Advanced Algorithms

Lecture 15: Locality Sensitive Hashing

Lecturer: Nathan Klein

1 Locality Sensitive Hashing

1.1 Introduction and Recap

Last time we discussed how hashing can be used to answer membership queries in $O(1)$ time in the word RAM model. In particular, we introduced the 2-universal property, showed a 2-universal distribution over hash functions, and used these to construct "perfect" collision-free hash tables in the static setting.

For the simple family of hash functions we discussed, $h(x)$ had no particular meaning, and in the ideal case we would let $h(x)$ be a uniformly random output that is independent of x. In applications of hashing in cryptography, such as storing passwords, this is also the idealized form of a hash function. In this lecture, we will instead study hash functions where the output is related to x, which will help us solve the *nearest neighbor* problem. In particular, we will design a distribution over hash functions which ensures that if x, y are "close" with respect to some distance measure, then $h(x) = h(y)$ with good probability, and otherwise we are likely to have $h(x) \neq h(y)$.

1.2 Nearest Neighbor Problem

Let $S \subseteq \mathbb{R}^d$ be a collection of points. In most applications, the points live in a high dimensional space. For example, maybe this is a set of movies, and the dimensions encode things like the movie's genre, director, lead actors, and so on. We now are given a distance function $d : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ which gives the distance between two points in some manner. For this lecture we will look at ℓ_1 distance, where remember the ℓ_1 distance between $x, y \in \mathbb{R}^d$ is $\|x - y\|_1 = \sum_{i=1}^d |x_i - y_i|$.

Now, the nearest neighbor problem is simple: given a point $x \in \mathbb{R}^d$, find the closest point in S to x, i.e., find the $y \in S$ which minimizes $d(x, y)$ among all points in S.

The problem is easy if you don't care about running time: just check $d(x, y)$ for all possible n points $y \in S$, for a running time of $O(n \cdot d)$. But this is the "big data" portion of the course, so we're not happy with this: we want something that works well even when n is massive.

How can we do better? First, suppose that there was only one dimension. Then this problem is easy: store the numbers in sorted order, then use binary search to find the closest point, which will take $O(\log n)$ time and the data structure will use $O(n)$ memory.

There is an analog of this in higher dimensions called a k-d tree, but for nearest neighbor search these trees run into issues when the number of dimensions grows and will have a worst-case query time of $O(n \cdot d)$ when the number of dimensions grows to $\Omega(\log n)$. The idea behind a k-d tree is to have the tree split on one dimension at a time. While this is fine for searching for a particular point, it's not at all clear which branch to follow in the tree for nearest neighbor search, as the closest point could be far in this dimension but close in other dimensions.

1.3 Approximate Nearest Neighbors

Just like when dealing with NP-Hardness, it can be useful to relax the requirement of optimality to achieve better running times in big data problems. So, we will now consider a relaxation of the problem which returns a point at most α times the distance of the closest point.

For simplicity in the next section, notice that it suffices to solve the following (slightly easier) radius-constrained problem which will fix a radius r and an approximation factor α. Here, given a query x, if there exists a point with distance at most r to x, we must return a point with distance at most $\alpha \cdot r$ to x. Otherwise we can return nothing.

Let $\delta > 0$ be the minimum distance between non-equal points in S and Δ the maximum distance. We can think of $\frac{\Delta}{\delta}$ as the *precision* of the model. In practice, this quantity is often not very big. In the movie example, there may be (say) 100 possible scales of relatedness, with a score of 1 being an extremely similar movie and 100 extremely different. But getting further granularity may not be so important.

Lemma 1.1. *Suppose there is an α approximation for the radius constrained nearest neighbors problem. Then for any $\epsilon > 0$ one can obtain a $\alpha(1+\epsilon)$ approximation for the nearest neighbors problem by running $O(\log_{1+\epsilon}(\Delta/\delta))$ copies of the radius constrained algorithm in parallel.*

Proof. Solve the radius-constrained problem with approximation factor α for radii $\delta, (1+\epsilon)\delta, (1+\epsilon)^2\delta, \dots, \Delta$. Create data structures for all of the above radii-constrained problems, which is $O(\log_{1+\epsilon}(\Delta/\delta)) = O(\log(\Delta/\delta)/\log(1+\epsilon))$ copies. Upon getting a point $x \in \mathbb{R}^d$, run a query in each one and return the closest obtained point.

Suppose the closest point had distance k, where $\delta(1+\epsilon)^c \le k \le \delta(1+\epsilon)^{c+1}$ to x. Then we return a point with distance at most $\alpha\delta(1+\epsilon)^{c+1} \le \alpha(1+\epsilon)k$. $\square$

1.4 Locality Sensitive Hashing: LSH

This brings us to LSH, which is a tool to solve the radius-constraint nearest-neighbor problem.

Definition 1.2 (Locality Sensitive Hashing)**.** *Given a radius r, a distance metric d, and an approximation factor α, as well as two probabilities p_{close}, p_{far}, a distribution over hash functions $h : \mathbb{R}^n \to \mathbb{Z}$ is ρ-locality sensitive (with the above parameters) if:*

$$\mathbb{P}\left[h(x) = h(y)\right] \ge p_{close} \qquad \forall x, y : d(x,y) \le r \tag{1}$$

$$\mathbb{P}\left[h(x) = h(y)\right] \le p_{far} \qquad \forall x, y : d(x,y) \ge \alpha \cdot r \tag{2}$$

for $p_{close} > p_{far}$ with $p_{close} = p_{far}^{\rho}$ for $\rho < 1$. We will call the randomly sampled function a ρ-locality sensitive hash function or an LSH.

Let's construct an LSH for $d(x,y) = \|x-y\|_1$ over $\{0,1\}^d$ for now. In the homework you will work with more general domains and distance functions, but in this case there is a natural LSH scheme. It is simply to pick a uniformly random integer $i \in [d]$, and let $h_i(x) = x_i$ be the hash function which outputs the ith bit of x.

What parameters do we get? Well, if $\|x-y\|_1 \le r$, then the probability $h_i(x) = h_i(y)$ over the randomness of i is at least $1 - \frac{r}{d} \approx e^{-r/d}$, where we use that for small x (which we assume will be the case) $e^{-x} \approx 1 - x$.[1] Otherwise, it is at most $1 - \frac{\alpha r}{d} \le e^{-\alpha r/d}$ using the inequality $1 - x \le e^{-x}$.

[1] If you want to be more precise, you can use $1 - x \ge e^{-x/2}$ for $x \in [0, 1.59]$ which will lead to a $\frac{2}{\alpha}$-LSH.

So, as long as r is relatively small compared to d, we get an LSH over $\{0,1\}^d$ for ℓ_1, and any approximation factor α with $p_{close} = e^{-r/d}$ and $p_{far} = e^{-\alpha r/d}$, so $\rho = \frac{1}{\alpha}$. Simple enough! On the homework you will see how to construct LSH schemes for other distances and domains.

1.5 Solving Radius-Constrained Nearest Neighbors with LSH

Why is this useful? Remember, all we want to do is this: given a set S and a point x, figure out if there's a point within radius r (or αr) of x in S. For intuition, suppose that $p_{close} \approx 1$ and $p_{far} \approx 0$. But then our algorithm can just be this: run every point in S through an LSH and record the outputs in a hash table. Then given x, compute $h(x)$ and check if there is a collision with anything in S, which we can do in time $O(1)$. If so, we output the point y with $h(x) = h(y)$. Otherwise, we report that there is no point that is close to x in S, and this algorithm works with high probability over the choice of hash function from the locality sensitive family.

In reality, it will not be the case that $p_{close} \approx 1$ and $p_{far} \approx 0$. But at least we know that $p_{close} > p_{far}$. The point now will be to run many copies of this LSH to "boost" these probabilities to arbitrarily close to 1 and 0. In particular, take k independent copies of the ρ-LSH to create a hash function $h^* : U \to [k]$. Now, the probability that $h^*(x) = h^*(y)$ when $d(x,y) \geq \alpha \cdot r$ is at most p_{far}^k. So by choosing k sufficiently large, we will be able to send p_{far} to 0.

Unfortunately, this will also send p_{close} to 0, which here is now at least p_{close}^k since we have a probability of p_{close} of matching on each coordinate independently. So instead, we will now take ℓ independent copies of our hash function h^*. And now the algorithm is simple. Initialize ℓ independent hash functions $h_1^*, \ldots, h_\ell^*$, where each is the above k independent copies of our ρ-LSH. Using these, create ℓ hash tables $T_1, \ldots, T_\ell$, where in hash table i, we will store the value of $h_i^*(x)$ for all $x \in S$.

Now, to query a point x, we will search through all of the ℓ hash tables and check if $h_i^*(x) = h_i^*(y)$ for all ℓ and any $y \in S$. This will take time $O(\ell \cdot k)$. If we find any collisions, we will check if $d(x,y) \leq \alpha \cdot r$, and if so, we will output y as a nearest neighbor. Each such test will take time $O(d)$. Our space will be that of the ρ-LSH (which in the running example is $O(1)$) times $\ell \cdot k$, so essentially $\ell \cdot k$.

Lemma 1.3. *Given a ρ-LSH, there is an algorithm to solve the radius-constrained nearest neighbors problem which finds a point with probability $1 - \frac{1}{n}$ if it exists in query time and space $n^{1+\rho}$.*

Proof. We need to pick ℓ and k. We will pick k so that $p_{far}^k = \frac{1}{n}$, so treating p_{far} as a constant this is $\Theta(\log(n))$. Then we will choose $\ell = n^\rho \ln(n)$. In this way, if the query point x is at least $\alpha \cdot r$ away from all points, the expected number of collisions is at most $n \cdot p_{far}^k \leq 1$, so there are few points to query to check if $d(x,y) \leq \alpha \cdot r$. In particular there are at most $O(\ell)$ points in expectation to check. It's worth noting that since we always check, we will never report a false nearest neighbor.

On the other hand, if there is a point $y \in S$ within $\alpha \cdot r$ of x, we want to find it. So we want to prove that for some $i \in [\ell]$, $h_i^*(x) = h_i^*(y)$ with good probability. For each i, we get a collision with probability at least p_{close} for this point y. So,

$$\begin{aligned}
\mathbb{P}\left[\exists i : h_i^*(x) = h_i^*(y)\right] &= 1 - (1 - p_{close}^k)^\ell \\
&= 1 - (1 - p_{far}^{\rho k})^\ell && \text{Definition of } \rho\text{-LSH} \\
&= 1 - (1 - n^{-\rho})^\ell && \text{By our choice of } k \\
&\geq 1 - e^{n^{-\rho}\ell} && \text{Using } 1 - x \leq e^{-x} \\
&= 1 - e^{n^{-\rho}n^{\rho}\ln(n)} = 1 - \frac{1}{n}
\end{aligned}$$

Putting things together, our space complexity is $O(n \cdot \ell \cdot k) = O(n^{1+\rho}\log^2(n))$ (assuming p_{far} is a constant). We will make $O(\ell \cdot k)$ hash calls, which is $O(n^\rho \log^2(n)$. Then in expectation, we will have $O(\ell)$ queries associated with far away points, which will take time $O(d \cdot \ell) = O(dn^\rho \log(n))$. So, ignoring logs, we have a scheme with query time $O(dn^\rho)$ and memory $O(n^\rho)$. $\square$

Plugging this in for our LSH on $\{0,1\}^d$ under ℓ_1 distance, we obtain an α approximation for nearest neighbors using query time $O(dn^{\frac{1}{\alpha}})$ and space $n^{1+\frac{1}{\alpha}}$. This is pretty good if you're OK with, say, a 5-approximation: we are reducing the query time from about dn to about $dn^{1/5}$. On a data set with a billion elements and $d = 1000$ this would correspond to a massive decrease in running time for a query, from trillions of operations to under a million. That's a runtime improvement on a standard computer from around 15 minutes (useless for most applications) to a millisecond (very reasonable). And this is a reasonable input size for nearest neighbors: eBay has around a billion items listed, and when you look at an item on eBay, it will show you related items using a nearest neighbor search.