Locality Sensitive Hashing

Subhash Suri

November 19, 2019

1 Nearest Neighbor Searching in High Dimensions

- The NNS problem is the following. Given a set of points $P \in R^d$, organize them in some data structure to answer following *nearest neighbor* queries. For any query point $q \in R^d$, find the point in P closest to q, namely, the argmin for the following:

$$\min_{p \in P} \text{dist}(p, q)$$

  We will write $p = NN(q)$.

- Many different distance functions are used in applications, from Euclidean ℓ_2 norm to Manhattan ℓ_1 norm, or ℓ_∞, or angular distance etc. We will focus mostly on ℓ_1, ℓ_2 but the techniques apply broadly.

- Clearly, one can find the $NN(q)$ in $O(dn)$ time by simply searching all the points of P. (The dim $d \gg 1$ is large, and needs to be explicitly factored in the time complexity.) But in many applications, linear-time per search is unacceptably slow, and we want a sub-linear query time.

- Two standard approaches for sub-linear query time for NNS are: Voronoi diagrams, and KD-trees. (Observe that WSPD is not useful here, since that solves off-line version of NNS.)

  1. The Voronoi diagram of P partitions the space into voronoi cells so that finding $NN(q)$ is equivalent to point locating q in $VD(P)$.
  2. The KD-tree of P organizes the points into a binary tree, and to search for $NN(q)$, we first the leaf node (cell with singleton point) containing q. We then perform a spiraling search around this cell, until we are guaranteed that none of the remaining boxes can contain the nearest neighbor of q.

- **Drawbacks of VoD and KD-tree.**

  1. The problem with these two approaches is that either the space complexity explodes exponentially with d, or the search time rapidly approaches $O(n)$.
  2. In particular, VoD of n points in d-dim can have size $\Omega(n^{\lceil d/2 \rceil})$. We need to store the VoD in a point-location data structure, and point location in d-space is also not trivial.
  3. The KD-tree search structure has good space bound $O(n)$, but the search has exponential dependence on d, and so already at $d = \log n$ the search time becomes linear.
  4. In fact, for all known approaches, either the space complexity or the search time grows exponentially with d. This is called the *Curse of Dimensionality.* Recall our earlier discussion about subdividing a unit cube into quadtree style children. Just one round of partition produces 2^d children.

- **Applications of High Dim NNS**. There are many applications that involve solving NNS problem in fairly high dim. We briefly mention two:

  1. **Image Processing.** Digital images are often divided into smaller regions, and storing the average values of color intensities for each region. Then the image can be represented as a vector in R^d, where d is the number of regions. The dim d can be in millions if regions are at pixel resolution, or in 10^3–10^5 for more coarse partitions. With this representation, Euclidean distance between two vectors is a good measure of their similarity.
  2. **Salton's word model for Document Retrieval.** Each English language document represented as a vector in R^d, where the ith entry is the frequency of the ith word in the English language vocabulary database. Typically these databases have 50K to 100K words. Or, vectors may be just 0-1 vectors, indicating which words are present. ℓ_1 or ℓ_2 distances between vectors v_i and v_j a measure of similarities between documents i and j.
  3. Thus, the dimension of the space over which NNS takes places is quite large in these applications, in 1000s to $100,000$s, if not millions. For comparison, if one could do a trillion (10^{12}) ops in one sec, even 2^{21} will require 100 years. 2^{100} is incomprehensibly large.

1.1 Approximate NN Search

- Therefore, practical approaches for NNS with sub-linear query time and reasonable space complexity resort to solving the problem approximately. In stead of trying to

compute the exact NN, we are content to find one that is almost as close. In particular, we define an *approximate NN* of q as follows: find a point $p \in P$ such that

$$\text{dist}(p, q) \ \leq \ c \cdot \min_{s \in P} \text{dist}(s, q)$$

That is, the distance from q to p is at most c times the distance to q's true nearest neighbor in P. The constant $c > 1$ is our approximation factor. Our search and data structure space complexity will be a function of c.

- **$r-$ANNS problem. (Fixed Radius NNS).** In fact, we will solve a more specialized problem, defined as follows. Input is a set of point $P \in R^d$, and $c > 1$, and $r > 0$ (our target distance).

  The goal is to construct a data structure on P such that given any query point q, if there is a point $p \in P$ with $\text{dist}(p, q) \leq r$, then return a point $p' \in P$ with

  $$\text{dist}(p', q) \ \leq \ cr$$

  Otherwise (no point p with $\text{dist}(p, q) \leq r$), and we are free to return nothing.

  In other words, we have a target distance r in mind, and want our data structure to solve NNS problems for distance $\leq r$.

  For any query point q satisfying this requirement, our algorithm returns an approximate NN with distance at most cr; for other, we don't care.

- **From r-ANNS to ANN.** It is not hard to see that if we can solve r-ANNS, then we can solve ANN by repeatedly guessing the values of r. Without loss of generality, by scaling we assume that $diam(P) \leq 1$, and so $\text{dist}(p, p') \leq 1$, for all $p, p' \in P$. Let $\delta > 0$ be the min distance between any two points in our data set.

  We solve $ANNS(c(1-\varepsilon), r)$ for the following values of r

  $$\delta, \ (1+\varepsilon)\delta, \ (1+\varepsilon)^2\delta, \ \ldots$$

  and report smallest value of r for which we find a point at distance $c(1-\varepsilon)$ from q. (The true distance r lies in some range $[(1+\varepsilon)^j \det, \ (1+\varepsilon)^{j+1}\delta]$, and so we lose one factor of $(1+\varepsilon)$ in our distance estimate. By using a smaller approximation factor, we get an estimate that is $c(1-\varepsilon)(1+\varepsilon)r \approx cr$, ignoring the lower order term.

  This reduction introduces a factor $\log 1/\delta$ overhead in our query algorithm and space complexity, since we need a separate data structure for each value of r.

1.2 Locality Sensitive Hash Functions

- We will solve ANNS using an interesting class of hash functions, called LSH, introduced by Indyck and Motwani in 1998. First some remarks:

  1. In traditional hashing, the goal is to map a set of keys to a hash table, so that we can perform Lookup, Insert, Delete in $O(1)$ expected time.
  2. The size of the input domain N is huge while the hash table size m is quite small, $m \ll N$. (For instance, storing IP addresses or user names to a table of size 10^3.)
  3. There we want the hash function to "spread" our data out randomly, so that similar keys do not end up hashing to the same location.
  4. Interestingly, in LSH we want *locality* in the sense that we want "nearby points" to hash to same value, and "far away" points to hash to different values.

- More specifically, we want two points p and q (with high probability)

  1. to hash to same value if $\quad \text{dist}(p, q) \leq r$ and
  2. to hash to different values if $\quad \text{dist}(p, q) > cr$

- More formally, let $\mathcal{H} = \{h \mid P \to Z^+\}$ be a family of hash functions that maps points of P to the set of positive integers Z^+. The integer in Z will be our hash table size.

  We say that $\mathcal{H}$ is a (c, cr, p_1, p_2)-LSH if for all $p, q \in P$, the following holds

  1. $\text{dist}(p, q) \leq r \implies Pr[h(p) = h(q)] \geq p_1$, and
  2. $\text{dist}(p, q) > cr \implies Pr[h(p) = h(q)] \leq p_2$,

  where $p_1 > p_2$ and the probabilities are over random choices of $h \in \mathcal{H}$. That is, if we choose a random hash function h from $\mathcal{H}$, we have these success probabilities.

- Ideally, we want $p_1 \gg p_2$, but as we will see this highly depends on the magnitude of c. The key idea of Indyck-Motwani is that *even if p_1 is only slightly larger than p_2, it is possible to use many independent hash functions from $\mathcal{H}$ to* **boost** *p_1 close to 1, and shrink p_2 close to $1/n$.*

- Before describing the LSH scheme, let us consider some examples.

- **Hamming or Manhattan Distance over binary vectors.**

  1. Suppose $P \subseteq \{0, 1\}^d$ with Manhattan distance function $\text{dist}(p, q) = \|p - q\|_1$. That is, $\text{dist}(p, q)$ is the number of coordinates at which p and q differ.

2. Consider the hash function family $\mathcal{H} = \{h_i\}_{i=1}^{d}$ where

$$h_i(p) \;=\; p_i, \qquad \text{where } p_i \text{ is the } i\text{th bit of } p$$

3. Then, for each pair of points $p, q \in \{0,1\}^d$, we have

$$Pr[h(p) = h(q)] \;=\; \frac{\#\text{bits in common}}{\text{total bits}} \;=\; \frac{(d - \|p-q\|_1)}{d} \;=\; 1 - \frac{\|p-q\|_1}{d}$$

4. Therefore,

$$Pr[h(p) = h(q)] \;\geq\; 1 - r/d \;\approx\; e^{-r/d} \qquad \text{if } \text{dist}(p,q) \leq r$$

and

$$Pr[h(p) = h(q)] \;\leq\; 1 - cr/d \;\approx\; e^{-cr/d} \qquad \text{if } \text{dist}(p,q) \geq cr$$

5. Hence, $\mathcal{H}$ is a $(c, cr, e^{-r/d}, e^{-cr/d})$-LSH.

- **Jaccard Similarity.**

1. Jaccard similarity is used to measure how similar two sets are.

$$J(S_1, S_2) \;=\; \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

2. To define a hash function family for Jaccard, choose a random permutation π on the universe U (from which sets are drawn).
3. Then, for any set $S \subset U$, the LSH for Jaccard similarity is

$$h(S) \;=\; \text{first element in } S \text{ according to permutation } \pi$$

4. Now, it is easy to see that

$$Pr[h(S_1) = h(S_2)] \;=\; J(S_1, S_2),$$

where the prob is over the choice of permutation π.

5. If we define the distance function as

$$\text{dist}(S_1, S_2) \;=\; 1 - J(S_1, S_2),$$

then our hash family is LSH for any $r > 0$ and $c > 1$ because

$$\text{dist}(S_1, S_2) \leq r \implies Pr[h(S_1) = h(S_2)] \geq 1 - r,$$

and

$$\text{dist}(S_1, S_2) \geq cr \implies Pr[h(S_1) = h(S_2)] \leq 1 - cr$$

1.3 Reducing the ANN problem to LSH.

- Suppose we are given a hash function family $\mathcal{H}$ that is (r, cr, p_1, p_2)-LSH. Let use first consider solving the ANN problem assuming an *Ideal Probability Gap*, namely, that $p_1 \approx 1$ and $p_2 \approx 0$. In that case, we would solve the ANN problem as follows.

  1. Choose a hash function $h \in \mathcal{H}$ uniformly at random, and store $h(p)$ for all points $p \in P$.
  2. Given a query point q, compute $h(q)$, and see if we have $h(q) = h(p)$, for some p. (This lookup is constant time using standard hashing data structure.)
  3. More precisely, what we will show is this: if there is a point p with $\text{dist}(p, q) \leq r$, then with prob $> 1 - 1/n$ it will hash to $h(q)$, and for all the points p with $\text{dist}(p, q) > cr$ only $O(\ell)$ will, in expectation, hash to $h(q)$.

- So, we first show that we can approach the Ideal Prob. Gap even if we only have (r, cr, p_1, p_2)-LSH, where we only have $p_1 > p_2$.

- **Boosting.** We do the boosting in two steps. Intuitively, we will use logical AND to reduce p_2, and then logical OR over many copies to improve p_1.

  1. First, we just try to make p_2 small. It suffices to take k independent hash functions from $\mathcal{H}$, and hash each point $p \in P$ to a k-dim vector:

     $$h(p) = [h_1(p), h_2(p), \ldots, h_k(p)]$$

  2. Then, by the independence of $h_1, \ldots, h_k$, for any two points p and q, we have

     $$\text{dist}(p, q) \geq cr \implies Pr[h(p) = h(q)] \leq p_2^k$$

     So, we can drive down p_2 arbitrarily close to 0, by increasing k.

  3. But this does not help us boost p_1 towards 1. *In fact, this k-wise hash function maps two close points to the same vector with prob. only p_1^k.*

4. To improve p_1, we use multiple such families. Specifically, we choose ℓ independent copies of the above k-dim hash functions: $f_1, \ldots, f_\ell$ where each f_i is one of the k-dim hash function.
5. Then, if ℓ is sufficiently large and $\text{dist}(p,q) \leq r$, we get $f_i(p) = f_i(q)$, for some i with high prob. More specifically,

$$\begin{aligned} Pr[\exists i,\ f_i(p) = f_i(q)] &= 1 - Pr[\forall i,\ f_i(p) \neq f_i(q)] \\ &= 1 - Pr[f_i(p) \neq f_i(q)]^\ell \\ &\geq 1 - (1 - p_1^k)^\ell \end{aligned}$$

1.4 LSH Algorithm

1. Preprocessing.
   (a) Choose k, ℓ and hash functions $h_{1,1}, \ldots, h_{\ell,k}$ uniformly from $\mathcal{H}$
   (b) Construct ℓ hash tables.
   (c) For $i = 1, 2, \ldots, \ell$, store $f_i(p) = (h_{i,1}(p), \ldots, h_{i,k}(p))$ in the ith hash table
2. Query.
   (a) for $i = 1$ to ℓ do
      i. compute $f_i(q)$
      ii. scan all points p with $f_i(p) = f_i(q)$, and if anyone has $\text{dist}(p,q) \leq cr$, output p.

1.5 LSH Design Parameters and Analysis

- How should we choose the parameters k, ℓ?
  1. We choose k so that $p_2^k \approx 1/n$.
  2. Now write $p_1 = p_2^\rho$, for some $\rho < 1$
  3. This ρ determines the time/space bounds of LSH algorithm.
  4. We choose
  $$\ell \ \propto \ n^\rho \ln n$$
- Consider a query point q. By the linearity of expectation, for any i we have

$$E[p \mid \text{dist}(p,q) > cr \ \text{ and } \ f_i(p) = f_i(q)] \ \leq \ np_2^k \ \leq \ 1$$

1. Summing over all i, in expectation, there are $O(\ell)$ points in our data set which map to the same hash function as q for some i. We may need to examine these points to check if any of them is really within distance cr from q. This implies an overhead of $O(\ell)$ distance calculations in our query time.
2. On the other hand, if $\text{dist}(p, q) \leq r$, for some $p \in P$, then

$$\begin{aligned} Pr[\exists i, f_i(p) = f_i(q)] &\geq 1 - (1 - p_1^k)^\ell \\ &= 1 - (1 - p_2^{\rho k})^\ell \\ &= 1 - (1 - n^{-\rho})^\ell \\ &\approx 1 - e^{-\ell n^{-\rho}} \qquad \text{using } (1 - x) \leq e^{-x} \\ &= 1 - 1/n \qquad \text{using } \ell = n^\rho \ln n \end{aligned}$$

3. In summary, if there is a point p with $\text{dist}(p, q) \leq r$, then with prob $> 1 - 1/n$ we get $f_i(p) = f_i(q)$, for some $i = 1, 2, \ldots, \ell$. On the other hand, there are only $O(\ell)$ points with $\text{dist}(p, q) > cr$ for which we get $f_i(p) = f_i(q)$, for some i..

- **Space and Time Complexity Analysis.**

1. The space complexity is $O(\ell nk)$, since we have $O(\ell)$ hash tables, each with n points and for each point we store a k-dim hash vector.
2. Ignoring lower order terms, the space complexity is $O(n^{1+\rho})$.
3. The query time is $O(\ell k)$ for computing $f_i(q)$, for all $1 \leq i \leq \ell$. For each candidate close to q, we spend $O(d)$ time to compute the distance $\text{dist}(p, q)$.
4. Let x be the size of the output, the number of points at distance $\leq cr$ form q. Plus, in expectation, we examine $O(\ell)$ faraway points that we don't output.
5. So the total query time is $O(d(\ell+x+\ell k))$. This works out to $O(n^\rho(d+\log(n/p_2))+xd)$.
6. Ignoring lower order terms, the query time is $O(n^\rho)$.

- Example analysis for binary vector case.

1. Let us calculate the performance of LSH for the binary vectors example. Recall that ρ is chosen so that $p_1 = p_2^\rho$. Thus,

$$\rho \;=\; \log_{p_2} p_1 \;=\; \frac{\log p_1}{\log p_2} \;=\; \frac{-r/d}{-cr/d} \;=\; \frac{1}{c}$$

2. If we choose $c = 2$, we need $O(n^{1.5})$ memory to achieve $O(\sqrt{n})$ query time.
3. If we choose $c = 4$, we need $O(n^{1.25})$ memory to achieve $O(n^{1/4})$ query time.

1.6 References

1. Approximate Nearest Neighbor: Towards removing the curse of dimensionality. P. Indyk and R. and Motwani. In Proceedings of the Symposium on Theory of Computing (STOC), 1998.

2. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. Alexandr Andoni and Piotr Indyk. Communications of the ACM, Jan 2008.