Calculating Adjusted Rank Index using Locality Sensitive Hashing (LSH): A Gaussian Approach

Aritra Banerjee[1]

[1]4th year B.Tech CSE, School of Computer Science and Engineering, VIT University, Chennai
Vandalur-Kelambakkam Road, Chennai -600048, Tamil Nadu.

***Abstract**: Locality Sensitive Hashing (LSH) is a technique which is generally used to reduce the dimensionality of the given data. In this paper, I have used the Gaussian approach to reduce the dimensionality of a given massive dataset. Then used the binary matrix generated and created a neighbourhood graph for the given dataset. From the neighbourhood graph derived we can calculate the Adjusted Rank Index (ARI) value of the given dataset after applying Locality Sensitive Hashing. Since LSH is an Approximate Nearest Neighbour (ANN) calculation we approximately find the nearest neighbours of the given training dataset and check ARI value to see how closely the value approximately is from the actual neighbours present of different classes in the dataset.*

Keywords: *Locality Sensitive Hashing, Adjusted Rank Index, Gaussian, Approximate Nearest Neighbour*

I. Introduction

Nearest neighbour search (**NNS**), as a form of proximity search, is the optimization problem of finding the point in a given set that is closest (or most similar) to a given point.For datasets, we need to check the similarity with each point in the entire dataset from the given query point.**Approximate Nearest Neighbor search (ANN)** on the other hand decides fewer number of candidate pairs which are hashed to the same "bucket" based on hash lengths and only among those candidate pairs in the same "bucket" or the nearby buckets are checked thus reducing the unnecessary scanning of the entire dataset to a minimum.**Locality-sensitive hashing** (**LSH**) reduces the dimensionality of high-dimensional data. LSH hashes input items so that similar items map to the same "buckets" with high probability.Each bucket has similar items hashed inside it.Given a query point, we wish to find the points in a large database that are closest to the query.We find the k-nearest neighbors for a particular point.Before finding the k-neighbor for a query point we initially make neighborhood graph with the points given in the training dataset.**[2] Bawa M et al**The intuition behind LSH-based indexes is to hash points into buckets, such that "nearby" points are much more likely to hash to the same bucket than points that are far apart. We could then find the approximate nearest neighbours of any point, simply by finding the bucket that it hashes to, and returning the other points in the bucket.

II. Existing Works on LSH

Jefferey Ullman et al [1] suggested a method called the Minhash algorithm for finding similar sets among documents from massive datasets. They used a method called the Jaccard Similarity to find the similarity among two documents. Jaccard similarity of two sets S and T is defined as

$$\frac{|S \cap T|}{|S \cup T|}$$

which can also be stated as the ratio of the size of intersection of S and T to the size of their union. After calculating the similarities between two sets for the entire dataset, a signature matrix is created. Again think of a collection of sets represented by their characteristic matrix M. To represent sets, we pick at random some number n of permutations of the rows of M. Perhaps 100 permutations or several hundred permutations will do. Call the Minhash functions determined by these permutations h_1, h_2. . . h_n. From the column representing set S, construct the Minhash signature for S, the Vector [$h_1(S)$, $h_2(S)$ . . . $h_n(S)$]. We normally represent this list of hash-values as a column. Thus, we can form from matrix M a signature matrix, in which the ith column of M is replaced by the Minhash signature for (the set of) the ith column.A hash function that maps integers 0, 1, . . . , k −1 to bucket numbers 0 through k−1 typically will map some pairs of integers to the same bucket and leave other buckets unfilled. However, the difference is unimportant as long ask is large and there are not too many collisions. We can maintain the fiction that our hash function h "permutes" row r to position h(r) in the permuted order. Thus, instead of picking n random permutations of rows, we pick n randomly chosen hash functions h_1, h_2. . . h_n on the rows. We construct the signature matrix by considering each row in their given order. Let SIG(i, c) be the

element of the signature matrix for the ith hash function and column c. Initially, set SIG(i, c) to ∞ for all i and c. We handle row r by doing the following:
1. Compute $h_1(r), h_2(r) \dots h_n(r)$.
2. For each column c do the following:
(a) If c has 0 in row r, do nothing.
(b) However, if c has 1 in row r, then for each i = 1, 2. . . n set SIG (i, c)to the smaller of the current value of SIG (i, c) and $h_i(r)$.**[7]**One general approach to LSH is to "hash" items several times, in such a way that similar items are more likely to be hashed to the same bucket than dissimilar items are. We then consider any pair that hashed to the same bucket for any of the hashings to be a candidate pair. We check only the candidate pairs for similarity. The hope is that most of the dissimilar pairs will never hash to the same bucket, and therefore will never be checked. Those dissimilar pairs that do hash to the same bucket are false positives; we hope these will be only a small fraction of all pairs. We also hope that most of the truly similar pairswill hash to the same bucket under at least one of the hash functions. Those that do not are false negatives; we hope these will be only a small fraction of the truly similar pairs. If we have Minhash signatures for the items, an effective way to choose the hashings is to divide the signature matrix into b bands consisting of r rows each. For each band, there is a hash function that takes vectors of r integers (the portion of one column within that band) and hashes them to some large number of buckets. We can use the same hash function for all the bands, but we use a separate bucket array for each band, so columns with the same vector in different bands will not hash to the same bucket.

III. Gaussian Approach to LSH

A. Algorithm Proposed

- **[5] Datar, Met al** Generate random hyperplanes.
- The hyperplanes define the hash code for each sample.
- Similar elements hash to same bucket.
- Make the neighborhood graph for k-nearest neighbors for the training dataset.
- For incoming query points, calculate the hash codes according the hyperplanes and search the approximate k-neighbors.
- Calculate nearest neighbors from the matched bucket using cosine distance.
- Use hamming distance to check the nearest bucket available if exactly matched bucket doesn't have k-samples within itself and then repeat the above step for nearest bucket.
- To avoid false negatives and false positives generate hyper planes and hash buckets for L tables.

B. Hyperplanes generated

Lavrenko, V [4] et al proposed the below diagram**.** This diagram shows the random hyperplanes generated using random hash functions. The random hash functions were created using Gaussian distribution and then projected as hyperplanes for different tables using different functions.

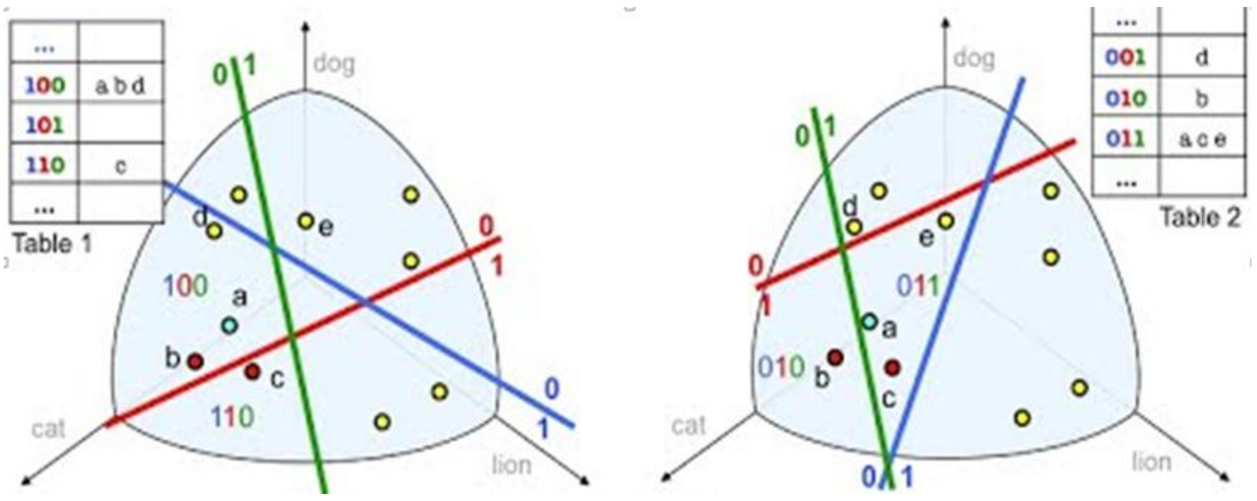


Figure 1: Random Hyperplanes generated using Hash Functions.

C. Required Formulae

The **Hamming distance** between two strings of equal length is the number of positions at which the corresponding symbols are different. In other words, it measures the minimum number of *substitutions* required to change one string into the other, or the minimum number of *errors* that could have transformed one string into the other.**The Cosine Similarity** of two vectors is given by similarity = cos (θ) =$\frac{A.B}{\|A\|\|B\|}$.

D. Calculation of ARI

Datasets Used:

I used mostly datasets which were included in mlbench library including Iris, Sonar, Ionosphere, Glass etc. Calculated the neighbourhood graphs for the given data sets and then calculated the ARI value at end. For this paper itself I am going to show the working and output for the Iris dataset.

Iris dataset:

In the iris dataset we broadly have three classes: *setosa, versicolor, virginica*. Each class has 50 samples each constituting a total of 150 samples and there are 4 different features: *sepal length, sepal width, petal length, petal width*.We now use to code developed in R to generate the 10 nearest neighbour matrix for each sample and finally calculate the Adjusted Rank Index value for the Iris dataset.

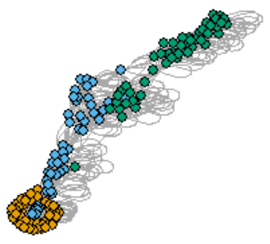
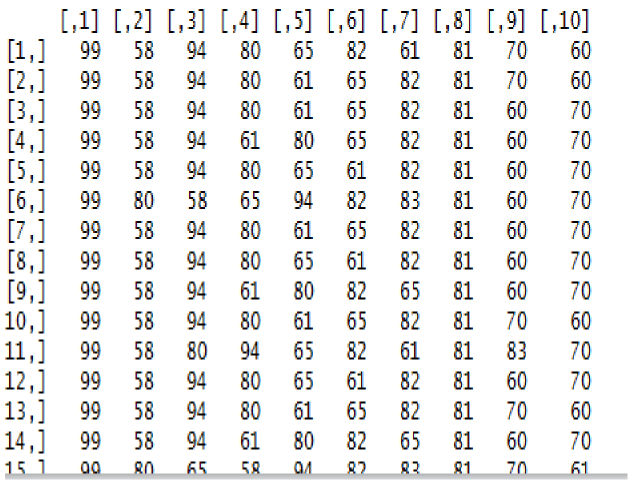


Figure 2: Clusters formed for 3 classes of Iris Dataset

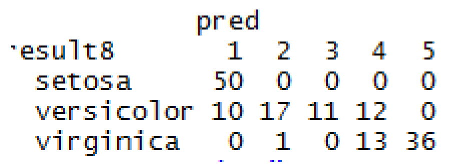
As we can clearly see that around 5 clusters are formed for the 3 classes of Iris dataset. The different species of Iris are shown by different colours in the graph.ARI value = **0.5475719**

E. Output values and discussion of Results



```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   99   58   94   80   65   82   61   81   70    60
[2,]   99   58   94   80   61   65   82   81   70    60
[3,]   99   58   94   80   61   65   82   81   60    70
[4,]   99   58   94   61   80   65   82   81   60    70
[5,]   99   58   94   80   65   61   82   81   60    70
[6,]   99   80   58   65   94   82   83   81   60    70
[7,]   99   58   94   80   61   65   82   81   60    70
[8,]   99   58   94   80   65   61   82   81   60    70
[9,]   99   58   94   61   80   82   65   81   60    70
10,]   99   58   94   80   61   65   82   81   70    60
11,]   99   58   80   94   65   82   61   81   83    70
12,]   99   58   94   80   65   61   82   81   60    70
13,]   99   58   94   80   61   65   82   81   70    60
14,]   99   58   94   61   80   82   65   81   60    70
15,]   99   80   65   58   94   82   83   81   70    61
```

Figure 3: Neighbourhood Matrix for first 15 samples



```
              pred
'esult8         1  2  3  4  5
  setosa       50  0  0  0  0
  versicolor   10 17 11 12  0
  virginica     0  1  0 13 36
```

Figure 4: Division of 5 clusters among 3 species

We can observe that the **ARI value is 0.5475719** and the five clusters is divided among 3 species of Iris dataset as stated above in the diagram. So the Adjusted Rank Index value is approximately 55% accurate which means the LSH algorithm devised using Gaussian approach is approximately accurate in finding the nearest neighbours of the samples in the IRIS dataset.

F. Complexity and Computational Cost

- **[5]**N points, D- dimensional, K hyperplanes
- D*K …. To find bucket where point lands
- 2^K …. Possible no. of hash codes
- N/2^K …. Points in that bucket (on average)
- D*N/2^K …. Cost of comparisons
- Repeat everything L times (hash tables)

So,Total cost = (L*D*K) + (L*D*N/2^K) $\rightarrow$ O (log N) if K ~ log N

IV. Conclusion and Future Work

As we can see in the above example where I used IRIS dataset it returned a 55% accurate result for finding the nearest neighbours of the given samples. LSH is actually an Approximate Nearest Neighbour (ANN) algorithm and hence its accuracy is compromised for the speed. LSH basically helps reduce the time complexity and search space for a given dataset to huge extent and hence sometimes the accuracy may not be as high as other Nearest Neighbour Search (NNS) algorithms. But, we can surely work towards improving the accuracy of LSH provided we keep in mind not to increase the search space. This method is largely used for massive datasets and hence there lies the application of LSH.

REFERENCES

[1] Anand RajaRaman, Jefferey Ullman, "*Mining of Massive Datasets*"
[2] Bawa, M., Condie, T. and Ganesan, P., 2005, May. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web* (pp. 651-660). ACM.
[3] Lv, Q., Josephson, W., Wang, Z., Charikar, M. and Li, K., 2007, September. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases* (pp. 950-961). VLDB Endowment.
[4]Moran, S., Lavrenko, V. and Osborne, M., 2013, August. Variable Bit Quantisation for LSH. In *ACL (2)* (pp. 753-758).
[5]https://www.youtube.com/watch?v=Arni-zkqMBA
[6] Datar, M., Immorlica, N., Indyk, P. and Mirrokni, V.S., 2004, June. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry* (pp. 253-262). ACM.
[7]http://www.slaney.org/malcolm/yahoo/Slaney2008-LSHTutorial.pdf