CSE 547/Stat 548: Machine Learning for Big Data **Lecture**

Locality sensitive hashing

Instructor: Sham Kakade

1 SK notes

- quick sort (check)
- 'physical' sorting
- Voronoi diagram

2 The Nearest Neighbor Problem

We have n points, $D = \{p_1, \ldots p_n\} \subset \mathcal{X}$ (possibly in d dimensions or possibly more abstract). We also have a distance function $d(p, p')$ on our points. Given some new point q we would like to find either: 1) an exact nearest neighbor or 2) a point in our $p \in D$ that is "close" to q.

Voronoi diagrams

One natural data structure is the Voronoi diagram. With each point, construct the region that contains all it's closest neighbors:

$$R_i := \{x | x \in \mathcal{X} \text{ s.t. } d(x, p_i) \leq d(x, p_j) \text{ for } i \neq j\}$$

For Euclidean distance, these regions are convex. This data-structure takes $n^{d/2}$ space; $O(n^{d/2} + n \log n)$ to construct the data-structure (for search, based Kirkpatrick's point location data-structure); $O(n^d \log n)$ time to query a point.

For constant dimension, the scaling with n for the query time is great. For $d = 2$, it is a great algorithm. For high dimensions, this trade off is harsh.

What do we want?

Naively, given a new point q finding its nearest neighbor would take us $O(n)$ time. If we are doing multiple nearest neighbor queries, we may hope for a data-structure in which the time to find a nearest neighbor is scaling *sublinear* in n, maybe even $O(\log n)$. And we would like to 1) construct such a data-structure quickly and 2) have this data-structure not take up too much memory.

3 Data-structure Review: Hash functions and Hash Tables

Suppose that $D = (p, v)$ is a set of names or a set of images, and that we also have associated information (e.g. metadata) about these documents or the images, that we want to store in memory in a manner that is easy for us to "recall" this information. Think of this associated information as being quite large (in terms of memory) Given some $p \in D$, how we would we construct a data-structure which makes it easy for us to retrieve the associated information v. One manner is to sort all the names (if the set of p could be ordered), and then associate with each element in the sorted order a "bucket", which will contain all the information of the i-th person in sorted order. Then it takes us $O(\log(n))$ to recall this information for some person $q \in D$: given q we just go find the corresponding bucket (in sorted order).

Another idea, which works extremely well, is to use a *hash table*. The idea is as follows: suppose we have a "hash" function g which of the form $g : \mathcal{X} \to \{0,1\}^k$, e.g. it maps names to binary strings. Suppose this function is *cheap* to compute, which can often be achieved easily. For example, map p to a binary string and then, for each coordinate function $g_i(\cdot)$, we could use some modulo arithmetic function. Let us come back to how large k should be.

Now let us index our buckets with binary strings (think of these like locations in memory). Our hash table is going to be set of these buckets: a set of the keys (e.g. the binary strings) and associated values (the information we put in the corresponding bucket). So recall is easy: we take p, apply our hash function g to get a key (easy to compute), then find the bucket corresponding to this key (think of the key, i.e. $g(p)$, like a pointer to the bucket), and we have our info.

When does this work? We need k to be large enough so that we do not get "collisions". It would be bad if we accidentally put the information of two different people in the same bucket. Suppose our hash function was 'random', in g is random function (though it is fixed after we have constructed it). What is the chance that any p and p' collide? It is $1/2^k$. And what is the chance that any two p's in our dataset collie? It is less than $n^2/2^k$ (as there are $O(n^2)$ pairs). Now it is easy to see that $O(\text{poly} \log n)$ suffices to avoid collisions, where the probability of a collision of *any* two elements is at most $1/n^{\text{constant}}$ (depending on the polynomial).

4 Locality Sensitive Hashing

Let us return to our nearest neighbor problem. What type of function g might we want? Ideally, a function which just points us to the index of nearest neighbor would be ideal! This is clearly too ambitious for points in a general metric space. Instead, suppose our function could point us to a bucket which only contained nearby points to q, say points R-close to q. Then we could just brute force search in this bucket, or maybe just return any one of the points (if we knew the bucket only contained close by points). Suppose we want the following approximate guarantee: If there exists a point that is R close to q, then we would like to 'quickly' return a point that is cR close to q, where $c > 1$. Ideally we would like $c = 1$, though this is too stringent, as we would like to allow the algorithm some wiggle room (well, we don't know how to do that efficiently, at least with these techniques).

Suppose we could randomly construct some binary functions, where h is a random function from $\mathcal{X}$ to $\{0, \dots m-1\}$ (we can consider m-array hash functions, not just binary), with following properties: For example, h may be a random projection or a random component of p. Let us suppose that

1. If $d(p, p') < R$, then $\Pr(h(p) = h(p')) \geq P_1$.
2. If $d(p, p') > cR$, then $\Pr(h(p) = h(p')) \leq P_2$.
3. Assume $P_2 < P_1$.

Let's look at an example. Suppose our points are binary strings of length d. And suppose our distance function $d(p, p')$ is the Hamming distance, namely, the number of entries in which p and p' disagree. Now let us take a naive stab at constructing a function randomly. Suppose that we construct each coordinate function of g randomly: we pick say some random index $a_i \in \{1, \dots d\}$, and we say $g_i(p) = h_{a_i}$, i.e. it is the the a_i-th coordinate of p. We do this k times to construct the function g. Now, suppose $d(p, p') = \|p - p'\|_1 \leq R$ (so the two points are near to each other), we have that $P_1 \geq 1 - R/d$. By doing this k times,

$$\Pr(g(p) = g(p')) \geq P_1^k = \exp(-k \log 1/P_1)$$

Of course, $k = 1$ would be ideal for placing nearby points in the same bucket, though this also place points far away into the same bucket.

Now suppose we have p and p' that are 'far' away. Say $d(p, p') > cR$ for some c. We would like far away points to not end up in the same bucket. Here, we see that $P_2 \leq 1 - cR/d$. So for these far away points we have:

$$\Pr(g(p) = g(p')) \leq P_2^k = \exp(-k \log 1/P_1)$$

Now the expected number of collisions will be:

$$n \exp(-k \log 1/P_2)$$

and, by $k = \log n / \log(1/P_2)$, this expected number is a constant.

However, for this $k = \log n / \log(1/P_2)$, the chance that a nearby point nearby point p' that is R close to p fails to land in this bucket is then:

$$\Pr(g(p) = g(p')) \geq \exp(-k \log 1/P_1) \leq n^{-\log 1/P_1 / \log 1/P_2}$$

which could be very small. However, if we do this $1/\Pr(g(p) = g(p'))$ such hash functions, then we are in good shape! Since one is expected to succeed.

The algorithm is as follows: For the data-structure construction, we:

- we construct L hash functions $g_1, \ldots g_L$ each of length k.
- We place each point $p \in D$ to L buckets (as specified by g_1 to g_L).

For recall we:

- check each bucket $g_i(q)$ in order. Step when we find a cR nearest neighbor.
- If after $3L$ points have been checked (and we have not found a cR point), then we give up posit that there does not exist any $p \in D$ which is R close to q.

Note that we do *not* check every point in all $g_1(q)$ to $g_L(q)$ buckets. This is due to that all the points may be cR close to q.

Note the following theorem is not limited to these bit-wise hash functions. We just need to satisfy our two conditions above.

Theorem 4.1. *Suppose the two numbered conditions hold for the above. Suppose $k = \log n / \log(1/P_2)$, and $L = n^\rho$, where $\rho = \log 1/P_1 / \log 1/P_2$. Then (with probability greater than $1 - 1/poly(n)$, our query time is $O(n^\rho)$. The space of our datastructure is $O(n^{1+\rho})$ and the construction time is $O(n^{1+\rho})$.*

Proof. For the far away points, we have already shown that the for each g, only a constant number of collisions will occur. However, for the close by points, there is $n^{-\log 1/P_1 / \log 1/P_2} = n^{-\rho}$ probability our nearby point (if it exists) falls into the same bucket as q. So with $L = n^\rho$ hash functions, one of these buckets will contain this nearby point. So for the query time, there are at most $L = n^\rho$ errors, so we just keep searching until we find a point cR close or we give up. If we give up, we are guaranteed (with high probability), that there is no point that in D that R close to q. □

4.1 Example: LSH for strings

Let's return to our example, where our points are binary strings of length d, with the Hamming distance. Here, we said $P_1 \geq 1 - R/d$. And $P_2 \leq 1 - cR/d$. So our $\rho = \log 1/P_1 / \log 1/P_2 \approx 1/c < 1$. So we have a *sublinear* query algorithm for approximate search.

4.2 Example: LSH for vectors

For vectors, we wil construct each component function g_i using a one dimensional random projection. Specifically, we will consider the projection $zz = \frac{1}{\sqrt{d}} v \cdot p$, with v being sampled from a standard normal $N(0, \text{Identity})$. Note that this makes the random quantity $\frac{1}{\sqrt{d}} v \cdot p$ be Gaussian distributed. Specifically, $z/\|p\|$ has a distribution of $N(0,1)$, and, more relevant, is that $\|z\|^2/\|p\|^2$ has a χ^2 distribution. We be concerned with the distribution of $\|z\|^2$.

Example: Euclidean distance

Suppose each p lives in unit norm ball. We might want to recall based on cosine similarity (e.g. find a point closest in angle). This is identical to Euclidean distance if we scale each point to be unit norm. Now instead of hashing ot $\{0,1\}$ we could hash to an integer, and we still think of this string as corresponding to a bucket. For example:

$$g_i(p) = \lfloor \frac{u_i \cdot p}{2R} \rfloor$$

where $u_i = \frac{1}{\sqrt{d}} v$, with being sampled from a standard normal $N(0, \text{Identity})$. Note that $u_i p$ is a random projection. For two nearby points, R close in distance, to hash into the same bucket, we would like $u_i p - u_i p' = u_i(p - p')$ to be small, which we view as a random projection of $p - p'$. For this, we desire that $\|u_i(p - p')\| \leq \|p - p'\|$, which implies we seek $\|u_i(p-p')\|^2 \leq R^2$. This will happen with constant probability, due to this having a χ^2-distribution. Furthermore if their projections are R close, then there is a $1/2$ probability they end up in the same bucket (of width $2R$), since even if one point is on the boundary there is even odds as which side the other point will land.

For points cR apart, we would like $\|u_i p - u_i p'\| \leq (c-1)R$, which implies $g_i(p) \neq g_i(p')$. This will happen if $\|u_i p - u_i p'\| \leq \frac{c-1}{c}\|p - p'\|$, since $\|p - p'\| \leq cR$. So if the event

$$\|u_i p - u_i p'\|^2 \leq (1 - \frac{1}{c})^2 \|p - p'\|^2 \leq (1 - \frac{1}{c})\|p - p'\|^2$$

occurs, then p and p' will not hash into the same bucket. Hence, we can take P_2 to be an upper bound on the probability that the following event occurs:

$$\|u_i p - u_i p'\|^2 \geq (1 - \frac{1}{c})\|p - p'\|^2$$

By JL, we can take $\epsilon = 1/c$. And we have that $P_2 \leq 2\exp(\text{constant}/c^2)$.

Thus we have shown that $\rho = \text{constant}/c^2$.