UNIVERSITY OF CALIFORNIA, BERKELEY

Lecture 25 - CS: 189

Oscar Ortega

July 16, 2021

1 NEAREST NEIGHBOR ALGORITHM

1.1 EXHAUSTIVE K-NN ALG

Given query point q:

- scan through all n sample pts, computing (squared) distances to q
- Maintain a max-heap with the k shortest distances seen so far.
- Time to construct classifier: 0. Only classifier with this property of the semester
- Query time: $O(nd + n\log k)$ need to look at every point at least once and each point has d features. If k is large, performing the heap operations generate the $n\log k$ term. Expected $O(nd + k\log n\log k)$ if we shuffle the points. Not performed often in practice due to computational issues such as cache misses.

Can we prepossess training pts to obtain sub linear query time?

- 2-5: dimensions: Voronoi diagrams
- (up to - 30 dimensions): k-d trees
- if more than 30 dimensions: exhaustive k-NN, but can use PCA or random projections
- **Locality Sensitive hashing**: look into this, might be something that is researched more in the future

1.2 Voronoi Diagrams

Let X be a point set. The **Voronoi cell** of $w \in X$ is the following:

$$\text{Vor}(w) = \{p \in \mathbf{R}^d : \|pw\|_2 \leq \|pv\|_2 \forall v \in X\}$$

The **Voronoi Diagram** of X is the set of X's Voronoi Cells. Each cell forms a convex polyhedron and for any two cells the division between the two voronoi cells is the maximum separating hyper plane between them. The size of the voronoi diagram in terms of the number of vertices is in $O(n^{\text{ceiling}(d/2)}$.. but in practice it is often $O(n)$ because the number of features is not typically too large. From there, we need a data structure that performs a query on the diagram:

1.3 Point Location

Given query point $q \in \mathbf{R}^d$, find the point $w \in X$ for which $q \in V$ or W

- 2d: O(n log n) time to compute V.D and a **trapezoidal map** for pt location.
- $Olog(n)$ query time
- dD: use **use binary space partition tree** for pt location. Good information on this online.

The unfortunate thing about this structure is that it only handles $1-NN$, take note that for nearest neighbor queries one does not need to necessarily use the 2-norm. Overall, this method is great for low-dimensional data. Often better to use K-D trees for these sorts of problems.

2 K-D Trees

"Decison trees" for NN search. Differences;

- choose splitting feature w/greatest width: feature i in $\max_{i,j,k}(X_{j,i} - X_{k,i})$
- Cheap Alternative: rotate through the features at each level.
- Choose splitting value: median point for feature i **OR** perform a halfway split of the box. $X_{j,i} + X_{k,i}/2$ where the two points are the two extreme values of the dataset with respect to that feature. Choosing the median guarantees floor$log_2 n$ tree depth $O(ndlogn)$ tree-building time.
- rotating the features guarantees a faster tree building time of $O(ND\log N)$
- We do not store the points in the leaves, each internal node stores a sample point.
- can think of a higher dimensional Binary Search tree where each level of the tree has some feature in which you perform the split.

Goal: given query pt. q, find a sample pt w such that $\|qw\| \leq (1+\epsilon)\|qs\|$, where s is the true nearest neighbor.

$$\epsilon = 0 \rightarrow \text{ exact } NN : \epsilon > 0 \rightarrow \text{ approximate NN}$$

Query algorithm maintains:

- Nearest neighbor found so far (or k nearest)
- binary heap of unexplored subtrees, keyed by distance from q.
- $Q \leftarrow$ heap containing root note with key zero
- $r \leftarrow \infty$
- while Q not empty and $(1+\epsilon) * \text{minkey}(Q) \leq r$
  - $B \rightarrow$ removemin(Q)
  - $w \rightarrow$ B's sample point.
  - $r \leftarrow \min(r, \text{dist}(q, w))$
  - B', B' $\rightarrow$ child boxes of B.
  - if $1+\epsilon \text{dist}(q, B') \leq r$ then insert $(Q, B, \text{dist}(q, B'))$
  - if $1+\epsilon \text{dist}(q, B'') \leq r$ then insert $(Q, B', \text{dist}(q, B''))$

In other words we take the smallest distance box of the q, we find the sample point inside the box and update r such that r is equal to either the challenging point or the current smallest point. We then grab the two child boxes of B and check to see if it might contain a point we want to investigate, then we insert it into the priority queue. We then end up replacing r with a max-heap holding the k nearest neighbors. Works with any l_p norm.

2.1 Why $\epsilon-$ approximate NN?

in some cases, we notice that we'll be traverse through every box in the circle defined by r. So if we allow a bit of slack, we limit the number of boxes we need to search through.