High Dimensional Indexing by Clustering

Yufei Tao

ITEE
University of Queensland

Recall that, our discussion so far has assumed that the dimensionality d is "moderately high", such that it can be regarded as a constant. This means that d should be smaller a sufficiently slow-growing function of the dataset size n (such as $\log_2 \log_2 n$), such that even an exponential value like 2^d is not too large.

Today we will enter a new chapter of the course, where we will discuss how to deal with very high dimensionality d that defeats the aforementioned assumption. In general, the higher d is, the harder data analysis becomes. In computer science, typically two vastly different sets of techniques are used to process data with moderately high and truly high d, respectively. Indeed, when d reaches as low as 10, the techniques we have discussed so far would not work well in practice.

This lecture will revisit the topic of indexing, namely, how to build a data structure to accelerate query processing.

Applications of High Dimensional Data

To be effective, many machine learning techniques require extracting a large number of features from the underlying objects.

For example, in evaluating the similarity of documents, each dimension may be a word, such that the coordinate of a document on the dimension is the number of occurrences of the word. The dimensionality can be as high as the number of English words.

Similar examples can be found in speech recognition, face recognition, sentiment analysis, image retrieval, customer profiling, etc.

Curse of Dimensionality

Recall that, when d is moderately high, the R-tree is a useful structure that can be used to accelerate many types of queries. Unfortunately, the R-tree gradually becomes useless as d increases. As we analyzed in a previous lecture, in an optimal R-tree, each leaf MBR should be a square whose side length is at the order of (each dimension has a length of 1)

$$(B/n)^{1/d}$$

where B is the number of points a leaf node should contain.

Note: Although our analysis was based on the "uniform distribution", similar conclusions can be obtained on other distributions as well (hint: replace 0.99 in the derivation of that lecture with any smaller constant gives the same result).

Curse of Dimensionality

Setting $B = 1$ for convenience and considering $n = 10^6$, we have:

- $d = 2$: $(1/n)^{1/d} = 0.001$
- $d = 10$: $(1/n)^{1/d} = 0.251$
- $d = 100$: $(1/n)^{1/d} = 0.871$
- $d = 1000$: $(1/n)^{1/d} = 0.986$.

In other words, when d is large, each leaf MBR (even of an optimal R-tree) is nearly as large as the data space, such that the R-tree becomes almost useless!

The phenomenon that (essentially) every index structure becomes increasingly useless when d grows is known as the curse of dimensionality.

Today we will discuss an alternative approach—called the clustering method—to index multidimensional data. This approach complements the advantages and disadvantages of the R-tree:

- It is less effective than the R-tree when the dimensionality is relatively low (e.g., ≤ 5).
- However, on practical data distributions, the approach deteriorates at a slower rate than the R-tree, such that it actually performs better than the R-tree in a higher range of d.

We will use nearest neighbor search as the example query type. Recall:

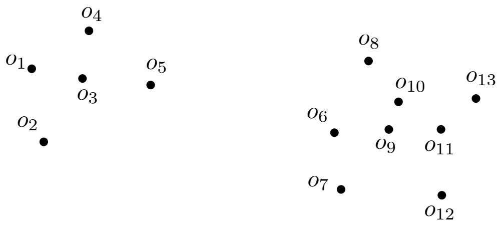
Let P be a set of d-dimensional points in $\mathbb{R}^d$. The (Euclidean) nearest neighbor (NN) of a query point $q \in \mathbb{R}^d$ is the point $p \in P$ that has the smallest Euclidean distance to q.

We denote the Euclidean distance between p and q as $\|p, q\|$.

Clustering

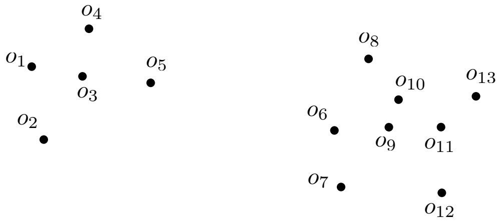
A cluster is a group of points that are (relatively) close to each other. A practical dataset often consists of several clusters, where points in different clusters are far away from each other (think: why so?). The act of clustering is to discover such clusters.

For example, the dataset below (intuitively) consists of 2 clusters:



Index by Clustering

Suppose that we have found a number, say k, of clusters. For each cluster, identify a centroid object, which intuitively is a point at the "middle" of the cluster.



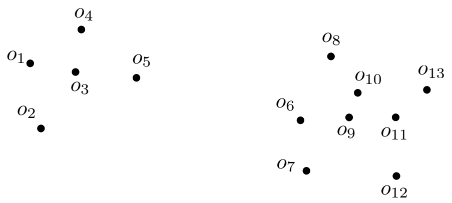
In the above example, o_3 (or o_9) is a good centroid of the left (or right, resp) cluster.

Index by Clustering

For each cluster, sort the points therein in ascending order of their distances to the centroid.

All the sorted lists constitute our index structure.

Example



- Left cluster: o_3, o_1, o_4, o_2, o_5.
- Right cluster: $o_9, o_{10}, o_6, o_{11}, o_7, o_8, o_{12}, o_{13}$.

Next we introduce two pruning rules that permit us to deploy the index to answer NN queries efficiently. Both rules find their roots in the triangle inequality. Namely, for any three points o, p, q, it must hold that $\|q, p\| + \|o, p\| \geq \|q, o\|$.

Pruning Rule 1

Lemma: Consider a cluster C with centroid object c. Let p be an arbitrary point in C, and q an arbitrary point in the data space. For any point p_{aft} that ranks after p in the sorted list of C, it must hold that:

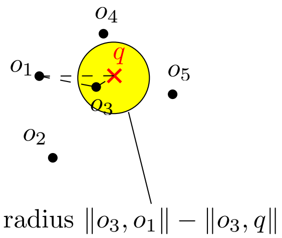
$$\|q, p_{aft}\| \geq \|p, c\| - \|c, q\|.$$

Proof:

$$\begin{aligned} \|q, p_{aft}\| &\geq \|p_{aft}, c\| - \|c, q\| && \text{(triangle inequality)} \\ &\geq \|p, c\| - \|c, q\| && \text{(by the definition of } p\text{).} \end{aligned}$$

$\square$

Example



Recall that the ordering of the cluster is o_3, o_1, o_4, o_2, o_5. The pruning rule says that o_4, o_2, o_5 must all be outside the yellow circle.

Pruning Rule 2

Corollary: Consider a cluster C with centroid object c. Let p_{far} be the point in C that is the farthest to c. Let q be an arbitrary point in the data space. Then, any point in C must have a distance to q at least:

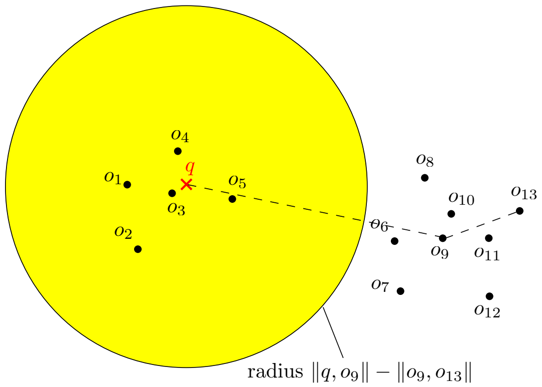
$$\|q, c\| - \|c, p_{far}\|.$$

Proof: Let p be an arbitrary point in C. Then:

$$\begin{aligned}\|q, p\| &\geq \|q, c\| - \|c, p\| && \text{(triangle inequality)} \\ &\geq \|q, c\| - \|c, p_{far}\| && \text{(by the definition of } p_{far}\text{).}\end{aligned}$$

□

Example



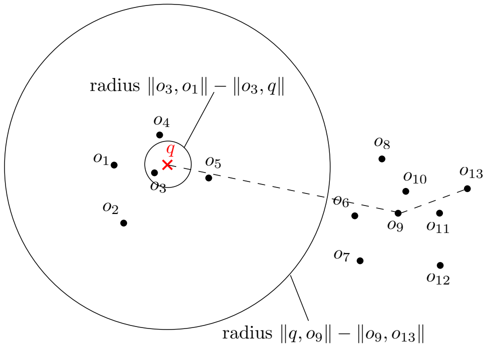
The pruning rule says that all the points in the right cluster must all be outside the yellow circle.

NN Algorithm

Given a query point q, we can find its NN as follows. Let p_{nn} the point found to be nearest to q so far (at the beginning, set p_{nn} to NULL and define $\|q, \text{NULL}\| = \infty$).

1. Let C be the cluster whose centroid c is the closest to q.
2. Scan the points of C by the order as stored in the index. Stop as soon as coming across a point p such that $\|p, c\| - \|c, q\| > \|q, p_{nn}\|$.
3. For every other cluster C' with centroid c':
    - 3.1 Prune C' if $\|q, c'\| - \|c', p_{far}\| > p_{nn}$ where p_{far} is the last point in the sorted order of C'.
    - 3.2 Otherwise, scan everything in C'.

Example



First, we scan the left cluster. After seeing o_3, p_{nn} is updated to o_3. Then, the algorithm reads o_1, which does not change o_3. The scan of the cluster terminates right here.

The right cluster is pruned directly.

In practice, we usually partition the dataset into more clusters (e.g., somewhere from 10 to 100). The hope is that only a small number of clusters need to be scanned in answering a NN query. This is often the case when the query point falls in some cluster, which in turn often happens in practice (think: why?).

But there is one important issue left: how to perform the clustering?

Next, we introduce the k-center algorithm which is a simple and yet effective algorithm with attractive theoretical guarantees.

Let us first formalize the problem:

Let P be a set of n points in $\mathbb{R}^d$, and k be an integer at most n. Let S be a set of objects in $\mathbb{R}^d$; we refer to S as a centroid set. Define for each object $p \in P$, its centroid distance as

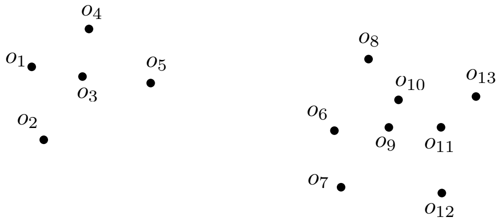
$$d_S(p) \quad = \quad \min_{c \in S} \|p, c\|.$$

The radius of C is defined to be

$$r(S) \quad = \quad \max_{o \in P} d_C(o).$$

The goal of the k-center problem is to find a centroid set S of size k with the minimum radius.

Example



For $k = 2$, the optimal solution is $S = \{o_3, o_9\}$ with $r(S) = \|o_9, o_{13}\|$.
A suboptimal solution is $S' = \{o_1, o_{13}\}$ with $r(S') = \|o_7, o_{13}\|$.

This problem is NP-hard, namely, no algorithm can solve the problem in time polynomial to both n and k (unless P = NP). Hence, we will aim to find approximate answers with precision guarantees.

Let S^* be an optimal centroid set for the k-center problem. A set S of k objects is ρ-approximate if $r(S) \leq \rho \cdot r(S^*)$. We will give an algorithm that guarantees to return a 2-approximate solution.

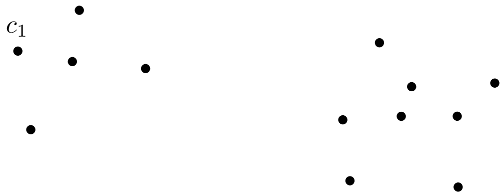
A 2-Approximate Algorithm

```
algorithm $k$-center ($P$)
/* this algorithm returns a 2-approximate subset $S$ */
1.  $S \leftarrow \emptyset$
2.  add to $S$ an arbitrary object in $P$
2.  for $i = 2$ to $k$
3.      $o \leftarrow$ an object in $P$ with the maximum $d_S(o)$
4.      add $o$ to $S$
5.  return $S$
```

The algorithm can be easily implemented in $O(nk)$ time.

Example

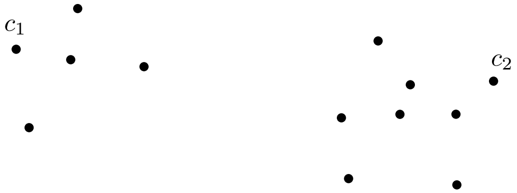
Set $k = 3$.



Initially, $S = \{c_1\}$

Example

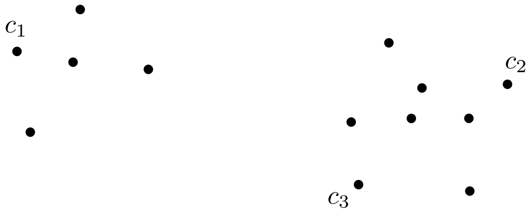
Set $k = 3$.



After a round, $S = \{c_1, c_2\}$

Example

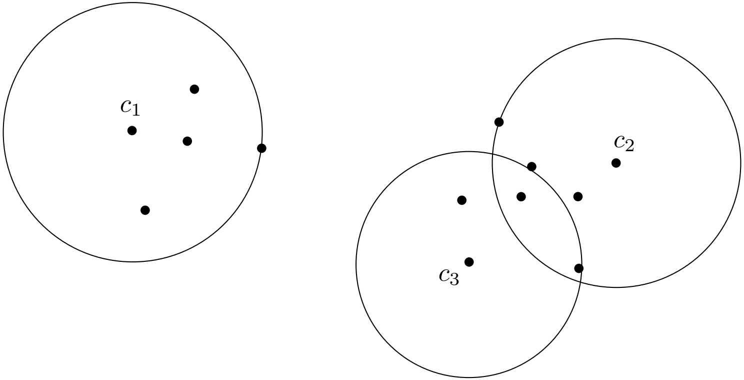
Set $k = 3$.



After another round, $S = \{c_1, c_2, c_3\}$

Example

Set $k = 3$.



$r(S)$ is the radius of the largest circle.

Theorem: The k-center algorithm is 2-approximate.

Proof: Let $S^* = \{c_1^*, c_2^*, ..., c_k^*\}$ be an optimal centroid set, i.e., it has the smallest radius $r(S^*)$. Let $C_1^*, C_2^*, ..., C_k^*$ be the optimal clusters, namely, C_i^* $(1 \leq i \leq k)$ contains all the objects that find c_i^* as the closest centroid among all the centroids in S^*.

Let $S = \{c_1, c_2, ..., c_k\}$ be the output of our algorithm. We want to prove $r(S) \leq 2r(S^*)$.

Case 1: S has an object in each of $C_1^*, C_2^*, \ldots, C_k^*$.

Take any object $o \in P$. We will prove that $d_S(o) \leq 2r(S^*)$, which implies that $r(S) \leq 2r(S^*)$.

Suppose that $o \in C_i^*$ (for some $i \in [1, k]$), and c is an object in $S \cap C_i^*$. It holds that:

$$\begin{aligned} d_S(o) &\leq \|c, o\| \\ &\leq \|c, c^*\| + \|c^*, o\| \\ &\leq 2r(S^*). \end{aligned}$$

Case 2: At least one of $C_1^*, ..., C_k^*$ covers no object in S. By the pigeon hole principle, one of $C_1^*, ..., C_k^*$ must cover at least two objects $c_1, c_2 \in S$. It thus follows that

$$\|c_1, c_2\| \leq 2r(S^*).$$

Next we will prove $r(S) \leq \|c_1, c_2\|$ which will complete the whole proof.

Without loss of generality, assume that c_2 was picked after c_1 by our algorithm. Hence, c_2 has the largest centroid distance at this moment (by how our algorithm runs). Therefore, any object $o \in P$ has a centroid distance at most $dist(c_1, c_2)$ at this moment. Its centroid distance can only decrease in the rest of the algorithm. It thus follows that $r(S) \leq dist(c_1, c_2)$. □