Ανάπτυξη Λογισμικού για Δυσεπίλυτα Αλγοριθμικά Προβλήματα

Ενότητα 1: Nearest neighbors and Clustering

Γιάννης Εμίρης

Τμήμα Πληροφορικής & Τηλεπικοινωνιών
Πανεπιστήμιο Αθηνών

October 10, 2021

Outline

Exact and Approximate Nearest Neighbors

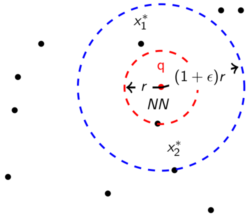
A d-dimensional space D is equipped with distance function $\text{dist}(\cdot,\cdot)$. Given pointset $P \subset D$, for any query point $q \in D$, its exact NN is any point $p_0 \in P$ s.t.:

$$\text{dist}(p_0, q) \leq \text{dist}(p, q), \quad \forall p \in P.$$

For fixed $P \subset D$, and approximation factor $1 > \epsilon > 0$, given a query point q, an ϵ-NN, or Approximate NN, of q is any point $p_0 \in P$:

$$\text{dist}(p_0, q) \leq (1+\epsilon) \cdot \text{dist}(p, q), \quad \forall p \in P.$$

Exact NN is practically linear in n,
while ANN is sublinear or logarithmic in n



Near neighbors

Approximation factor $c = 1 + \epsilon > 1$.

Definition

Given: finite set of points $P \subset D$, approximation factor $c > 1$, radius r.
Input: query point $q \in D$.

(r, c)-Near Neighbor Decision problem with witness:
If $\exists\, p_0$ within radius r, output any $p : dist(q, p) \leq c \cdot r$;
If $\nexists p$ within radius cr, then report NULL;
otherwise output any point p within cr or NULL.

(r, c)-Near Neighbor Range search: Report any subset of $\{p \in P \;:\; dist(q, p) \leq c \cdot r\}$ that includes all $\{p \in P \;:\; dist(q, p) \leq r\}$.

In practice, if c is not given, take $c = 1$.

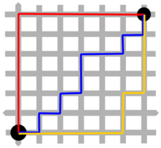
Distances in vector spaces

Definition

The family of ℓ_k norms, for vectors $x, y \in \mathbb{R}^d$, defines various distances:

$$dist_{\ell_k}(x, y) = \|x - y\|_k = \sqrt[k]{\sum_{i=1}^{d} |x_i - y_i|^k}, \quad k \geq 0.$$

Examples:
-- ℓ_2: Euclidean distance,
-- ℓ_1: Manhattan (taxicab) distance,
-- ℓ_∞: max distance.
-- $\ell_0 : \|v\|_0 = \#$nonzero entries in v.



Distance Measure

Definition (Metric)

A distance measure $d : D^2 \to \mathbb{R}$ *is a function that satisfies:*

- *Non-negativity:* $d(x, y) \geq 0$
- *Isolation:* $x \neq y \Leftrightarrow d(x, y) > 0$
- *Symmetry:* $d(x, y) = d(y, x)$
- *Triangle inequality:* $d(x, y) \leq d(x, z) + d(z, y)$

It follows that $d(x, x) = 0$, and $|d(x, z) - d(z, y)| \leq d(x, y)$.

Distances in vector spaces (e.g. ℓ_k, Hamming) are distance measures. Thanks to their (vector) representation, we can also compute attributes, s.t. the mean of a set, or its total order.

NN in $\mathbb{R}$

Sort/store the points (in balanced binary search tree), use binary search for queries, then:

- Prepreprocessing in $O(n \log n)$ time
- Data structure requiring $O(n)$ space
- Answer the query in $O(\log n)$ time

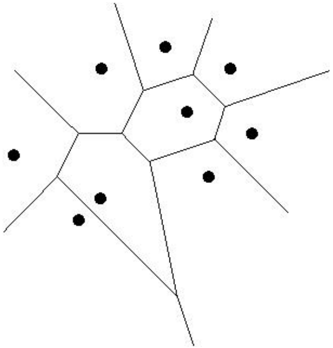
A hash-table with M buckets offers a solution with

- preprocessing in $O(M + n) = O(n)$ time
- space $O(M + n) = O(n)$
- query time $O(1)$

assuming constant time for hashing and constant number of items per bucket.

NN in $\mathbb{R}^2$

- Preprocessing: Voronoi Diagram in $O(n \log n)$.
- Storage $= O(n)$.
- Given query q, find the cell it belongs (point location) in $O(\log n)$.
  NN = site of cell containing q.



NN in $\mathbb{R}^d$

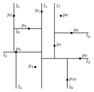
Exact NN:

- Voronoi diagram uses space/preprocessing $= O(n^{\lceil d/2 \rceil})$; point location methods hardly extend to high dimensions.
- State of the art: kd-trees: Sp $= O(dn)$, Query $\simeq O(d \cdot n^{1-1/d})$, tends to linear in n for large d.
- Randomized (Clarkson'88): S $= O(n^{\lceil d/2 \rceil + \delta})$, Q $= O(2^d \log n)$.

Hence the Curse of Dimensionality:
Can we solve NN in poly-time in d and faster than linear-time in n?

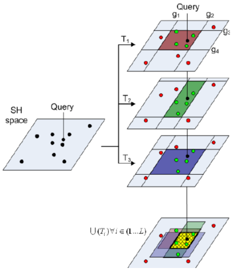
Approximate nearest neighbor (ANN)

- Tree-based data structures: kd-, BBD-trees



  (Arya,Mount et al.'98)
  - Space $= O(dn)$
  - Query $= O((1/\epsilon)^d \log n)$

  . . . still plagued by the curse of dimensionality



- LSH (Indyk,Motwani'98) (Andoni,Indyk et al.'08)
  - Space $= O(dn^{1+\rho})$,
  - Query $= O(dn^{\rho})$, $\rho = 1/(1+\epsilon)^2 < 1$.

  . . . "beats the curse of dimensionality" (Chazelle)

- Dimensionality reduction (E,Psarros et al.'15-19)
  - Space $= O^*(dn)$,
  - Query $= O^*(dn^{\rho})$, $\rho = 1 - \Theta(\epsilon^2)$

  . . . beats the curse in optimal space

Outline

LSH definition

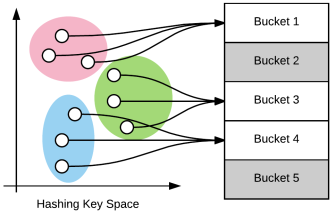
Idea: use hash-table for proximity query, mapping nearby vectors to same bucket

LSH Family

Let $r_1 < r_2$, probabilities $P_1 > P_2$. A family H of functions is (r_1, r_2, P_1, P_2)-sensitive if, for any points $p \neq q$ and any randomly selected function $h \in_R H$,

- if $\mathrm{dist}(p, q) \leq r_1$, then $\mathrm{prob}[h(q) = h(p)] \geq P_1$,
- if $\mathrm{dist}(p, q) \geq r_2$, then $\mathrm{prob}[h(q) = h(p)] \leq P_2$.

$h \in_R H$ is chosen uniformly at random from H.

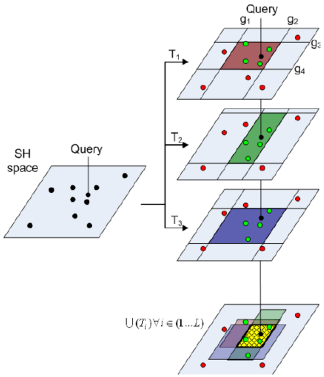


Construction

Preprocess

- Having defined hash-function family H we can construct several $h_j \in_R H$
- Randomly specify L amplified hash-functions $g_1, \ldots, g_L$, each using k h_j's
- Store all points to i-th (1-dim) hash-table using g_i, $i = 1, \ldots, L$

Figure shows intuition behind h_j's:



Practical choices:
$L = 5$ (or 6),
HashTable size $n/8$ (or $n/16$),
aiming at $\Theta(1)$ points per bucket.

NN search

Approximate kNN

```
Input: query $q$
Let $b$ ←Null; $d_b \leftarrow \infty$; initialize $k$ best candidates and distances;
for $i$ from 1 to $L$ do
    for each item $p$ in bucket $g_i(q)$ do
        if dist$(q, p) < d_b = k$-th best distance then $b \leftarrow p$; $d_b \leftarrow$ dist$(q, p)$
        end if
        if large number of retrieved items (e.g. $> 10L$) then return $b$        // optional
        end if
    end for
    return $b$; $k$ best candidates;
end for
```

For $k > 1$ must maintain k best candidate distances sorted so as to always replace k-th farthest candidate neighbor.

Approximate Range Search

Approximate (r, c) Range search

```
Input: $r$, query $q$
for $i$ from 1 to $L$ do
    for each item $p$ in bucket $g_i(q)$ do
        if $\mathrm{dist}(q, p) < r$ then output $p$
        end if
        if large number of retrieved items (e.g. $> 20L$) then return        // optional
        end if
    end for
end for
return
```

This may miss points within radius r.

Amplification

Hash-tables

LSH creates hash-tables using amplified index function g by combining k functions $h_i \in_R H$, chosen uniformly at random with repetition from H. So some h_i may be chosen more than once for some g, or for different g's.

Notice both h_i, g map objects to integers (indices in the hash-table).

Setting $g(p) = [h_1(p), \ldots, h_k(p)]$ defines k-dimensional table with many empty buckets. So, we apply concatenation

$$g(p) = [h_1(p)|h_2(p)| \cdots |h_k(p)],$$

or random linear combination, for some large $M \in \mathbb{N}$, random $r_i \in \mathbb{N}$:

$$g(p) = \sum_{i=1}^{k} r_i h_i(p) \bmod M \quad \in [0, \ldots, M).$$

Practical choice $k = 4$ to 6.

Known LSH-able metrics

- Hamming distance,
- ℓ_2 (Euclidean) distance,
- ℓ_1 (Manhattan) distance,
- ℓ_k distance for any $k \in (1, 2)$,
- ℓ_2 distance on a sphere,
- Cosine similarity,
- Jaccard coefficient.

Recall ℓ_k distance: $$\mathrm{dist}_{\ell_k}(x, y) = \sqrt[k]{\sum_{i=1}^{d} |x_i - y_i|^k}.$$

Outline

Euclidean Space

Recall: $\text{dist}_{\ell_2}(x,y)^2 = \sum_{i=1}^{d}(x_i - y_i)^2$.

Definition

Let d-vector $v \sim \mathcal{N}(0,1)^d$ have coordinates identically independently distributed (i.i.d.) by the standard normal (next slide).
Set "window" $w \in \mathbb{N}^*$ for the entire algorithm, pick single-precision `real` *t uniformly $\in_R [0, w)$. For point $p \in \mathbb{R}^d$, define:*

$$h(p) = \left\lfloor \frac{p \cdot v + t}{w} \right\rfloor \in \mathbb{Z}.$$

-- Essentially project p on the line of v, shift by t, partition into cells of length w.
-- In general, $w = 4$ is good but should increase for range queries with large r.
-- Also $k = 4$ (but can go up to 10), and L may be 5 (up to 30).

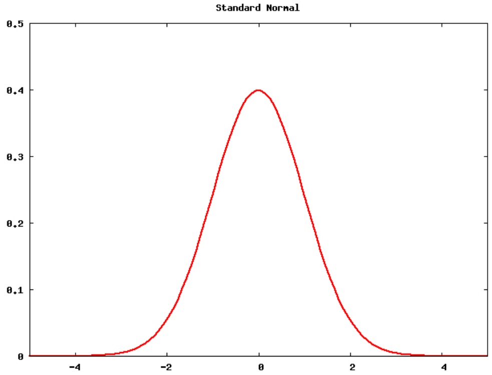
Normal distribution

Vector $v \sim \mathcal{N}(0,1)^d$ has single-precision `real` coordinates distributed according to the standard normal (Gaussian) distribution:

$$v_i \sim \mathcal{N}(0,1), \quad i = 1, 2, \ldots, d,$$

with mean $\mu = 0$, variance $\sigma^2 = 1$ (σ is the standard deviation).

The bell curve:



Hash-table

To avoid many empty buckets we define ϕ as random combination of the h_i's:

Classic hash-function

Build 1-dim hash-table with classic index:

$$\phi(p) = [(r_1 h_1(p) + r_2 h_2(p) + \cdots + r_k h_k(p)) \bmod M] \bmod \text{TableSize},$$

s.t. `int` $r_i \in_R \mathbb{Z}$, prime $M = 2^{32} - 5$ if $h_i(p)$ are `int`, TableSize$= n/4$ (or $n/8$).

ϕ computed in `int` arithmetic, if all $h_i(p), r_i$ are `int` ($\leq$ 32 bits).
Can have smaller TableSize$= n/8$ or $n/16$ (heuristic choice).

Recall $(a \square b) \bmod m = ((a \bmod m) \square (b \bmod m)) \bmod m$

Querying trick

Store object ID along with pointer to object, for all bucket elements.

Object ID

For every p, store

$$\mathrm{ID}(p) = r_1 h_1(p) + r_2 h_2(p) + \cdots + r_k h_k(p) \bmod M.$$

Then indexing hash-function is $\phi(p) = \mathrm{ID}(p) \bmod \mathrm{TableSize}$.

ID is locality sensitive: depends on w-length cells on the v-lines.
To avoid computing Euclidean distance to all elements in the bucket, do it only for p: $\mathrm{ID}(p) = \mathrm{ID}(q)$, assuming such p exists.

Complexity

Theorem

For $P \subset \mathbb{R}^d$, $n = |P|$, the hashtable for $(r, 1+\epsilon)$-Neighbors offers, whp,

$$\textit{queries in } O(dn^{\frac{1}{1+\epsilon}} \log n), \ \textit{space in } O(dn + n^{1+\frac{1}{1+\epsilon}}) \log n.$$

Theorem

We solve $(1+\epsilon)$-ANN by performing $O(\log \frac{n}{\epsilon})$ many queries to $(r, 1+\epsilon)$-Neighbor structures. Binary search is not sufficient.

Corollary

$(1+\epsilon)$-ANN performs queries in $O(dn^{\frac{1}{1+\epsilon}} \log n \log \frac{n}{\epsilon})$ using space in $O(dn + n^{1+\frac{1}{1+\epsilon}} \log^3(\frac{n}{\epsilon})/\epsilon^2)$.

Outline

Randomized projection

Lemma (Johnson,Lindenstrauss'82)

Given $P \subset \mathbb{R}^d$, $|P| = n$, $\epsilon > 0$, $\exists$ *randomized linear* $f : \mathbb{R}^d \to \mathbb{R}^{d'}$, *for*

$$d' = O(\log n/\epsilon^2),$$

s.t., whp, $(1-\epsilon)\|p-q\|_2 \leq \|f(p) - f(q)\|_2 \leq (1+\epsilon)\|p-q\|_2$.

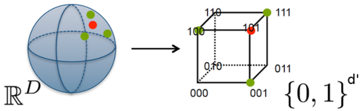
For randomized $d' \times d$ matrix M, projected vectors are Mp for input $p \in \mathbb{R}^d$.

However, the JL Lemma cannot remedy the curse of dimensionality for ANN.

Binary (0/1) hypercube

Projection

- Input: Metric space admitting family of LSH functions h_i.
- Let f_i s.t: for each h_i, $f_i(h_i)$ maps buckets to $\{0, 1\}$ uniformly.
- Preprocess: Store points $p \mapsto [f_1(h_1(p)), \ldots, f_{d'}(h_{d'}(p))] \in \{0, 1\}^{d'}$, so data at hypercube vertices, dimension $d' = \lfloor \lg n \rfloor - 1$ to $\lfloor \lg n \rfloor - 3$.



Search

1. Project query point to corresponding hypercube vertex.
2. Check points in same vertex and nearby vertices in increasing Hamming distance (=1, then 2, etc), until some threshold reached (next slide).
3. ANN returns closest candidate, range search all p within query ball.

Complexity

Theorem

For ℓ_1 and ℓ_2 metrics, a (single) hypercube yields an efficient solution for the ϵ-ANN problem with space and preprocessing in $O^*(dn)$, query time in $O^*(dn^\rho)$, $\rho = 1 - \Theta(\epsilon^2)$. The data structure succeeds with constant probability.

Implementation Parameters

- d': larger implies finer mapping so search can stop earlier (lower thresholds); increases storage and preprocessing.
- Thresholds: $\#$points to be checked in $\mathbb{R}^d$, max $\#$vertices probed, bound on Hamming distance of vertices probed.

The Hamming distance of strings $x, y \in \{0,1\}^k$ equals $\#$ different bits in x, y. Here the vertices of the (unique) hypercube are the hash-table buckets.

Outline

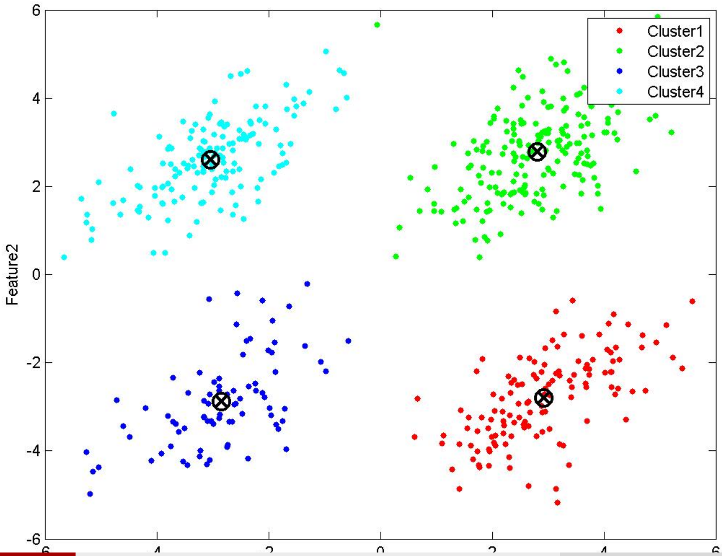
Clustering

Definition (k clusters)

Given n objects (and k > 1) partition the objects into k clusters so as to optimize some objective function.

- Objects in the same cluster are more "similar" (or closer) to each other than to those in other clusters.
- Possible criteria: minimizing the total distance among all cluster points, minimizing the distance of cluster points to some center, etc.
- Variations: k is unknown and computed, e.g., by the Silhouette method.

Good Clustering, with centroids



Approaches

- hierarchical (agglomerative): each point initializes a cluster, merge until stopping criterion, e.g., predetermined number of clusters, or if merging creates cluster with points too far apart.
- centroid-based (point-assignment): given some initial clusters (with centroids), assign points to "best" cluster (nearest centroid); redefine centroids and repeat. Might allow combining / splitting clusters. Example: k-means (our focus).

(Ullman et al:Mining Massive datasets)
(Theodoridis-Koutroumbas:Pattern Recognition)

Outline

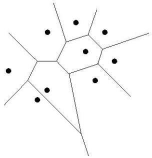
Vector spaces

Problem definition

- Clustering that minimizes objective function.
- k is given.
- Centroids may not be in the input; computed per cluster.

k-means

- k-means is the most common problem: Main algorithms:
  -- Lloyd's algorithm is standard.
  -- Elkan's uses triangular inequality to accelerate updates.
- In Euclidean space, assignment is point location to k Voronoi cells.



Lloyd´s for k-means

Algorithm (EM)

Initialize k centers randomly (or using some strategy).

1. Assignment (Expectation): Assign each object to its nearest center.
2. Update (Maximization): Calculate mean $\frac{1}{T}\sum_{i=1}^{T}\vec{v}_i$ per cluster, make it new center, where $T = \#$objects $\vec{v}_i$ in cluster.

Repeat the two steps until there is no, or little, change in the assignments.

Properties

- Each distance calculation $= O(d)$ because vectors in $\mathbb{R}^d$.
- Assignment $= O(nkd)$, Update $= O(nd)$,
- $\#$iterations unknown, in practice $\ll n$.
- Converges to local minimum in Euclidean space (depends on initialization)

k-means: Objective function

Typical ambient space is $\mathbb{R}^d$ but can generalize to metric space $\mathcal{Z}$.

Minimization function

In any metric space over vectors $\mathcal{Z}$ with distance metric d, let the dataset be $X = \{x_1, \ldots, x_n\} \subseteq \mathcal{X} \subseteq \mathcal{Z}$, and $k > 1$. Given centroids $C \subset \mathcal{Z}$, let

$$\mathrm{d}(x_i, C) = \min_{c \in C} \mathrm{d}(x_i, c).$$

Consider vector $v(C) = [\mathrm{d}(x_1, C), \ldots, \mathrm{d}(x_n, C)]$. The k-means objective is:

$$\min_{C \subseteq \mathcal{Z}, |C| = k} \|v(C)\|_2^2 = \min_{C \subseteq \mathcal{Z}, |C| = k} \sum_{i=1}^{n} \mathrm{d}(x_i, C)^2.$$

The k-means objective is NP-hard to check, but for the ℓ_2 metric, Lloyd's algorithm converges "quickly" to a *local* minimum.

Variations

Recall $X = \{x_i\}$ is input, $v(C) = [\mathrm{d}(x_1, C), \ldots, \mathrm{d}(x_n, C)]$, where $C \subset \mathcal{Z}$ are the k centroids in the ambient space $\mathcal{Z}$. The k-median objective is:

$$\min_{C\subseteq\mathcal{Z},|C|=k} \|v(C)\|_1 = \min_{C\subseteq\mathcal{Z},|C|=k} \sum_{i=1}^{n} \mathrm{d}_{\ell_1}(x_i, C).$$

Further objective functions exist when the centroids belong to the dataset:
k-medoid: $\min_{C\subseteq X,|C|=k} \|v(C)\|_1$.
k-center: $\min_{C\subseteq X,|C|=k} \|v(C)\|_\infty$.

k-medians clustering

Lloyd's Algorithm (EM)

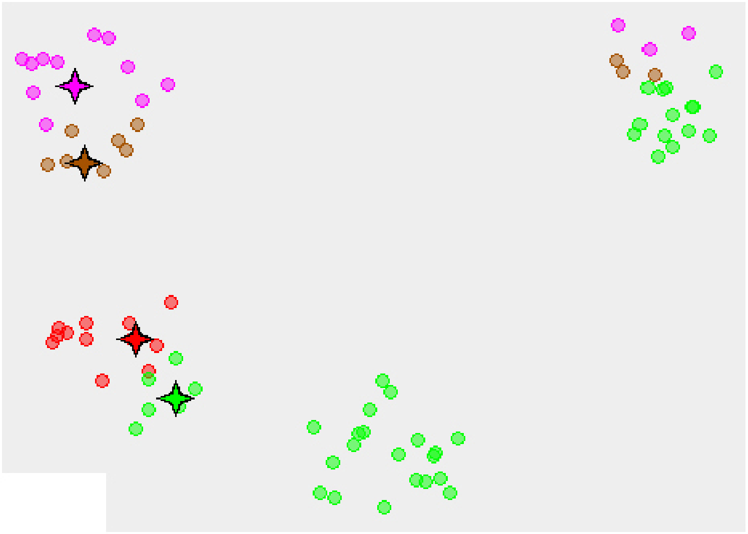
Initialize k centers.

1. Expectation: Assignment of vectors to nearest center in ℓ_1 metric.
2. Maximization: Update center by (marginal) median $c \in \mathbb{R}^d$ defined s.t. c_j is median of $\{v_{1j}, v_{2j}, \ldots, v_{Tj}\}$, $1 \leq j \leq d$, $T =$ cluster cardinality.

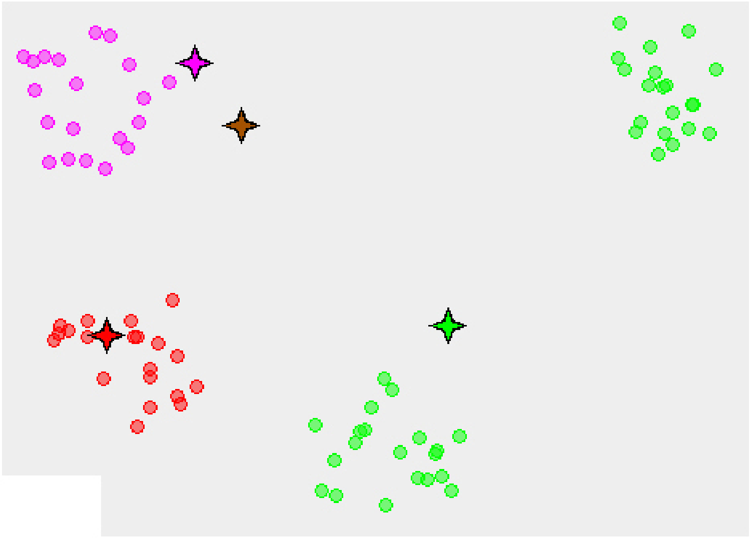
Repeat the two steps until there is no change in the assignments.

Recall the median of n sorted values is the $\lceil n/2 \rceil$-th largest value.

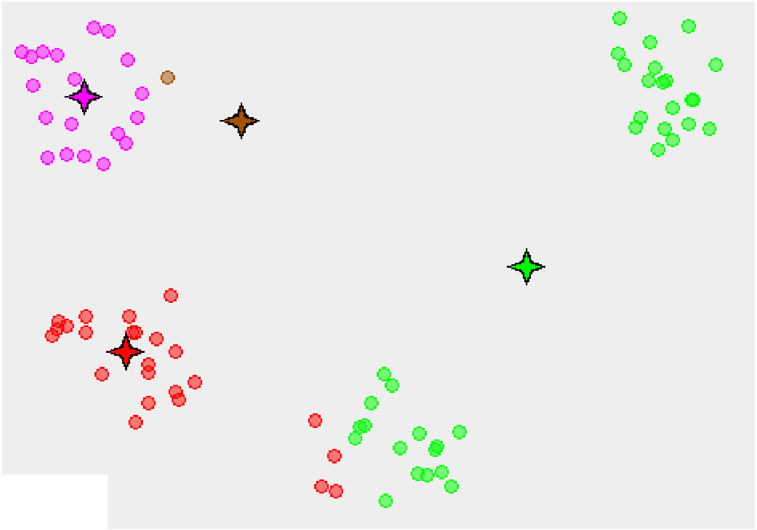
Random initialization $k = 4$



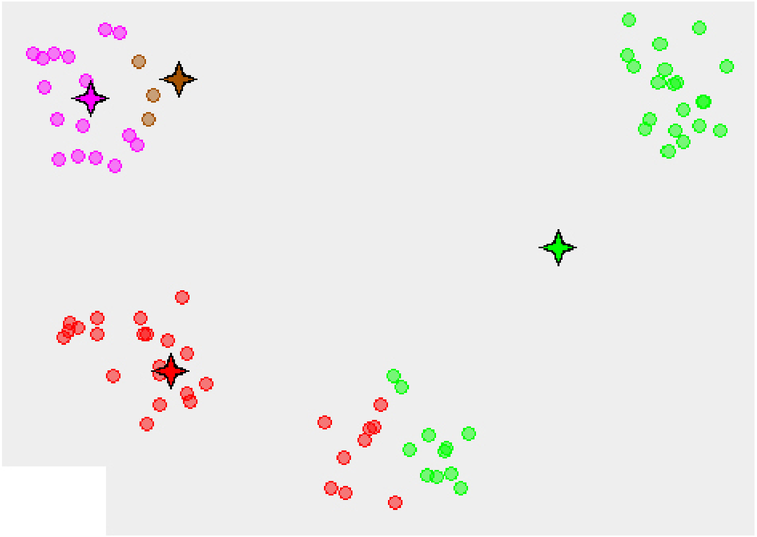
Lloyd's 1



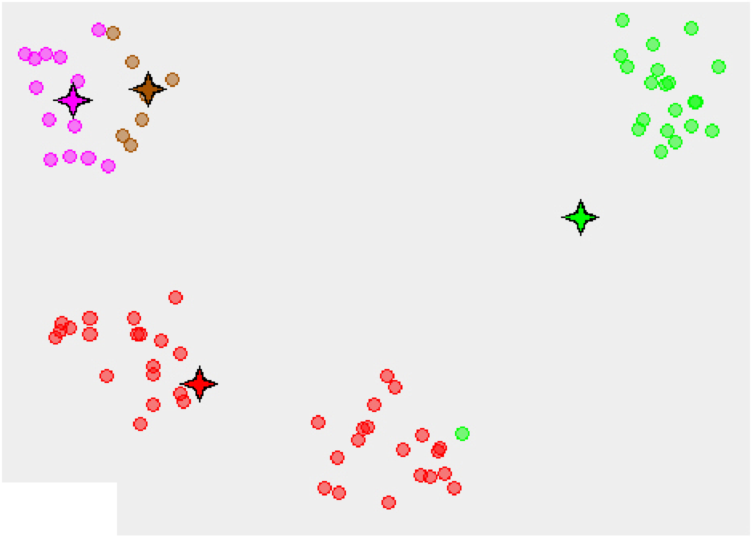
Lloyd’s 2



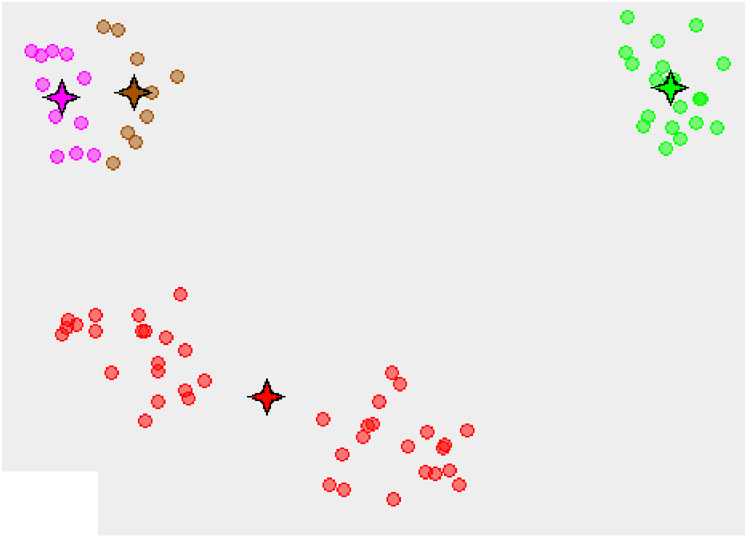
Lloyd's 3



Lloyd's 4



Fails to find 4 clusters



Outline

Outline

Improve Initialization 1: Spread-out

```
INITIALIZATION++ : K-MEANS++ / K-MEDOIDS++:
    (1) Choose a centroid uniformly at random; $t \leftarrow 1$.
    (2) $\forall$ non-centroid point $i = 1, \ldots, n - t$,
            let $D(i) \leftarrow \min$ distance to some centroid, among $t$ chosen centroids.
    (3) Choose new centroid: $r$ chosen with probability proportional to $D(r)^2$:
```

$$\text{prob}[\text{choose } r] = D(r)^2 / \sum_{i=1}^{n-t} D(i)^2.$$

```
    Let $t \leftarrow t + 1$.
    (4) Go to (2) until $t = k =$ given #centroids.
```

Expected approximation ratio $= O(\log k)$ (Arthur-Vassilvitskii:SODA'07)

Similar algo for 2-approx of k-center (NP-hard prob)

Implement initialization++

Given $D(i) > 0$, $i = 1, \ldots, n-t$, compute $n-t$ (`float`) partial sums

$$P(r) = \sum_{i=1}^{r} D(i)^2, \quad r = 1, \ldots, n-t,$$

and store them in binary tree (or array) P. To avoid the $P(r)$'s being very large, we may normalize all $D(i)$'s by dividing them by $\max_i D(i)$.

Pick a uniformly distributed `float` $x \in [0, P(n-t)]$ and return

$$r \in \{1, 2, \ldots, n-t\} \, : \, P(r-1) < x \leq P(r),$$

where $P(0) = 0$; r chosen with probability proportional to $P(r) - P(r-1) \sim D(i)^2$. Can find r by binary search in P.

Improve Initialization 2: Concentrate

Select centroids close to dataset's center of mass (and to each other) as follows.

(1) Calculate symmetric $n \times n$ distance matrix of all objects, i.e. all distances d_{ij} from every object $i = 1, \ldots, n$ to every other object $j = 1, \ldots, n, i \neq j$.
(2) For object i compute

$$v_i = \sum_{j=1}^{n} \frac{d_{ij}}{\sum_{t=1}^{n} d_{jt}}, \qquad i = 1, \ldots, n.$$

(3) Return the k objects with k smallest v_i values.

Algorithm proposed in (Park-Jun'09).

Outline

Assignment by direct method

Exact approach -- for little data

At each iteration (Lloyd's):

1. For every point, compute distance to every centroid.
2. Return (exact) nearest centroid.

Approximate approach -- for big data

At each iteration:

1. Index k centroids into data-structure, e.g. LSH hashtables.
2. For every non-centroid point, run ANN to find nearest centroid.
3. Return (approximate) nearest centroid.

This is the standard approach in almost all big data implementations today.

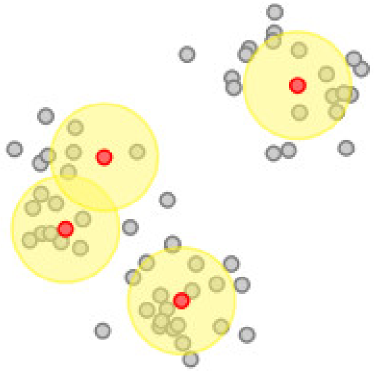
Assignment by Range search

Reverse approach (ANN)

- Index n points into L hashtables: once for entire algorithm.
- LSH TableSize $\leq n/8$: avoid buckets with very few items.
- At each iteration, for each centroid c, range/ball queries centered at c.
- Mark assigned points: either move them at end of LSH buckets (and insert "barrier", or mark them using "flag" field).
- Multiply radii by 2, start with min(dist between centers)/2; centers mapped to buckets once. Until most balls get no new point.
- For a given radius, if a point lies in ≥ 2 balls, compare its distances to the respective centroids, assign to closest centroid.
- At end: for every unassigned point, compare its distances to all centroids

Reverse assignment: example

Double radii until all points assigned, or most balls get no new point, or radius reaches some threshold.



Outline

Silhouette

-- For $1 \leq i \leq n$, $a(i) =$ average ℓ_1 distance of *i* to objects in same cluster
-- Let $b(i) =$ average ℓ_1 distance of *i* to objects in *next best* (neighbor) cluster, i.e. cluster of 2nd closest centroid.

Silhouette of Object *i*

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} = \left\{ \begin{array}{ll} 1 - a(i)/b(i), & \text{if } a(i) < b(i) \\ 0, & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1, & \text{if } a(i) > b(i) \end{array} \right\} \in [-1, 1].$$

Interpret silhouette

$s(i) \rightarrow 1$: *i* seems correctly assigned to its cluster;
$s(i) \simeq 0$: borderline assignment (but not worth to change);
$s(i) \rightarrow -1$: *i* would be better if assigned to next best cluster.

Silhouette: Cluster and clustering

Specific clusters

-- Evaluate a cluster: Compute average $s(i)$ over all i in some cluster.
-- If k is too large or too small, some clusters shall display much smaller silhouettes than the rest.
-- Silhouette plots are used to improve k: try different k's and see if clusters have roughly equal silhouettes.

Overall Clustering

Overall Silhouette coefficient = average $s(i)$ over $i = 1, \ldots, n$.

High if well clustered, low may indicate bad k (or existance of outlier points).