CSE 584A Class 23

Jeremy Buhler

April 28, 2016

1 Near-Neighbor Search in High-Dimensional Space

- Let S be a collection of "items."
- Let $d(\cdot, \cdot)$ be a metric distance on pairs of items.
- Given a query item q, find all items in S that are close to q. More specifically, find all items within some distance bound R of q.
- This is the "R-near neighbor" problem.

Applications?

- points in Euclidean space
- feature vectors with, e.g., L_1 or L_2 metric.
- bit vectors, or vectors over any discrete alphabet, with Hamming distance (e.g. DNA!)
- The last implies search with "at most k mismatches" (not indels)

You may have heard of some index data structures for solving the Euclidean version of the problem.

- Example: k-d tree
- In the plane ($k = 2$), this structure is very effective at rapidly eliminating items far from a given query.
- How does it scale with dimension, though?
- The higher the dimension, the more ways an item can be "near" a given query.
- Near-neighbor search algorithms that rapidly eliminate irrelevant items in low dimensions tend to pull in ever more such items as the dimensionality increases.
- Cost tends to rise *exponentially* with dimension.
- In the limit, cost of near-neighbor search can be nearly linear in collection size S (worst case!).
- This phenomenon is called the *curse of dimensionality.*

- *Example*: Unless you index many more than 2^k items, k-d trees don't provide much cost savings over linear search.

If the dimensionality of your data is more than about 10-20, classical strategies for near-neighbor search tend to degenerate to worst case (i.e. linear search) in practice.

2 A High-Dimension-Friendly Approach to Near Neighbor Search

What do you do if you have high-dimensional data and want to solve near-neighbor problems?

- We will discuss an approach called locality-sensitive hashing (LSH).
- Pioneered by Piotr Indyk in late 1990's. See, e.g., Andoni and Indyk 2008, CACM.
- LSH is a *randomized*, *approximate* strategy for near neighors.
- *randomized*: the indexing algorithm makes some random choices, independent of the data being indexed.
- With *high probability* (over the random choices), the resulting index yields a fast, accurate near neighbor search algorithm.
- *approximate*: if you want all items within some distance R of a query, you might also get most or all items within some distance bound $R' > R$.

To make this idea precise, define *randomized c-approximate R-nn problem* as follows.

- Let S be a collection of items with distance measure d.
- For any query item q, we seek to return all items $p \in S$ such that $d(p, q) \leq R$, for some given distance R.
- For any p with $d(p, q) \leq R$, the probability that the search misses p must be at most some fixed δ.
- We might also return items p with $d(p, q)$ as big as cR, but we are *not* guaranteed to find all or most of them.
- We need c to define the cost of solving the problem, but if you really want items within R, you can just "throw out" any returned items at larger distance.

What are our performance goals?

- Running time clearly depends on output size, i.e. number of items close to q (in particular, the number of p s.t. $d(p, q) \leq cR$.)
- Call the time to enumerate all results the *output time*.
- There is also a *search time*, independent of output.

- For a naive algorithm, search time would be $\Theta(|S|)$, even if we found nothing close to our query.
- Goal is to keep search time $o(|S|)$, and preferably highly sublinear in $|S|$.
- OK, what about building the index?
- We probably don't want to build an index that takes a ridiculous amount of space to store (or time to build).
- Ideally, the index would take only $O(|S|)$ space and $O(|S|)$ time to build.
- For large S, even $O(|S|^2)$ bounds may be unacceptable.
- We will try to get as close to $O(|S|)$ space and time as we can.

3 Locality-Sensitive Hash Functions

- Let $\mathcal{H}$ be a family of hash functions mapping items to some universe U of values.
- **Defn**: $\mathcal{H}$ is a (R, c, P_1, P_2)-LSH family if, for any two items p and q,
  1. If $d(p, q) \leq R$, then $\Pr_{h \in \mathcal{H}}[h(p) = h(q)] \geq P_1$.
  2. If $d(p, q) \geq cR$, then $\Pr_{h \in \mathcal{H}}[h(p) = h(q)] \leq P_2$.
- *Intuition*: locality-sensitive hash functions tend to map nearby items to the same value, while mapping far-away items to distinct values.

Here's a simple example for Hamming space.

- Suppose our items are ℓ-bit vectors.
- Distance between vectors is # of posns in which they disagree.
- Let $h_i(\cdot)$ be a function that extracts the bit in the ith posn of its input vector.
- Define $\mathcal{H} = \{h_i \mid 1 \leq i \leq \ell\}$.
- A randomly chosen function from $\mathcal{H}$ projects all vectors to their ith bits.
- Now suppose $d(p, q) \leq R$.
- p and q agree in at least $\ell - R$ positions, so

$$\begin{aligned} P_1 &= \Pr_{h \in \mathcal{H}}[h(p) = h(q)] \\ &\geq (\ell - R)/\ell \\ &= 1 - R/\ell. \end{aligned}$$

- Now suppose that $d(p, q) \geq cR$.
- p and q agree in at most $\ell - cR$ positions, so

$$\begin{aligned} P_1 &= \Pr_{h \in \mathcal{H}}[h(p) = h(q)] \\ &\leq (\ell - cR)/\ell \\ &= 1 - cR/\ell. \end{aligned}$$

4 Using LSH for Near-Neighbors Problem

Claim: Let $\mathcal{H}$ be an (R, c, P_1, P_2)-LSH family of hash functions. Let $n = |S|$. Then we can solve the randomized c-approximate R-nn problem in

- expected search time $\widetilde{O}(n^{\rho} \log \frac{1}{\delta})$,
- deterministic index construction time $\widetilde{O}(n^{1+\rho} \log \frac{1}{\delta})$,
- deterministic index space $O(n^{1+\rho} \log \frac{1}{\delta})$

where

$$\rho = \frac{\log(1/P_1)}{\log(1/P_2)}.$$

- (Here, "$\widetilde{O}(f(n))$" means $O(f(n)\text{polylog}(n))$.)
- Note that, if $P_1 > P_2$, then for any fixed δ, search time is sublinear in n, and index space is subquadratic in n.
- *Examples*: $P_1 = 0.9$ and $P_2 = 0.8$ imply search time better than $\sqrt{n}$ and index space better than $n^{3/2}$.
- Amplifying the gap to 0.9 vs 0.7 would yield search time better than $n^{1/3}$ and space better than $n^{4/3}$.
- Note also that probability δ of failure is guaranteed for *each* item to be returned.
- So, a reasonable δ (say, 0.99) ensures that we will return almost every nearby item.

OK, time for the index construction.

- We will use two parameters L and k, to be fixed later.
- For $1 \leq j \leq L$, choose functions

  $$g_j = [h_{1j} \dots h_{kj}]$$

  mapping items to a vector in U^k.
- (For example, g_j defined over our example family above maps ℓ-bit vectors to k-bit vectors.)
- Each function's component hashes are chosen independently and uniformly at random from $\mathcal{H}$.
- To construct an index from S, we will construct a table T_j of "buckets" from each g_j.
- For each j, compute $g_j(p)$ for all $p \in S$ and store all items with the same g_j in the same bucket of T_j.
- We can use conventional hashing to store T_j in space $O(n)$, since it can have at most $O(n)$ distinct nonempty buckets.

- The complete index is the set of L tables $\{T_1 \dots T_L\}$, which uses space $O(nL)$ (and requires time $O(nLk)$ to build).

OK, now how do we search?

- Let q be a query item.
- For each $1 \leq j \leq L$, compute $d(p, q)$ for all p such that $g_j(p) = g_j(q)$.
- (We can find the bucket of all such p in T_j with a single hash lookup.)
- Finally, return all p found such that $d(p, q) \leq R$.

Time for cost analysis!

- To make things work out, we are going to choose k and L carefully.
- First, we want to ensure that the buckets we compare to q don't have too many irrelevant items (i.e. items more than cR away).
- For table T_j, a far-away item ends up in the same bucket as q with probability at most P_2^k.
- Hence, expected number of such items from table T_j is at most nP_2^k.
- If we set this expectation to 1, then we expect to compare at most L total far-away items to q over the whole search.
- (This makes search cost per table $O(k)$ beyond the output cost: $O(k)$ to form the hash value $g_j(q)$, and $O(1)$ to retrieve the bucket and eliminate the false positives.)
- Setting $nP_2^k = 1$ and solving for k, we get

$$k = \frac{\log n}{\log(1/P_2)}.$$

- (Note that k is $O(\log n)$.)
- Second, we want to ensure that, for any p with $d(p, q) \leq R$, the chance that we fail to discover p is at most δ.
- Probability that we fail to discover p after L tries is $(1 - P_1^k)^L$.
- Setting $(1 - P_1^k)^L \leq \delta$ and solving for L, we get

$$L \geq \frac{\log \delta}{\log(1 - P_1^k)}.$$

- To simplify, we use the fact that $1 - x \leq e^{-x}$ for $0 \leq x \leq 1$ to derive a (slightly larger than needed) bound on the smallest feasible L:

$$\begin{aligned} L &\approx \frac{\log \delta}{-P_1^k} \\ &= \frac{\log(1/\delta)}{P_1^k}. \end{aligned}$$

- Subbing in the value of k we chose above, we get

$$
\begin{aligned}
L &= \log\left(\frac{1}{\delta}\right)\left(\frac{1}{P_1}\right)^{\log n/\log(1/P_2)} \\
&= \log\left(\frac{1}{\delta}\right) n^{\log(1/P_1)/\log(1/P_2)} \\
&= \log\left(\frac{1}{\delta}\right) n^{\rho}.
\end{aligned}
$$

- Since the index size is $O(nL)$ and the expected search time is $O(Lk)$, the above values for L and k yield the claimed space and time bounds. QED