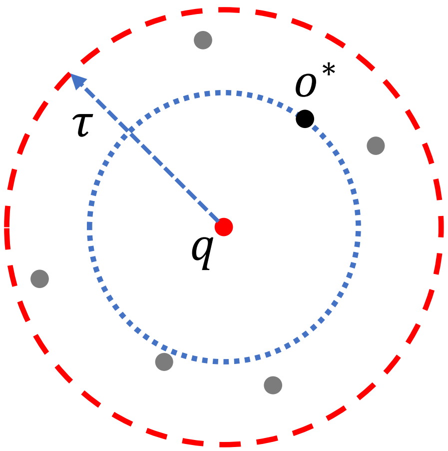
COMP9313: Big Data Management

High Dimensional Similarity Search

Similarity Search

- Problem Definition:
  - Given a query q and dataset D, find o $\in D$, where o is similar to q
- Two types of similarity search
  - Range search:
    - $dist(o, q) \leq \tau$
  - Nearest neighbor search
    - $dist(o^*, q) \leq dist(o, q), \forall o \in D$
    - Top-k version
- Distance/similarity function varies
  - Euclidean, Jaccard, inner product, …
- Classic problem, with mutual solutions



High Dimensional Similarity Search

- Applications and relationship to Big Data
  - Almost every object can be and has been represented by a high dimensional vector
    - Words, documents
    - Image, audio, video
    - …
  - Similarity search is a fundamental process in information retrieval
    - E.g., Google search engine, face recognition system, …
- High Dimension makes a huge difference!
  - Traditional solutions are no longer feasible
  - This lecture is about why and how
    - We focus on high dimensional vectors in Euclidean space

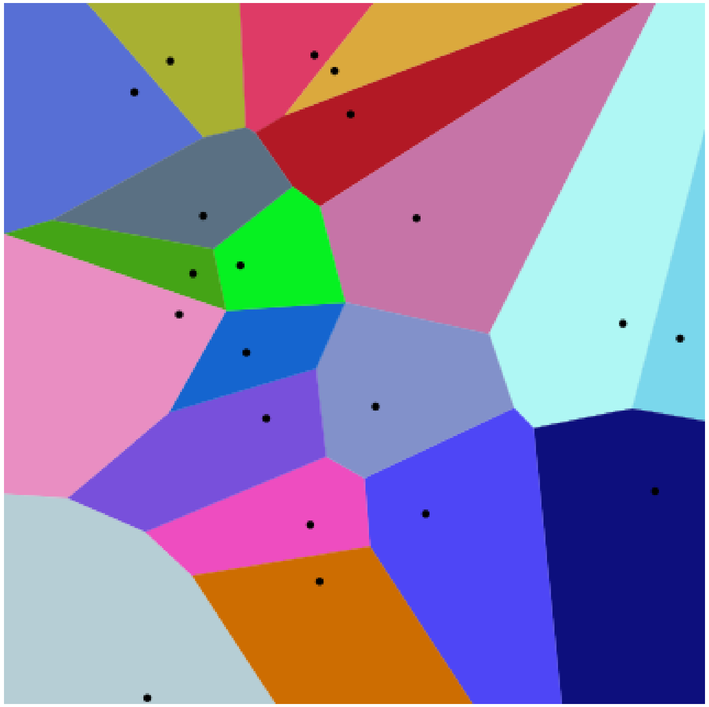
Similarity Search in
Low Dimensional Space

Similarity Search in One Dimensional Space

- Just numbers, use binary search, binary search tree, B+ Tree…
- The essential idea behind: objects can be sorted

Similarity Search in Two Dimensional Space

- Why binary search no longer works?
  - No order!
- Voronoi diagram

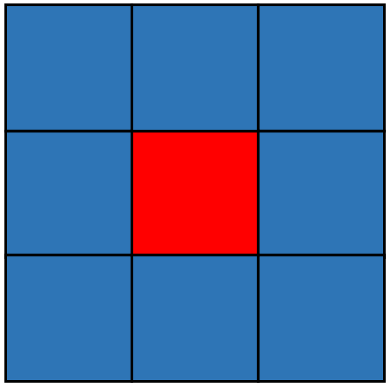


Euclidean distance

Manhattan distance

Similarity Search in Two Dimensional Space

- Partition based algorithms
  - Partition data into “cells”
  - Nearest neighbors are in the same cell with query or adjacent cells



- How many “cells” to probe on 3-dimensional space?

Similarity Search in Metric Space

- Triangle inequality
  - $dist(x, q) \leq dist(x, y) + dist(y, q)$
- Orchard's Algorithm
  - for each $x \in D$, create a list of points in increasing order of distance to x
  - given query q, randomly pick a point x as the initial candidate (i.e., pivot p), compute $dist(p, q)$
  - walk along the list of p, and compute the distances to q. If found y closer to q than p, then use y as the new pivot (e.g., $p \leftarrow y$).
  - repeat the procedure, and stop when
    - $dist(p, y) > 2 \cdot dist(p, q)$

Similarity Search in Metric Space

- Orchard's Algorithm, stop when $dist(p, y) > 2 \cdot dist(p, q)$

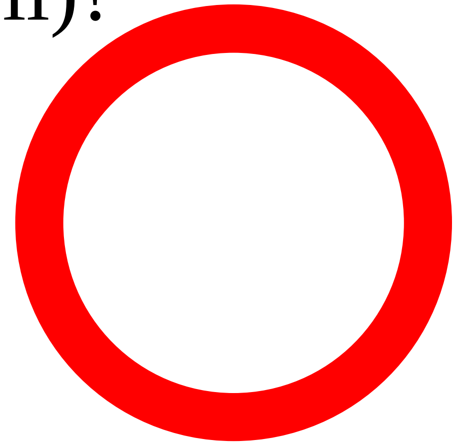
$$2 \cdot dist(p, q) < dist(p, y) \text{ and}$$
$$dist(p, y) \le dist(p, q) + dist(y, q)$$
$$\Rightarrow 2 \cdot dist(p, q) < dist(p, q) + dist(y, q)$$
$$\Leftrightarrow dist(p, q) < dist(y, q)$$

- Since the list of p is in increasing order of distance to p, $dist(p, y) > 2 \cdot dist(p, q)$ hold for all the rest y's.

None of the Above Works
in
High Dimensional Space!

Curse of Dimensionality

- Refers to various phenomena that arise in high dimensional spaces that do not occur in low dimensional settings.
- Triangle inequality
  - The pruning power reduces heavily
- What is the volume of a high dimensional "ring" (i.e., hyperspherical shell)?
  - $\frac{V_{ring}(w=1,d=2)}{V_{ball}(r=10,d=2)} = 29\%$
  - $\frac{V_{ring}(w=1,d=100)}{V_{ball}(r=10,d=100)} = 99.997\%$

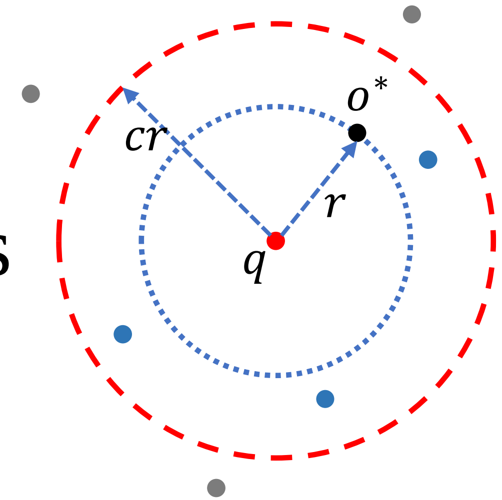


Approximate Nearest Neighbor Search in High Dimensional Space

- There is no sub-linear solution to find the exact result of a nearest neighbor query
  - So we relax the condition
- approximate nearest neighbor search (ANNS)
  - allow returned points to be not the NN of query
  - Success: returns the true NN
  - use success rate (e.g., percentage of succeed queries) to evaluate the method
  - Hard to bound the success rate

c-approximate NN Search

- Success: returns o such that
  - $dist(o, q) \leq c \cdot dist(o^*, q)$
- Then we can bound the success probability
  - Usually noted as $1 - \delta$
- Solution: Locality Sensitive Hashing (LSH)



Locality Sensitive Hashing

- Hash function
  - Index: Map data/objects to values (e.g., hash key)
    - Same data ⇒ same hash key (with 100% probability)
    - Different data ⇒ different hash keys (with high probability)
  - Retrieval: Easy to retrieve identical objects (as they have the same hash key)
    - Applications: hash map, hash join
  - Low cost
    - Space: $O(n)$
    - Time: $O(1)$
  - Why it cannot be used in nearest neighbor search?
    - Even a minor difference leads to totally different hash keys

Locality Sensitive Hashing

- Index: make the hash functions error tolerant
  - Similar data ⇒ same hash key (with high probability)
  - Dissimilar data ⇒ different hash keys (with high probability)
- Retrieval:
  - Compute the hash key for the query
  - Obtain all the data has the same key with query (i.e., candidates)
  - Find the nearest one to the query
  - Cost:
    - Space: $O(n)$
    - Time: $O(1) + O(|cand|)$
- It is not the real Locality Sensitive Hashing!
  - We still have several unsolved issues…

LSH Functions

- Formal definition:
  - Given point o_1, o_2, distance r_1, r_2, probability p_1, p_2
  - An LSH function $h(\cdot)$ should satisfy
    - $\Pr[h(o_1) = h(o_2)] \geq p_1$, if $dist(o_1, o_2) \leq r_1$
    - $\Pr[h(o_1) = h(o_2)] \leq p_2$, if $dist(o_1, o_2) > r_2$
- What is $h(\cdot)$ for a given distance/similarity function?
  - Jaccard similarity
  - Angular distance
  - Euclidean distance

MinHash - LSH Function for Jaccard Similarity

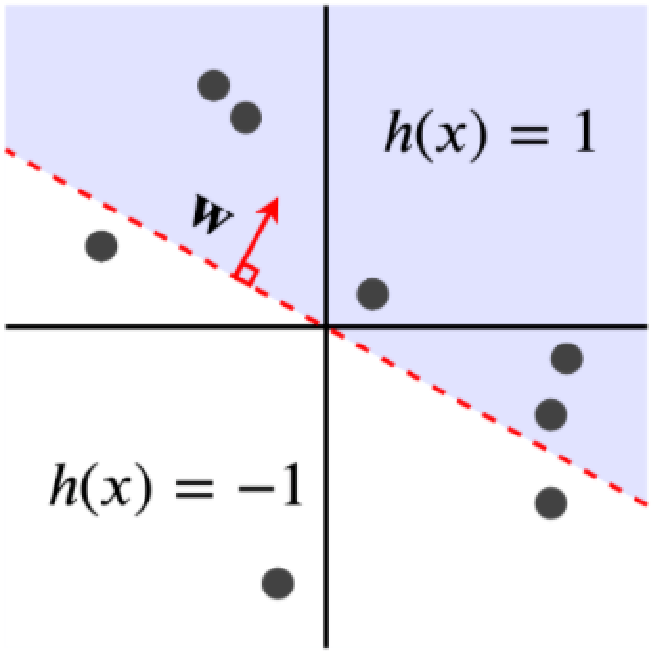
- Each data object is a set
  - $Jaccard(S_1, S_2) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$
- Randomly generate a global order for all the elements in $\mathrm{C} = \bigcup_1^n S_i$
- Let $h(S)$ be the minimal member of S with respect to the global order
  - For example, $S = \{b, c, e, h, i\}$, we use inversed alphabet order, then re-ordered $S = \{i, h, e, c, b\}$, hence $h(S) = i$.

MinHash

- Now we compute $\Pr[h(S_1) = h(S_2)]$
- Every element $e \in S_1 \cup S_2$ has equal chance to be the first element among $S_1 \cup S_2$ after re-ordering
- $e \in S_1 \cap S_2$ if and only if $h(S_1) = h(S_2)$
- $e \notin S_1 \cap S_2$ if and only if $h(S_1) \neq h(S_2)$
- $\Pr[h(S_1) = h(S_2)] = \frac{|\{e_i | h_i(S_1) = h_i(S_2)\}|}{|\{e_i\}|} = \frac{|S_i \cap S_j|}{|S_i \cup S_j|} = Jaccard(S_1, S_2)$

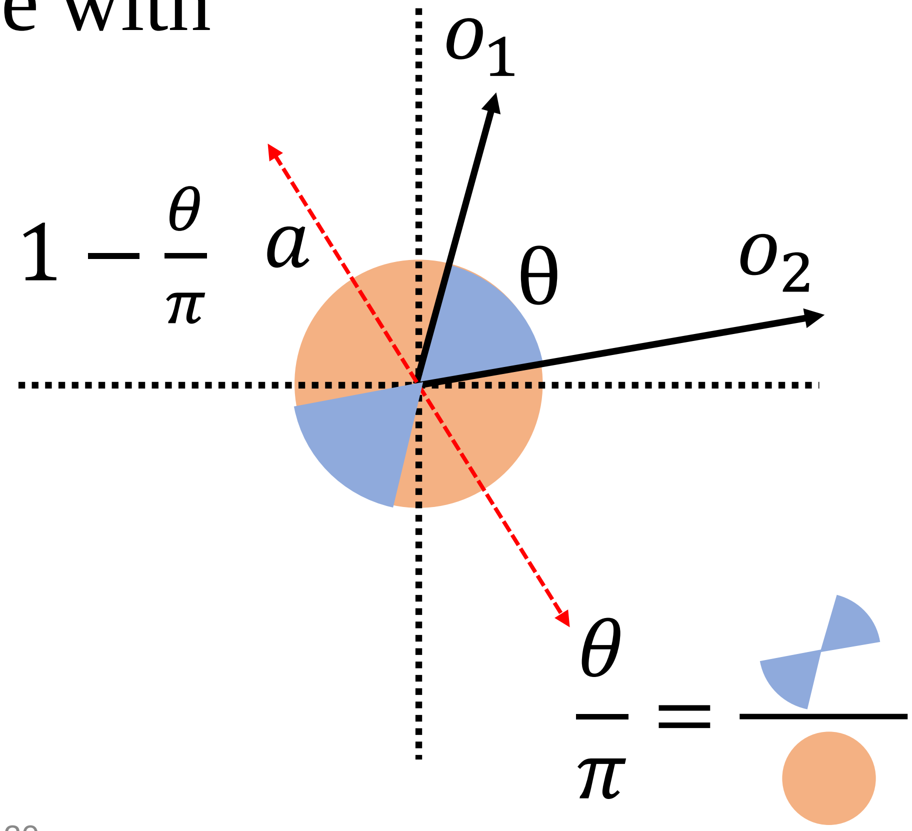
SimHash – LSH Function for Angular Distance

- Each data object is a d dimensional vector
  - $\theta(x, y)$ is the angle between x and y
- Randomly generate a normal vector a, where $a_i \sim N(0,1)$
- Let $h(x; a) = \text{sgn}(a^T x)$
  - $\text{sgn(o)} = \begin{cases} 1; if\ o \geq 0 \\ -1; if\ o < 0 \end{cases}$
  - x lies on which side of a's corresponding hyperplane



SimHash

- Now we compute $\Pr[h(o_1) = h(o_2)]$
- $h(o_1) \neq h(o_2)$ iff o_1 and o_2 are on different sides of the hyperplane with a as its normal vector
- $\Pr[h(o_1) = h(o_2)] = 1 - \frac{\theta}{\pi}$

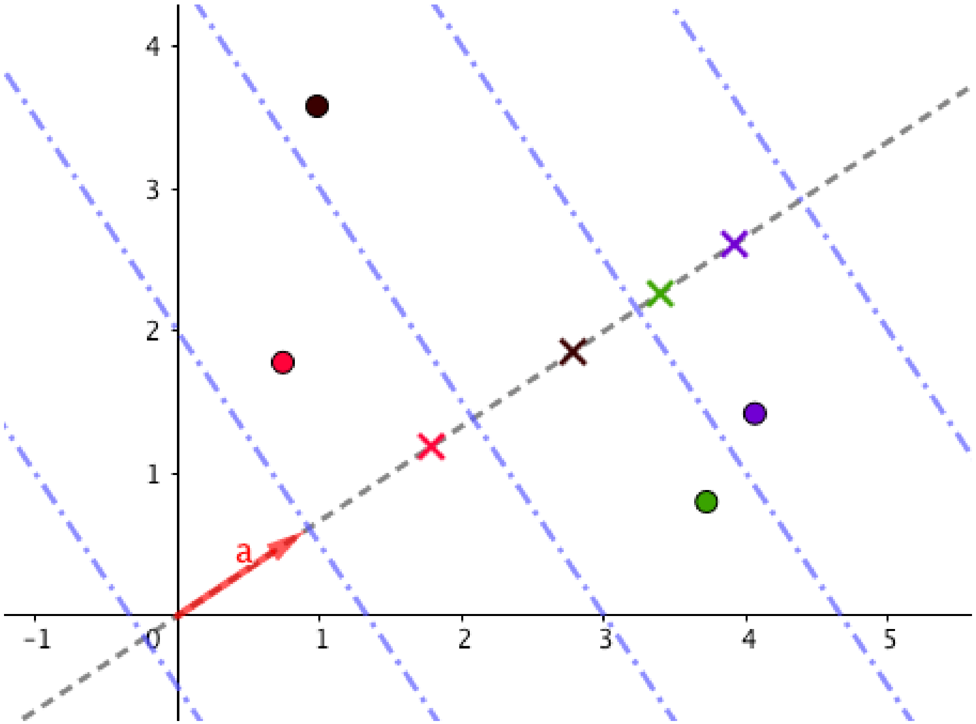


p-stable LSH - LSH function for Euclidean distance

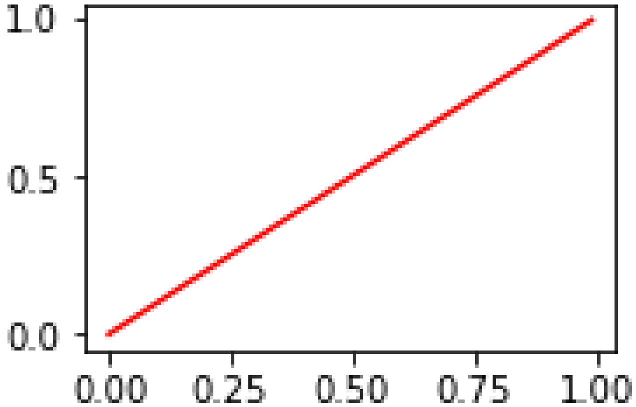
- Each data object is a d dimensional vector
  - $dist(x, y) = \sqrt{\sum_1^d (x_i - y_i)^2}$
- Randomly generate a normal vector a, where $a_i \sim N(0,1)$
  - Normal distribution is 2-stable, i.e., if $a_i \sim N(0,1)$, then $\sum_1^d a_i \cdot x_i \sim N(0, \|x\|_2^2)$
- Let $h(x; a, b) = \left\lfloor \frac{a^T x + b}{w} \right\rfloor$, where $b \sim U(0,1)$ and w is user specified parameter
  - $\Pr[h(o_1; a, b) = h(o_2; a, b)] = \int_0^w \frac{1}{\|o_1, o_2\|} f_p\left(\frac{t}{\|o_1, o_2\|}\right)\left(1 - \frac{t}{w}\right) dt$
  - $f_p(\cdot)$ is the pdf of the absolute value of normal variable

p-stable LSH

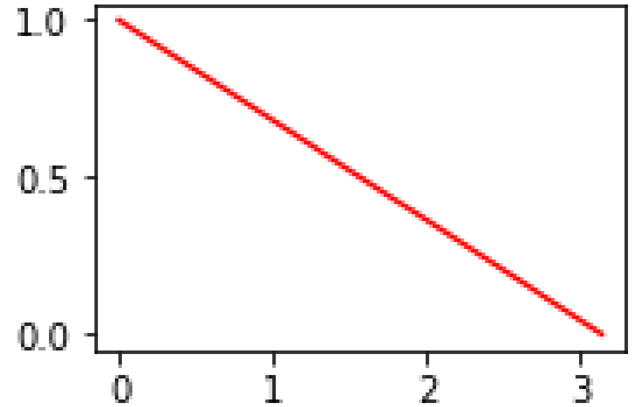
- Intuition of p-stable LSH
  - Similar points have higher chance to be hashed together



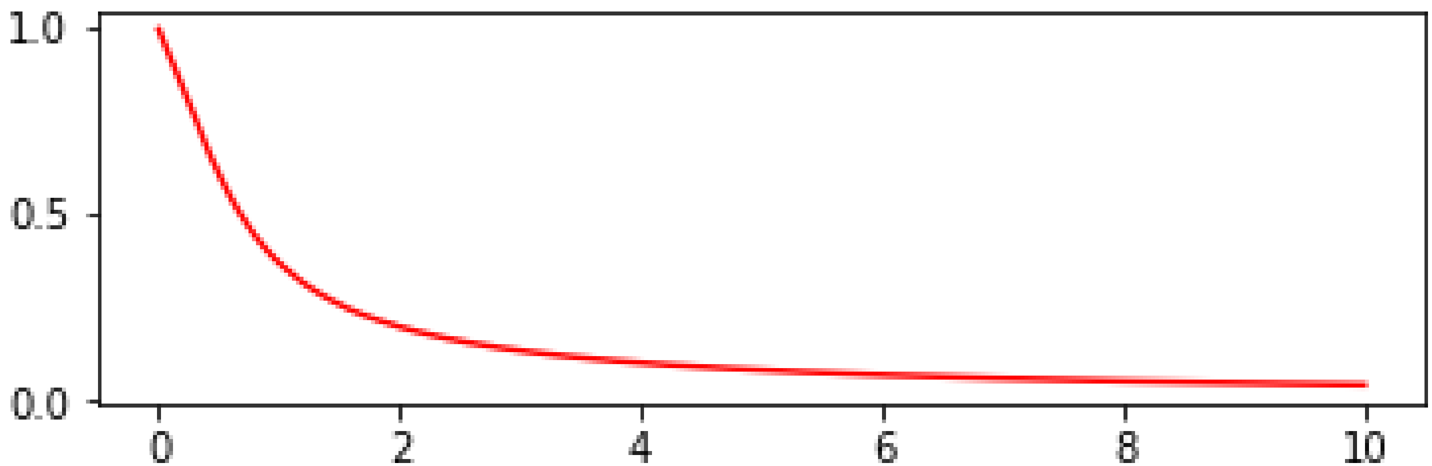
$\Pr[h(x) = h(y)]$ for different Hash Functions



MinHash



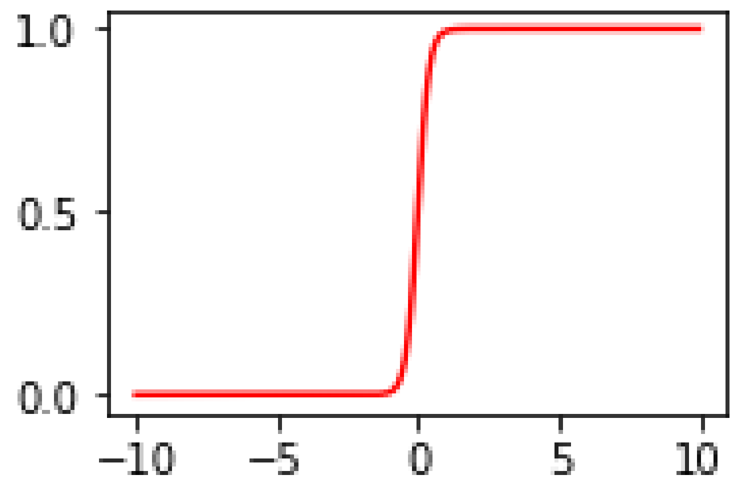
SimHash



p-stable LSH

Problem of Single Hash Function

- Hard to distinguish if two pairs have distances close to each other
  - $\Pr[h(o_1) = h(o_2)] \geq p_1$, if $dist(o_1, o_2) \leq r_1$
  - $\Pr[h(o_1) = h(o_2)] \leq p_2$, if $dist(o_1, o_2) > r_2$
- We also want to control where the drastic change happens…
  - Close to $dist(o^*, q)$
  - Given range

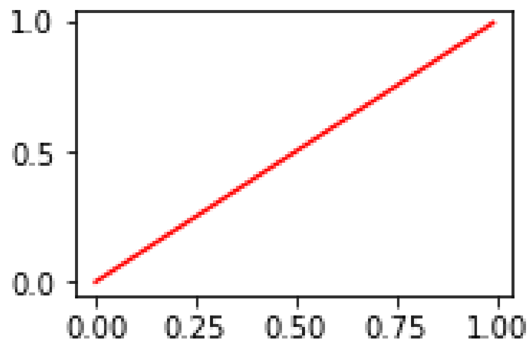


AND-OR Composition

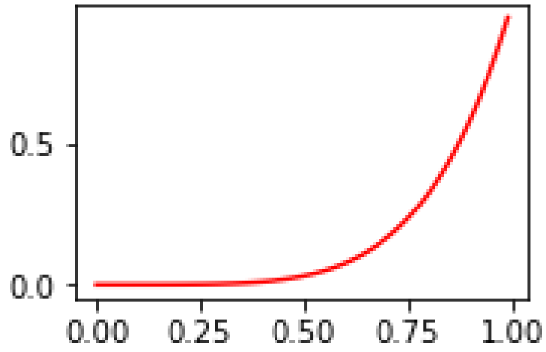
- Recall for a single hash function, we have
  - $\Pr[h(o_1) = h(o_2)] = p(dist(o_1, o_2))$, denoted as p_{o_1,o_2}
- Now we consider two scenarios:
  - Combine k hashes together, using AND operation
    - One must match all the hashes
    - $\Pr[H_{AND}(o_1) = H_{AND}(o_2)] = {p_{o_1,o_2}}^k$
  - Combine l hashes together, using OR operation
    - One need to match at least one of the hashes
    - $\Pr[H_{OR}(o_1) = H_{OR}(o_2)] = 1 - (1 - p_{o_1,o_2})^l$
      - Not match only when all the hashes don't match

AND-OR Composition

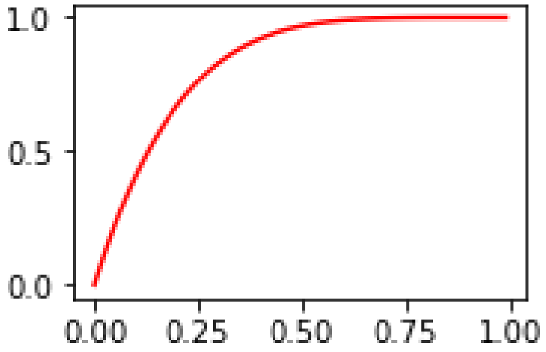
- Example with minHash, $k = 5, l = 5$



$\Pr[h(o_1) = h(o_2)]$



$\Pr[H_{AND}(o_1) = H_{AND}(o_2)]$



$\Pr[H_{OR}(o_1) = H_{OR}(o_2)]$

AND-OR Composition in LSH

- Let $h_{i,j}$ be LSH functions, where $i \in \{1,2,\ldots,l\}, j \in \{1,2,\ldots,k\}$
- Let $H_i(o) = [h_{i,1}(o), h_{i,2}(o), \ldots, h_{i,k}(o)]$
  - super-hash
  - $H_i(o_1) = H_i(o_2) \Leftrightarrow \forall j \in \{1,2,\ldots,k\}, h_{i,j}(o_1) = h_{i,j}(o_2)$
- Consider query q and any data point o, o is a nearest neighbor candidate of q if
  - $\exists i \in \{1,2,\ldots,l\},\ H_i(o) = H_i(q)$
- The probability of o is a nearest neighbor candidate of q is
  - $1 - (1 - p_{q,o}^{k})^{l}$

The Effectiveness of LSH

- $1-(1-p_{q,o}^{k})^{l}$ changes with $p_{q,o}$, $(k=20, l=5)$

$p_{q,o}$	$1-(1-p_{q,o}^{k})^{l}$
0.2	0.002
0.4	0.050
0.6	0.333
0.7	0.601
0.8	0.863
0.9	0.988

- E.g., we are expected to retrieve 98.8% of the data with Jaccard > 0.9

False Positives and False Negatives

- False Positive:
  - returned data with $\mathrm{dist}(\mathrm{o}, \mathrm{q}) > r_2$
- False Negative
  - not returned data with $\mathrm{dist}(\mathrm{o}, \mathrm{q}) < r_1$
- They can be controlled by carefully chosen k and l
  - It’s a trade-off between space/time and accuracy

The Framework of NNS using LSH

- Pre-processing
  - Generate LSH functions
    - minHash: random permutations
    - simHash: random normal vectors
    - p-stable: random normal vectors and random uniform values
- Index
  - Compute $H_i(o)$ for each data object o, $i \in \{1, \ldots, l\}$
  - Index o using $H_i(o)$ as key in the i-th hash table
- Query
  - Compute $H_i(q)$ for query q, $i \in \{1, \ldots, l\}$
  - Generate candidate set $\{o \mid \exists i \in \{1, \ldots, l\}, H_i(q) = H_i(o)\}$
  - Compute the actual distance for all the candidates and return the nearest one to the query

The Drawback of LSH

- Concatenate k hashes is too "strong"
  - $h_{i,j}(o_1) \neq h_{i,j}(o_2) \Rightarrow H_i(q) \neq H_i(o)$ for any j
- Not adaptive to the distribution of the distances
  - What if not enough candidates?
  - Need to tune w (or build indexes different w's) to handle different cases

Multi-Probe LSH

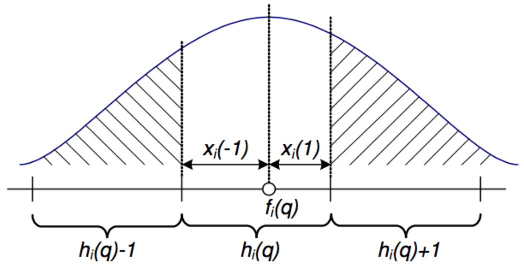
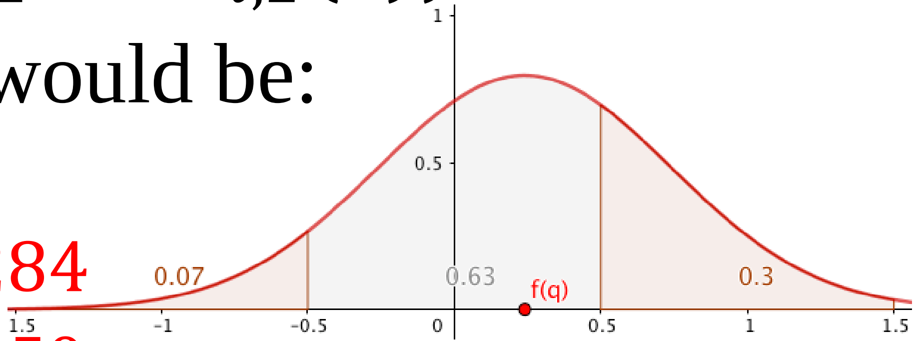


Figure 3: Probability of q's nearest neighbors falling into the neighboring slots.

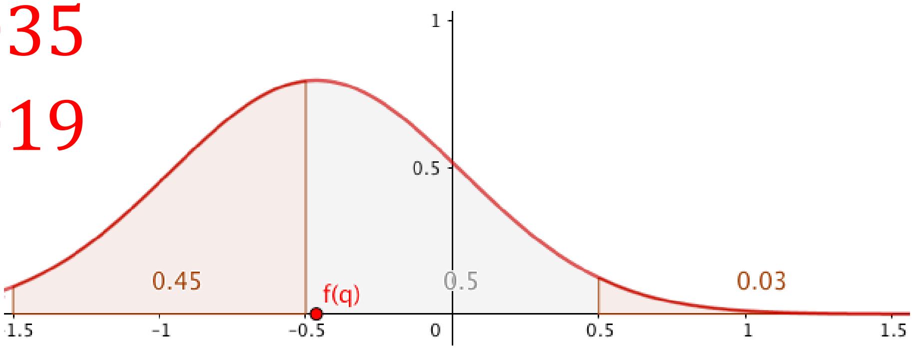
- Observation:
  - If q's nearest neighbor does not falls into q's hash bucket, then most likely it will fall into the adjacent bucket to q's
  - Why? $\sum_1^d a_i \cdot x_i - \sum_1^d a_i \cdot q_i \sim N(0, \|x, q\|_2^2)$
- Idea:
  - Not only look at the hash bucket where q falls into, but also those adjacent to it
- Problem:
  - How many such bucket? 2^k
  - And they are not equally important!

Multi-Probe LSH

- Consider the case when $k = 2$:
- Note that $H_i(o) = (h_{i,1}(o), h_{i,2}(o))$
- The ideal probe order would be:
  - $h_{i,1}(q), h_{i,2}(q)$: 0.315
  - $h_{i,1}(q), h_{i,2}(q) - 1$: 0.284
  - $h_{i,1}(q) + 1, h_{i,2}(q)$: 0.150
  - $h_{i,1}(q) - 1, h_{i,2}(q)$: 0.035
  - $h_{i,1}(q), h_{i,2}(q) + 1$: 0.019



$h_{i,1}$



$h_{i,2}$

We don't have to compute the integration, but use the offset between f(q) and the boundaries.

Multi Probe LSH

- Pros:
  - Requires less l
    - Because we use hash tables more efficiently
  - More robust against the unlucky points
- Cons:
  - Lose the theoretical guarantee about the results
  - Not parallel-friendly

Collision Counting LSH (C2LSH)

- C2LSH (SIGMOG'12 paper)
  - Which one is closer to q?

q	1	1	1	1	1	1	1	1	1	1	1	1
o_1	1	1	1	2	1	2	1	1	2	1	1	1
o_2	1	1	1	1	2	2	3	4	1	2	3	4

- We will omit the theoretical parts hence leads to a slightly different version to the paper.
  - But the essential ideas are the same
- Project 1 is to implement C2LSH using PySpark!

Counting the Collisions

- Collision: match on a single hash function
  - Use number of collisions to determine the candidates
  - Match one of the super hash with q → collides at least αm hash values with q
- Recall in LSH, The probability of o with $\text{dist}(o, q) \leq r_1$ is a nearest neighbor candidate of q is $1 - (1 - p_1^k)^l$
- Now we compute the case with collision counting…

The Collision Probability

- $\forall o$ with $\text{dist}(o, q) \leq r_1$, we have
  - $\Pr[\#collision(o) \geq \alpha m] = \sum_{i=\alpha m}^{m} \binom{m}{i} p^i (1-p)^{m-i}$
  - $p = \Pr\left[h_j(o) = h_j(q)\right] \geq p_1$
- We define m Bernoulli random variables $X_i \sim B(m, 1-p)$ with $1 \leq i \leq m$.
  - Let X_i equal 1 if o does not collide with q
    - i.e., $\Pr[X_i = 1] = 1 - p$
  - Let X_i equal 0 if o collides with q
    - i.e., $\Pr[X_i = 0] = p$
  - Hence $\mathrm{E}[X_i] = 1 - p$
    - Thus $E(\bar{X}) = 1 - p$, where $\bar{X} = \frac{\sum_{i=1}^{m} X_i}{m}$.
- Let $t = p - \alpha > 0$, we have:
  - $\Pr[\bar{X} - E(\bar{X}) \geq t] = \Pr\left[\frac{\sum_{i=1}^{m} X_i}{m} - (1-p) \geq t\right] = \Pr[\sum X_i \geq (1-\alpha)m]$

The Collision Probability

- From Hoeffding's Inequality, we have
  - $\Pr[\bar{X} - E(\bar{X}) \geq t] = \Pr[\sum X_i \geq (1-\alpha)m] \leq \exp\left(-\frac{2(p-\alpha)^2 m^2}{\sum_{i=1}^{m}(1-0)^2}\right) = \exp\left(-2(p-\alpha)^2 m\right) \leq \exp\left(-2(p_1-\alpha)^2 m\right)$
- Since the event "#collision(o) $\geq \alpha m$" is equivalent to the event "o misses the collision with q less than $(1-\alpha)m$ times",
  - $\Pr[\#collision(o) \geq \alpha m] = \Pr[\sum X_i < (1-\alpha)m] \geq 1 - \exp\left(-2(p_1-\alpha)^2 m\right)$
- Now you can compute the case for o with $\mathrm{dist}(o,q) \geq r_2$ in a similar way…
- Then we can accordingly set α to control false positives and false negatives

Virtual Rehashing

- When we are not getting enough candidates…
  - E.g., # of candidates < top-k
- Observation:
  - A close point *o* usually falls into adjacent hash buckets of q's if it does not collide with q
  - Why?
    - $\sum_1^d a_i \cdot x_i - \sum_1^d a_i \cdot q_i \sim N(0, \|x, q\|_2^2)$
- Idea:
  - Include the adjacent hash buckets into consideration
    - So you don't need to re-hash them again…

Virtual Rehashing

- At first consider $h(o) = h(q)$

q	1	1	1	1	1	1	1	1	1	1
o_1	1	0	2	-1	1	2	4	-3	0	-1

- Consider $h(o) = h(q) \pm 1$ if not enough candidates

q	1	1	1	1	1	1	1	1	1	1
o_1	1	0	2	-1	1	2	4	-3	0	-1

- Then $h(o) = h(q) \pm 2$ and so on…

q	1	1	1	1	1	1	1	1	1	1
o_1	1	0	2	-1	1	2	4	-3	0	-1

The Framework of NNS using C2LSH

- Pre-processing
  - Generate LSH functions
    - Random normal vectors and random uniform values
- Index
  - Compute and store $h_i(o)$ for each data object o, $i \in \{1, \dots, m\}$
- Query
  - Compute $h_i(q)$ for query q, $i \in \{1, \dots, m\}$
  - Take those o that shares at least αm hashes with q as candidates
  - Relax the collision condition (e.g., virtual rehashing) and repeat the above step, until we got enough candidates

Pseudo code of Candidate Generation in C2LSH

```
candGen(data_hashes, query_hashes, αm, βn):
    offset ← 0
    cand ← ∅
    while true:
        for each (id, hashes) in data_hashes:
            if count(hashes, query_hashes, offset) ≥ αm:
                cand ← cand ∪ {id}
        if |cand| < βn:
            offset ← offset + 1
        else:
            break
    return cand
```

Pseudo code of Candidate Generation in C2LSH

count(hashes_1, hashes_2, offset):

    counter ← 0

    for each $hash_1$, $hash_2$ in hashes_1, hashes_2:

        if $|hash_1 - hash_2| \leq$ offset:

            counter ← counter + 1

    return counter

Project 1

- Spec has been released, deadline: 18 Jul, 2020
  - Late Penalty: 10% on day 1 and 30% on each subsequent day.
- Implement a light version of C2LSH (i.e., the one we introduced in the lecture)
- Start working ASAP
- Evaluation: Correctness and Efficiency
- Must use PySpark, some python modules and PySpark functions are banned.
  - E.g., numpy, pandas, collect(), take(), …
  - Use transformations!

Project 1

- There will be a bonus part (max 20 points) to encourage efficient implementations.
  - Details in the spec
- Make sure you have valid output
- Make your own test cases, a real dataset would be more desirable
  - Toy example in the spec is a real “toy” (e.g., for babies…)
- Won’t accept excuses like “it works on my own computer”
- Don’t violate the Student Conduct!!!

Product Quantization

and K-Means Clustering

Recall: NNS in High Dimensional Euclidean Space

- Naïve (but exact) solution:
  - Linear scan: compute $dist(o,q)$ for all $o \in D$
  - $dist(o,q) = \sqrt{\sum_{i=1}^{d}(o_i - q_i)^2}$
  - $O(nd)$
    - n times (d subtractions + $d-1$ additions + d multiplications)
  - Storage is also costly: $O(nd)$
    - Could be problematic in DBMS and distributed systems

- This motivates the idea of **compression**

Vector Quantization

- Idea: compressed representation of vectors
  - Each vector o is represented by a representative
    - Denoted as $QZ(o)$
    - We will discuss how to get the representatives later
  - We control the total number of representatives for the dataset (denoted as k)
    - One representative represents multiple vectors
  - Instead of store o, we store its representative id
    - d floats => 1 integer
  - Instead of compute $dist(o, q)$, we compute $dist(QZ(o), q)$
    - We only need k computations of distance!

How to Generate Representatives

- Assigning representatives is essentially a partition problem
  - Construct a "good" partition of a database of n objects into a set of k clusters
- How to measure the "goodness" of a given partitioning scheme?
  - Cost of a cluster
    - $Cost(C_i) = \sum_{o_j \in C_i} \left\| o_j - center(C_i) \right\|_2^2$
  - Cost of k clusters: sum of $Cost(C_i)$

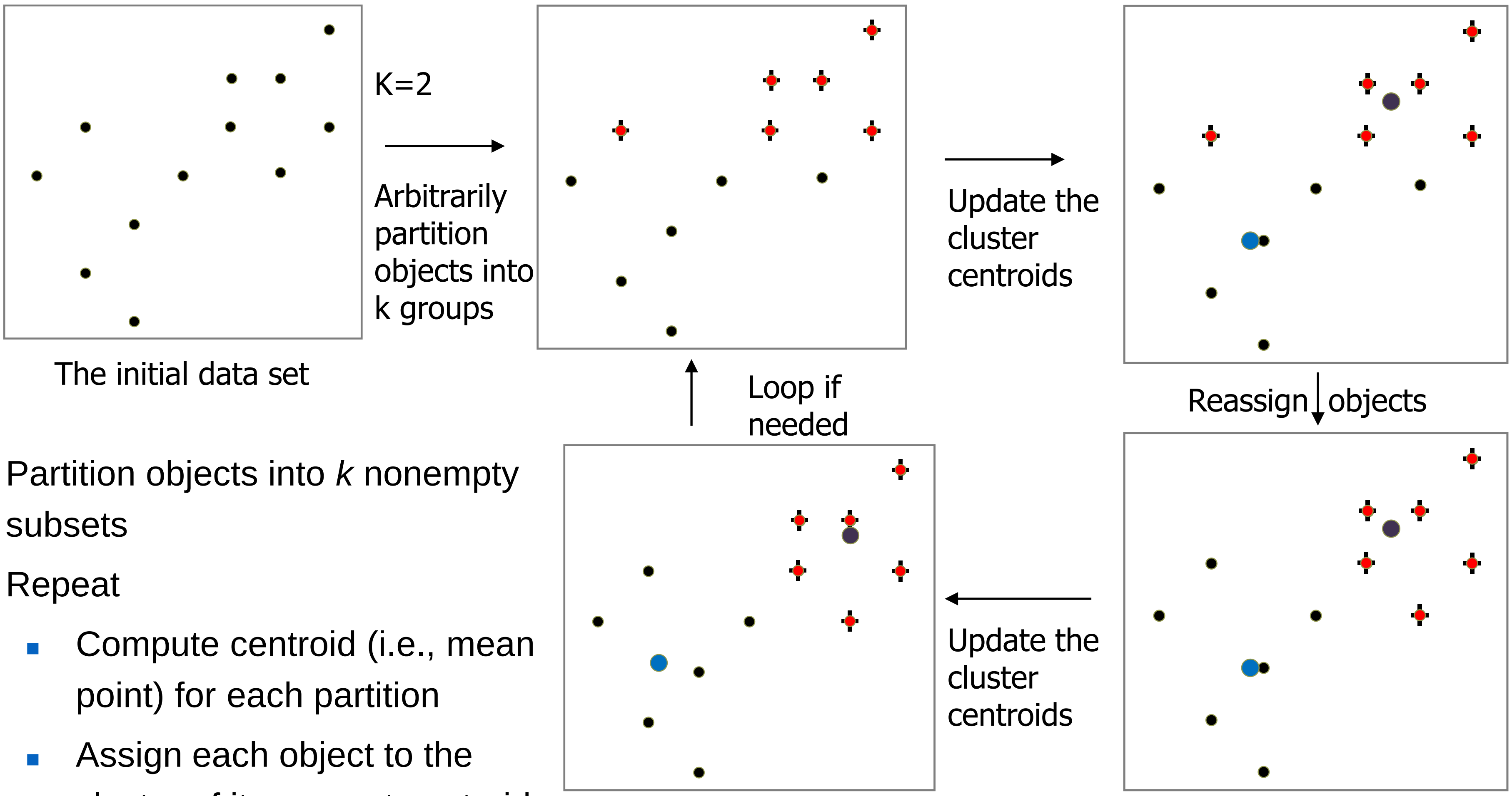
Partitioning Problem: Basic Concept

- It's an optimization problem!
  - Global optimal:
    - NP-hard (for a wide range of cost functions)
    - Requires exhaustively enumerate all $\binom{n}{k}$ partitions
      - Stirling numbers of the second kind
      - $\binom{n}{k} \sim \frac{k^n}{k!}$ when $n \to \infty$
  - Heuristic methods:
    - k-means
    - Many variants

The k-Means Clustering Method

- Given k, the k-means algorithm is implemented in four steps:
  1. Partition objects into k nonempty subsets (randomly)
  2. Compute seed points as the centroids of the clusters of the current partitioning (the centroid is the center, i.e., mean point, of the cluster)
  3. Assign each object to the cluster with the nearest seed point
  4. Go back to Step 2, stop when the assignment does not change

An Example of k-Means Clustering



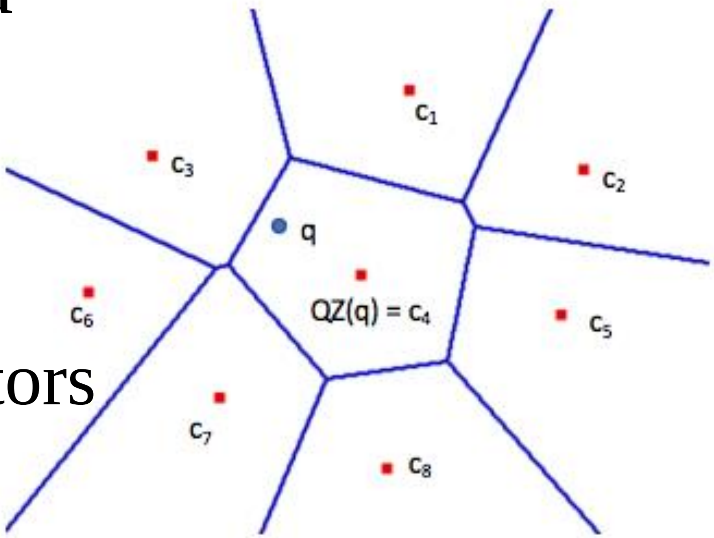
- Partition objects into *k* nonempty subsets
- Repeat
  - Compute centroid (i.e., mean point) for each partition
  - Assign each object to the cluster of its nearest centroid
- Until no change

Vector Quantization

- Encode the vectors
  - Generate a codebook $W = \{c_1, \ldots, c_k\}$ via k-means
  - Assign o to its nearest codeword in W
    - E.g., $QZ(o) = c_i$ $(i \in 1 \ldots k)$ such that $dist(o, c_i) \le dist(o, c_j) \forall j$
  - Represent each vector o by its assigned codeword
- Assume $d = 256, k = 2^{16}$
  - Before: 4 bytes * 256 = 1024 bytes for each vector
  - Now:
    - data: 16 bits = 2 bytes
    - codebook: 4 * 256 * 2^{16}

Vector Quantization – Query Processing

- Given query q, how to find a point close to q?
- Algorithm:
  - Compute $QZ(q)$
  - Candidate set C = all data vectors associated with $QZ(q)$
  - Verification: compute distance between q and $o_i \in C$
    - Requires loading the vectors in C
- Any problem/improvement?



Inverted index: a hash table that maps c_j to a list of $\mathrm{o_i}$ that are associated with c_j

Limitations of VQ

- To achieve better accuracy, fine-grained quantizer with large k is needed
- Large k
  - Costly to run K-means
  - Computing $QZ(q)$ is expensive: $O(kd)$
  - May need to look beyond $QZ(q)$ cell

- Solution:
  - Product Quantization

Product Quantization

- Idea
  - Partition the dimension into m partitions
    - Accordingly a vector => subvectors
  - Use separate VQ with k codewords for each chunk
- Example:
  - 8-dim vector decomposed into $m = 2$ subvectors
  - Each codebook has $k = 4$ codewords, (i.e., $c_{i,j}$)
  - Total space in bits:
    - Data: n $\cdot\, m \cdot log(k)$
    - Codebook: $m \cdot \frac{d}{m} \cdot k \cdot 32$

Example of PQ

2	4	6	5	-2	6	4	1
1	2	1	4	9	-1	2	0
…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…

00	00
01	11
…	…
…	…
…	…

$c_{1,0}$	2.1	3.6	5.3	6.6
$c_{1,1}$	1.2	1.5	2.4	3.3
$c_{1,2}$	…	…	…	…
$c_{1,3}$	…	…	…	…

$c_{2,0}$	3.3	4.1	2.7	1.4
$c_{2,1}$	…	…	…	…
$c_{2,2}$	…	…	…	…
$c_{2,3}$	…	…	…	…

Distance Estimation

- Euclidean distance between a query point q and a data point encoded as t
  - Restore the virtual joint center by looking up each partition of t in the corresponding codebooks => p
  - $d^2(q,t) = \sum_{i=1}^{d}(q_i - p_i)^2$
- Known as Asymmetric Distance Computation (ADC)
  - $d^2(q,t) = \sum_{i=1}^{m}\left(q_{(i)} - c_{i,t(i)}\right)^2$

q	1	3	5	4	-3	7	3	2

t	01	00

$c_{1,0}$	2.1	3.6	5.3	6.6
$c_{1,1}$	1.2	1.5	2.4	3.3
$c_{1,2}$	…	…	…	…
$c_{1,3}$	…	…	…	…
$c_{2,0}$	3.3	4.1	2.7	1.4
$c_{2,1}$	…	…	…	…
$c_{2,2}$	…	…	…	…
$c_{2,3}$	…	…	…	…

p	1.2	1.5	2.4	3.3	3.3	4.1	2.7	1.4

Query Processing

- Compute ADC for every point in the database
  - How?
- Candidate = those with the l smallest AD
- [Optional] Reranking (if $l > 1$):
  - Load the data vectors and compute the actual Euclidean distance
  - Return the one with the smallest distance

Query Processing

q							
1	3	5	4	-3	7	3	2

t_1	01	00
t_2	11	10
	…	…
	…	…

$$\left(q_{(1)} - c_{1,t_1(1)}\right)^2 + \left(q_{(2)} - c_{2,t_1(2)}\right)^2$$

$$\left(q_{(1)} - c_{1,t_2(1)}\right)^2 + \left(q_{(2)} - c_{2,t_2(2)}\right)^2$$

$c_{1,0}$	2.1	3.6	5.3	6.6
$c_{1,1}$	1.2	1.5	2.4	3.3
$c_{1,2}$	…	…	…	…
$c_{1,3}$	…	…	…	…
$c_{2,0}$	3.3	4.1	2.7	1.4
$c_{2,1}$	…	…	…	…
$c_{2,2}$	…	…	…	…
$c_{2,3}$	…	…	…	…

Framework of PQ

- Pre-processing:
  - Step 1: partition data vectors
  - Step 2: generate codebooks (e.g., k-means)
  - Step 3: encode data
- Query
  - Step 1: compute distance between q and codewords
  - Step 2: compute AD for each point and return the candidates
  - Step 3: re-ranking (optional)