_neighbors' . For us to be able to track the result we wrote a method called "generateCloseNeighbors". This method returns an array of size close_neighbors which contains vectors of size d, which is the length of our query vector. After this stage, we generate a random sample set of vectors of size d but first we transplant our close neighbors array, so we will be able to draw conclusions from the result.
** (-) represent minus sign.

Experiment #1

We will define k as the num of hash function and the dimension of the newly set of vectors. Our query and sample vectors of size d – are turning into vectors of size k. The k represent for us the number of buckets in the Pair table.

We chose to look what is the influence of increasing k when we are looking on one table, means:
L = 1 , 0 < K < SampleSize/4 , SampleSize = 1000, number of close neighbors = 3;



We want to examine first the number of comparisons. The number of comparisons represent us the division of our sample into similar buckets. We see that when K is very small the number of comparisons is large. At first when we set k = 1, the number of comparisons was 867. We can conclude that in the average case, when increasing k, the number of comparisons respectively descends.

Secondly, we want to examine the influence of increasing k on the false positive value. Means, we want to examine the number of the neighbors that the algorithms gave us vs the real number of closest neighbors. We assigned to the heading of 'false positive' in the chart, the value : number of comparisons (–) num of neighbors, because the comparisons we made is the number of the closest neighbors that the algorithm gave us, and

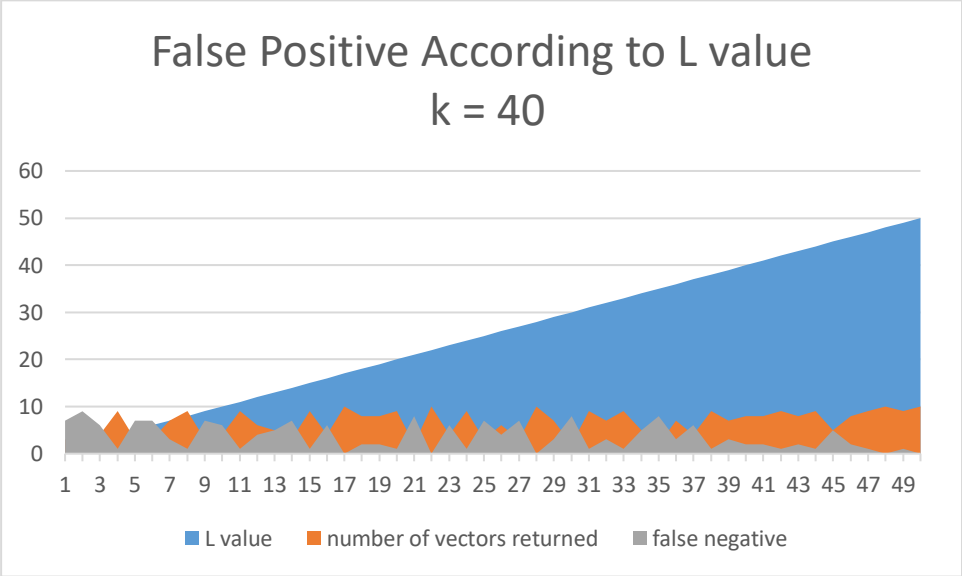
num of neighbors is the number of closest neighbors we transplanted in the sample. We determine that if the algorithm gave us less than the num of neighbors in the chart we set the value to -100 so the change will stand out.

the result show us that when K is very small, the algorithm alerts vectors to be the closest neighbors while they are not actual are. When we increase the value of k we see that the algorithm is giving us less redundant results. We also see that when K is very large, there is a large divisions of the sample to buckets and there is a large probability that the query vector will fall in a bucket alone, or without his real neighbors.

Experiment #2

We will define L as the number of times we repeat the algorithm each time with different K hash functions. At the end of the process we will have L tables, we will apply our query vectors on each table and reflect on the union of the resulting vectors.

We chose to look what is the influence of increasing L while k is constant.
0 < L < 50 , K = 40 , SampleSize = 1000, number of close neighbors = 10;

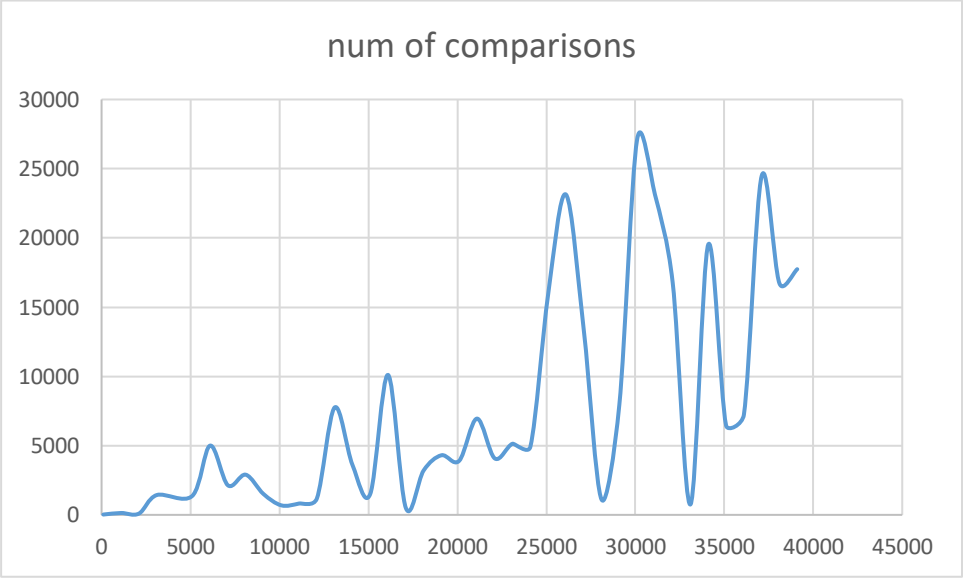


We want to examine the false negative value. We will define false negative as the number of vectors which are a part of the query vector's closest vectors but the algorithm did not detect them.
We will set the value of 'false negative' to be: num of closest neighbors (–) number of vectors returned (which are part of our closest neighbors). We were supposed to get all 10 vectors we transplanted but some of them are missing, this is exactly
The value of L is in charge of the amount of repeats of the algorithm. We see that if we increase L the algorithm has more 'opportunities' of finding the closest once, because we are using different hash functions. According to our result we see that large value of L gives us range of best results.

Experiment #3

In this experiment we want to explore the affect of out sample size on the number of comparisons.
To check this parameter, we will define constant size of L and K.
We chose L to be the average of experiment 2 and K to be the average of experiment 1 according to sample size 100, which is the starting sample size on the current experiment.
We checked the sample size in range 99 < x < 40,000 while every check we increased the sample size by 1000, this was for us to see more clear results in jumps of large numbers.

L = 20, K = 5, 99 < SampleSize < 40,000, number of close neighbors = 1;



In the graph above we can see that when we increase the number of the sample size the number of comparisons also increase.
Of course we see that in some places when there is high amount of data the number of comparisons can be small because a right divisions to buckets has occurred but if we will eliminate those cases and look at the overall result we can clearly see that if we increase the size of data the number of comparisons will act accordantly.

We can conclude from this results that the parameter K should be increase with respect of the size of the data. E.g will mark |Si| as the size of the sample i, so if we will look at |Di| = 100 and |Dj| = 10,000 and we will preform the algorithm with the same k, it means we will divide the samples to the same amount of buckets, so the probability that in the same bucket as our query vector we will have larger amount of vectors is greater on the experiment on Dj rather in Di.

Conclusion and Future Work

Conclusions

After we have done the experiments, we found that most of what we thought that would happen in the experiments – did happened.

From experiment 1, we conclude that if we will choose K to be small, the number of comparisons in most cases will be large because we divide the sample into buckets, if there are fewer buckets, it is logical that on every bucket could be large group of vectors, in particular in the with our query vector. So that means that if we will increase K so the number of false positive will decline due to the fact that the number of vectors spread in a larger number of buckets.
In addition, we saw that picking K to be too large can cause an isolation of our query vector, means he will be the only vector in our bucket so we will not report his neighbors, or we will report part of them.

From experiment 2, we conclude that choosing large value of L can give us better results. This caused because we repeat the hashing process multiple time with different hash functions and combine the results on each table from each stage, So for example if in one stage our query vector was isolated, there is a probability that in another stage he isn't, so the combination can give us better result of false negative, meaning the probability of tracking all the query vector neighbors, increases.

From experiment 3, we conclude that if we increase the sample size, the value of K should increase accordingly to maintain the number of comparisons.

Future work

In the future we would like to explore the influence of several more parameters.

We want to explore the effect of the value of D in the D-dimensional vectors.
We would like to conduct experiments of finding the optimal K and L value.
We want to explore the search time of the algorithm with optimal K and L.
We also want to explore the memory cost of different sizes of the sample group.

Bibliography

https://www.cs.princeton.edu/courses/archive/spr04/cos598B/bib/IndykM-curse.pdf

http://www.mit.edu/~andoni/thesis/main.pdf

http://mlwiki.org/index.php/Euclidean_LSH

https://www.slaney.org/malcolm/yahoo/Slaney2012%28OptimalLSH%29.pdf