CS 49/149: 21st Century Algorithms (Fall 2018): Lecture 12

Date: 23rd October, 2018

Topic: Approximate Near(est) Neighbors. Locality Sensitive Hashing

Scribe: Prantar Ghosh

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please email errors to prantar.ghosh.gr@dartmouth.edu.

1 The Nearest Neighbor Problem

The NEAREST NEIGHBOR (NN) problem is a standard problem in multiple areas of Computer Science. In this problem, we are given a data set S, which we can preprocess to answer queries of a certain type. The points in S can be thought to be lying in a metric space U, i.e. there is a notion of distance (given by a metric D) between any two points in S. So U can be assumed to be $\mathbb{R}^d$, where d is very large, say $d \approx 10^6$. The query that we have to answer is of the following form: Given a point q in the metric space U, we need to return the point in S that is closest to q. Formally, the problem is given as follows:

NEAREST NEIGHBOR (NN)
Input: Data Set $S \subseteq U$, $|S| = n$. Distance metric $D(x, y)$ $\forall x, y \in U$.
Query: Given $q \in U$, output $x \in S$ such that $x = \arg\min_{y \in S} D(y, q)$.

Now, suppose we consider the following relaxed version of the problem, which we call APPROXIMATE NEAREST NEIGHBOR (ANN). In this problem, given a query point q, we are allowed to return a point in S whose distance from q is within a constant factor c of the smallest distance from q to S. Formally, the problem is given by:

APPROXIMATE NEAREST NEIGHBOR (ANN)
Input: Data Set $S \subseteq U$, $|S| = n$. Distance metric $D(x, y)$ $\forall x, y \in U$. $c > 1$.
Query: Given $q \in U$, output $x \in S$ such that $D(x, q) \leq cD(y^*, q)$, where $y^* = \arg\min_{y \in S} D(y, q)$.

How do we go about solving this problem? Let's consider the NN problem again. The trivial algorithm to solve NN does $O(n)$ distance computations. For standard norms, a single distance computation takes time proportional to the dimension d and hence the query time of the trivial algorithm would be $O(nd)$. This is very inefficient since both n and d are large. We want to solve the problem making sublinear (in n) distance computations. While that seems extremely hard for the exact NN problem, the approximate version might enable us to achieve this query time.

What if d was small, say $d = 2$? Is the problem easy then? The answer is Yes. In that case, we can do a preprocessing by Voronoi decomposition which partitions the $\mathbb{R}^2$ plane into multiple regions, with each region R containing one point p_R from S such that p_R is the point closest to any point in R. So, for each query q, we only need to check which region it falls into and can answer the query in $O(1)$ time. One way to do this Voronoi decomposition is by using kd-trees, which is widely used for low dimensional data.

But what happens if d is large? Is the Voronoi decomposition still an efficient way to preprocess? The answer is No. The decomposition takes time exponential in d. And hence, for large d, the Voronoi partition is infeasible.

We present the Algorithm for ANN by Indyk, Motwani 1998.

Consider the following variation of the ANN Problem, which we call the APPROXIMATE NEAR NEIGHBOR problem.

APPROXIMATE NEAR NEIGHBOR
Input: Data Set $S \subseteq U$, $|S| = n$. Distance metric $D(x, y)$ $\forall x, y \in U$. Parameter R. $c > 1$.
Output: Given a query $q \in U$, do exactly one of the following:

(i) Return $x \in S$ such that $D(x, q) \leq cR$

(ii) Return NO, i.e. assert that there does not exist $x \in S$ such that $D(x, q) \leq R$.

Note that to solve the ANN problem, we need to find the smallest R for which APPROXIMATE NEAR NEIGHBOR outputs (i), i.e. returns some point. Hence, given an algorithm for APPROXIMATE NEAR NEIGHBOR, we can use it as a blackbox and solve the ANN problem by doing a binary search with the parameter R and incur a (negligible) factor of $O(\log D^*)$ in the runtime where D^* is the smallest distance from the query q to the data set S. So, in the next section, we focus on solving the APPROXIMATE NEAR NEIGHBOR problem and we look at a hashing technique that we use to solve it.

2 Locality Sensitive Hashing (LSH)

In standard hashing, we have the property that two distinct elements rarely collide. But here, we consider a type of hashing such that if two elements are "close", then they *will* collide with high probability, and if they are "far away", then they do not collide with high probability. This is useful for ANN because when a query comes, we can hash it, and if it collides with some point, then there's high likelihood that that point is close to the query.

Definition 1 (Locality Sensitive Hashing (LSH))**.** A family H of hash functions ($h : U \to [r]$ for a metric space U with distance metric D and some range r) is (R, cR, p_1, p_2)-LSH if the following hold:

- $\forall x, y \in U$ with $D(x, y) \leq R$, we have $\mathbf{Pr}_{h \in H}[h(x) = h(y)] \geq p_1$.
- $\forall x, y \in U$ with $D(x, y) > cR$, we have $\mathbf{Pr}_{h \in H}[h(x) = h(y)] \leq p_2$.

Also, we have $p_1 > p_2$.

For an (R, cR, p_1, p_2)-LSH hash family, its quality is measured by the parameter $\rho := \frac{\ln p_1}{\ln p_2}$ (We shall see in Section 2.1 why it is defined this way). Note that since $p_1 > p_2$, $\rho < 1$ (We shall also see later why we need this).

Example

Let us the consider the following example of an LSH.

$U = \{0, 1\}^n$. The family H of hash functions is given by $H = \{h_1, \ldots, h_n\}$ with $h_i : U \to \{0, 1\}$ and $h_i(x) = x_i$. $D(x, y)$ is the Hamming distance between x and y, denoted by $||x - y||_1$.

Then, for all $x, y \in U$,

$$\begin{aligned}\mathbf{Pr}_{h\in H}[h(x) = h(y)] &= \mathbf{Pr}[h_i \text{ is chosen s.t. } x_i = y_i] \\ &= \frac{\text{No. of coordinates in which } x \text{ and } y \text{ agree}}{n} \\ &= \frac{n - D(x,y)}{n} \\ &= 1 - \frac{D(x,y)}{n}.\end{aligned}$$

Hence, by definition, this family H is $(R, cR, 1-\frac{R}{n}, 1-\frac{cR}{n})$-LSH. Also, $1-\frac{R}{n} > 1-\frac{cR}{n}$ is satisfied since $c > 1$.

We note that for an $(R, cR, 1-\frac{R}{n}, 1-\frac{cR}{n})$-LSH hash family, $\rho = \dfrac{\ln(1-\frac{R}{n})}{\ln(1-\frac{cR}{n})} \approx \dfrac{-\frac{R}{n}}{-\frac{cR}{n}} = \frac{1}{c}$, assuming that $R << n$.

2.1 Solving APPROXIMATE NEAR NEIGHBOR using LSH

First, given an (R, cR, p_1, p_2)-LSH with constant probabilities p_1 and p_2, we want to boost them to vanishingly large and small respectively. We do that in two steps:

STEP 1 OF PROBABILITY BOOST
Define a new hash function $g(x) = (h_1(x), \ldots, h_k(x))$ (k is to be set in the analysis), where each h_i is a randomly and independently drawn function from the (R, cR, p_1, p_2)-LSH hash family H.

As a result, we make the following observations.

Observation 1. If $D(x,y) > cR$, then $\mathbf{Pr}[g(x) = g(y)] = \mathbf{Pr}[h_i(x) = h_i(y)\ \forall i = 1, \ldots, k] \leq p_2^k$. This follows from definition of (R, cR, p_1, p_2)-LSH and from the fact that h_i's are drawn independently. We want this probability p_2^k to be at most $1/n$. Hence, we set $k = \frac{\ln n}{\ln \frac{1}{p_2}}$.

Observation 2. If $D(x,y) \leq R$, then, by definition of (R, cR, p_1, p_2)-LSH and since h_i's are drawn independently, we get $\mathbf{Pr}[g(x) = g(y)] \geq p_1^k = p_1^{\frac{\ln n}{\ln \frac{1}{p_2}}} = n^{\frac{\ln p_1}{\ln \frac{1}{p_2}}} = n^{-\frac{\ln p_1}{\ln p_2}} = n^{-\rho}$.

So, we now see the reason behind the definition of ρ. However, the probability $n^{-\rho}$ is very small for constant ρ, and we would like to boost it up further. We take care of this in Step 2.

Now, we try to use the hash function g for the APPROXIMATE NEAR NEIGHBOR problem. We would like to keep a hash table T and store $x \in S$ in the slot $T[g(x)]$. That way, we ensure with high probability that points within a distance of R are hashed to same slot, while the ones which are at least cR distance apart are hashed to different slots. But then, the size of the hash table would have to be the range of g, which is n^k since the range r of each h_i is taken to be n. The space requirement of n^k is infeasible as k can be $O(\log n)$. But note that the number of non-empty slots in the hash table will be at most $|S| = n$. So, we use "standard hashing" to maintain this hash table in a space efficient way, still accessing the slot for each x in $O(1)$ time. Let us recall what the standard hashing algorithm does.

STANDARD HASHING
Given a universe U and a static dictionary $D \subseteq U$ with $|D| = s$, the standard hashing algorithm answers queries of the form "Does $q \in D$?" (and hence accesses the slot for q in the hash table) in

- $O(1)$ time
- $O(s)$ space
- $O(s)$ preprocessing time.

In our case, the universe is the set of all possible n^k slots, and the dictionary is the set of all non-empty slots, i.e. $\{g(x) : x \in S\}$. Thus, using standard hashing, given $x \in S$, we can access the $g(x)$th slot in $O(1)$ time, maintaining the hash table using $O(n)$ space and $O(n)$ preprocessing time.

We proceed to describe Step 2. We want the "close" points to collide with high probability, instead of with probability only $n^{-\rho}$. For achieving this, the idea is to run multiple independent trials of Step 1, and it is enough to have at least one of them succeed. This is because the only bad case is when all the trials fail, and that happens with very low probability. So, we do the following:

STEP 2 OF PROBABILITY BOOST
Repeat Step 1 for $L = 10n^{\rho}$ times, i.e.

- Choose L independent copies $g^{(1)}(x), \ldots, g^{(L)}(x)$ for each $x \in S$.
- Keep L different hash tables $T_1, \ldots, T_L$ s.t. for each $i \in [L]$, T_i is the hash table that $g^{(i)}$ hashes to.

Hence, we use $O(nL)$ space and $O(nL)$ preprocessing time.

We now show that as a result of this step, the probability of collision of "close" points in at least one T_i is high.

Claim 1. Given $D(x, y) \leq R$, $\mathbf{Pr}[\exists i \in [L] : g^{(i)}(x) = g^{(i)}(y)] \geq 1 - e^{-10}$.

Proof. Let $x, y \in U$ have $D(x, y) \leq R$. Then,

$$\begin{aligned}
\mathbf{Pr}[\exists i \in [L] : g^{(i)}(x) = g^{(i)}(y)] &= 1 - \mathbf{Pr}[\forall i \in [L] : g^{(i)}(x) \neq g^{(i)}(y)] \\
&\geq 1 - (1 - \frac{1}{n^{\rho}})^{L} && \text{by Obs. 2 \& independence of } g^{(i)}\text{'s} \\
&= 1 - (1 - \frac{1}{n^{\rho}})^{10n^{\rho}} \\
&\geq 1 - e^{-10}.
\end{aligned}$$

□

However, similarly the probability of collision of "far away" points also increases as we now have multiple hash tables. We handle it in our algorithm. We are now ready to complete our algorithm for APPROXIMATE NEAR NEIGHBOR.

APPROXIMATE NEAR NEIGHBOR ALGORITHM

Given a query q,

1. Compute $g^{(1)}(q), \ldots, g^{(L)}(q)$.
2. Go to corresponding locations in T_i for each $i \in [L]$.
3. Keep collecting points from the tables till you get $4L$ points (or till you exhaust the slots).
4. Check if any of these $\leq 4L$ points are $\leq cR$ distance away from q.
   - If yes, then return it.
   - Otherwise, return NO, i.e. assert there does not exist $x \in S$ such that $D(x, q) \leq R$.

Query time: Steps 1,2 and 3 clearly take $O(L)$ time. Step 4 requires $O(L)$ distance computations. Hence, the query time of this algorithm is the time taken by $O(L) = O(n^\rho)$ distance computations.

Analysis

We prove that the algorithm returns the correct answer for the APPROXIMATE NEAR NEIGHBOR problem with high probability. We see that if the algorithm actually returns a point, it is definitely within a distance of at most cR from q. So the only way the algorithm can be incorrect is when there is a point $x \in S$ such that $D(x, q) \leq R$, but it still returns NO. This may happen in two ways:

(a) x is not hashed with q by any g, i.e. $\forall i \in [L],\ g^{(i)}(x) \neq g^{(i)}(q)$.

(b) There are $4L$ far away points which have hashed with q in some table, i.e. there are $4L$ points z with $D(z, q) > cR$ such that $g^{(i)}(q) = g^{(i)}(z)$ for some $i \in [L]$.

Note that if neither (a) nor (b) occurs, then the algorithm correctly returns some point y such that $D(y, q) \leq cR$.

So we calculate the probability of either (a) or (b) happening. By Claim 1, the complement of event (a) has probability at least $1 - e^{-10}$. Hence, $\mathbf{Pr}[(a)] \leq e^{-10}$.

Now, consider event (b). We call a point z with $D(z, q) > cR$ as a "far away" point. We fix a far away point z and an $i \in [L]$. Then, by Observation 1, $\mathbf{Pr}[g^{(i)}(q) = g^{(i)}(z)] \leq \frac{1}{n}$.

For $i \in [L]$, let N_i be a random variable that denotes the number of far away points z with $g^{(i)}(z) = g^{(i)}(q)$. It follows that $\mathbb{E}[N_i] \leq \dfrac{\text{Total number of far away points}}{n} \leq 1$.

Hence, $\mathbb{E}[\text{No. of far away points that get hashed with } q \text{ by } g^{(i)} \text{ for some } i \in L] \leq \sum_{i=1}^{L} \mathbb{E}[N(i)] \leq L$. So, by Markov inequality, $\mathbf{Pr}[4L \text{ far away points collide with } q] \leq 1/4$. Hence, $\mathbf{Pr}[(b)] \leq 1/4$. This implies that $\mathbf{Pr}[(a) \text{ or } (b)] \leq e^{-10} + 1/4 < 1/2$ and so the algorithm has error probability $< 1/2$. Now, we can use the standard trick of running the algorithm independently $\log 1/\delta$ times to make the error $\leq \delta$.

Space and time complexity: To get error $\leq \delta$, this algorithm takes $O(nL \log 1/\delta) = O(n^{1+\rho} \log 1/\delta)$ space and preprocessing time. The total query time is $O(n^\rho \log 1/\delta)$ distance computations. Now we use the fact that $\rho < 1$ and argue that the number of distance computations we make is sublinear in n and so is an improvement over the trivial algorithm. Recall that for the $(R, cR, 1 - \frac{R}{n}, 1 -$

$\frac{cR}{n}$)-LSH that we constructed for n-length binary strings, $\rho = \frac{1}{c}$, where c is the approximation factor we are aiming for. We can expect ρ to be inversely proportional to c for the LSH we use as well, giving a natural trade-off between the approximation factor and the query time.

2.2 Min-hash

In this section, we see another example of an LSH family.

Consider the following setting. Each point in set S (can be thought of as a set of documents) is a (multi)subset of a huge set W (a set of words). Hence, the universe U is the power set 2^W, and $S \subseteq U$. The distance metric between $A, B \in S$ is given by $D(A, B) = 1 - J(A, B)$, where $J(A, B)$ is the "Jaccard similarity" between A & B, given by $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$. So when a query $q \in U$ comes, we want to find the subset in S that is closest to q with respect to the Jaccard similarity.

We want an LSH for this metric. It is obtained by using "min-hash", which we define next. Let H be a family of hash functions where each function $h \in H$ is from 2^W to $[W]$. For a permutation π of W and $A \subseteq W$, define $h_\pi(A) = \min_{a \in A} \pi(a)$. Define $H = \{h_\pi : \pi \text{ is a permutation of } W\}$.

To prove that H is an LSH, we need to find $\mathbf{Pr}_{h_\pi \in H}[h_\pi(A) = h_\pi(B)]$. Note that this is equal to the probability that for a permutation π of W picked uniformly at random, $\min_{a \in A} \pi(a) = \min_{b \in B} \pi(b)$. To find this, we only need to consider the numbers assigned by π to the elements in $A \cup B$. Since every element in W is assigned a unique number, the minimum among numbers assigned to elements in A will be equal to that in B if and only if some element in $A \cap B$ is assigned the minimum number among the elements in $A \cup B$. Since the permutation π is drawn uniformly at random, it is equally likely to assign any element in $A \cup B$ the minimum number. Thus, $\mathbf{Pr}_{h_\pi \in H}[h_\pi(A) = h_\pi(B)] = \frac{|A \cap B|}{|A \cup B|} = J(A, B) = 1 - D(A, B)$. Hence, by definition, the hash family H is $(R, cR, 1-R, 1-cR)$-LSH.