Database Seminar Report - Similarity Search in High Dimensions via Hashing[1]

Timothy Pescatore

April 21, 2018

1 Introduction

Problem: There are many applications which need to run a similarity search in order to find a nearest neighbor. For example if one must search for a similar image in a database, based on its content and not metadata like tags or descriptions. Such a search will end in comparing points which have several dimensions. Indexing structures which solve this problem well in low dimension, like a k-d tree, will lose their efficiency quickly as soon as the dimension increases. This is the reason, why they often do not have better performance than a brute force linear search. However, often it is sufficient to just find an approximate nearest neighbor. The paper [1] offers an algorithm to find an approximate nearest neighbor through indexing the data set.

Overview: The paper [1] uses s hashing approach for the indexing. This is done in two stages. The first hashing function is responsible for the indexing structure of the similarity search, the second one is a basic hash into a smaller table, so space will not be wasted. The difference from the first hash to a basic hash is that, the similarity of two points is closely related to the probability that they will be hashed into a same bucket. The technique is based on randomness and will therefore not always return the exact nearest neighbor but an approximate one, which for us, is good enough. We will do this hashing several times to increase the probability that a near neighbor will be in the same bucket as the query point. This way we will still have a set of points over which we need to do a linear search, but much smaller than the original set.

2 Preliminaries

We use l_p^d to denote the Euclidean space $\Re^d$ under the l_p norm, i.e., when the length of a vector $(x_1, ..., x_d)$ is defined as $(|x_1|^p + ... + |x_d|^p)^{1/p}$. Further, denote $d_p(p, q) = ||p - q||_p$ the distance between the points p and q om l_p^d. H^d is used to denote the *Hamming metric space* of dimension d, which is the set of all binary strings of length d. The distance function $d_H(p, q)$ denotes the Hamming distance, i.e., the number of bits on which p and q differ.

The nearest neighbor search problem is defined as follows:

Definition 1 (Nearest Neighbor Search(NNS)) *Given a set P of n objects represented as points in a normed space l_p^d, preprocess P so as to efficiently answer queries by finding the point in P closest to a query point q.*

The definition generalizes naturally to the case where we want to return $K > 1$ points. Specifically, in the $K - NearestNeighborsSearch(K - NNS)$, we wish to return the k points in the database that are closest to the query point.
With the hash function discussed in this report, we will not solve the NNS-problem, but we want to find an approximate nearest neighbor with a small variance $\epsilon \geq 0$:

Definition 2 (ϵ-Nearest Neighbor Search (ϵ-NNS)) *Given a set P of points in a normed space l_p^d, preprocess P so as to efficiently return a point $p \in P$ for any given query point q, such that $d(q, p) \leq (1 + \epsilon)d(q, P)$, where $d(q, P)$ is the distance of q to its closest point in P.*

3 The Algorithm

In this section we present locality-sensitive hashing (LSH). This technique was originally introduced by Indyk and Motwani [2]. The new algorithm is an improved version, because it is more natural and guarantees a better running time.

For the data are two assumptions made:

1. the distance is defined by the l_1 norm,
2. all coordinates of points in P are positive integers.

The first assumption is not very restrictive, since they observed in the experiments that a nearest neighbor of an average query point computed under the l_1 norm was also an ϵ-approximate neighbor under the l_2 norm with an average value of ϵ less than 3%. As for the second assumption, the rounding error can be made arbitrarily small.

Now we can start with the algorithm:
Let C be the largest coordinate in all points in P. We can embed P into the Hamming cube $H^{d'}$ with $d' = Cd$, by transforming each point $p = (x_1, ..., x_d)$ into a binary vector $v(p)$, which means that that a transformed x is a sequence of x ones followed by $C - x$ zeroes.

Example:

We chose $P = \{(1,1), (5,4), (1,2)\}$

Here $d = 2$, $C = 5$, hence $d' = Cd = 10$

The transformation of p_1 into a binary vector is:

$v(p_1) = 10000,\ 10000 \rightarrow 1000010000$
$v(p_2) = 11111,\ 11110 \rightarrow 1111111110$
$v(p_3) = 10000,\ 10000 \rightarrow 1000011000$

Each coordinate becomes a string with 5 digits, let us call them bits since they are either 1s or 0s. These strings are always ordered, starting with x ones followed by C - x 0s. Afterwards we concatenate all coordinates from each point and get 10 bit strings. Keep in mind that the first to the 5th bits correspond to the first coordinate, and the 6th to the 10th bits correspond to the second coordinate.

To measure the distance between two points in the Hamming space we count the bits which differ when comparing both strings.

Fact 1: *For any pair of points p, q with coordinates in the set* $\{1...C\}$, $d_1(p,q) = d_H(v(p), v(q))$

This means that the actual distance from the l_1-norm is preserved in the Hamming space, so we can still identify a nearest neighbor. In the actual implementation it will not be actually necessary to convert the data into binary string. It could be expensive when C is large. But we use the transformation to illustrate how the hashing works.

3.1 The Hashing

First we chose l subsets $I_1, ..., I_l$ of $\{1, ..., d'\}$. Each subset contains $k < d'$ elements which are sampled uniformly at random with replacement. The proper choice of l and k will be discussed later. Let $p_{|I}$ denote the projection of vector p on the coordinate set I, i.e. we compute $p_{|I}$ by selecting the ith bit from the transformation of p, where $i \in I$.

For the sake of example we chose $k = 3$ and $l = 2$; We get two subsets with 4 random numbers of $\{1...10\}$ recall $d' = 10$

$I_1 = \{2, 4, 5\}$, $I_2 = \{3, 6, 10\}$.

Let us take $p_1(1, 1)$ and project it on the coordinate sets I_1 and I_2:

Transformed point $v(p_1)$	1	0	0	0	0	1	0	0	0	0
Position in string	1	2	3	4	5	6	7	8	9	10

$p_{1|I_1} = 0_2 0_4 0_5 : 000$

$p_{1|I_2} = 0_3 1_6 0_{10} : 010$

Let $g_j(p) = p_{|I_j}$ for $j = 1, ..., l$ be the bucket we store the hashed point in. For the preprocessing, we store each $p \in P$ in the bucket $g_j(p) = p_{|I_j}$ for $j = 1, ..., l$. As the total number of buckets may be large, we compress the buckets by resorting to standard hashing. For the sake of example we will not do this second step of hashing, since it is not essential for the illustration of the LSH.

Algorithm 1: Preprocessing

Input: A set of points P, l(number of hash tables)
Output: Hash tables $\mathcal{T}_i, i = 1, ..., l$
Foreach i = 1,...,l
Initialize hash table $\mathcal{T}_i$ by generating a random hash function $g_i()$ using the subset I_i
Foreach i = 1,...,l
Foreach j = 1,...,n
Store point p_j on bucket $g_i(p_j)$ of hash table $\mathcal{T}_i$

As mentioned before, the transformation of each point into the *Hamming space* could be expensive when C is large. To avoid this transformation there exists an auxiliary step.
Let the elements of I corresponding to the ith coordinate of p, $I_{|i}$ for $i = 1...d$, be in sorted order. When projecting p on $I_{|i}$, one will get monotone strings of say o_i ones followed by $C - o_i$ zeroes. Here o_i is equal to the number of elements in a subset which are smaller or equal to the ith coordinate of p. To represent the projection of p, it is sufficient to compute o_i for $i = 1...d$. Let us demonstrate it on an example:

Looking at I_2 there are 2 subsets each corresponding to one of the coordinates. Remember that the first 5 bits correspond to the first coordinate, while the 6th to the 10th bits correspond to the second coordinate.

$I_2\Big\{I_{|1} = \{3\}, I_{|2} = \{6, 10\}\Big\}$

Now keep in mind, that the subsets are ordered and the bits corresponding to a specific coordinate are ordered as well, since they are x 1s , x equals the corresponding coordinate, followed by $C - x$ 0s. Counting the elements smaller than x from the subset, we know how many elements point to a "1"-bit and how many points to a "0"-bit. With the point $p_1(1, 1)$ we get:

$I_{|1}\{\mathbf{1} < 3\} \to 0$: *Both elements are bigger than the corresponding coordinate*

$I_{|2}\{6, \leq (\mathbf{1} + 5) <, 10\} \to 10$: *We add 5 to the value of the coordinate, since the corresponding bits start after the 5th bit*

This results in finding the last element in subset which is smaller or equal to the corresponding coordinate and then counting the elements.

This auxiliary step can be done with a binary search in $\log C$, since we only need to split the subsets into two parts. So the total time needed to compute the hash function is $O(d \log C)$.

Finally to query for a point, one does the exact same hash, but instead of storing the point into the resulted buckets, we give back all points found in these buckets or stop when we found $2l$ points. The reason why exactly $2l$ will be discussed in the analysis. The last step is to do a linear search over the encountered points.

Algorithm 2: Approximate Nearest Neighbor Query

Input: A query point q, K (number of approximate nearest neighbors)
Access To hash tables $\mathcal{T}_i, i = 1, ..., l$ generated by the preprocessing algorithm
Output: K (or less) appr. nearest neighbors
$S \leftarrow \emptyset$
Foreach i = 1,...,l
$S \leftarrow S \cup \{$ points found in $g_i(q)$ = bucket of table $\mathcal{T}_i\}$
Return the K nearest neighbors of q found in set S /*Can be found with linear search*/

Now let's look at a complete example to illustrate the procedure which is needed to do this locality sensitive hash. Let us query for the point $q(2, 1)$.

Since we have a very small data set for the sake of example we will hash only in two buckets per table. Each table has 1 bucket B_1 for hashes, smaller than 100_2 and another one B_2 for all hashes greater or equal to 100_2.

Recall that $I_1\{2, 4, 5\}$, $I_2\{3, 6, 10\}$, $P = \{p_1(1, 1), p_2(5, 4), (1, 2)\}$

Preprocessing:
$g_1(p_1) = 000 \to I_{1B_1}$, $g_2(p_1) = 010 \to I_{2B_1}$
$g_1(p_2) = 111 \to I_{1B_2}$, $g_2(p_2) = 110 \to I_{2B_2}$
$g_1(p_3) = 000 \to I_{1B_1}$, $g_2(p_1) = 010 \to I_{2B_1}$

After preprocessing we get the following buckets filled with points

Bucket	I_{1B_1}	I_{1B_2}	I_{2B_1}	I_{2B_2}
Points stored	p_1	p_2	p_1	p_2
	p_3		p_3	

Query:
$g_1(q) = 100 \to I_{1B_2}$, $g_2(q) = 010 \to I_{2B_1}$

So the query would return I_{1B_2} and I_{2B_1} which contain p_1, p_2 and p_3. This is a good result, because the near neighbors p_1 and p_3 are both contained in the result set. The point p_2 is also in the result set, but just because we hashed only into two buckets. Even though this point is far away, it is not bad, since we do a linear search as a next step, to find the K nearest neighbor from the result set.

The data from the example is strongly reduced since the number of points is very small, the dimension is low and the value of k and l were adapted for illustration reasons. Now let us look at the analysis to answer open questions, like why this actually works or how many points will be returned.

4 Analysis of LHS

The principle behind the LSH is that the probability of collision of two points p and q is closely related to the distance between them. Thus the bigger the distance between a point and another one, the lower the probability they will be hashed into the same bucket.

Definition 3
A Family $\mathcal{H}$ of functions from S to U is called $(r_1, r_2, p_1, p_2)-$sensitive for $D(\cdot,\cdot)$ if for any $q, p \in S$

- *if $p \in \mathcal{B}(q, r_1)$ the $Pr_{\mathcal{H}}[h(q) = h(p)] \geq p_1$,*
- *if $p \notin \mathcal{B}(q, r_2)$ the $Pr_{\mathcal{H}}[h(q) = h(p)] \leq p_2$*

  In order for a locality-sensitive family to be useful, it has to satisfy the inequalities $p_1 > p_2$ and $r_1 < r_2$.

This means, that we want a higher probability (p_1) if a point is near to our query point and it will have the same hash result as the query point than, when (p_2) it is out of our "query area" and the hash result would be the same.

Fact 2 *Let S be $H^{d'}$ (the d'-dimensional Hamming cube) and $D(p,q) = d_H(p,q)$ for $p,q \in H^{d'}$. Then for any $r, \epsilon > 0$, the family $\mathcal{H}_{d'} = \{h_i : h_i((b_1, ..., b_{d'}))\} = b_i$, for $i = 1, ..., d'$ is $(r, r(1+\epsilon), 1 - \frac{r}{d'}, 1 - \frac{r(1+\epsilon)}{d'})$-sensitive*

Hence, r, ϵ and d give p_1 and p_2 it is guaranteed that $p_1 > p_2$ and also $r_1 < r_2$.

We can now show that the LSH algorithm can be used to solve what is called the (r, ϵ)**-Neighbor problem**: determine whether there exists a point p within a fixed distance $r_1 = r$ of q, or whether all points in the database are at least a distance $r_2 = r(r, \epsilon)$ away from q. Denote the set of all points $p' \notin B(q, r_2)$ by P'. The algorithm solves this problem correctly when the following two properties hold:

P1 If there exists p^* such that $p^* \in B(q, r_1)$, then $g_j(p^*) = g_j(q)$ for some $j = 1, ..., l$.

P2 The total number of blocks pointed to by q and containing only points from P' is less than cl.

Assume that $\mathcal{H}$ is a (r_1, r_2, p_1, p_2)-sensitive family; define $\rho = \frac{\ln 1/p_1}{\ln 1/p_2}$. The correctness of the LSH algorithm follows from the following theorem.

Theorem 1 *Setting $k = \log_{1/p_2}(n/B)$ and $l = (n/B)^\rho$ guarantees that properties* **P1** *and* **P2** *hold with probability at least $\frac{1}{2} - \frac{1}{e} \geq 0.132$.*

Remark 1 *Note that by repeating the LSH algorithm $O(1/\delta)$ times, we can amplify the probability of success in at least one trial to $1 - \delta$, for any $\delta > 0$.*

Proof: Let property **P1** hold with probability P_1, and property **P2** hold with probability P_2. We will show that both P_1 and P_2 are large. Assume that there exists a point p^* within distance r_1 of q. Set $k = \log_{1/p_2}(n/B)$. The probability that $g(p') = g(q)$ for $p' \in P - \mathcal{B}(q, r_2)$ is at most $\frac{B}{n}$. Since,

$$p_2^k = p_2^{\log_{1/p_2}(n/B)} = (1/p_2)^{-\log_{1/p_2}(n/B)} = (1/p_2)^{\log_{1/p_2}(B/n)} = \frac{B}{n}$$

The expected number of blocks allocated for all g_j which contain *exclusively* points from P' does

not exceed 1. The expected number of such blocks allocated for all g_j is at most l. Thus, by the Markov inequality [3], the probability that this number exceeds $2l$ is less than $1/2$. If we choose $c = 2$, the probability that the property P2 holds is $P_2 > 1/2$. Consider now the probability of $g_j(p^*) = g_j(q)$. Clearly, it is bounded from below by

$$p_1^k = p_1^{\log_{1/p_2} n/B} = (n/B)^{-\frac{log1/p_1}{log1/p_2}} = (n/B)^{-\rho}.$$

By setting $l = (\frac{n}{B})^\rho$, we bound from above the probability that $g_j(p^*) \neq g_j(q)$ for all $j = 1, ..., l$ by $1/e$. Thus the probability that one such g_j exists is at least $P_1 \geq 1-1/e$. Therefore, the probability that both properties P1 and P2 hold is at least $1 - [(1 - P_1) + (1 - P_2)] = P_1 + P_2 - 1 \geq \frac{1}{2} - \frac{1}{e}$. The theorem follows. □

In the following we consider the LSH family for the Hamming metric of dimension d' as specified in Fact 2. For this case, we show that $\rho \leq \frac{1}{1+\epsilon}$ assuming that $r < \frac{d'}{\ln n}$; the latter assumption can be easily satisfied by increasing the dimensionality by padding a sufficiently long string of 0s at the end of each point's representation.

Fact 3 *Let* $r < \frac{d'}{\ln n}$. *if* $p_1 = 1 - \frac{r}{d'}$ *and* $p_2 = 1 - \frac{r(1+\epsilon)}{d'}$, *then* $\rho = \frac{\ln 1/p_1}{\ln 1/p_2} \leq \frac{1}{1+\epsilon}$.

Proof: Observe that

$$\rho = \frac{\ln 1/p_1}{\ln 1/p_2} = \frac{\ln \frac{1}{1-r/d'}}{\ln \frac{1}{1-(1+\epsilon)r/d'}} = \frac{\ln(1-r/d')}{\ln(1-(1+\epsilon)r/d')}.$$

Multiplying both the numerator and the denominator by $\frac{d'}{r}$, we obtain:

$$\rho = \frac{\frac{d'}{r}\ln(1-r/d')}{\frac{d'}{r}\ln(1-1(1+\epsilon)r/d')} = \frac{\ln(1-r/d')^{d'/r}}{\ln(1-/1+\epsilon)r/d')^{d'/r}} = \frac{U}{L}$$

In order to upper bound ρ, we need to bound U from below and L from above; note that both U and L are negative. To this end we use the following inequality[3]:

$$(1 - (1 + \epsilon)r/d')^{d'/r} < e^{-(1+\epsilon)}$$

and

$$(1 - \frac{r}{d'})^{d'/r} > e^{-1}(1 - \frac{1}{d'/r}).$$

Therefore,

$$\frac{U}{L} < \frac{\ln(e^{-1}(1-\frac{1}{d'/r}))}{\ln e^{(1+\epsilon)}} = \frac{-1+\ln(1-\frac{1}{d'/r})}{-(1+\epsilon)} = 1/(1+\epsilon) - \frac{\ln(1-\frac{1}{d'/r}}{1+\epsilon} < 1/(1+\epsilon) - \ln(1-1/\ln n)$$

where the last step uses the assumption that $\epsilon > 0$ and $r < \frac{d'}{\ln n}$. We conclude that,

$$n^\rho < n^{1/(1+\epsilon)} n^{-\ln(1-1/\ln n)} = n^{1/(1+\epsilon)}(1 - 1/\ln n)^{-\ln n} = O(n^{1/(1+\epsilon)}) \ \square$$

5 Conclusion

In this paper [1] a significant improvement of the time is achieved. So the new LSH algorithm guarantees sublinearity with $O(dn^{1/(1+\epsilon)})$. This algorithm can be used in many applications, because the approximate-nearest neighbor search might provide a higher quality-efficiency trade-off. The improvement is achieved through a simple hashing technique, which is based on randomness to achieve locality-sensitivity. Additionally the hashing function is nicely illustrated in the paper by an interim step, which eventually is avoided in the implementation through an auxiliary step.

References

[1] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[2] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of computing*, 8(1):321–350, 2012.

[3] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.