

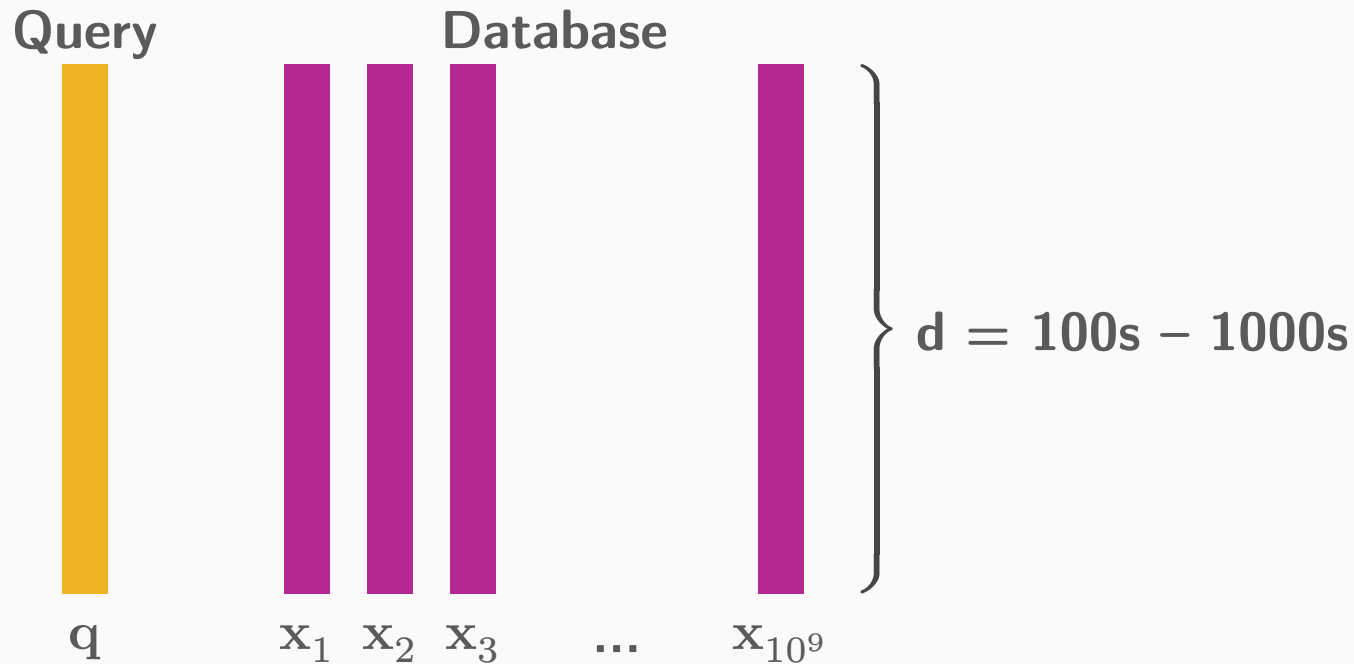
CS-GY 6763: Lecture 6

Near-Neighbor Search in High Dimensions

Pratyush Avi

New York University

Nearest Neighbor Search in High Dimensions



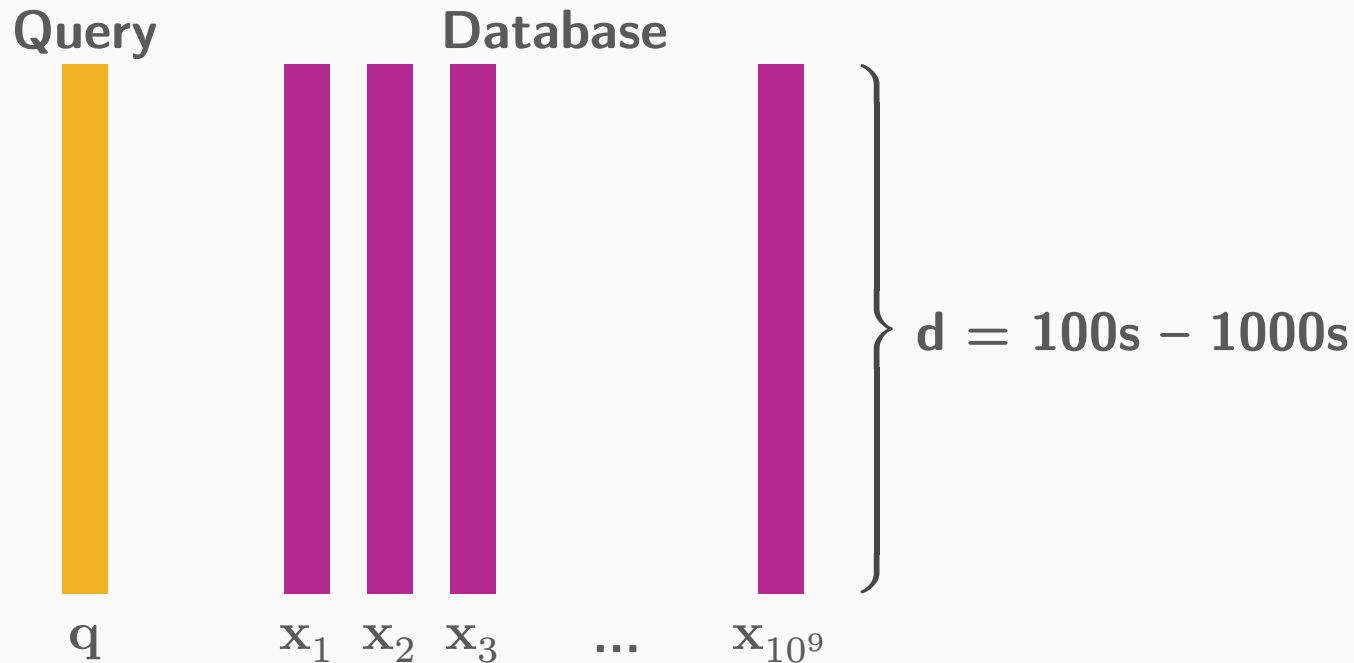
Problem

Find the k closest vectors to a given query $\mathbf{q}$ in a *vector database* $P = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$.

For $k = 1$, return

$$\operatorname{argmin}_{\mathbf{x} \in P} \operatorname{dist}(\mathbf{x}, \mathbf{q})$$

Nearest Neighbor Search in High Dimensions



Problem

Find the k closest vectors to a given query q in a *vector database* $P = \{x_1, \dots, x_n\}$.

For $k = 1$, return

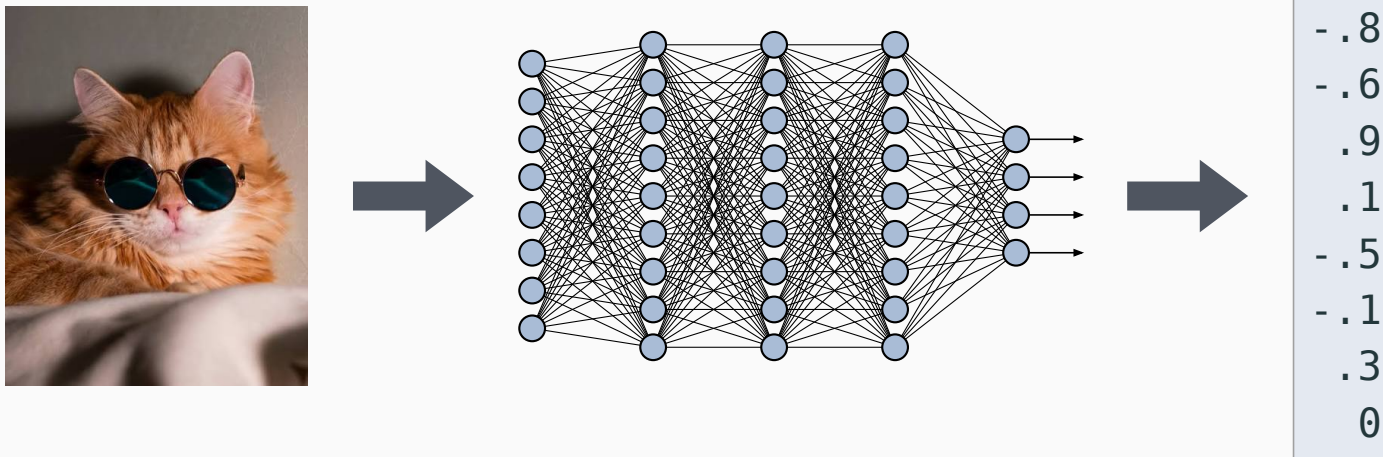
$$\operatorname{argmin}_{x \in P} \operatorname{dist}(x, q)$$

Classical problem with many modern applications.

E.g. multi-modal search, retrieval-augmented generation, Shazam, and so on...

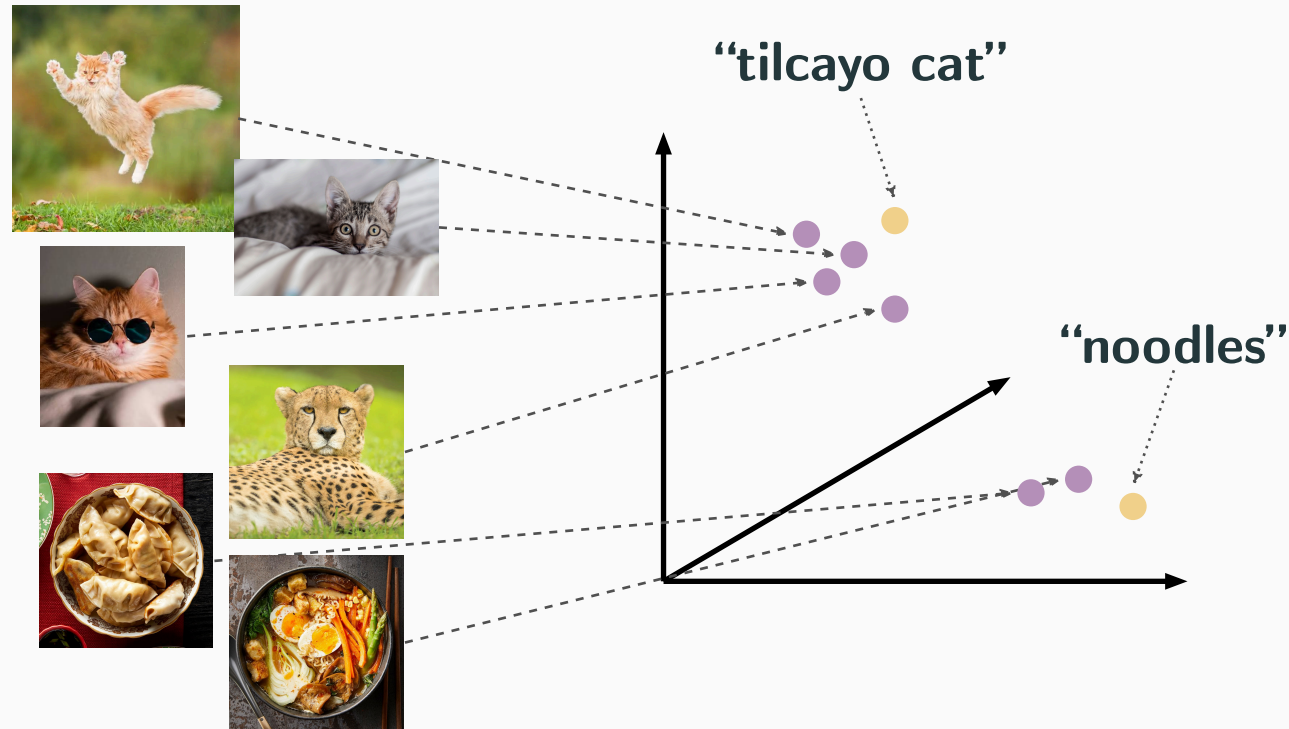
Motivation: AI-Based Search

Deep neural networks (BERT, CLIP, etc.) are used to convert documents, images, videos, and more into high-dimensional, numerical vectors.



AI-Based Search

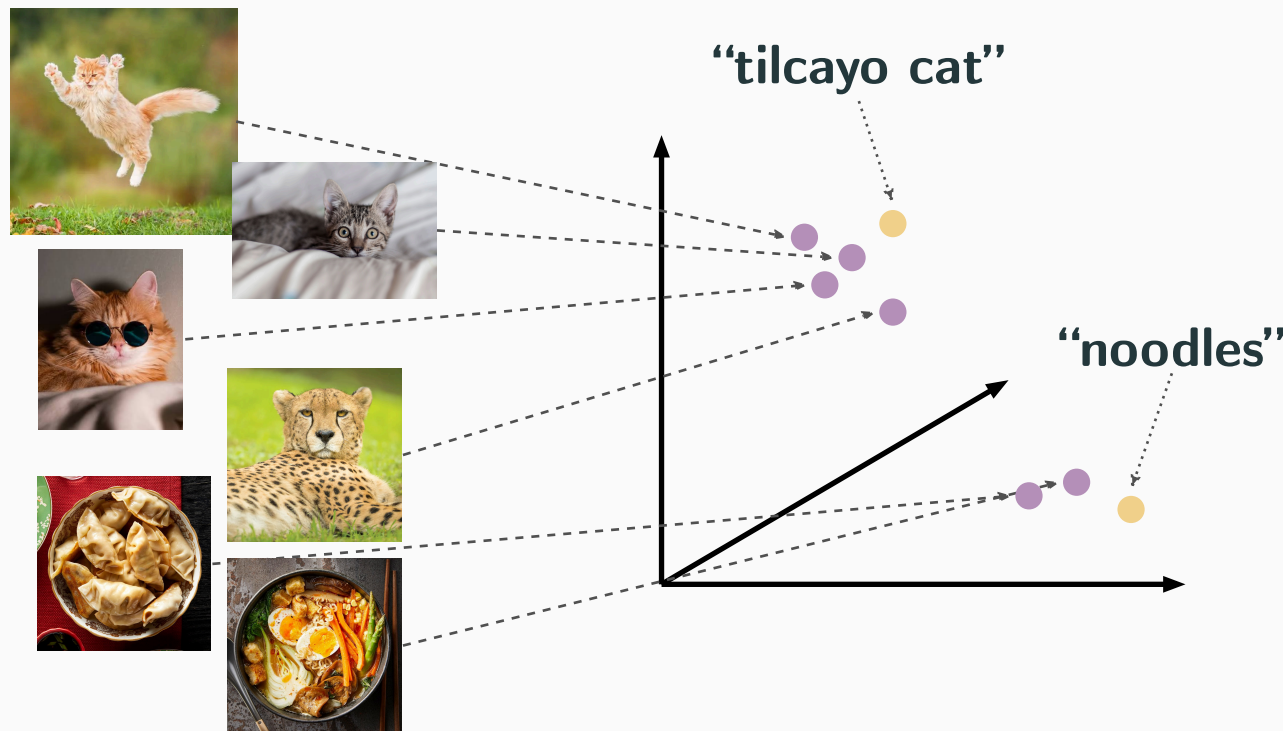
Deep neural networks (BERT, CLIP, etc.) are used to convert documents, images, videos, and more into high-dimensional, numerical vectors.



The network is trained so that semantically similar objects map to similar vectors.

AI-Based Search

Deep neural networks (BERT, CLIP, etc.) are used to convert documents, images, videos, and more into high-dimensional, numerical vectors.

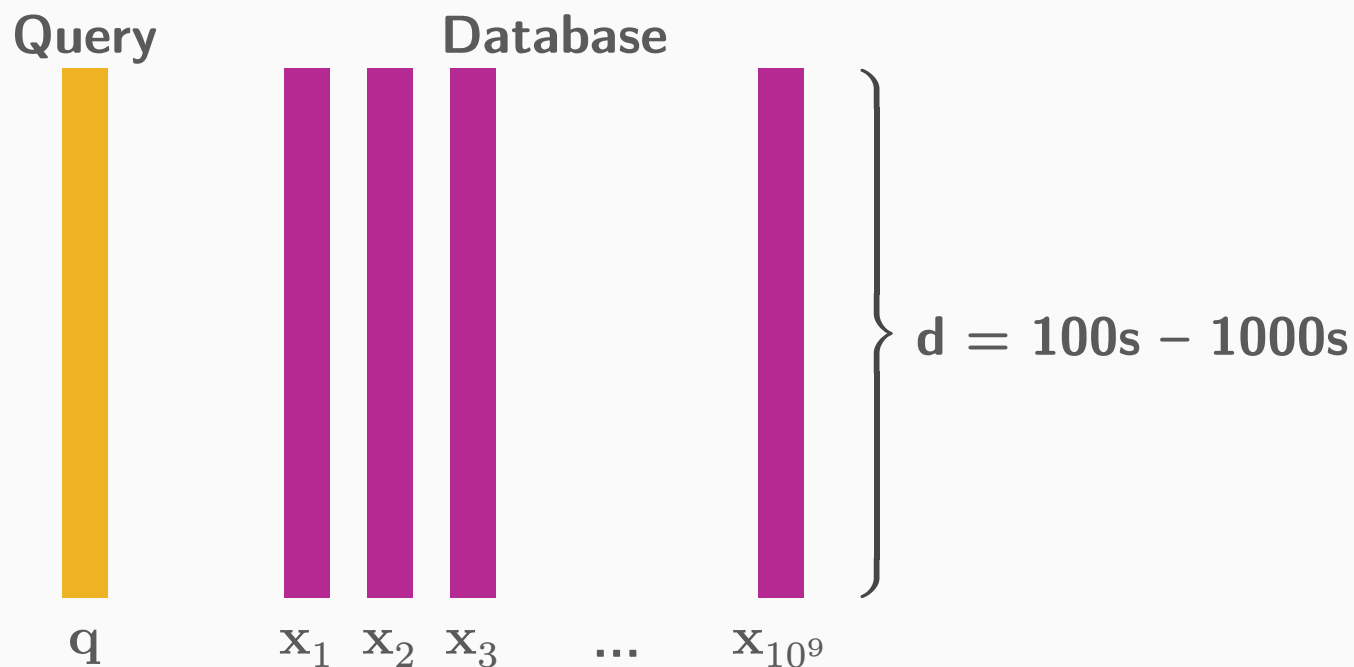


The network is trained so that semantically similar objects map to similar vectors.

We can create vector embeddings for queries as well: e.g. "cat", "cat with glasses", "spicy food", cat images, audio-files of meows etc.

Goal: Find vectors close to our query vector.

Nearest Neighbor Search in High Dimensions



Problem

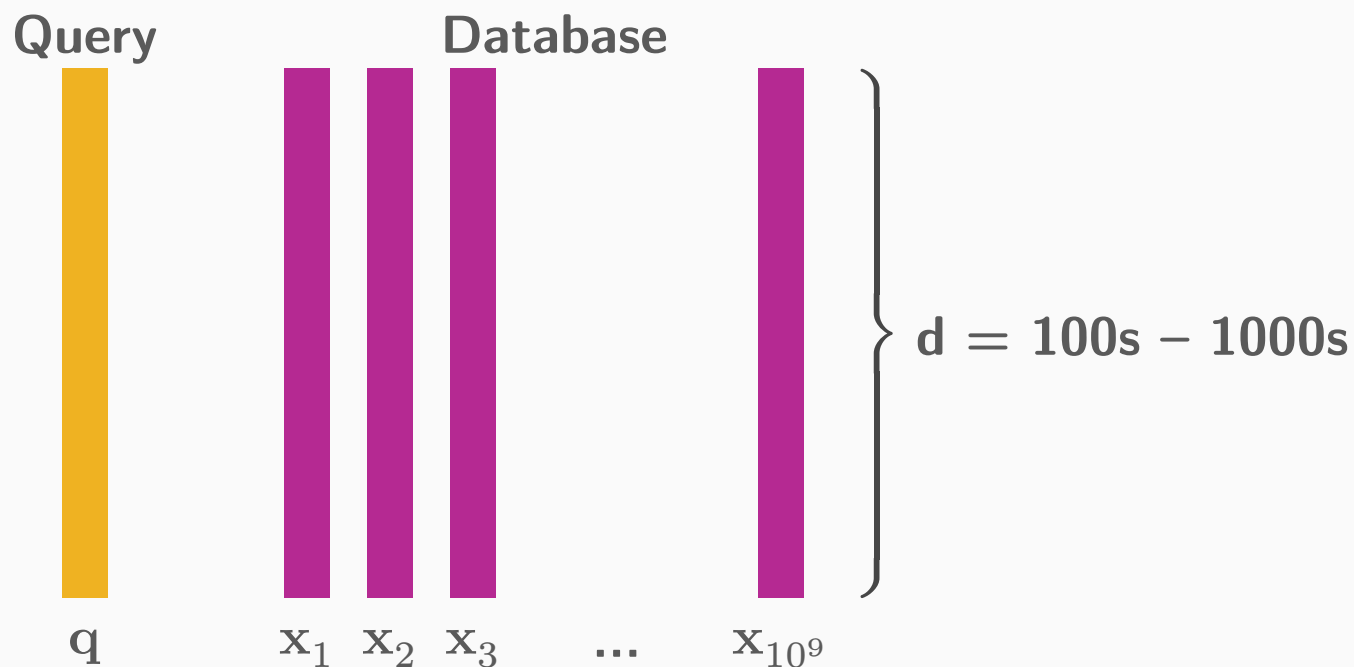
Find the k closest vectors to a given query q in a *vector database* $P = \{x_1, \dots, x_n\}$.

For $k = 1$, return

$$\operatorname{argmin}_{x \in P} \operatorname{dist}(x, q)$$

Naive Search Cost:

Nearest Neighbor Search in High Dimensions



Problem

Find the k closest vectors to a given query q in a *vector database* $P = \{x_1, \dots, x_n\}$.

For $k = 1$, return

$$\operatorname{argmin}_{x \in P} \operatorname{dist}(x, q)$$

Naive Search Cost: $O(nd)$

Dimensionality reduction: Given vectors $\mathbf{x}, \mathbf{y}$, compute small space compressions $C(\mathbf{x}), C(\mathbf{y})$ that can be used to estimate the distance or similarity between $\mathbf{x}$ and $\mathbf{y}$.

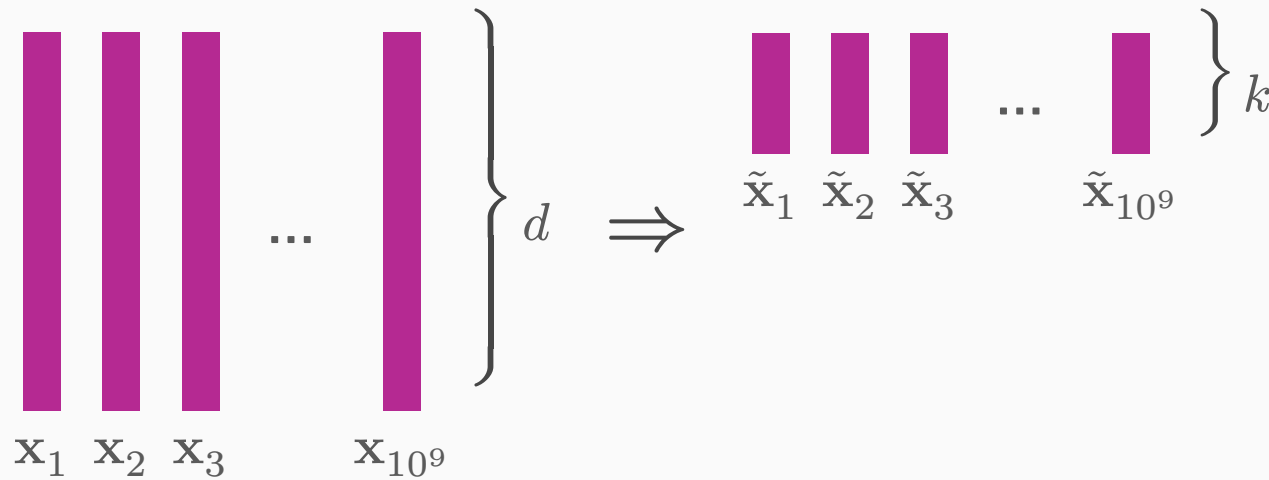
Lemma (Distributional JL Lemma)

Let Π be a random matrix that compresses to $k = O\left(\frac{\log(1/\delta)}{\varepsilon^2}\right)$ rows. Then with probability $(1 - \delta)$:

$$(1 - \varepsilon)\|\mathbf{x} - \mathbf{y}\|_2^2 \leq \|\Pi\mathbf{x} - \Pi\mathbf{y}\|_2^2 \leq (1 + \varepsilon)\|\mathbf{x} - \mathbf{y}\|_2^2$$

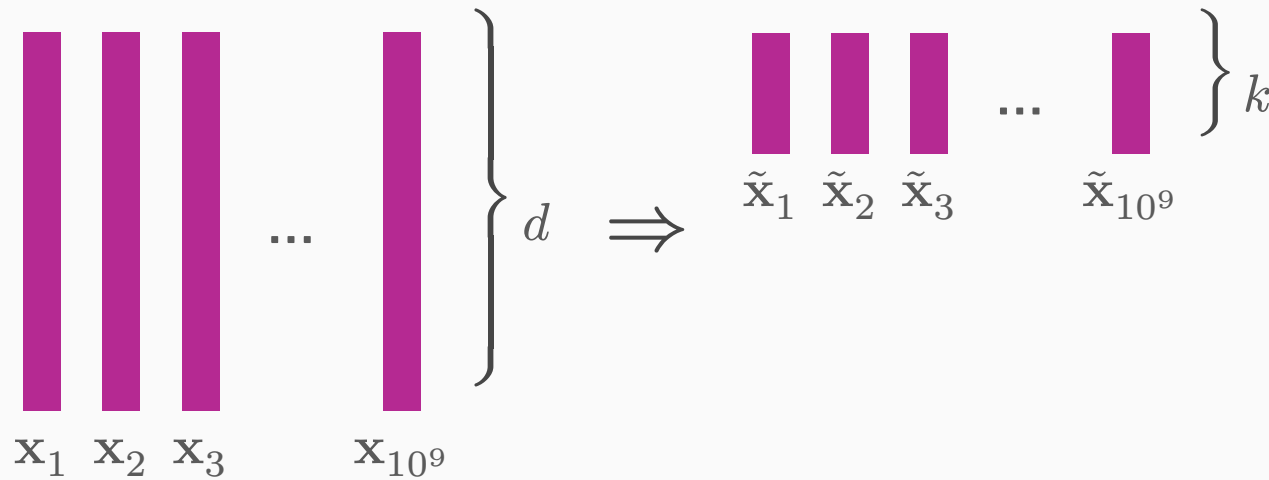
$$O(\log(1/\delta)/\varepsilon^2) \left\{ \begin{array}{c} \text{green box} \\ C(\mathbf{x}) \end{array} \right\} = \underbrace{\Pi}_{d} \begin{array}{c} \text{yellow box} \\ \mathbf{x} \end{array}$$

Key Application: Modern Vector Search



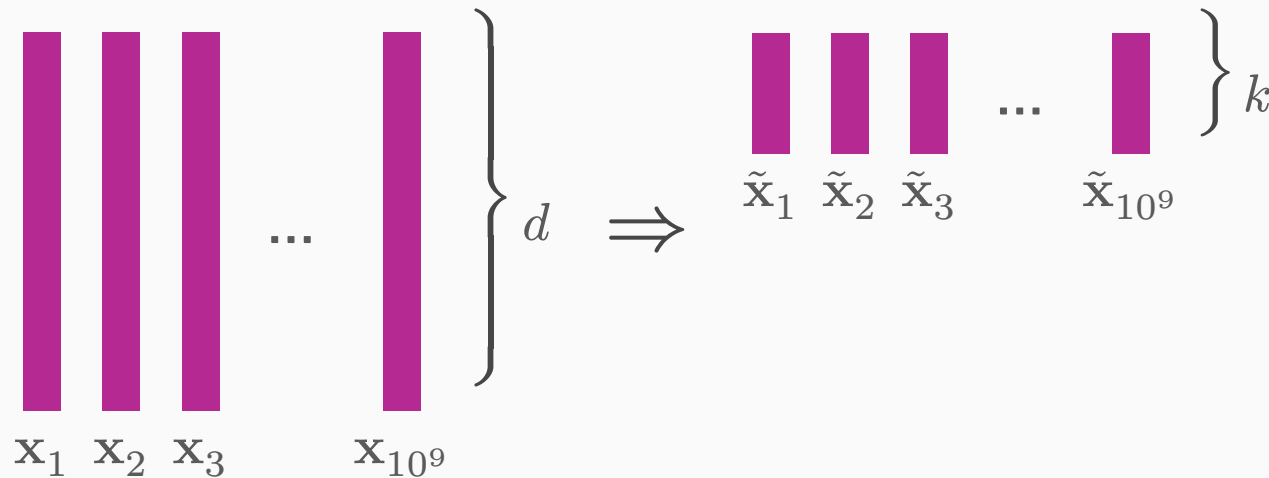
Dimensionality reduction reduces search cost to

Key Application: Modern Vector Search



Dimensionality reduction reduces search cost to $O(nk)$ and reduces space requirements.

Key Application: Modern Vector Search



Dimensionality reduction reduces search cost to $O(nk)$ and reduces space requirements.

All modern vector search systems use “fancier” versions of methods studied in this class:

- Quantized JL/SimHash
- b-bit MinHash
- Product Quantization
- PCA-based methods

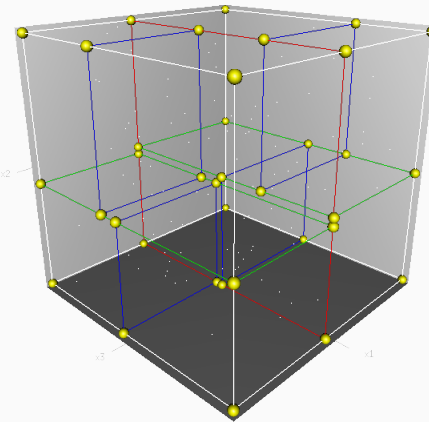
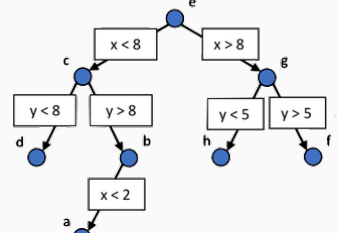
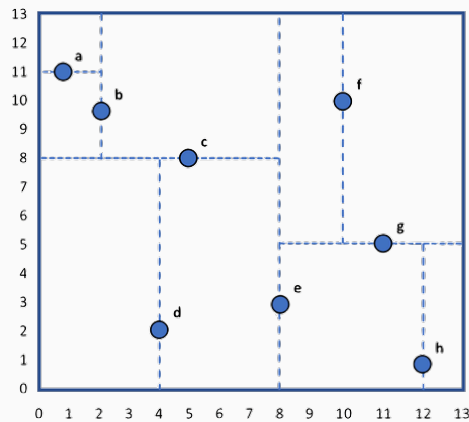
Dimensionality reduction methods are typically paired with *vector indexing methods*.

Goal of Dimensionality Reduction: Reduce **dependence on d** in $O(nd)$ search cost. Reduce space complexity.

Goal of Vector Indexing: Reduce **dependence on n** in $O(nd)$ search cost. Often at the cost of added space complexity.

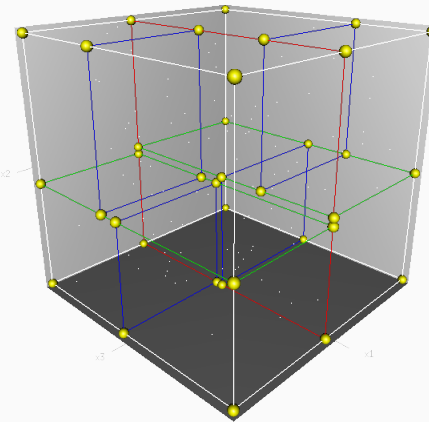
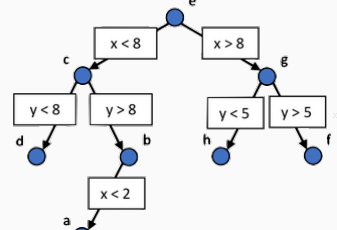
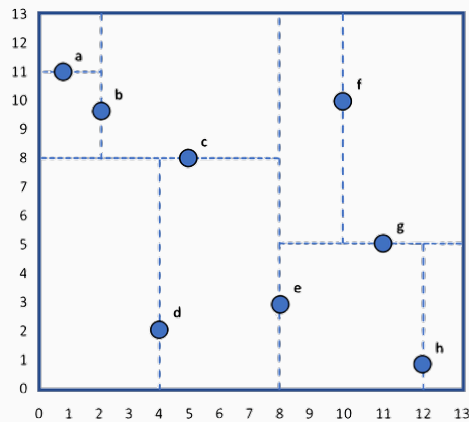
Beyond a Linear Scan

This problem can already be solved in low-dimensions using space partitioning approaches (namely, kd-trees).



Beyond a Linear Scan

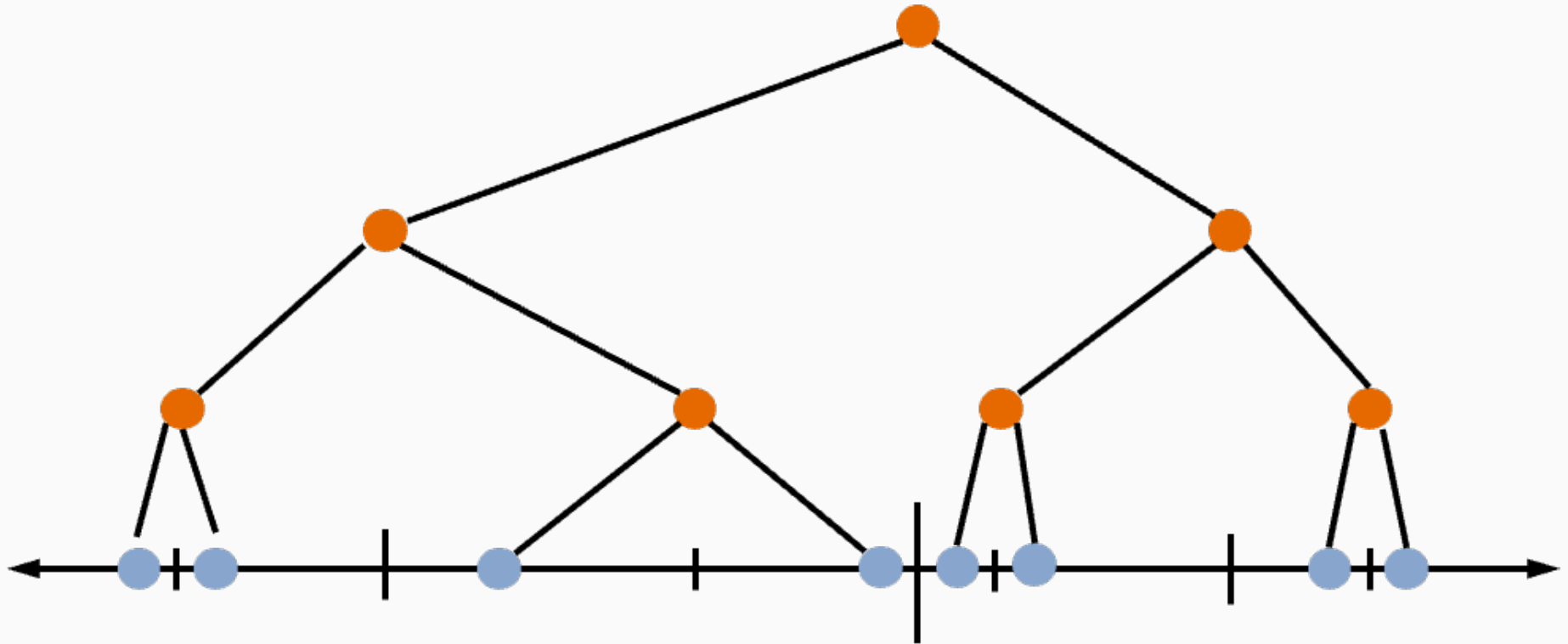
This problem can already be solved in low-dimensions using space partitioning approaches (namely, kd-trees).



Search time is roughly $O(d \cdot \log n \cdot 2^d)$, which is only sublinear for $d = o(\log n)$.

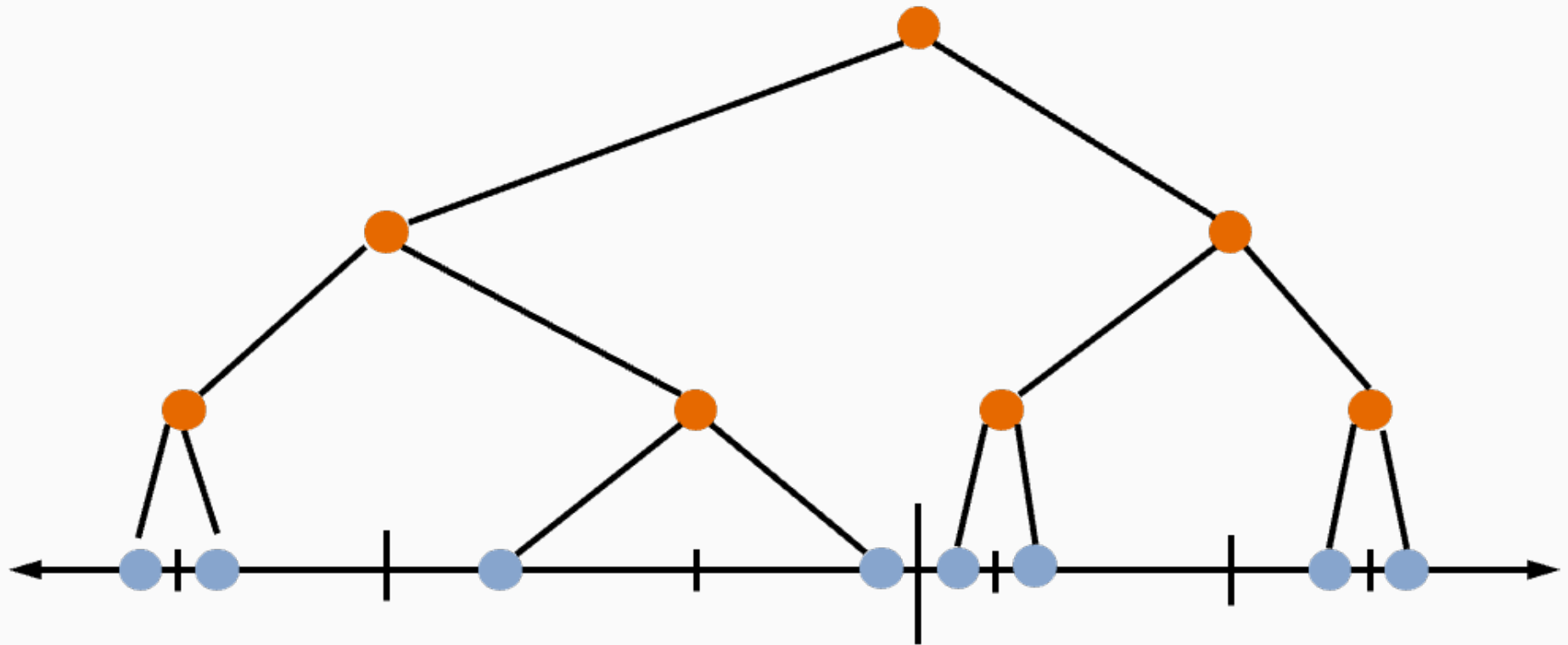
KD-Trees in 1-D

In one-dimension, we can construct a binary tree!



KD-Trees in 1-D

In one-dimension, we can construct a binary tree!

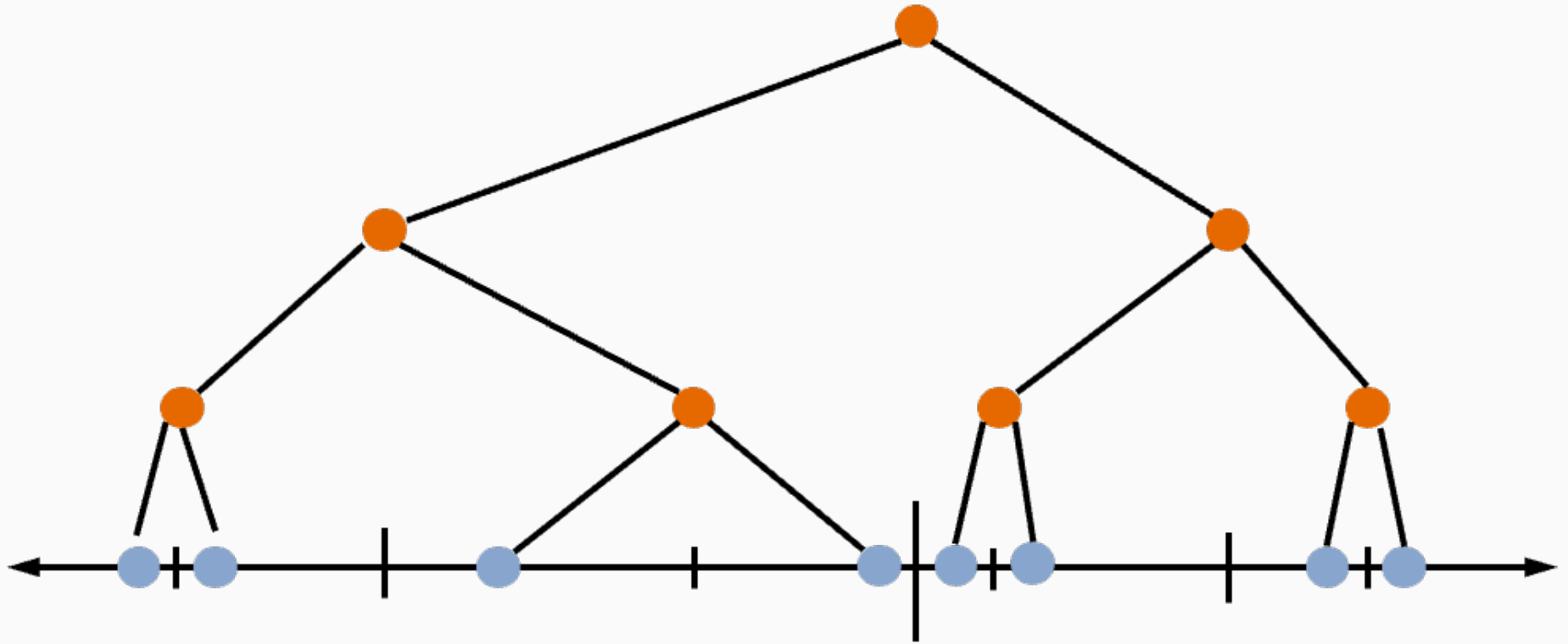


Construction Time: $O(n \log n)$

Space Complexity: $O(n)$

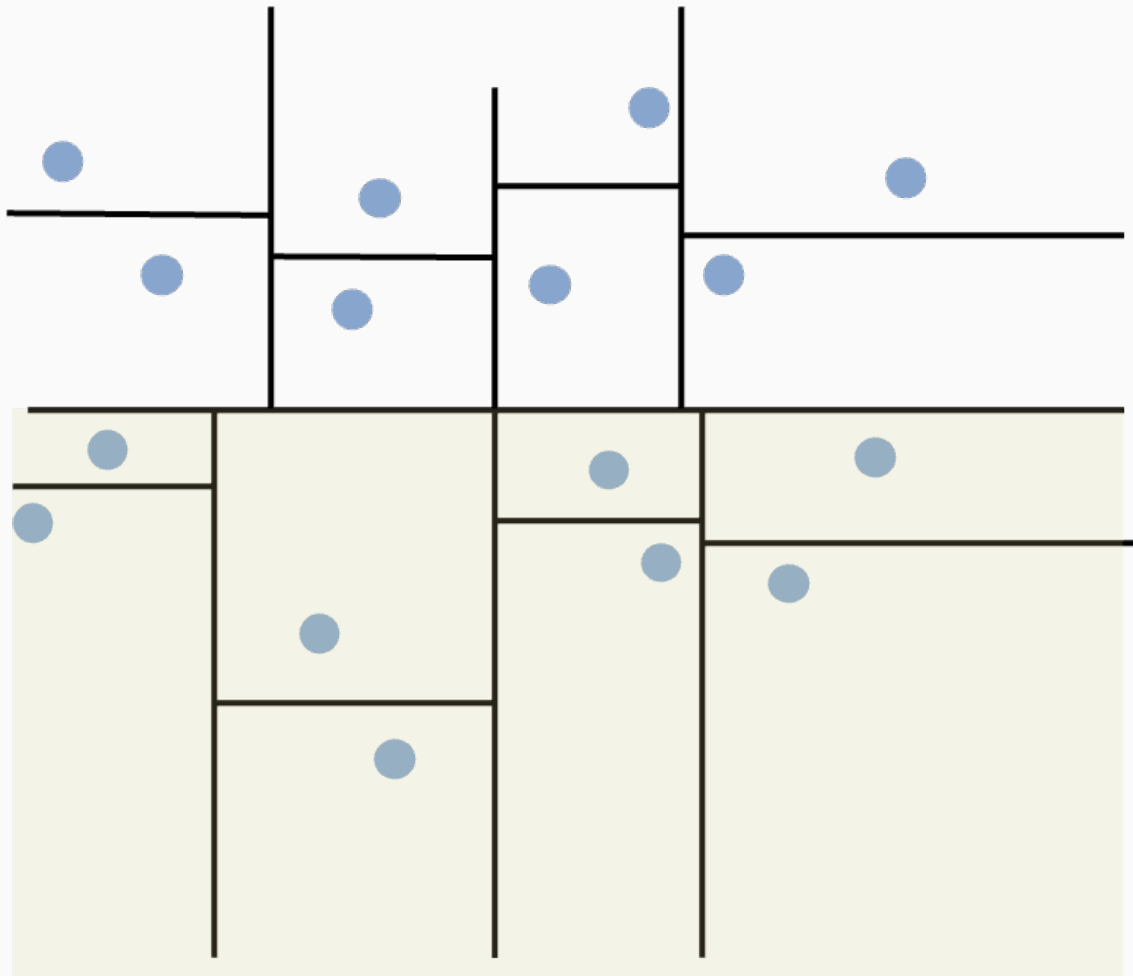
Query Time: $O(\log n)$

Issue with KD-Trees



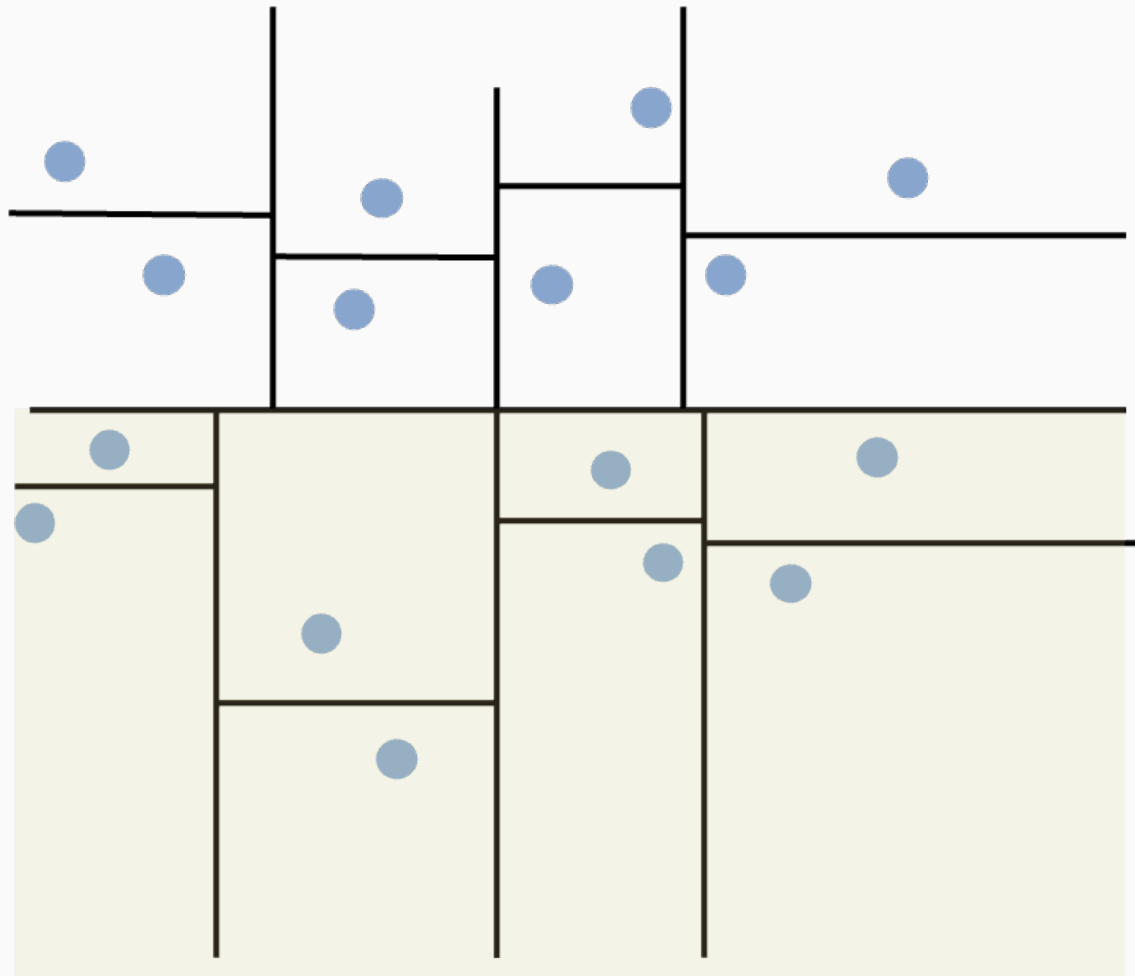
Are we guaranteed to get the nearest neighbor in the worst case?

Issue with KD-Trees



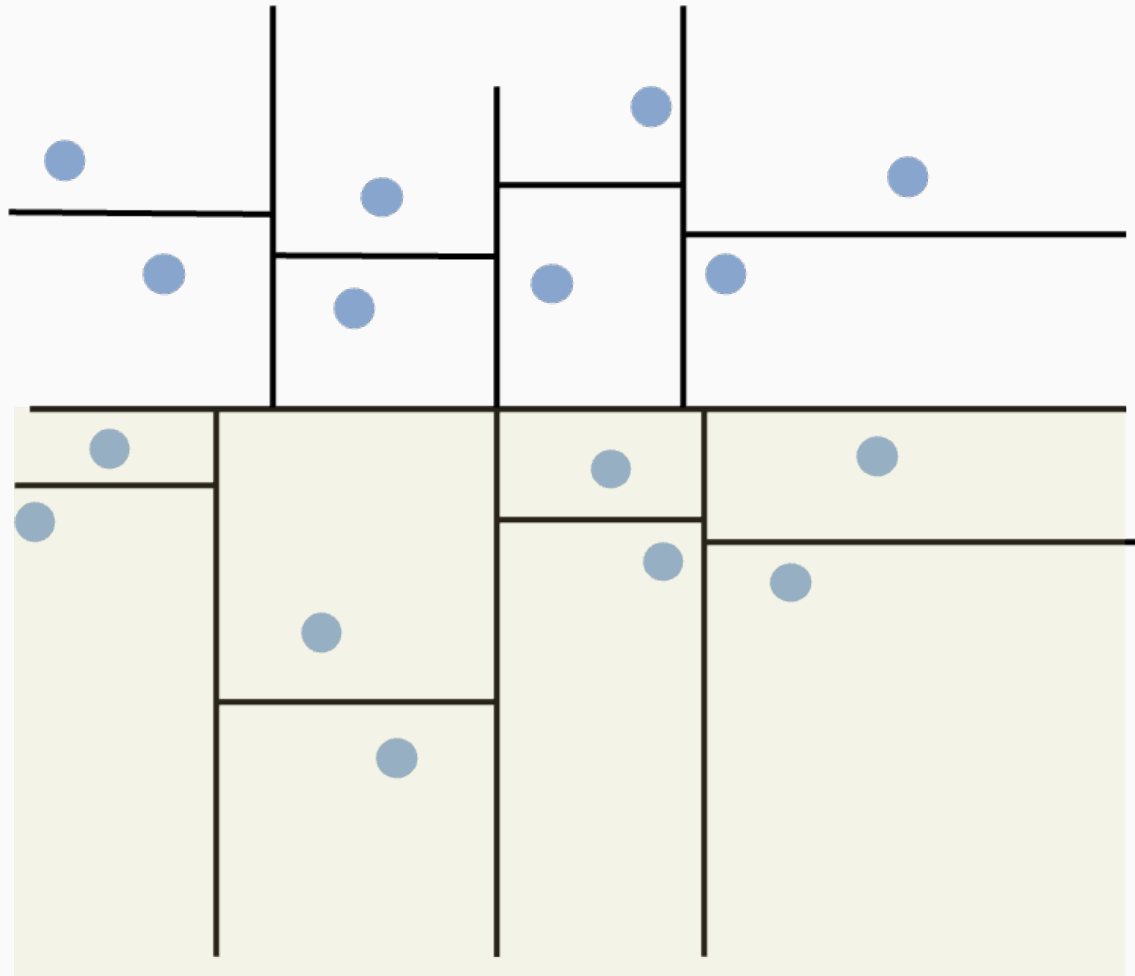
How many boxes do I need to check now?

Issue with KD-Trees



How many boxes do I need to check now?
 2^d in d dimensions, so query time is exponential in d .

Issue with KD-Trees



Theorem (informal): Exact nearest neighbor search in d dimensions requires exponential (in d) space and time.

Only been attacked much more recently:

- **Locality-sensitive hashing [Indyk, Motwani, 1998]**
- Spectral hashing [Weiss, Torralba, and Fergus, 2008]
- Vector quantization [Jégou, Douze, Schmid, 2009]
- **Graph-based vector search [Malkov, Yashunin, 2016, Subramanya et al., 2019]**

Key ideas behind all of these methods:

1. Trade worse space-complexity + preprocessing time for better query time.

Only been attacked much more recently:

- **Locality-sensitive hashing [Indyk, Motwani, 1998]**
- Spectral hashing [Weiss, Torralba, and Fergus, 2008]
- Vector quantization [Jégou, Douze, Schmid, 2009]
- **Graph-based vector search [Malkov, Yashunin, 2016, Subramanya et al., 2019]**

Key ideas behind all of these methods:

1. Trade worse space-complexity + preprocessing time for better query time.
2. Preprocess and create data structure that uses $\Omega(n)$ space.

Only been attacked much more recently:

- **Locality-sensitive hashing [Indyk, Motwani, 1998]**
- Spectral hashing [Weiss, Torralba, and Fergus, 2008]
- Vector quantization [Jégou, Douze, Schmid, 2009]
- **Graph-based vector search [Malkov, Yashunin, 2016, Subramanya et al., 2019]**

Key ideas behind all of these methods:

1. Trade worse space-complexity + preprocessing time for better query time.
2. Preprocess and create data structure that uses $\Omega(n)$ space.
3. Allow for approximation.

Intuitively Why Do Preprocessing and Space Help?

Why does extra space help?

Intuitively Why Do Preprocessing and Space Help?

Why does extra space help?

Question: Dataset is n points in the cube $[-1, 1]^d$. Given ε and a query $\mathbf{q} \in [-1, 1]^d$, you want to find $\tilde{\mathbf{y}}$ with

$$\|\mathbf{q} - \tilde{\mathbf{y}}\|_2 \leq \min_i \|\mathbf{q} - \mathbf{v}_i\|_2 + \varepsilon.$$

Intuitively Why Do Preprocessing and Space Help?

Why does extra space help?

Question: Dataset is n points in the cube $[-1, 1]^d$. Given ε and a query $\mathbf{q} \in [-1, 1]^d$, you want to find $\tilde{\mathbf{y}}$ with

$$\|\mathbf{q} - \tilde{\mathbf{y}}\|_2 \leq \min_i \|\mathbf{q} - \mathbf{v}_i\|_2 + \varepsilon.$$

Can you construct a data structure that supports $O(d)$ time search but uses *exponential space*?

Key Idea: Map close by points in $\mathbb{R}^d$ to the same bucket. Map far away points to different buckets.

Key Idea: Map close by points in $\mathbb{R}^d$ to the same bucket. Map far away points to different buckets.

Why does such a function even exist? It's the complete opposite of uniformly random hash functions.

Let $h : \mathbb{R}^d \rightarrow \{1, \dots, m\}$ be a random hash function.

Locality Sensitive Hash Functions

Let $h : \mathbb{R}^d \rightarrow \{1, \dots, m\}$ be a random hash function.

We call h *locality sensitive* for similarity function $s(\mathbf{q}, \mathbf{y})$ if $\Pr[h(\mathbf{q}) == h(\mathbf{y})]$ is:

- Higher when $\mathbf{q}$ and $\mathbf{y}$ are more similar, i.e. $s(\mathbf{q}, \mathbf{y})$ is higher.

Locality Sensitive Hash Functions

Let $h : \mathbb{R}^d \rightarrow \{1, \dots, m\}$ be a random hash function.

We call h *locality sensitive* for similarity function $s(\mathbf{q}, \mathbf{y})$ if $\Pr[h(\mathbf{q}) == h(\mathbf{y})]$ is:

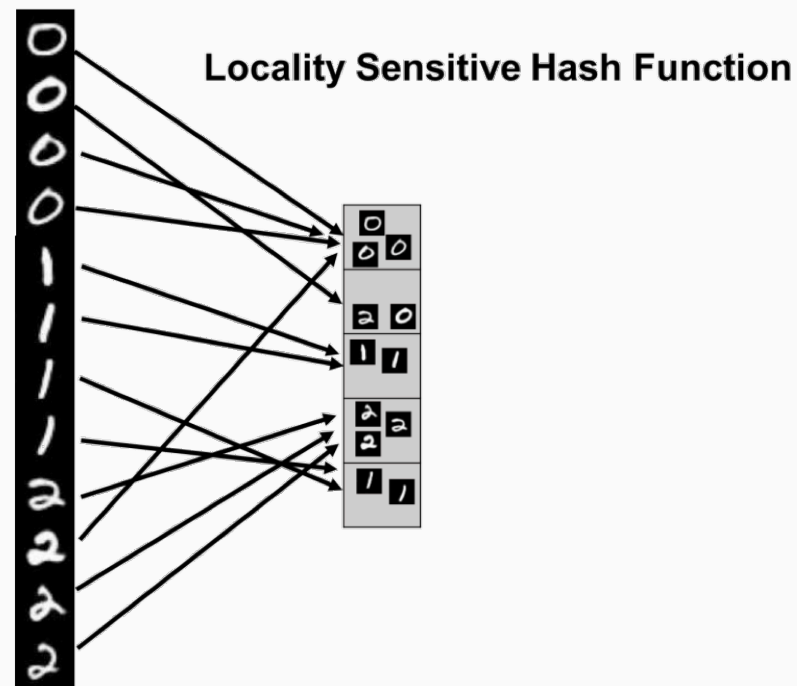
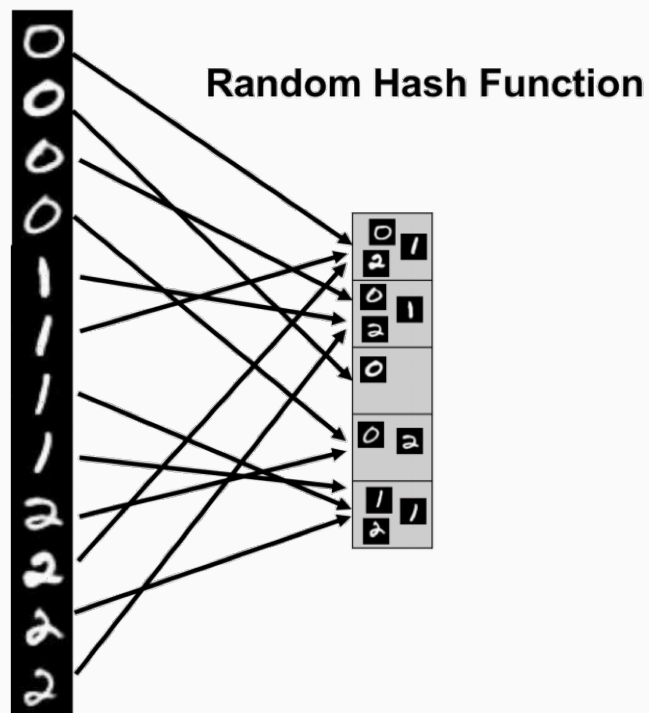
- Higher when $\mathbf{q}$ and $\mathbf{y}$ are more similar, i.e. $s(\mathbf{q}, \mathbf{y})$ is higher.
- Lower when $\mathbf{q}$ and $\mathbf{y}$ are more dissimilar, i.e. $s(\mathbf{q}, \mathbf{y})$ is lower.

Locality Sensitive Hash Functions

Let $h : \mathbb{R}^d \rightarrow \{1, \dots, m\}$ be a random hash function.

We call h *locality sensitive* for similarity function $s(\mathbf{q}, \mathbf{y})$ if $\Pr[h(\mathbf{q}) == h(\mathbf{y})]$ is:

- Higher when $\mathbf{q}$ and $\mathbf{y}$ are more similar, i.e. $s(\mathbf{q}, \mathbf{y})$ is higher.
- Lower when $\mathbf{q}$ and $\mathbf{y}$ are more dissimilar, i.e. $s(\mathbf{q}, \mathbf{y})$ is lower.



LSH for $s(\mathbf{q}, \mathbf{y})$ equal to Jaccard similarity:

- Let $c : \{0, 1\}^d \rightarrow [0, 1]$ be a single instantiation of MinHash.

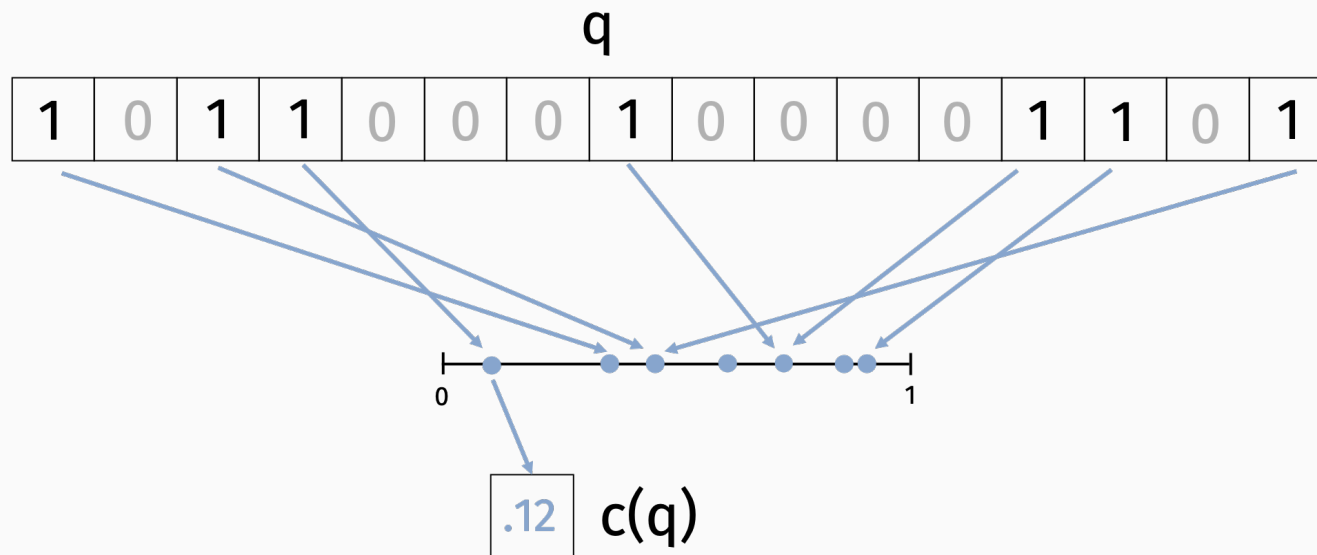
LSH for $s(\mathbf{q}, \mathbf{y})$ equal to Jaccard similarity:

- Let $c : \{0, 1\}^d \rightarrow [0, 1]$ be a single instantiation of MinHash.
- Let $g : [0, 1] \rightarrow \{1, \dots, m\}$ be a uniform random hash function.

Locality Sensitive Hash Functions

LSH for $s(\mathbf{q}, \mathbf{y})$ equal to Jaccard similarity:

- Let $c : \{0, 1\}^d \rightarrow [0, 1]$ be a single instantiation of MinHash.
- Let $g : [0, 1] \rightarrow \{1, \dots, m\}$ be a uniform random hash function.
- Let $h(\mathbf{q}) = g(c(\mathbf{q}))$.

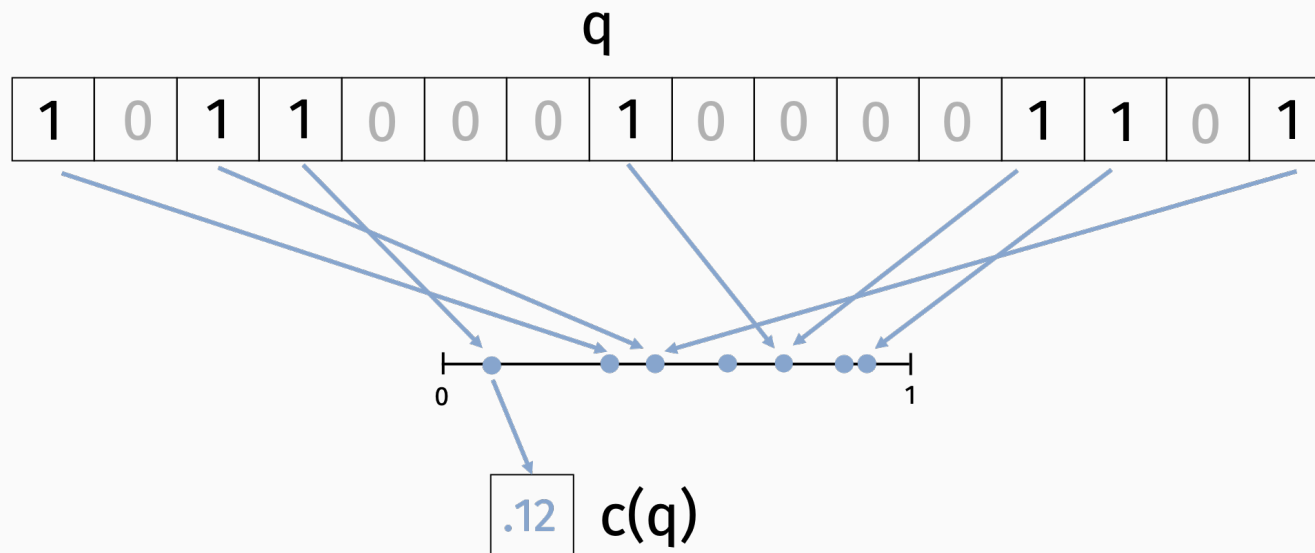


Why should this be locality sensitive?

Locality Sensitive Hash Functions

LSH for $s(\mathbf{q}, \mathbf{y})$ equal to Jaccard similarity:

- Let $c : \{0, 1\}^d \rightarrow [0, 1]$ be a single instantiation of MinHash.
- Let $g : [0, 1] \rightarrow \{1, \dots, m\}$ be a uniform random hash function.
- Let $h(\mathbf{q}) = g(c(\mathbf{q}))$.



$\Pr[c(\mathbf{q}) == c(\mathbf{y})] = J(\mathbf{q}, \mathbf{y})$ and $c(\mathbf{q}) = c(\mathbf{y})$ implies $g(c(\mathbf{q})) = g(c(\mathbf{y}))$.

LSH for Jaccard similarity:

- Let $c : \{0, 1\}^d \rightarrow [0, 1]$ be a single instantiation of MinHash.
- Let $g : [0, 1] \rightarrow \{1, \dots, m\}$ be a uniform random hash function.
- Let $h(\mathbf{x}) = g(c(\mathbf{x}))$.

If $J(\mathbf{q}, \mathbf{y}) = v$,

$$\Pr[h(\mathbf{q}) == h(\mathbf{y})] = ??$$

Locality Sensitive Hash Functions

LSH for Jaccard similarity:

- Let $c : \{0, 1\}^d \rightarrow [0, 1]$ be a single instantiation of MinHash.
- Let $g : [0, 1] \rightarrow \{1, \dots, m\}$ be a uniform random hash function.
- Let $h(\mathbf{x}) = g(c(\mathbf{x}))$.

If $J(\mathbf{q}, \mathbf{y}) = v$,

$$\Pr[h(\mathbf{q}) == h(\mathbf{y})] = ??$$

$$= J(\mathbf{q}, \mathbf{y}) + \underbrace{(1 - J(\mathbf{q}, \mathbf{y})) \frac{1}{m}}_{\text{negligible}}$$

Basic approach for LSH-based near neighbor search in a database.

Pre-processing:

- Select random LSH function $h : \{0, 1\}^d \rightarrow 1, \dots, m$.
- Create table T with $m = O(n)$ slots.¹
- For $i = 1, \dots, n$, insert $\mathbf{q}_i$ into $T(h(\mathbf{q}_i))$.

¹Enough to make the $O(1/m)$ term negligible.

Basic approach for LSH-based near neighbor search in a database.

Pre-processing:

- Select random LSH function $h : \{0, 1\}^d \rightarrow 1, \dots, m$.
- Create table T with $m = O(n)$ slots.¹
- For $i = 1, \dots, n$, insert $\mathbf{q}_i$ into $T(h(\mathbf{q}_i))$.

Query: Want to find near neighbors of input $\mathbf{y} \in \{0, 1\}^d$.

¹Enough to make the $O(1/m)$ term negligible.

Basic approach for LSH-based near neighbor search in a database.

Pre-processing:

- Select random LSH function $h : \{0, 1\}^d \rightarrow 1, \dots, m$.
- Create table T with $m = O(n)$ slots.¹
- For $i = 1, \dots, n$, insert $\mathbf{q}_i$ into $T(h(\mathbf{q}_i))$.

Query: Want to find near neighbors of input $\mathbf{y} \in \{0, 1\}^d$.

Retrieval:

- Hash $\mathbf{y}$ to get $h(\mathbf{y})$.
- Go to slot $T(h(\mathbf{y}))$. Linear scan through all vectors $\mathbf{q} \in T(h(\mathbf{y}))$ and return any that are close to $\mathbf{y}$.

¹Enough to make the $O(1/m)$ term negligible.

Basic approach for LSH-based near neighbor search in a database.

Pre-processing:

- Select random LSH function $h : \{0, 1\}^d \rightarrow 1, \dots, m$.
- Create table T with $m = O(n)$ slots.¹
- For $i = 1, \dots, n$, insert $\mathbf{q}_i$ into $T(h(\mathbf{q}_i))$.

Query: Want to find near neighbors of input $\mathbf{y} \in \{0, 1\}^d$.

Retrieval:

- Hash $\mathbf{y}$ to get $h(\mathbf{y})$.
- Go to slot $T(h(\mathbf{y}))$. Linear scan through all vectors $\mathbf{q} \in T(h(\mathbf{y}))$ and return any that are close to $\mathbf{y}$.

Time required is $O(d \cdot |T(h(\mathbf{y}))|)$.

¹Enough to make the $O(1/m)$ term negligible.

Two main considerations:

- **False Negative Rate:** What's the probability we do not find a vector that *is close* to y ?
 - A higher false negative rate means we miss near neighbors.

Two main considerations:

- **False Negative Rate:** What's the probability we do not find a vector that *is close* to $\mathbf{y}$?
 - A higher false negative rate means we miss near neighbors.
- **False Positive Rate:** What's the probability that a vector in $T(h(\mathbf{y}))$ *is not close* to $\mathbf{y}$?
 - A higher false positive rate means increased runtime – we need to compute $S(\mathbf{q}, \mathbf{y})$ for every $\mathbf{q} \in T(h(\mathbf{y}))$ to check if it's actually close to $\mathbf{y}$.

Reducing False Negative Rate

Let's use Jaccard similarity as a running example.

Suppose the nearest database point $\mathbf{q}$ has $J(\mathbf{y}, \mathbf{q}) = .4$.

What's the probability we do not find $\mathbf{q}$?

$$\Pr[\text{We find } \mathbf{q}] = \Pr[h(\mathbf{q}) == h(\mathbf{y})] \approx 0.4$$

Reducing False Negative Rate

Let's use Jaccard similarity as a running example.

Suppose the nearest database point $\mathbf{q}$ has $J(\mathbf{y}, \mathbf{q}) = .4$.

What's the probability we do not find $\mathbf{q}$?

$$\Pr[\text{We find } \mathbf{q}] = \Pr[h(\mathbf{q}) == h(\mathbf{y})] \approx 0.4$$

$$\Pr[\text{We do not find } \mathbf{q}] \approx 1 - 0.4 = 0.6$$

Reducing False Negative Rate

Let's use Jaccard similarity as a running example.

Suppose the nearest database point $\mathbf{q}$ has $J(\mathbf{y}, \mathbf{q}) = .4$.

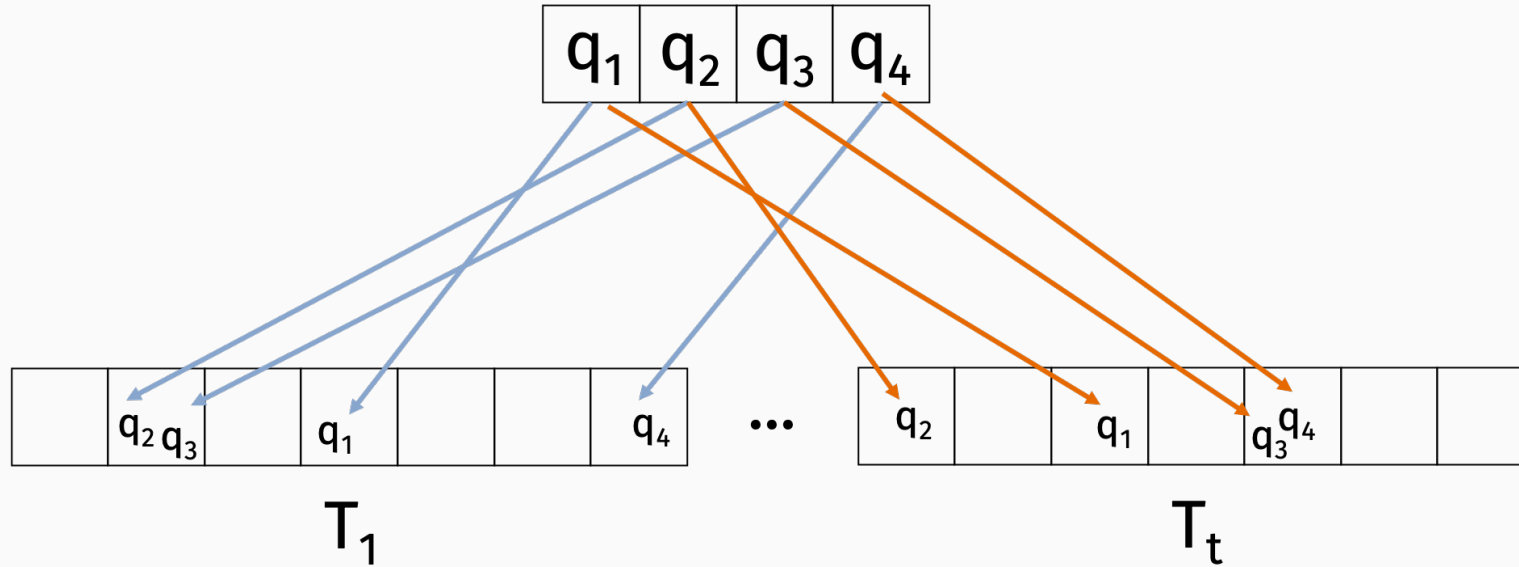
What's the probability we do not find $\mathbf{q}$?

$$\Pr[\text{We find } \mathbf{q}] = \Pr[h(\mathbf{q}) == h(\mathbf{y})] \approx 0.4$$

$$\Pr[\text{We do not find } \mathbf{q}] \approx 1 - 0.4 = 0.6$$

This is quite bad, how do we dampen the false negative rate?

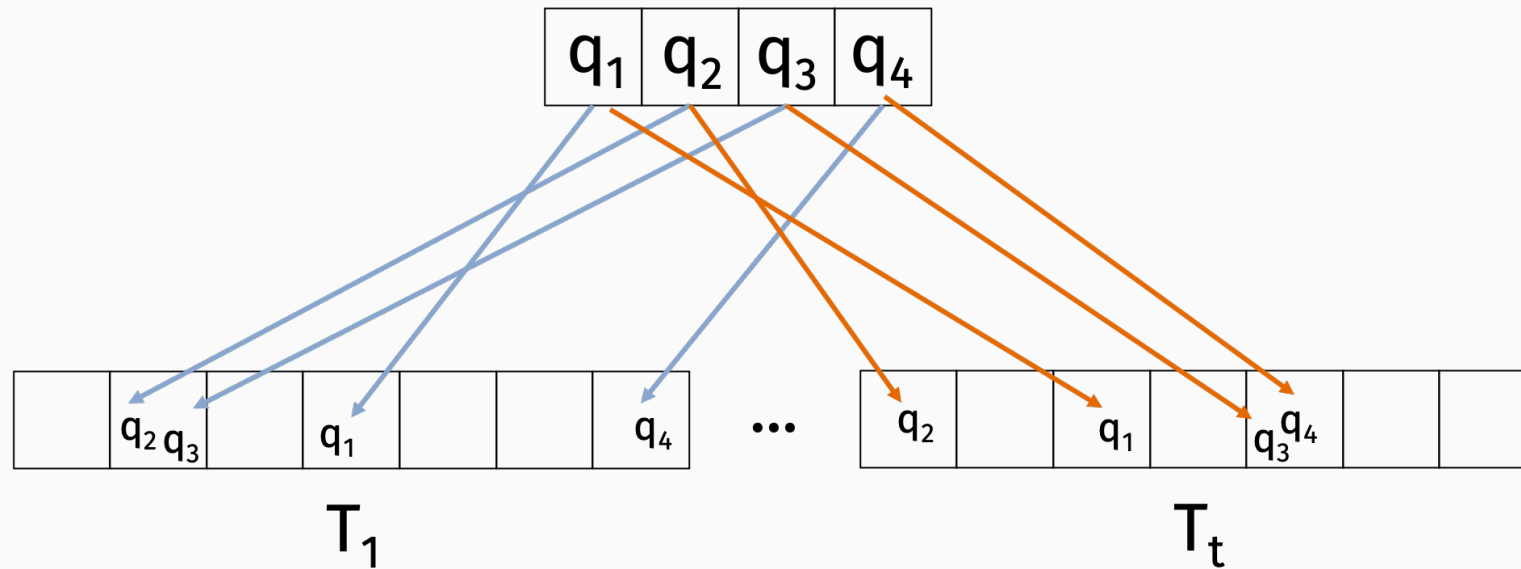
Reducing False Negative Rate



Pre-processing:

- Select t independent LSH's $h_1, \dots, h_t : \{0, 1\}^d \rightarrow 1, \dots, m$.

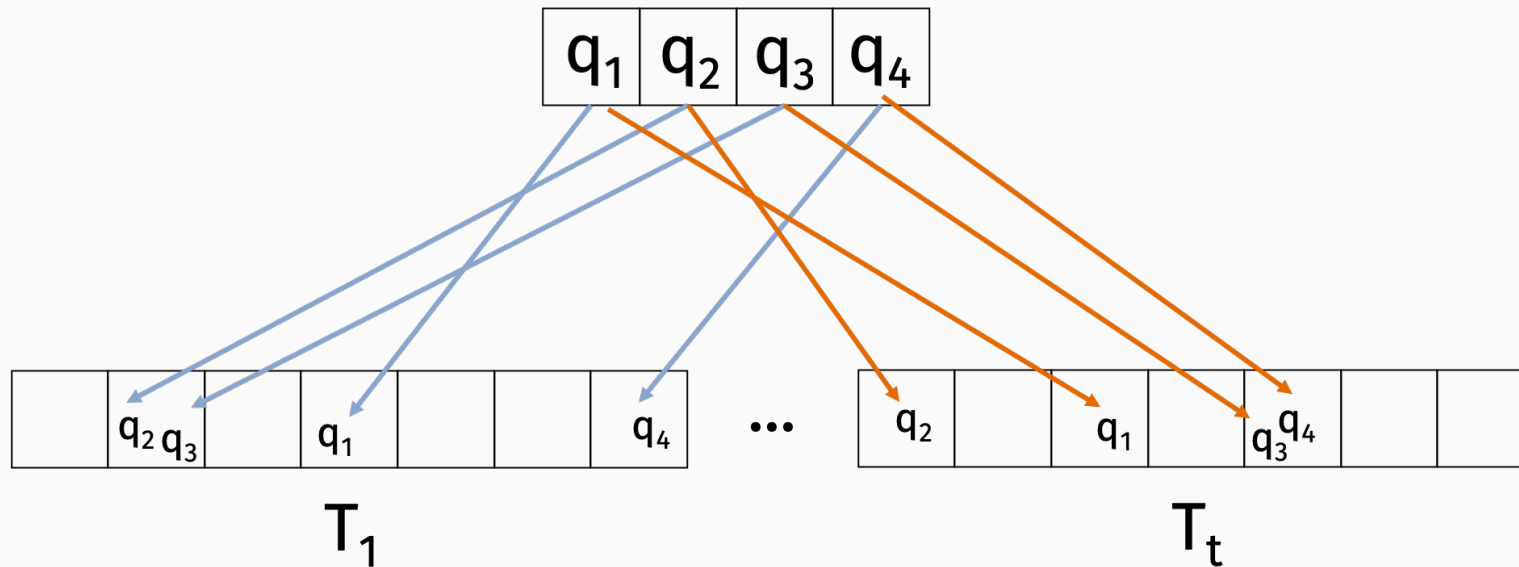
Reducing False Negative Rate



Pre-processing:

- Select t independent LSH's $h_1, \dots, h_t : \{0, 1\}^d \rightarrow 1, \dots, m$.
- Create tables $T_1, \dots, T_t$, each with m slots.

Reducing False Negative Rate



Pre-processing:

- Select t independent LSH's $h_1, \dots, h_t : \{0, 1\}^d \rightarrow 1, \dots, m$.
- Create tables $T_1, \dots, T_t$, each with m slots.
- For $i = 1, \dots, n$, $j = 1, \dots, t$,
 - Insert q_i into $T_j(h_j(q_i))$.

Reducing False Negative Rate

Query: Want to find near neighbors of input $\mathbf{y} \in \{0, 1\}^d$.

- Compute $h_1(\mathbf{y}), \dots, h_t(\mathbf{y})$.
- Linear scan through all vectors in $T_1(h_1(\mathbf{y})) \cup T_2(h_2(\mathbf{y})) \cup \dots, T_t(h_t(\mathbf{y}))$.

Reducing False Negative Rate

Query: Want to find near neighbors of input $\mathbf{y} \in \{0, 1\}^d$.

- Compute $h_1(\mathbf{y}), \dots, h_t(\mathbf{y})$.
- Linear scan through all vectors in $T_1(h_1(\mathbf{y})) \cup T_2(h_2(\mathbf{y})) \cup \dots, T_t(h_t(\mathbf{y}))$.

Suppose the nearest database point $\mathbf{q}$ has $J(\mathbf{y}, \mathbf{q}) = .4$.

What's the probability we find $\mathbf{q}$?

Reducing False Negative Rate

Query: Want to find near neighbors of input $\mathbf{y} \in \{0, 1\}^d$.

- Compute $h_1(\mathbf{y}), \dots, h_t(\mathbf{y})$.
- Linear scan through all vectors in $T_1(h_1(\mathbf{y})) \cup T_2(h_2(\mathbf{y})) \cup \dots, T_t(h_t(\mathbf{y}))$.

Suppose the nearest database point $\mathbf{q}$ has $J(\mathbf{y}, \mathbf{q}) = .4$.

What's the probability we find $\mathbf{q}$?

$$\Pr[\text{We find } \mathbf{q}] = 1 - \Pr[\text{We do not find } \mathbf{q} \text{ in any table}]$$

Reducing False Negative Rate

Query: Want to find near neighbors of input $\mathbf{y} \in \{0, 1\}^d$.

- Compute $h_1(\mathbf{y}), \dots, h_t(\mathbf{y})$.
- Linear scan through all vectors in $T_1(h_1(\mathbf{y})) \cup T_2(h_2(\mathbf{y})) \cup \dots, T_t(h_t(\mathbf{y}))$.

Suppose the nearest database point $\mathbf{q}$ has $J(\mathbf{y}, \mathbf{q}) = .4$.

What's the probability we find $\mathbf{q}$?

$$\begin{aligned}\Pr[\text{We find } \mathbf{q}] &= 1 - \Pr[\text{We do not find } \mathbf{q} \text{ in any table}] \\ &= 1 - (1 - 0.4)^t = 1 - 0.6^t\end{aligned}$$

Reducing False Negative Rate

Query: Want to find near neighbors of input $\mathbf{y} \in \{0, 1\}^d$.

- Compute $h_1(\mathbf{y}), \dots, h_t(\mathbf{y})$.
- Linear scan through all vectors in $T_1(h_1(\mathbf{y})) \cup T_2(h_2(\mathbf{y})) \cup \dots, T_t(h_t(\mathbf{y}))$.

Suppose the nearest database point $\mathbf{q}$ has $J(\mathbf{y}, \mathbf{q}) = .4$.

What's the probability we find $\mathbf{q}$?

$$\begin{aligned}\Pr[\text{We find } \mathbf{q}] &= 1 - \Pr[\text{We do not find } \mathbf{q} \text{ in any table}] \\ &= 1 - (1 - 0.4)^t = 1 - 0.6^t \\ &\approx 99\% \text{ for } t = 10\end{aligned}$$

What Happens to False Positives?

Now suppose there is some other database point $\mathbf{z}$ with $J(\mathbf{y}, \mathbf{z}) = .2$.

What Happens to False Positives?

Now suppose there is some other database point $\mathbf{z}$ with $J(\mathbf{y}, \mathbf{z}) = .2$.

What is the probability we will need to compute $J(\mathbf{z}, \mathbf{y})$ in our hashing scheme with one table?

What Happens to False Positives?

Now suppose there is some other database point $\mathbf{z}$ with $J(\mathbf{y}, \mathbf{z}) = .2$.

What is the probability we will need to compute $J(\mathbf{z}, \mathbf{y})$ in our hashing scheme with one table?

$$\Pr[\text{We find } \mathbf{z}] = 1 - (1 - 0.2)^t = 1 - 0.8^t$$

What Happens to False Positives?

Now suppose there is some other database point $\mathbf{z}$ with $J(\mathbf{y}, \mathbf{z}) = .2$.

What is the probability we will need to compute $J(\mathbf{z}, \mathbf{y})$ in our hashing scheme with one table?

$$\begin{aligned}\Pr[\text{We find } \mathbf{z}] &= 1 - (1 - 0.2)^t = 1 - 0.8^t \\ &\approx 89\% \text{ for } t = 10\end{aligned}$$

What Happens to False Positives?

Now suppose there is some other database point $\mathbf{z}$ with $J(\mathbf{y}, \mathbf{z}) = .2$.

What is the probability we will need to compute $J(\mathbf{z}, \mathbf{y})$ in our hashing scheme with one table?

$$\begin{aligned}\Pr[\text{We find } \mathbf{z}] &= 1 - (1 - 0.2)^t = 1 - 0.8^t \\ &\approx 89\% \text{ for } t = 10\end{aligned}$$

This is really bad! All the other points could have similarity .2

We have to search over $\Omega(n)$ points anyway.

Reducing False Positives

Secret Sauce: Change our locality sensitive hash function.

Reducing False Positives

Secret Sauce: Change our locality sensitive hash function.

Tunable LSH for Jaccard similarity:

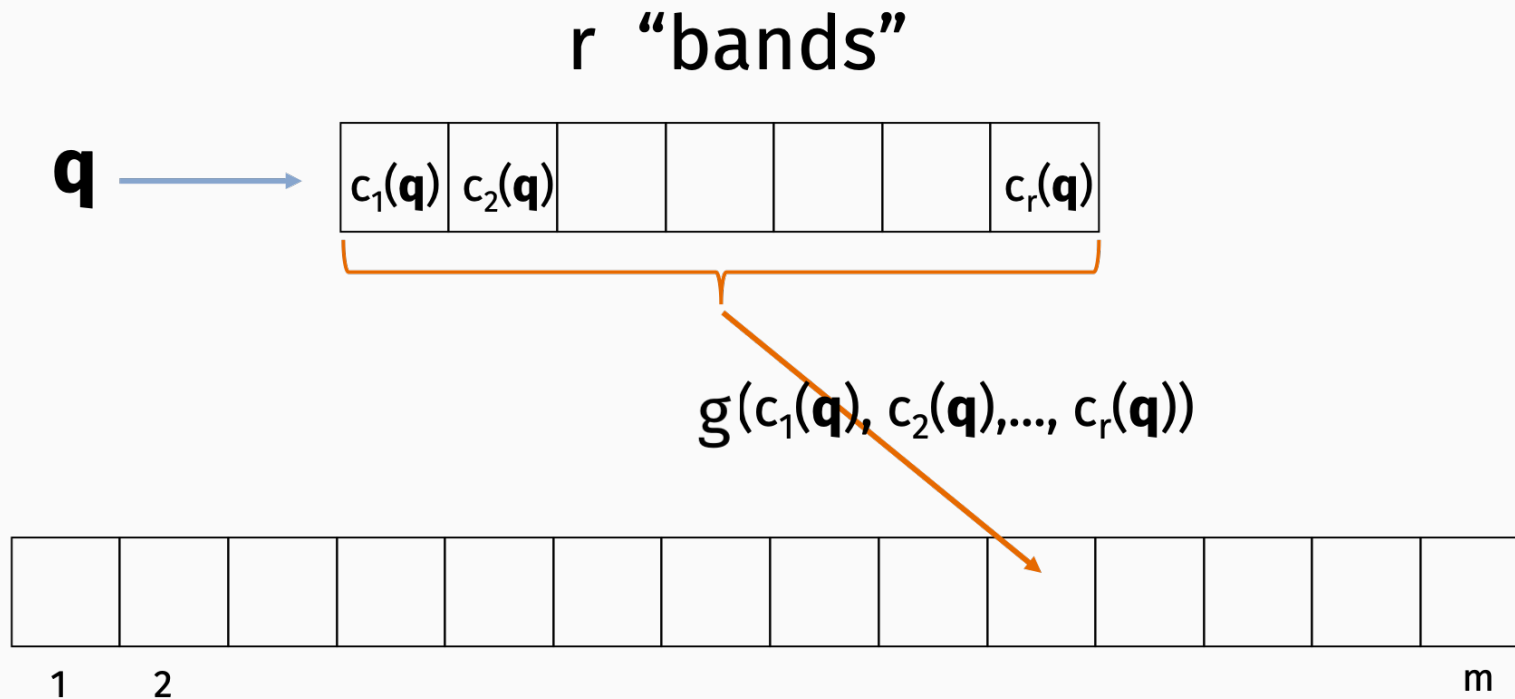
- Choose parameter $r \in \mathbb{Z}^+$.
- Let $c_1, \dots, c_r : \{0, 1\}^d \rightarrow [0, 1]$ be independent random MinHash's.
- Let $g : [0, 1]^r \rightarrow \{1, \dots, m\}$ be a uniform random hash function.
- Let $h(\mathbf{x}) = g(c_1(\mathbf{x}), \dots, c_r(\mathbf{x}))$.

Reducing False Positives

Secret Sauce: Change our locality sensitive hash function.

Tunable LSH for Jaccard similarity:

- Choose parameter $r \in \mathbb{Z}^+$.
- Let $c_1, \dots, c_r : \{0, 1\}^d \rightarrow [0, 1]$ be independent random MinHash's.
- Let $g : [0, 1]^r \rightarrow \{1, \dots, m\}$ be a uniform random hash function.
- Let $h(\mathbf{x}) = g(c_1(\mathbf{x}), \dots, c_r(\mathbf{x}))$.



Reducing False Positives

Tunable LSH for Jaccard similarity:

- Choose parameter $r \in \mathbb{Z}^+$. (number of bands)
- Let $c_1, \dots, c_r : \{0, 1\}^d \rightarrow [0, 1]$ be random MinHash.
- Let $g : [0, 1]^r \rightarrow \{1, \dots, m\}$ be a uniform random hash function.
- Let $h(\mathbf{x}) = g(c_1(\mathbf{x}), \dots, c_r(\mathbf{x}))$.

If $J(\mathbf{q}, \mathbf{y}) = v$, then $\Pr[h(\mathbf{q}) == h(\mathbf{y})] = ??$

Reducing False Positives

Tunable LSH for Jaccard similarity:

- Choose parameter $r \in \mathbb{Z}^+$. (number of bands)
- Let $c_1, \dots, c_r : \{0, 1\}^d \rightarrow [0, 1]$ be random MinHash.
- Let $g : [0, 1]^r \rightarrow \{1, \dots, m\}$ be a uniform random hash function.
- Let $h(\mathbf{x}) = g(c_1(\mathbf{x}), \dots, c_r(\mathbf{x}))$.

If $J(\mathbf{q}, \mathbf{y}) = v$, then $\Pr[h(\mathbf{q}) == h(\mathbf{y})] = ??$

$$\Pr[h(\mathbf{q}) == h(\mathbf{y})] = \Pr[g(c_1(\mathbf{q}), \dots, c_r(\mathbf{q})) = g(c_1(\mathbf{y}), \dots, c_r(\mathbf{y}))] + \text{correction}$$

Reducing False Positives

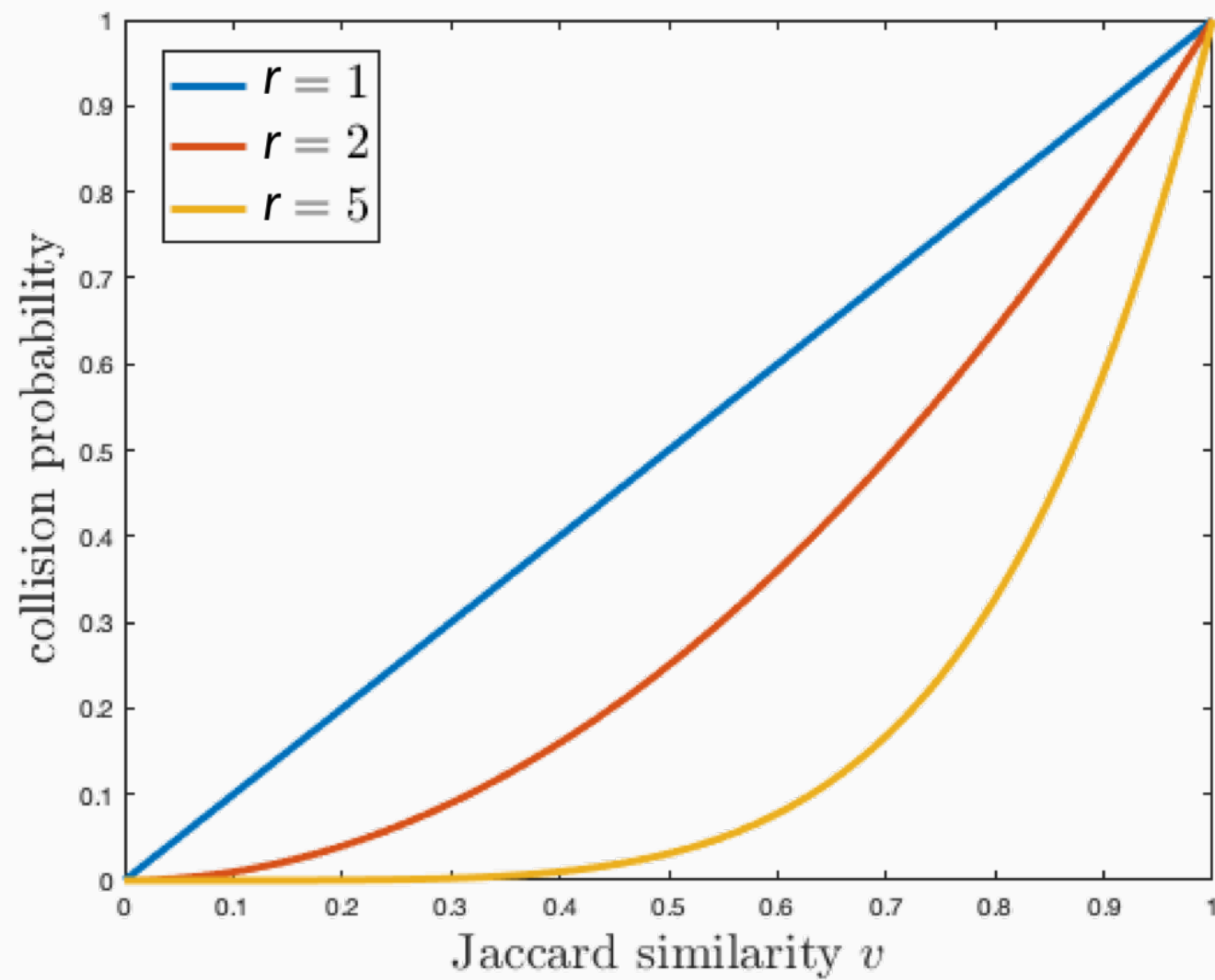
Tunable LSH for Jaccard similarity:

- Choose parameter $r \in \mathbb{Z}^+$. (number of bands)
- Let $c_1, \dots, c_r : \{0, 1\}^d \rightarrow [0, 1]$ be random MinHash.
- Let $g : [0, 1]^r \rightarrow \{1, \dots, m\}$ be a uniform random hash function.
- Let $h(\mathbf{x}) = g(c_1(\mathbf{x}), \dots, c_r(\mathbf{x}))$.

If $J(\mathbf{q}, \mathbf{y}) = v$, then $\Pr[h(\mathbf{q}) == h(\mathbf{y})] = ??$

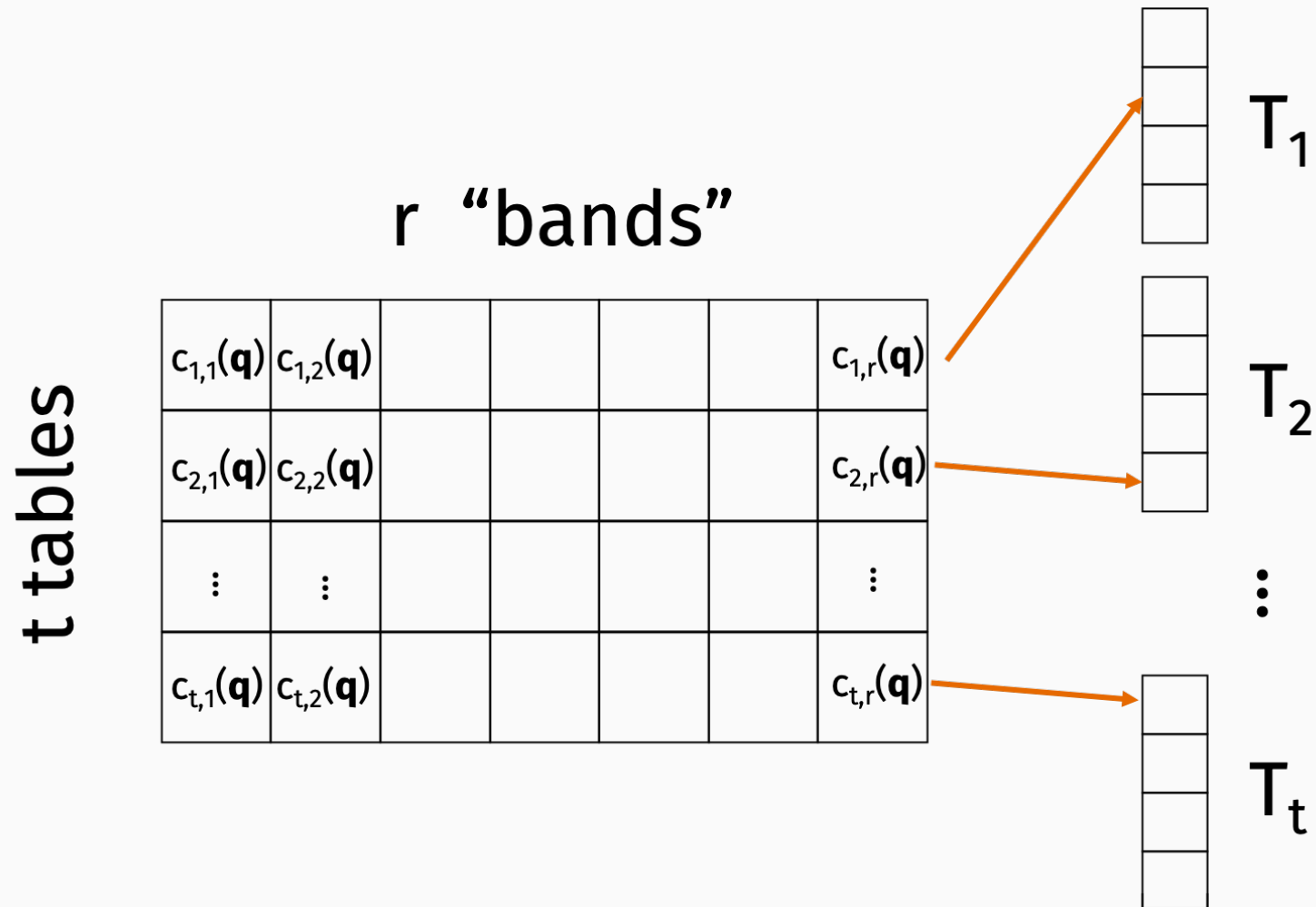
$$\begin{aligned}\Pr[h(\mathbf{q}) == h(\mathbf{y})] &= \Pr[g(c_1(\mathbf{q}), \dots, c_r(\mathbf{q})) = g(c_1(\mathbf{y}), \dots, c_r(\mathbf{y}))] + \text{correction} \\ &= v \cdot v \cdots v + \text{correction} = v^r + (1 - v^r) \frac{1}{m}\end{aligned}$$

Tunable LSH



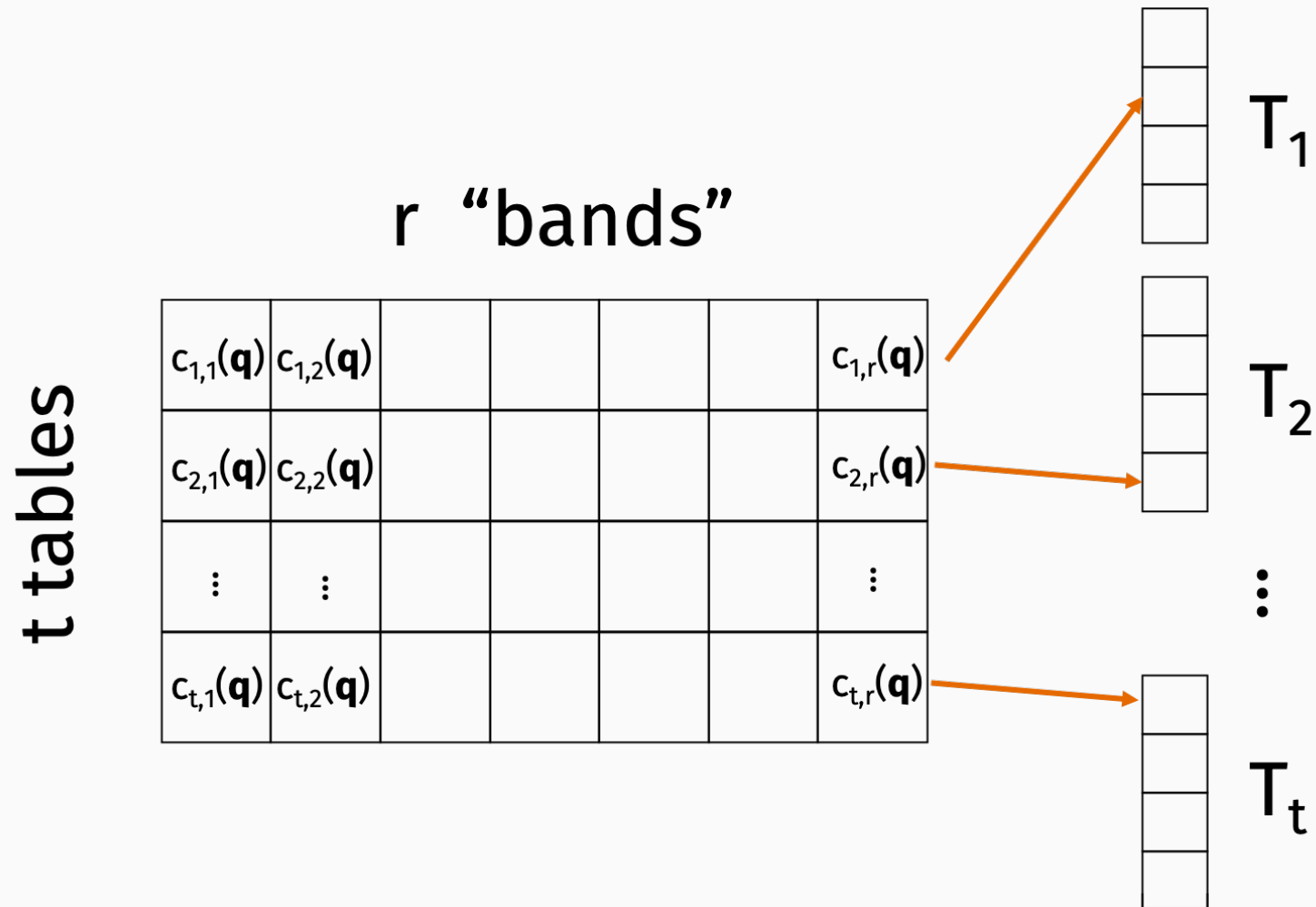
Tunable LSH

Full LSH scheme has two parameters to tune:



Tunable LSH

Full LSH scheme has two parameters to tune:



Still unclear why this is useful

Effect of **increasing number of tables** t on:

False Negatives

False Positives

Tunable LSH

Effect of **increasing number of tables** t on:

False Negatives

Decrease

False Positives

Increase

Effect of **increasing number of bands** r on:

False Negatives

False Positives

Tunable LSH

Effect of **increasing number of tables** t on:

False Negatives

Decrease

False Positives

Increase

Effect of **increasing number of bands** r on:

False Negatives

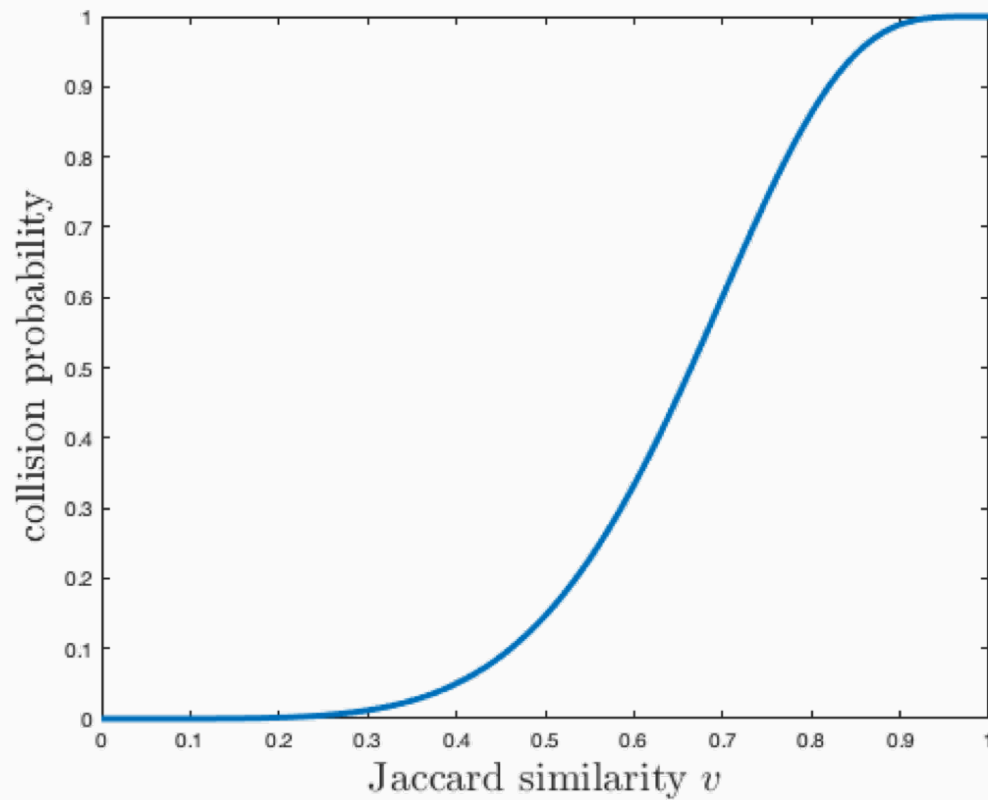
Increase

False Positives

Decrease

Probability we check $\mathbf{q}$ when querying $\mathbf{y}$ if $J(\mathbf{q}, \mathbf{y}) = v$:

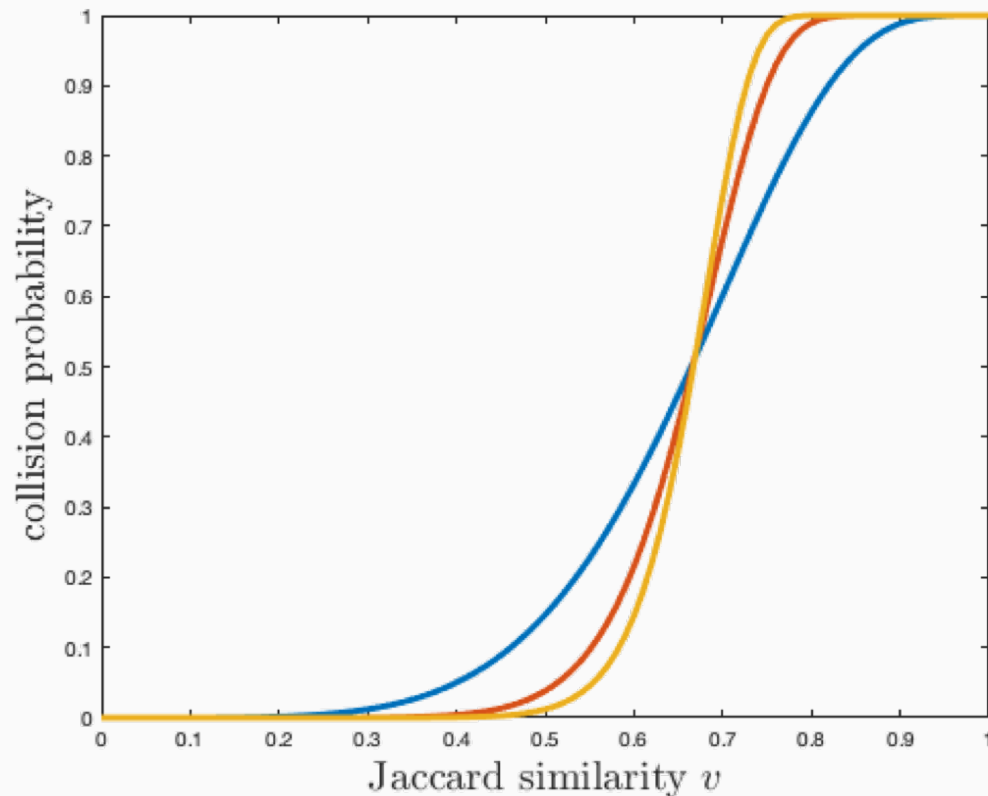
$$\text{Collision Probability} \approx 1 - (1 - v^r)^t$$



$$r = 5, t = 5$$

Probability we check $\mathbf{q}$ when querying $\mathbf{y}$ if $J(\mathbf{q}, \mathbf{y}) = v$:

$$1 - (1 - v^r)^t$$



Increasing both r and t gives a steeper curve, approaching a step function.

Better for search, but worse space complexity.

Use Case 1: Fixed threshold.

- Shazam wants to find match to audio clip y in a database of 10 million clips.

Use Case 1: Fixed threshold.

- Shazam wants to find match to audio clip $\mathbf{y}$ in a database of 10 million clips.
- There are 10 *true matches* with $J(\mathbf{y}, \mathbf{q}) > .9$.

Use Case 1: Fixed threshold.

- Shazam wants to find match to audio clip $\mathbf{y}$ in a database of 10 million clips.
- There are 10 *true matches* with $J(\mathbf{y}, \mathbf{q}) > .9$.
- There are 10,000 *near matches* with $J(\mathbf{y}, \mathbf{q}) \in [.7, .9]$.

Use Case 1: Fixed threshold.

- Shazam wants to find match to audio clip $\mathbf{y}$ in a database of 10 million clips.
- There are 10 *true matches* with $J(\mathbf{y}, \mathbf{q}) > .9$.
- There are 10,000 *near matches* with $J(\mathbf{y}, \mathbf{q}) \in [.7, .9]$.
- All other items have $J(\mathbf{y}, \mathbf{q}) < .7$.

With $r = 25$ and $t = 40$,

- Hit probability for $J(\mathbf{y}, \mathbf{q}) > .9$ is $\gtrsim 1 - (1 - .9^{25})^{40} = .95$

Use Case 1: Fixed threshold.

- Shazam wants to find match to audio clip $\mathbf{y}$ in a database of 10 million clips.
- There are 10 *true matches* with $J(\mathbf{y}, \mathbf{q}) > .9$.
- There are 10,000 *near matches* with $J(\mathbf{y}, \mathbf{q}) \in [.7, .9]$.
- All other items have $J(\mathbf{y}, \mathbf{q}) < .7$.

With $r = 25$ and $t = 40$,

- Hit probability for $J(\mathbf{y}, \mathbf{q}) > .9$ is $\gtrsim 1 - (1 - .9^{25})^{40} = .95$
- Hit probability for $J(\mathbf{y}, \mathbf{q}) \in [.7, .9]$ is $\lesssim 1 - (1 - .9^{25})^{40} = .95$

Use Case 1: Fixed threshold.

- Shazam wants to find match to audio clip $\mathbf{y}$ in a database of 10 million clips.
- There are 10 *true matches* with $J(\mathbf{y}, \mathbf{q}) > .9$.
- There are 10,000 *near matches* with $J(\mathbf{y}, \mathbf{q}) \in [.7, .9]$.
- All other items have $J(\mathbf{y}, \mathbf{q}) < .7$.

With $r = 25$ and $t = 40$,

- Hit probability for $J(\mathbf{y}, \mathbf{q}) > .9$ is $\gtrsim 1 - (1 - .9^{25})^{40} = .95$
- Hit probability for $J(\mathbf{y}, \mathbf{q}) \in [.7, .9]$ is $\lesssim 1 - (1 - .9^{25})^{40} = .95$
- Hit probability for $J(\mathbf{y}, \mathbf{q}) < .7$ is $\lesssim 1 - (1 - .7^{25})^{40} = .005$

Upper bound on total number of items checked:

$$10 + .95 \cdot 10,000 + .005 \cdot 9,989,990 \approx 60,000 \ll 10,000,000.$$

Space complexity: 40 hash tables $\approx 40 \cdot O(n)$.

Directly trade space for fast search.

Possible to prove concrete worst-case results for distance functions that satisfy triangle inequality.

Theorem (Indyk, Motwani, 1998. Point Location in Ball)

Fix a distance R . If there exists some $\mathbf{q}$ with $\|\mathbf{q} - \mathbf{y}\|_0 \leq R$, return a vector $\tilde{\mathbf{q}}$ with $\|\tilde{\mathbf{q}} - \mathbf{y}\|_0 \leq C \cdot R$ in:

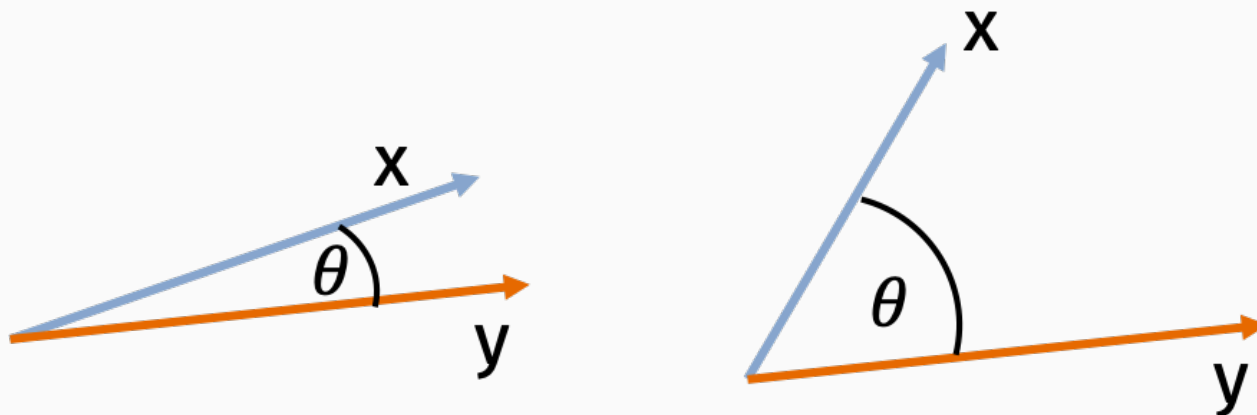
- Time: $O(n^{1/C})$.
- Space: $O(n^{1+1/C} + nd)$.

$\|\mathbf{q} - \mathbf{y}\|_0$ = “hamming distance” = number of elements that differ between $\mathbf{q}$ and $\mathbf{y}$.

If there is no point at distance R , algorithm does not need to return anything.

Good locality sensitive hash functions exist for other similarity measures.

Cosine similarity $\cos(\theta(\mathbf{x}, \mathbf{y})) = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2}$:



$$-1 \leq \cos(\theta(\mathbf{x}, \mathbf{y})) \leq 1.$$

Break

Milgram's Small-World Experiment

The Small World Problem

Stanley Milgram (1967)



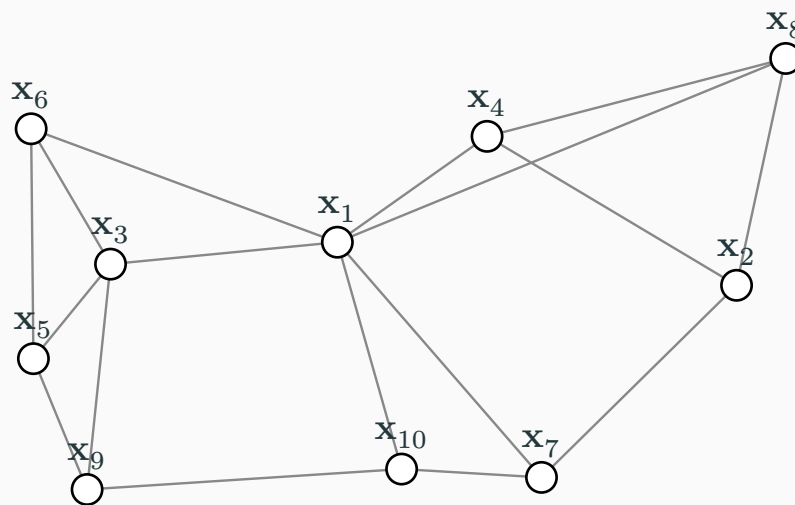
Associated with the phrase “six-degrees of separation”

Graph-Based Vector Search

New kid on the block: **Graph-based Vector Search**

Key Idea

Pre-process the dataset into a graph. Then, run greedy search on the graph.

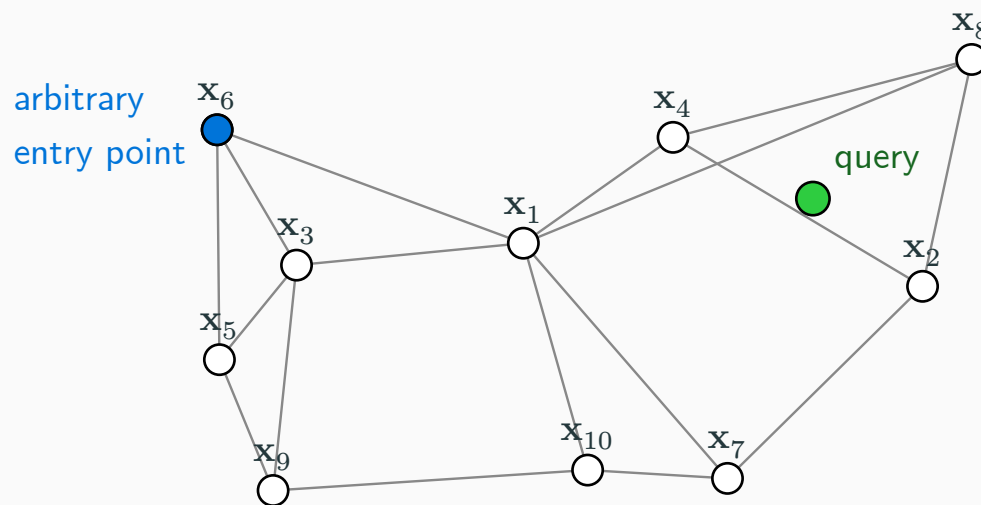


Graph-Based Vector Search

New kid on the block: **Graph-based Vector Search**

Key Idea

Pre-process the dataset into a graph. Then, run greedy search on the graph.

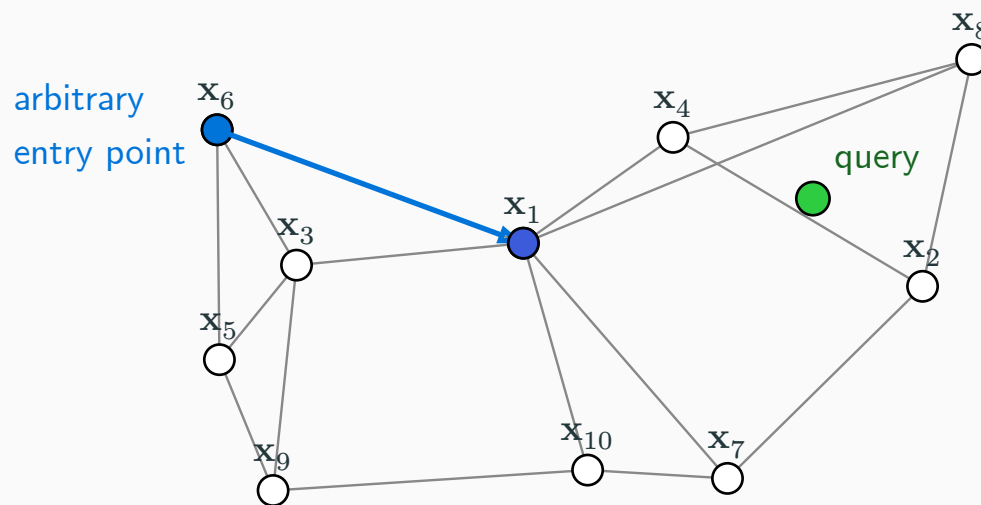


Graph-Based Vector Search

New kid on the block: **Graph-based Vector Search**

Key Idea

Pre-process the dataset into a graph. Then, run greedy search on the graph.

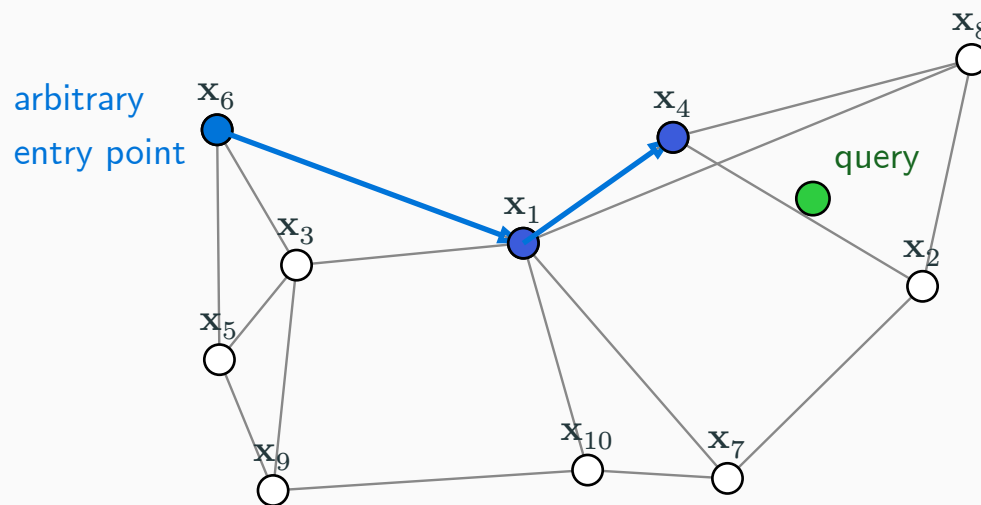


Graph-Based Vector Search

New kid on the block: **Graph-based Vector Search**

Key Idea

Pre-process the dataset into a graph. Then, run greedy search on the graph.

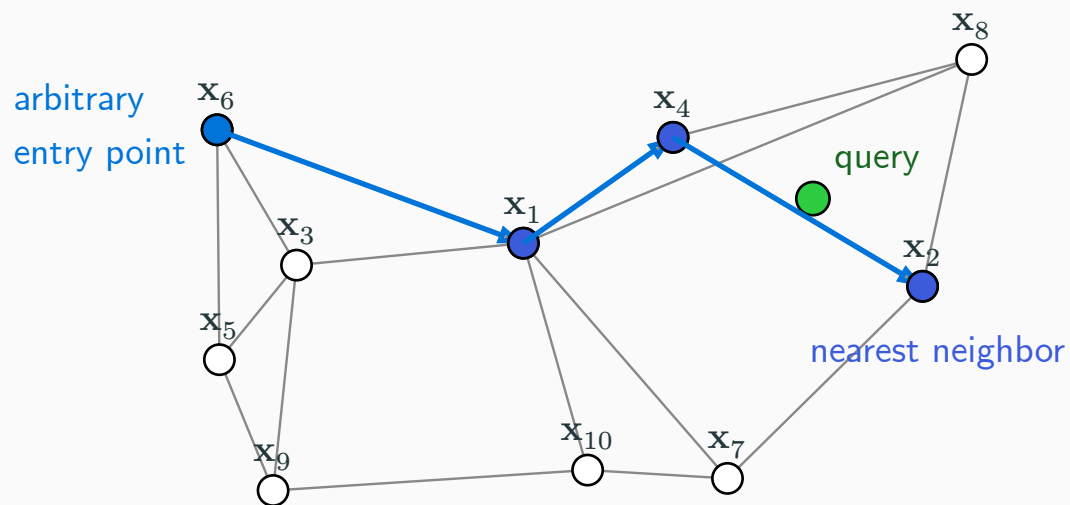


Graph-Based Vector Search

New kid on the block: **Graph-based Vector Search**

Key Idea

Pre-process the dataset into a graph. Then, run greedy search on the graph.

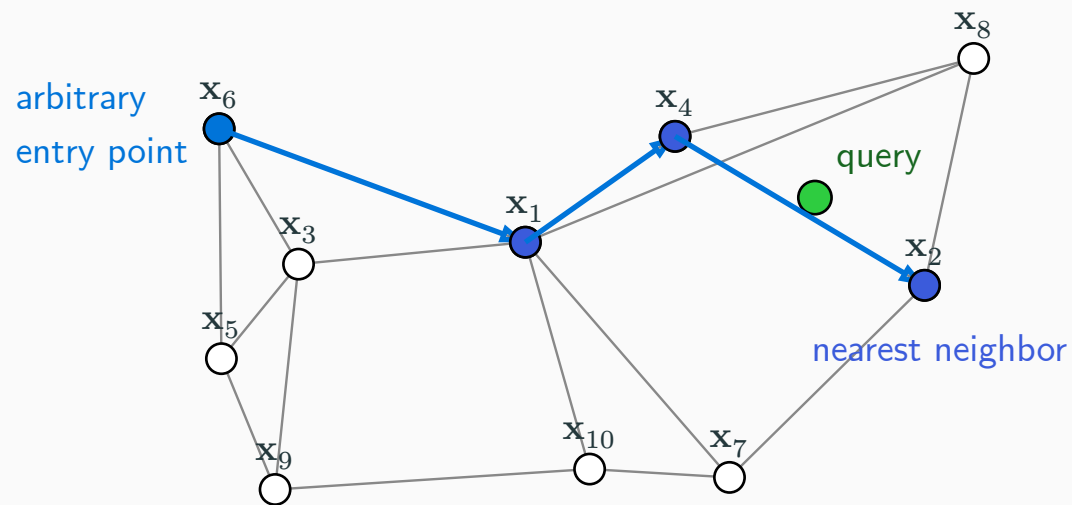


Graph-Based Vector Search

New kid on the block: **Graph-based Vector Search**

Key Idea

Pre-process the dataset into a graph. Then, run greedy search on the graph.



E.g. Microsoft DiskANN [Subramanya et al., 2019], NSG [Fu et al., 2019], and HNSW [Malkov, Yashunin, 2020].

Question:

What makes a graph good for vector search?

Question:

What makes a graph good for vector search?

Graph Navigability has been identified as a desirable property.

Informally: Every node in the graph has an out-edge that gets closer to any other node.

Informally: Every node in the graph has an out-edge that gets closer to any other node.

Definition (Navigable Graph)

A graph $G = (P, E)$ is navigable if, for all pairs $\mathbf{x}_i, \mathbf{x}_j \in P$, there is a directed edge $(\mathbf{x}_i, \mathbf{u}) \in E$ such that

$$\text{dist}(\mathbf{u}, \mathbf{x}_j) < \text{dist}(\mathbf{x}_i, \mathbf{x}_j).$$

Informally: Every node in the graph has an out-edge that gets closer to any other node.

Definition (Navigable Graph)

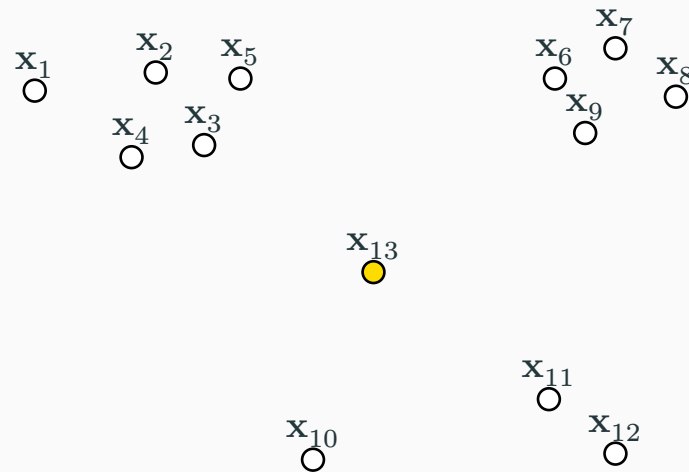
A graph $G = (P, E)$ is navigable if, for all pairs $\mathbf{x}_i, \mathbf{x}_j \in P$, there is a directed edge $(\mathbf{x}_i, \mathbf{u}) \in E$ such that

$$\text{dist}(\mathbf{u}, \mathbf{x}_j) < \text{dist}(\mathbf{x}_i, \mathbf{x}_j).$$

Consequence: Greedy search on navigable graphs always succeeds on *in-distribution* queries, i.e., if we query $\mathbf{x}_j \in P$, it exactly returns $\mathbf{x}_j$ no matter where the search begins.

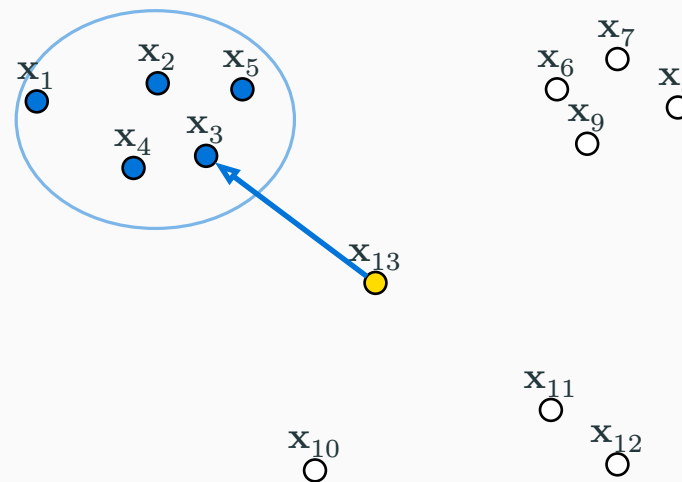
Visualizing Navigability

Consider the following pointset and focus on x_{13} 's out-neighborhood.



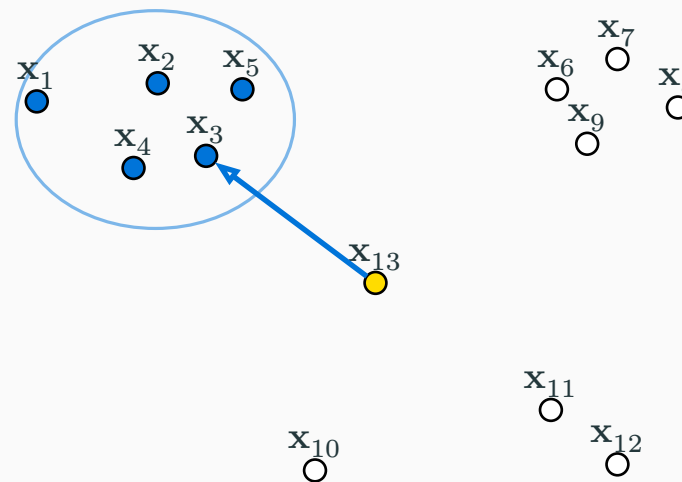
Visualizing Navigability

The edge (x_{13}, x_3) allows greedy navigation to points x_1, x_2, x_3, x_4, x_5 .



Visualizing Navigability

The edge (x_{13}, x_3) allows greedy navigation to points x_1, x_2, x_3, x_4, x_5 .

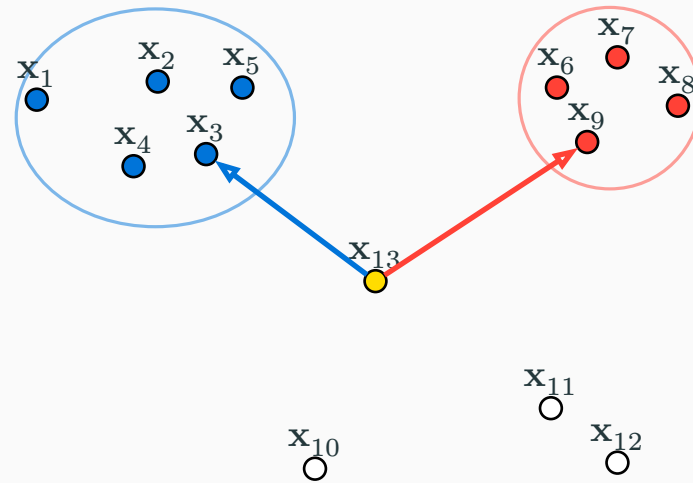


Rest of the lecture: An edge **covers** all the points it gets closer to.

E.g. (x_{13}, x_3) covers x_1, x_2, x_3, x_4, x_5

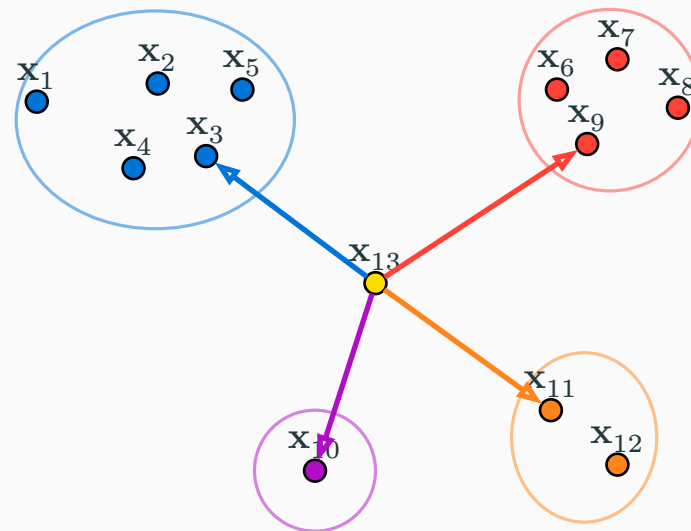
Visualizing Navigability

The edge (x_{13}, x_9) covers points x_6, x_7, x_8, x_9 .



Visualizing Navigability

Adding edges (x_{13}, x_{10}) and (x_{13}, x_{11}) covers the remaining points.



Sparse Navigable Graphs

What are the search costs associated with graph-based methods?

- Space:
- Time to explore a single neighborhood:

Sparse Navigable Graphs

What are the search costs associated with graph-based methods?

- Space:
- Time to explore a single neighborhood:

Theorem (Diwan, Gou, Musco, Musco, Suel, 2024)

For any points $\mathbf{x}_1, \dots, \mathbf{x}_n$ and any distance function, we can efficiently construct a navigable graph with average out-degree at most $O(\sqrt{n \ln n})$.

Moreover, greedy search always succeeds in at most 2 steps.

Compare to the trivial bound: the complete graph has out-degree $n - 1$.

Navigable Graph Construction

Imagine a **distance-based permutation matrix**, where the i^{th} row lists all points sorted in order of their distance to point i .

Node 1	1	10	2	3	5	9	6	8	4	7
--------	---	----	---	---	---	---	---	---	---	---

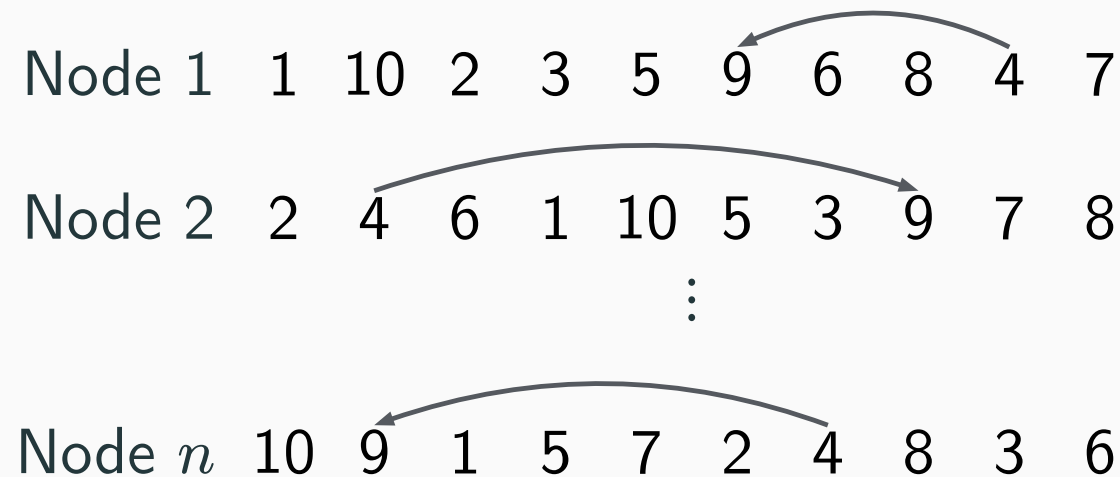
Node 2	2	4	6	1	10	5	3	9	7	8
--------	---	---	---	---	----	---	---	---	---	---

⋮

Node n	10	9	1	5	7	2	4	8	3	6
----------	----	---	---	---	---	---	---	---	---	---

Navigable Graph Construction

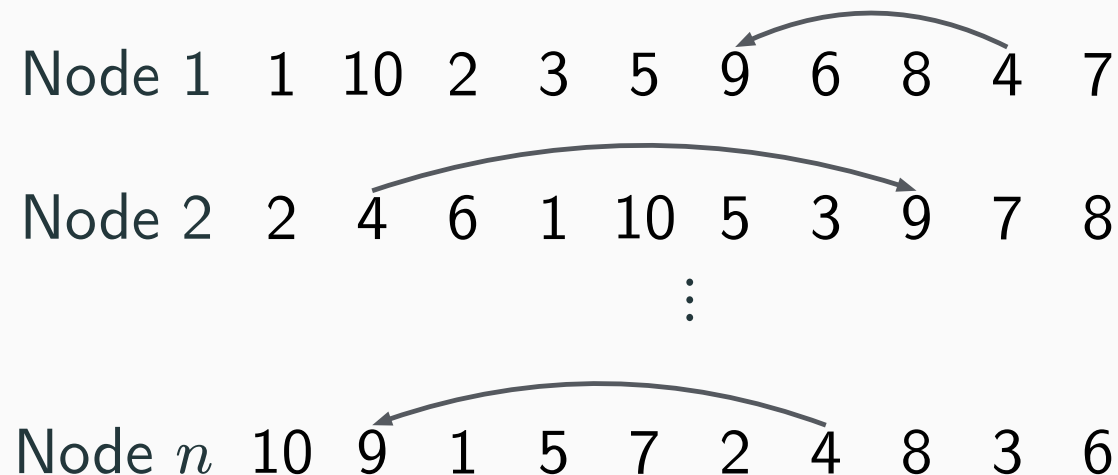
Imagine a **distance-based permutation matrix**, where the i^{th} row lists all points sorted in order of their distance to point i .



For every edge in G , add that edge to every row of the matrix.

Navigable Graph Construction

Imagine a **distance-based permutation matrix**, where the i^{th} row lists all points sorted in order of their distance to point i .



For every edge in G , add that edge to every row of the matrix.

Navigability requirement: Need at least one “left pointing” edge for every node in every list.

Navigable Graph Construction

Construction: Choose $m < n$.

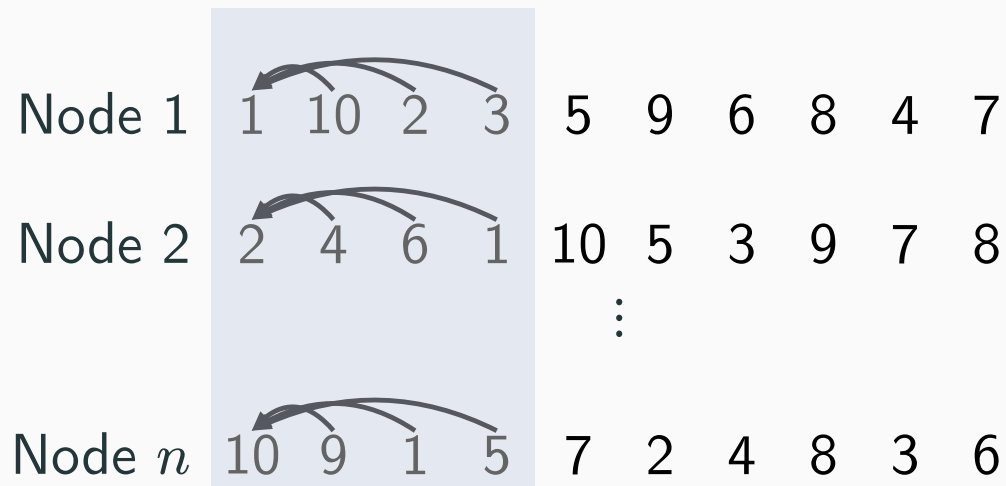
1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O\left(\frac{n}{m} \log n\right)$ uniformly random out-edges to every node.

	m									
Node 1	1	10	2	3	5	9	6	8	4	7
Node 2	2	4	6	1	10	5	3	9	7	8
					$\vdots$					
Node n	10	9	1	5	7	2	4	8	3	6

Navigable Graph Construction

Construction: Choose $m < n$.

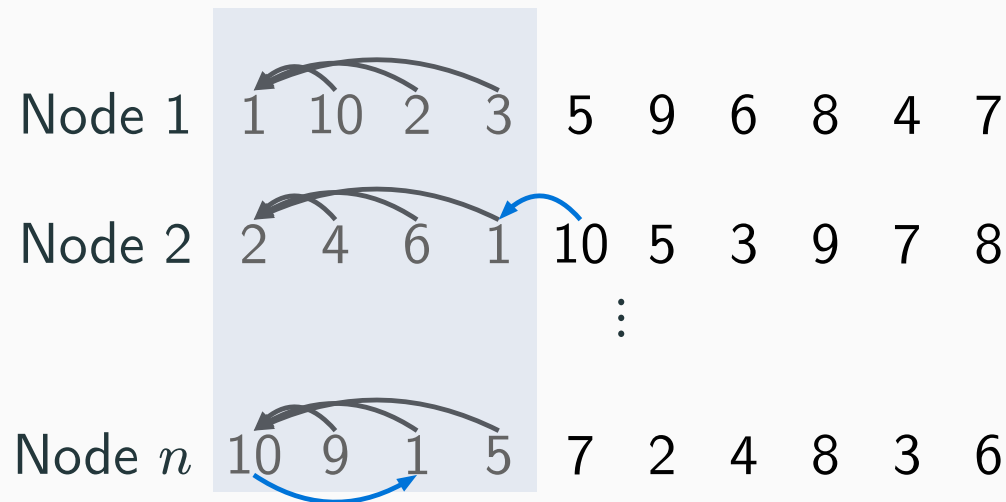
1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



Navigable Graph Construction

Construction: Choose $m < n$.

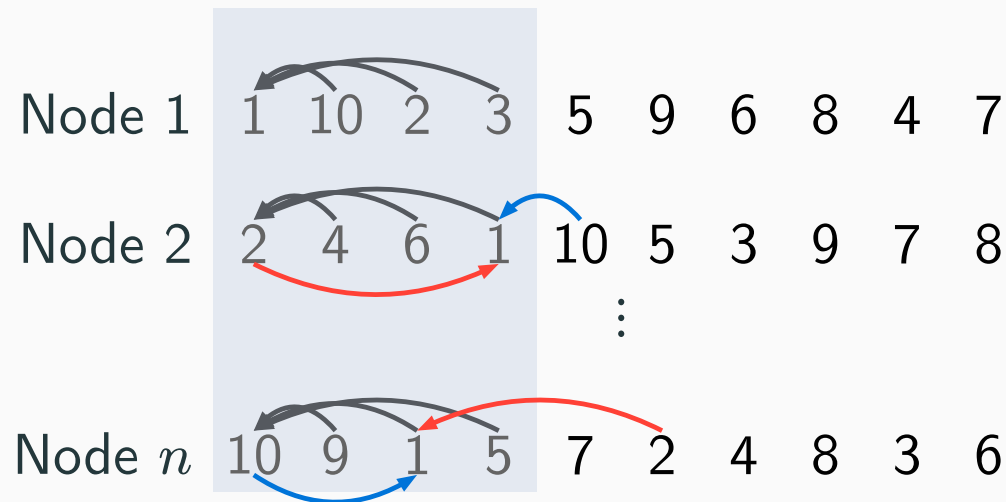
1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



Navigable Graph Construction

Construction: Choose $m < n$.

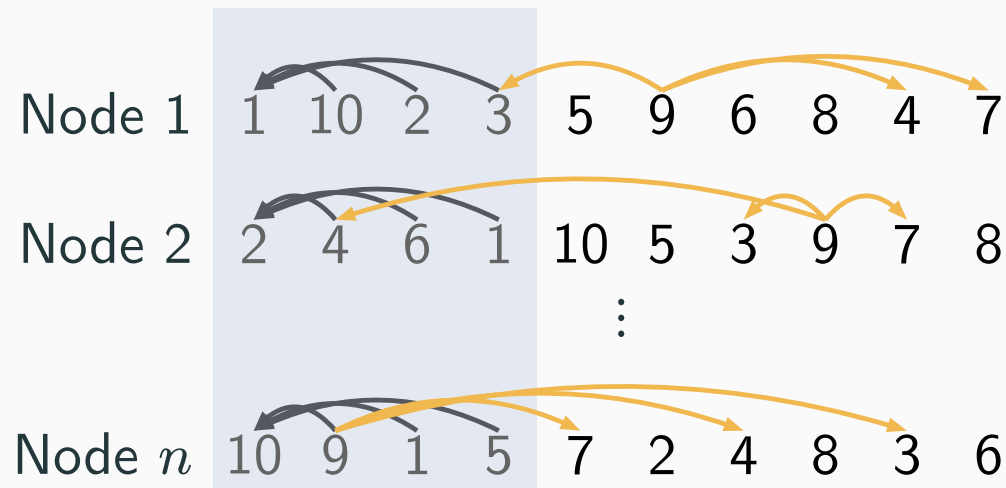
1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



Navigable Graph Construction

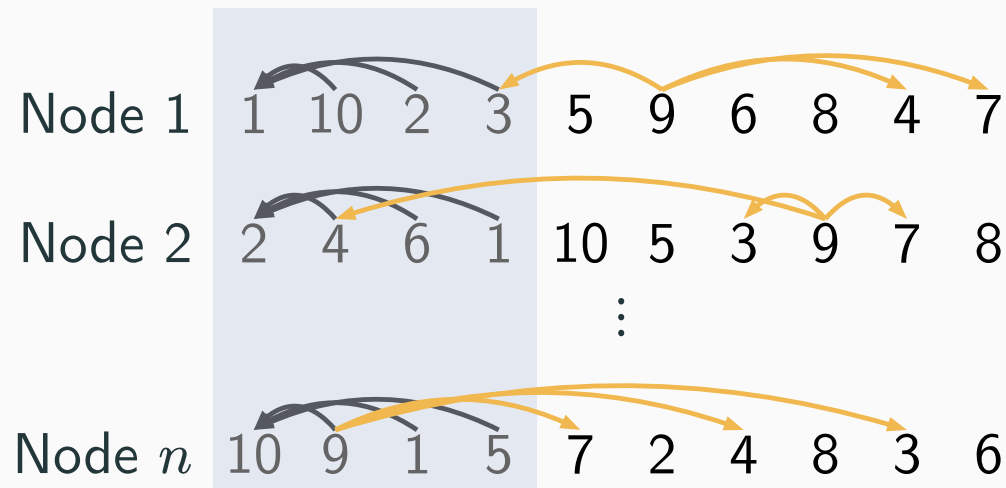
Construction: Choose $m < n$.

1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



Left Region

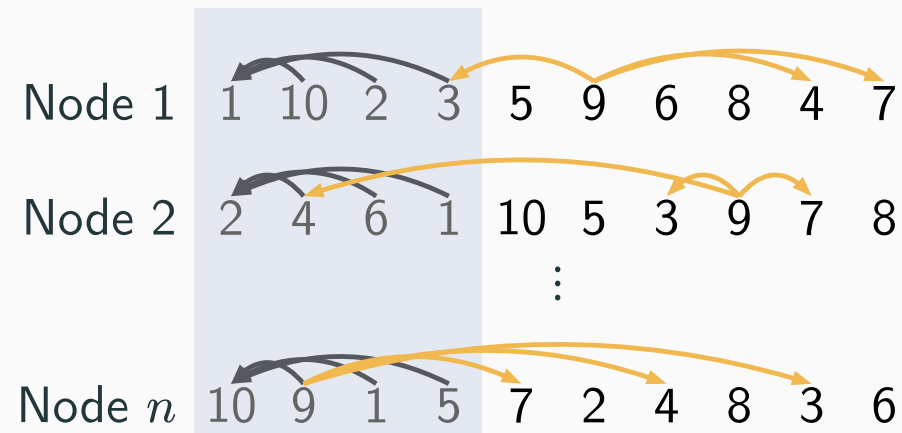
1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



Analysis: All constraints in the left shaded region are trivially satisfied.

Right Region

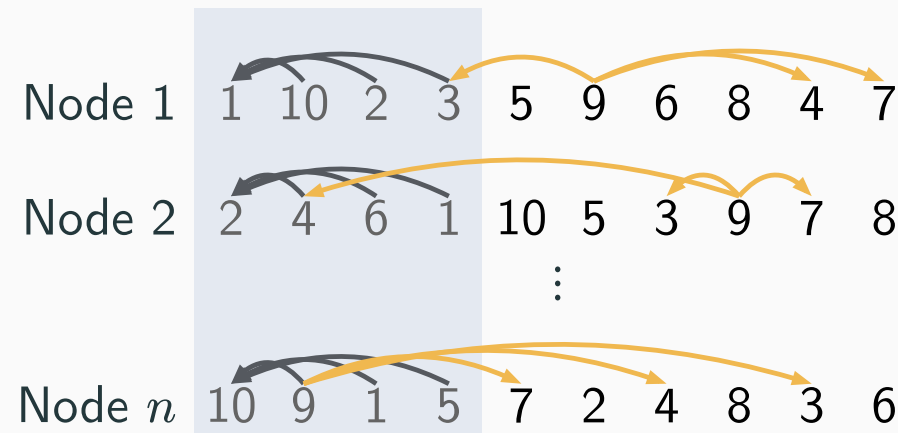
1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



Analysis: Consider any node outside the left region. The probability a random edge points left is at least $\frac{m}{n}$.

Right Region

1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O(\frac{n}{m} \log n)$ uniformly random out-edges to every node.



Analysis: Consider any node outside the left region. The probability a random edge points left is at least $\frac{m}{n}$.

So the probability that *none* point left from a given i is:

$$\leq \left(1 - \frac{m}{n}\right)^{O(\frac{n}{m} \log n)} \leq \frac{1}{n^c}.$$

Average Degree Bound

Construction: Choose $m < n$.

1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O\left(\frac{n}{m} \log n\right)$ uniformly random out-edges to every node.

Total number of edges: $nm + n \cdot O\left(\frac{n}{m} \log n\right)$.

Minimize by setting $m = O(\sqrt{n \log n})$.

Average out-degree: $O(\sqrt{n \log n})$.

Average Degree Bound

Construction: Choose $m < n$.

1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O\left(\frac{n}{m} \log n\right)$ uniformly random out-edges to every node.

Total number of edges: $nm + n \cdot O\left(\frac{n}{m} \log n\right)$.

Minimize by setting $m = O(\sqrt{n \log n})$.

Average out-degree: $O(\sqrt{n \log n})$.

In fact, these results are optimal. Diwan, Gou, Musco, Musco, Suel, 2024 also give a $\Omega(\sqrt{n})$ average degree lower bound.

Average Degree Bound

Construction: Choose $m < n$.

1. Add an edge from j to i if j is one of i 's m closest neighbors.
2. Add $O\left(\frac{n}{m} \log n\right)$ uniformly random out-edges to every node.

Total number of edges: $nm + n \cdot O\left(\frac{n}{m} \log n\right)$.

Minimize by setting $m = O(\sqrt{n \log n})$.

Average out-degree: $O(\sqrt{n \log n})$.

In fact, these results are optimal. Diwan, Gou, Musco, Musco, Suel, 2024 also give a $\Omega(\sqrt{n})$ average degree lower bound.

But this would be less than ideal in practice... (why?)

Question

Are there relaxations of navigability that allow for **sparser graphs**?

Question

Are there relaxations of navigability that allow for **sparser graphs**?

Most practical graph-based methods build *close to navigable* graphs, usually in some heuristic sense.

Let's see if we can achieve this more formally.

Definition (γ -almost navigability)

A graph $G = (P, E)$ is γ -almost navigable if every point $\mathbf{p} \in P$ has an out-neighborhood that covers at least γ fraction of the dataset.

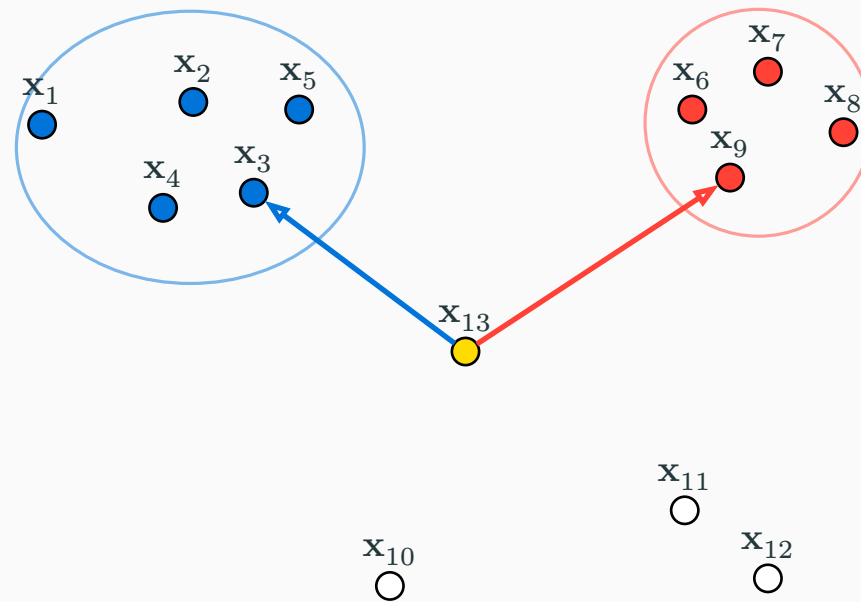
Definition (γ -almost navigability)

A graph $G = (P, E)$ is γ -almost navigable if every point $\mathbf{p} \in P$ has an out-neighborhood that covers at least γ fraction of the dataset.

Where true navigability requires that the out-neighborhood covers all $n - 1$ nodes, γ -almost navigability only requires that we cover $\geq \gamma \cdot (n - 1)$ nodes.

Almost Navigable Graphs: Example

For example, just selecting the edges (x_{13}, x_3) and (x_{13}, x_9) results in a $\frac{3}{4}$ -almost navigable out-neighborhood for x_{13} .



Selecting 2 edges covers *most* points.
Need 4 for full navigability.

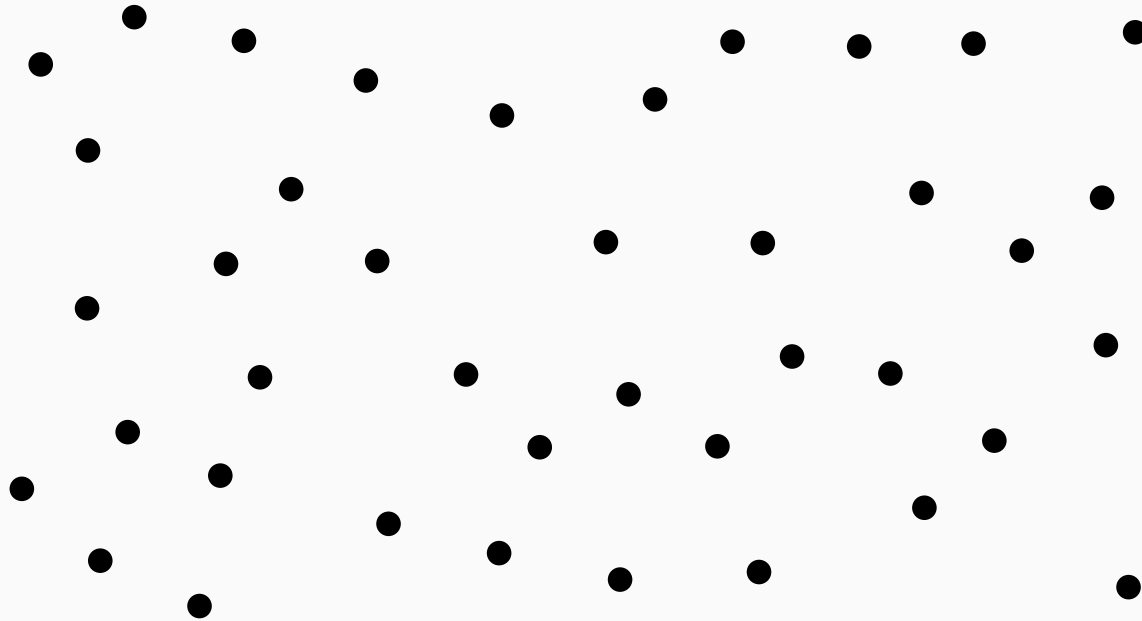
Theorem (Avi, Musco 2026)

Every point set P admits a γ -almost navigable graph with $\frac{4}{1-\gamma}$ average degree.

Notice that degree is independent of the number of points n and the vector dimension d !

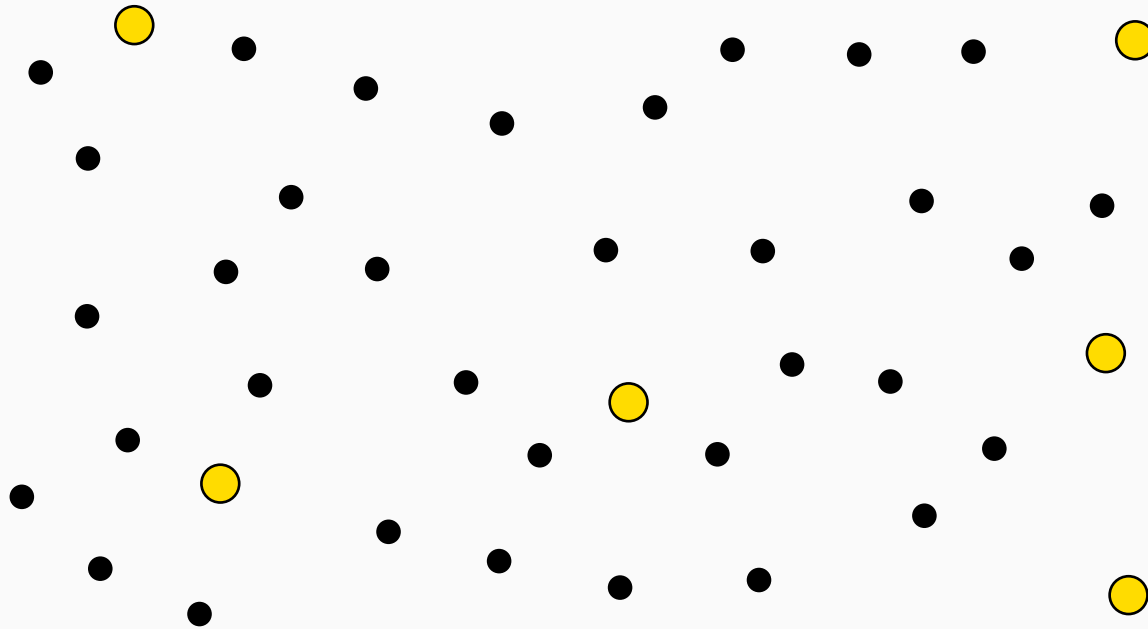
Main Idea: The Power of Cliques

Simple Idea: Choose a subset of points $S \subset P$ arbitrarily and add a clique over them.



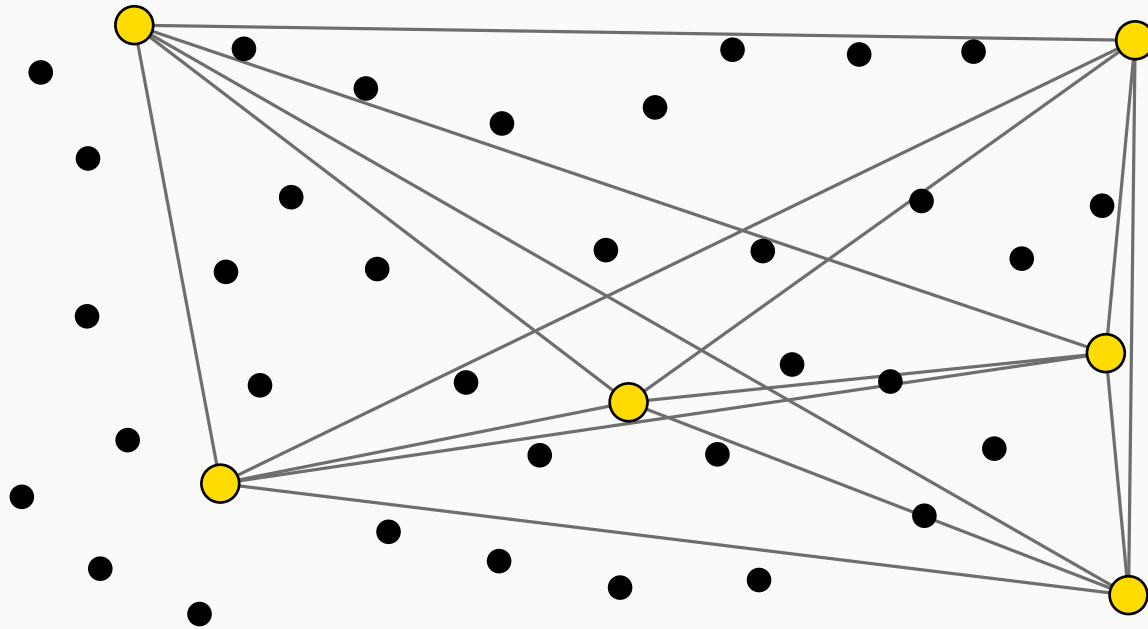
Main Idea: The Power of Cliques

Simple Idea: Choose a subset of points $S \subset P$ arbitrarily and add a clique over them.



Main Idea: The Power of Cliques

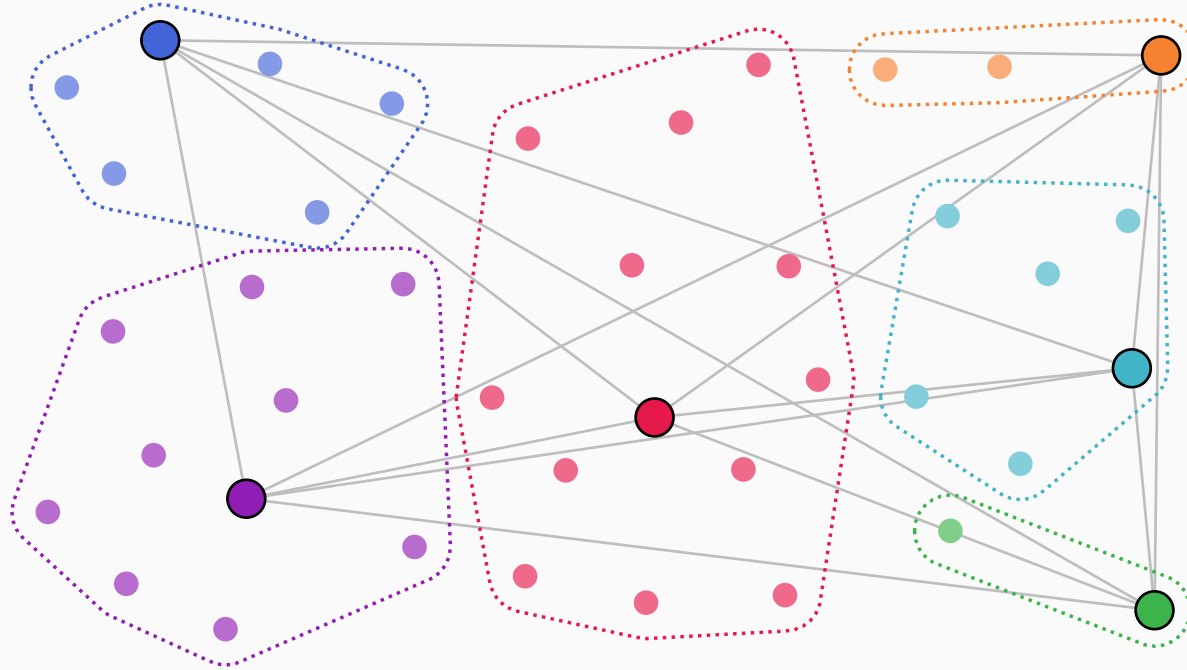
Simple Idea: Choose a subset of points $S \subset P$ arbitrarily and add a clique over them.



Ownership Sets

Each clique point $\mathbf{v} \in S$ owns the points that are closer to it than to any other point in S :

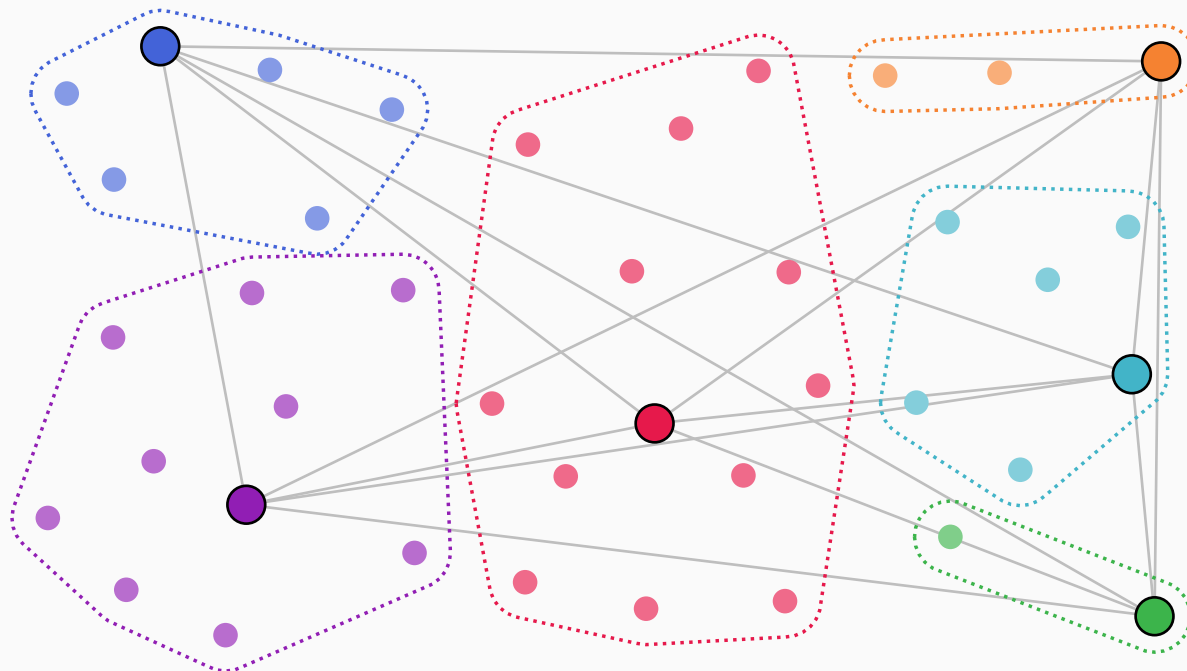
$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$



Ownership Sets

Each clique point $\mathbf{v} \in S$ *owns* the points that are closer to it than to any other point in S :

$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$



Observe: $\mathbf{v}$ has an edge closer to every point except the points in $U_{S,\mathbf{v}}$

The Power of Cliques

Claim (The Power of Cliques)

Let P be a set of n points and suppose $G = (P, E)$ contains a clique connecting all nodes in some subset $S \subseteq P$. For all $\mathbf{v} \in S$, define:

$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$

Then, for at least half the points in S , $|U_{S,\mathbf{v}}| \leq \frac{2n}{|S|}$.

The Power of Cliques

Claim (The Power of Cliques)

Let P be a set of n points and suppose $G = (P, E)$ contains a clique connecting all nodes in some subset $S \subseteq P$. For all $\mathbf{v} \in S$, define:

$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$

Then, for at least half the points in S , $|U_{S,\mathbf{v}}| \leq \frac{2n}{|S|}$.

Proof

The Power of Cliques

Claim (The Power of Cliques)

Let P be a set of n points and suppose $G = (P, E)$ contains a clique connecting all nodes in some subset $S \subseteq P$. For all $\mathbf{v} \in S$, define:

$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$

Then, for at least half the points in S , $|U_{S,\mathbf{v}}| \leq \frac{2n}{|S|}$.

Proof

Observe $\bigcup_{\mathbf{v} \in S} U_{S,\mathbf{v}} = P$

The Power of Cliques

Claim (The Power of Cliques)

Let P be a set of n points and suppose $G = (P, E)$ contains a clique connecting all nodes in some subset $S \subseteq P$. For all $\mathbf{v} \in S$, define:

$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$

Then, for at least half the points in S , $|U_{S,\mathbf{v}}| \leq \frac{2n}{|S|}$.

Proof

Observe $\bigcup_{\mathbf{v} \in S} U_{S,\mathbf{v}} = P$

$$\Rightarrow \text{average ownership set size} = \frac{1}{|S|} \sum_{\mathbf{v} \in S} |U_{S,\mathbf{v}}| = \frac{|P|}{|S|} = \frac{n}{|S|}$$

The Power of Cliques

Claim (The Power of Cliques)

Let P be a set of n points and suppose $G = (P, E)$ contains a clique connecting all nodes in some subset $S \subseteq P$. For all $\mathbf{v} \in S$, define:

$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$

Then, for at least half the points in S , $|U_{S,\mathbf{v}}| \leq \frac{2n}{|S|}$.

Proof

Observe $\bigcup_{\mathbf{v} \in S} U_{S,\mathbf{v}} = P$

$$\Rightarrow \text{average ownership set size} = \frac{1}{|S|} \sum_{\mathbf{v} \in S} |U_{S,\mathbf{v}}| = \frac{|P|}{|S|} = \frac{n}{|S|}$$

Fact: For positive data, the median is at most twice the mean.

The Power of Cliques

Claim (The Power of Cliques)

Let P be a set of n points and suppose $G = (P, E)$ contains a clique connecting all nodes in some subset $S \subseteq P$. For all $\mathbf{v} \in S$, define:

$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$

Then, for at least half the points in S , $|U_{S,\mathbf{v}}| \leq \frac{2n}{|S|}$.

Proof

Observe $\bigcup_{\mathbf{v} \in S} U_{S,\mathbf{v}} = P$

$$\Rightarrow \text{average ownership set size} = \frac{1}{|S|} \sum_{\mathbf{v} \in S} |U_{S,\mathbf{v}}| = \frac{|P|}{|S|} = \frac{n}{|S|}$$

Fact: For positive data, the median is at most twice the mean. Thus, for at least half the points in S ,

$$|U_{S,\mathbf{v}}| \leq \frac{2n}{|S|}$$

Claim (The Power of Cliques)

Let P be a set of n points and suppose $G = (P, E)$ contains a clique connecting all nodes in some subset $S \subseteq P$. For all $\mathbf{v} \in S$, define:

$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$

Then, for at least half the points in S , $|U_{S,\mathbf{v}}| \leq \frac{2n}{|S|}$.

Consequence

Choose $|S| = \frac{2}{1-\gamma}$. Then, at least $\frac{|S|}{2}$ points have γ -almost navigable out-neighborhoods.

Claim (The Power of Cliques)

Let P be a set of n points and suppose $G = (P, E)$ contains a clique connecting all nodes in some subset $S \subseteq P$. For all $\mathbf{v} \in S$, define:

$$U_{S,\mathbf{v}} := \left\{ \mathbf{p} \in P \mid \mathbf{v} = \operatorname{argmin}_{\mathbf{x} \in S} d(\mathbf{p}, \mathbf{x}) \right\}$$

Then, for at least half the points in S , $|U_{S,\mathbf{v}}| \leq \frac{2n}{|S|}$.

Consequence

Choose $|S| = \frac{2}{1-\gamma}$. Then, at least $\frac{|S|}{2}$ points have γ -almost navigable out-neighborhoods.

Algorithm Main Idea

Repeat this procedure over multiple subsets $S_1, \dots, S_k \subset P$ to get a complete γ -almost navigable graph with average degree $O\left(\frac{1}{1-\gamma}\right)$.