Lecture outline

- Nearest-neighbor search in low dimensions
  - kd-trees
- Nearest-neighbor search in high dimensions
  - LSH
- Applications to data mining

Definition

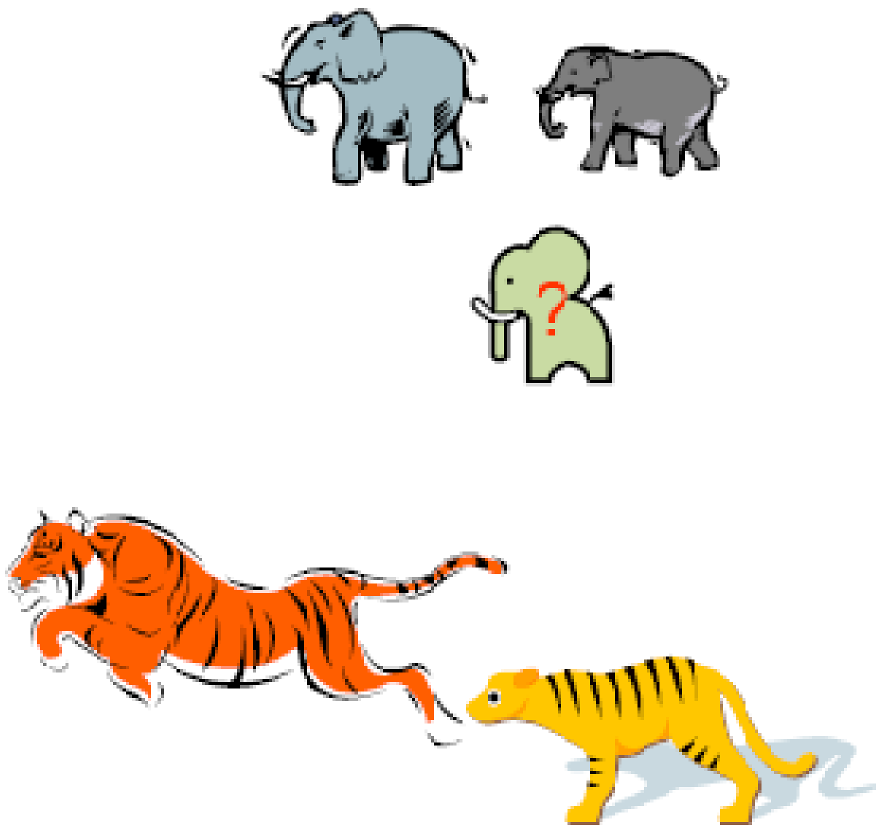
- Given: a set **X** of **n** points in $\mathbf{R}^d$
- Nearest neighbor: for any query point $\mathbf{q \in R}^d$ return the point $\mathbf{x \in X}$ minimizing $\mathbf{D(x,q)}$

- **Intuition:** Find the point in **X** that is the ***closest*** to **q**

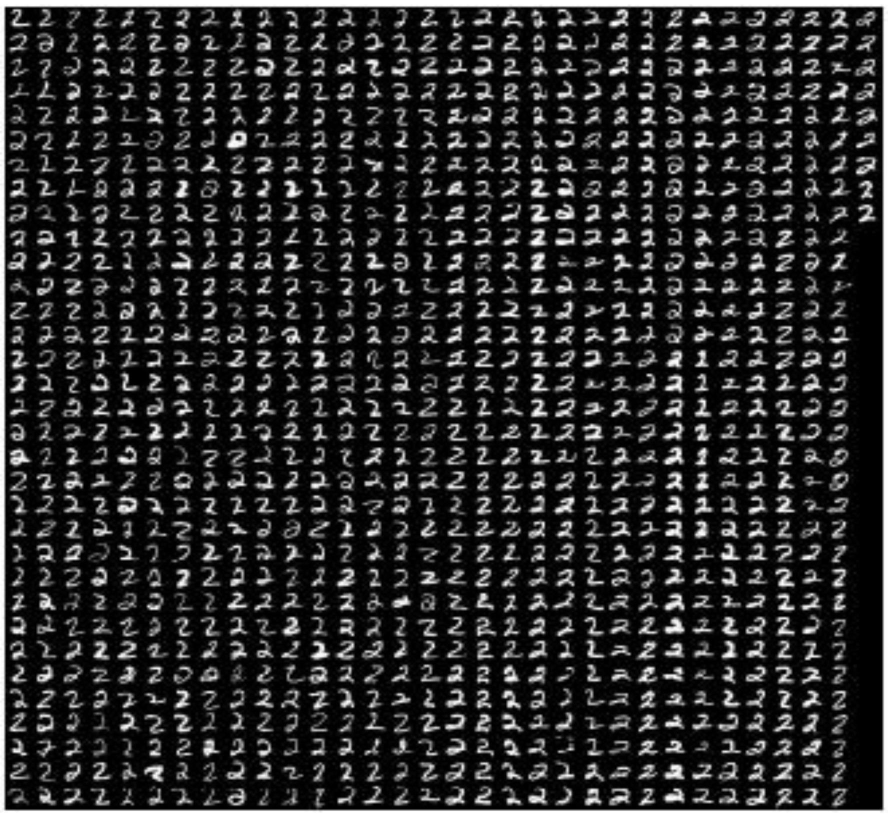
Motivation

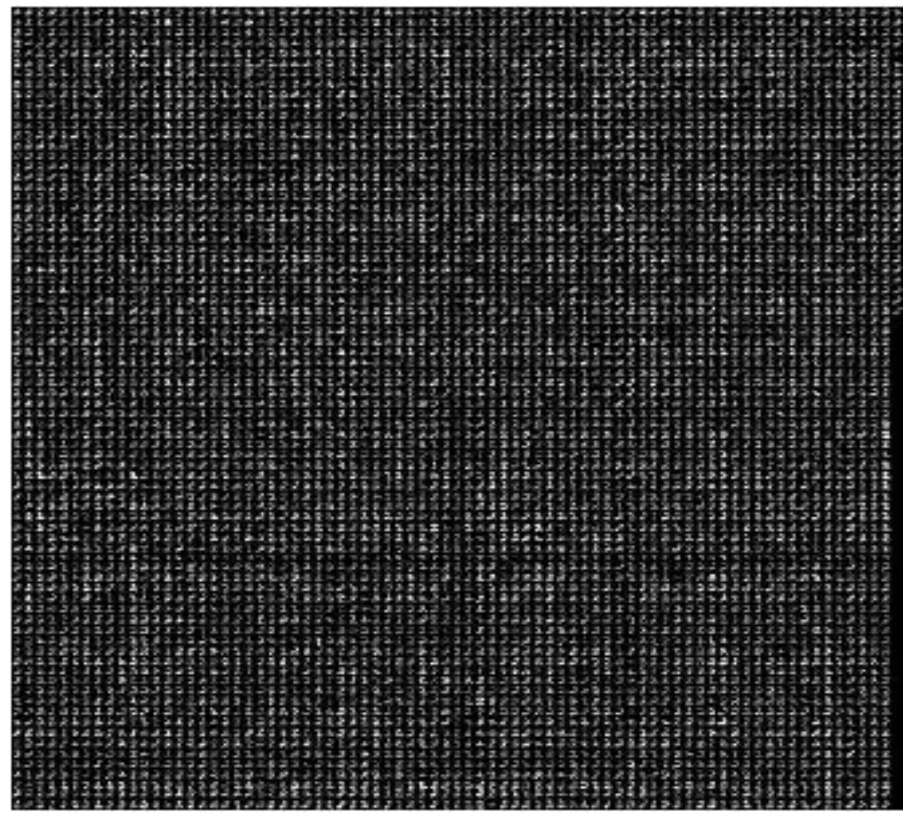
- **Learning:** Nearest neighbor rule
- **Databases:** Retrieval
- **Data mining:** Clustering
- Donald Knuth in vol.3 of ***The Art of Computer Programming*** called it the post-office problem, referring to the application of assigning a resident to the ***nearest-post office***

Nearest-neighbor rule



MNIST dataset “2”





Methods for computing NN

- **Linear scan: O(nd)** time

- This is pretty much all what is known for exact algorithms with theoretical guarantees

- In practice:
  - ***kd-trees*** work “well” in “low-medium” dimensions

2-dimensional kd-trees

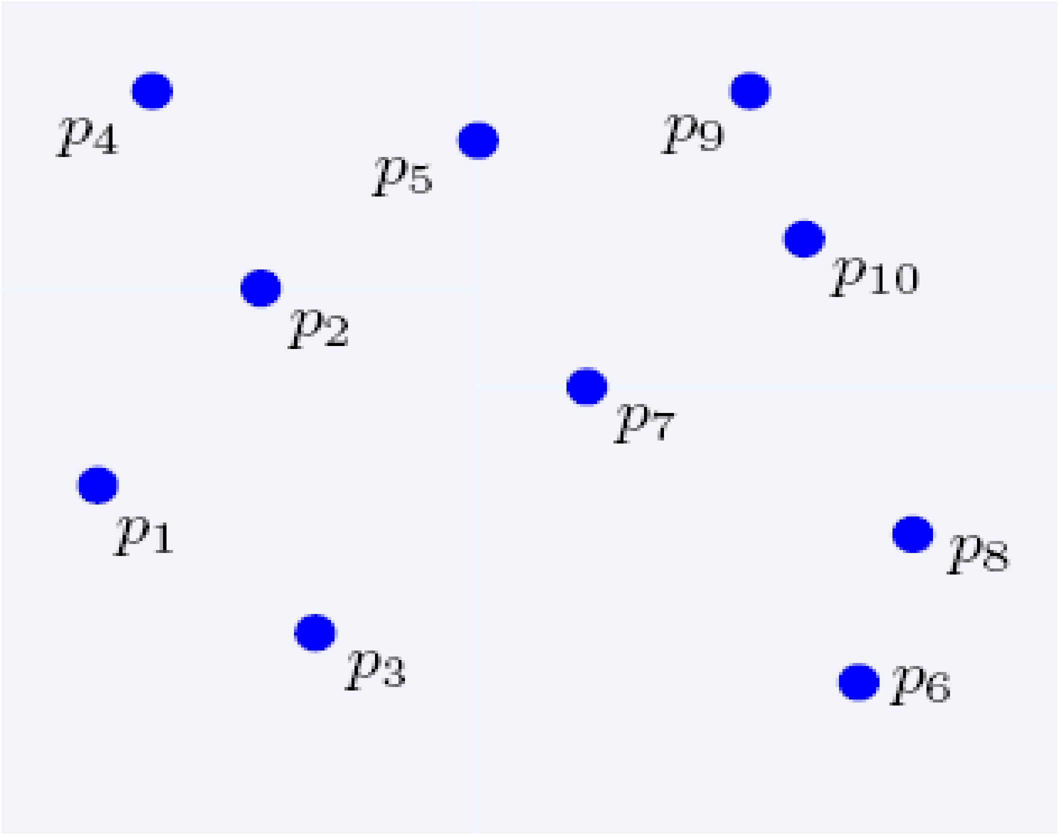
- A data structure to support range queries in $\mathbf{R^2}$
  - Not the most efficient solution in theory
  - Everyone uses it in practice

- Preprocessing time: **$O(n\log n)$**
- Space complexity: **$O(n)$**
- Query time: **$O(n^{1/2}+k)$**

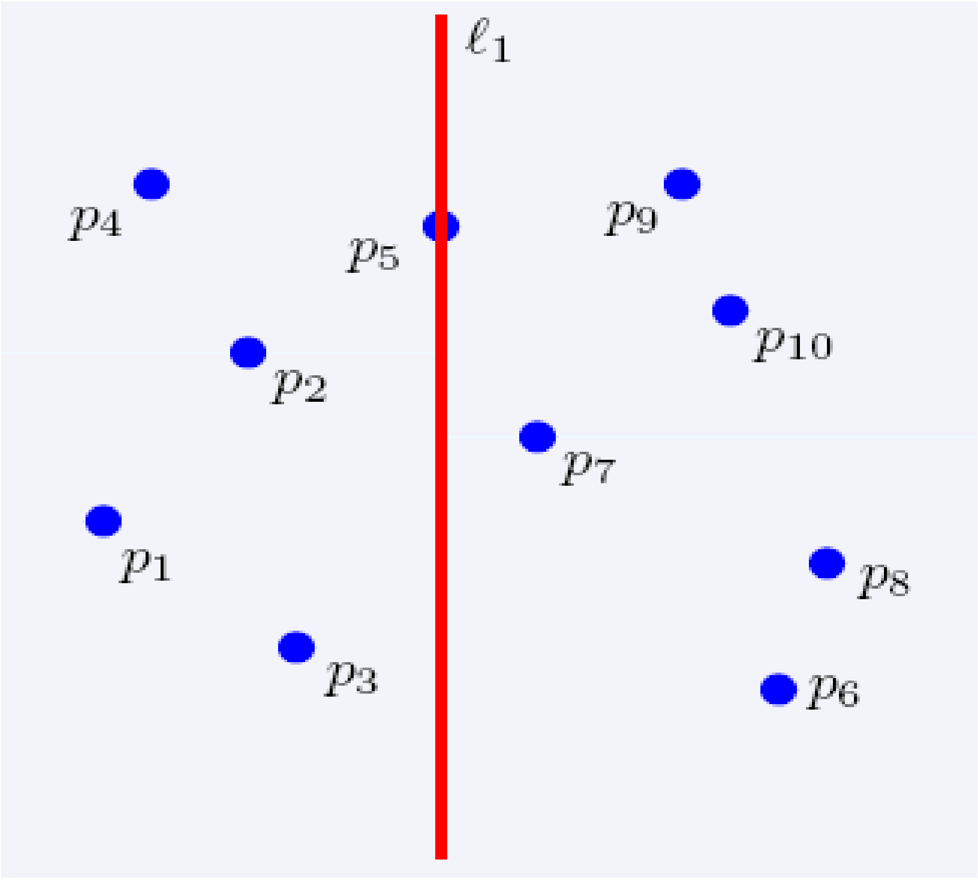
2-dimensional kd-trees

- Algorithm:
  - Choose **x** or **y** coordinate (alternate)
  - Choose the median of the coordinate; this defines a horizontal or vertical line
  - Recurse on both sides
- We get a binary tree:
  - Size **O(n)**
  - Depth **O(logn)**
  - Construction time **O(nlogn)**

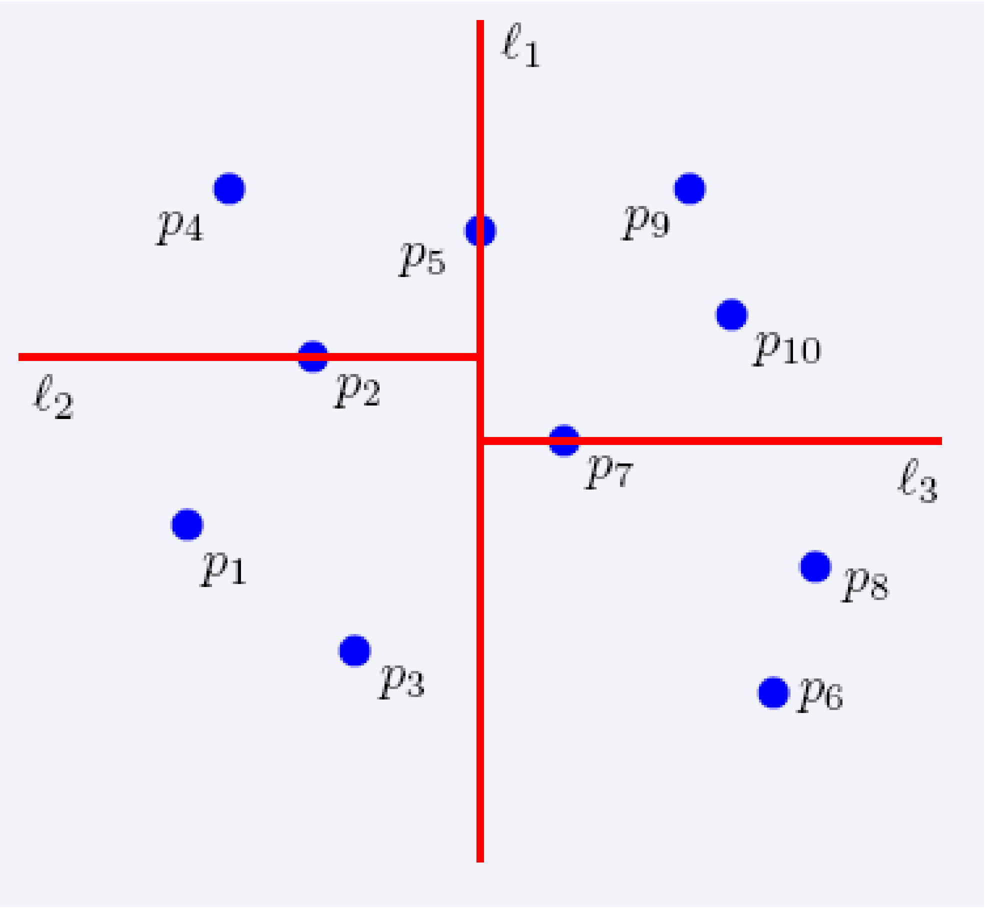
Construction of kd-trees



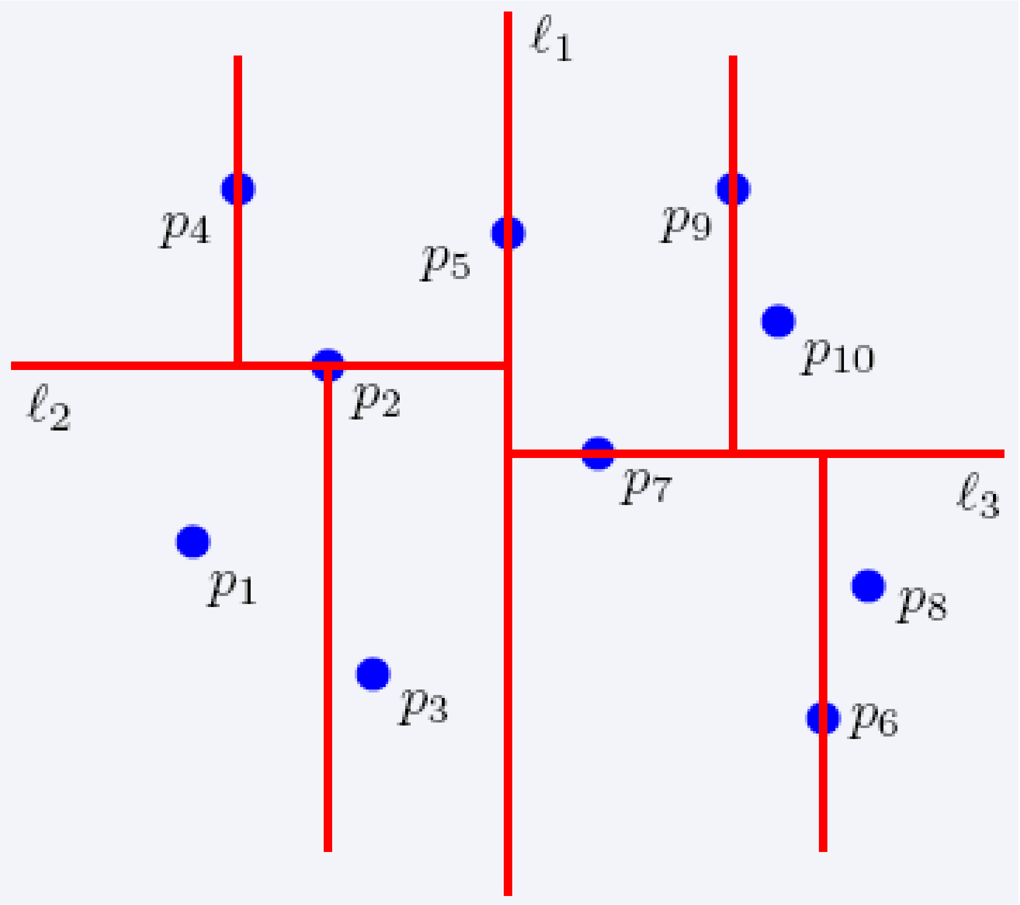
Construction of kd-trees



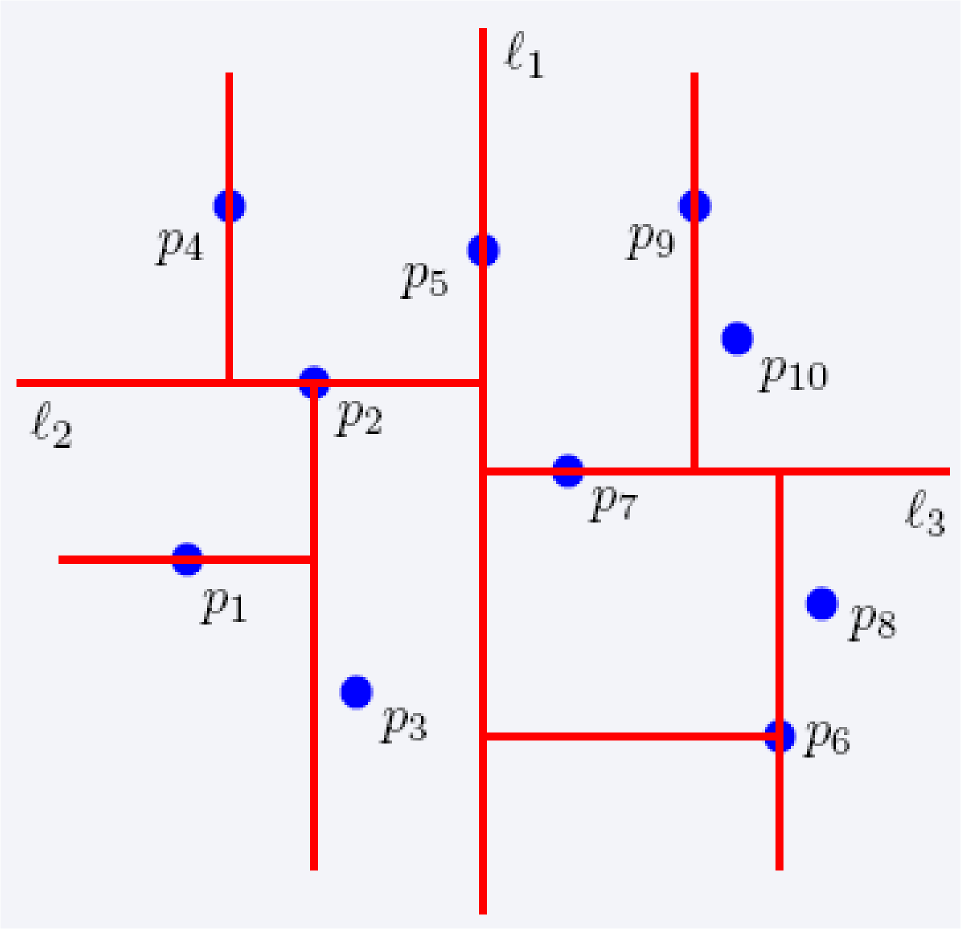
Construction of kd-trees



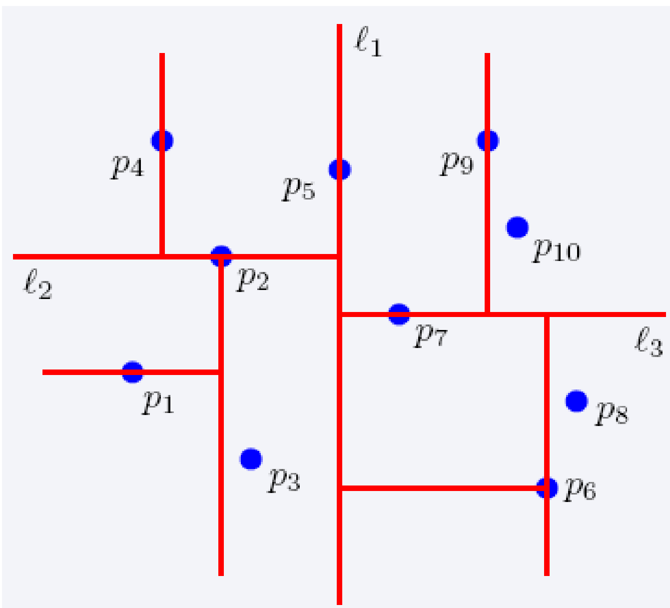
Construction of kd-trees

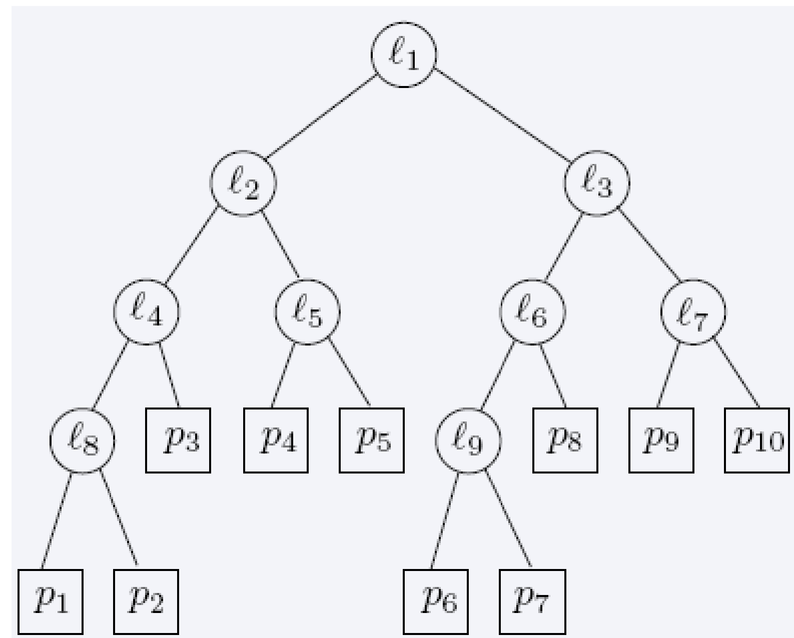


Construction of kd-trees

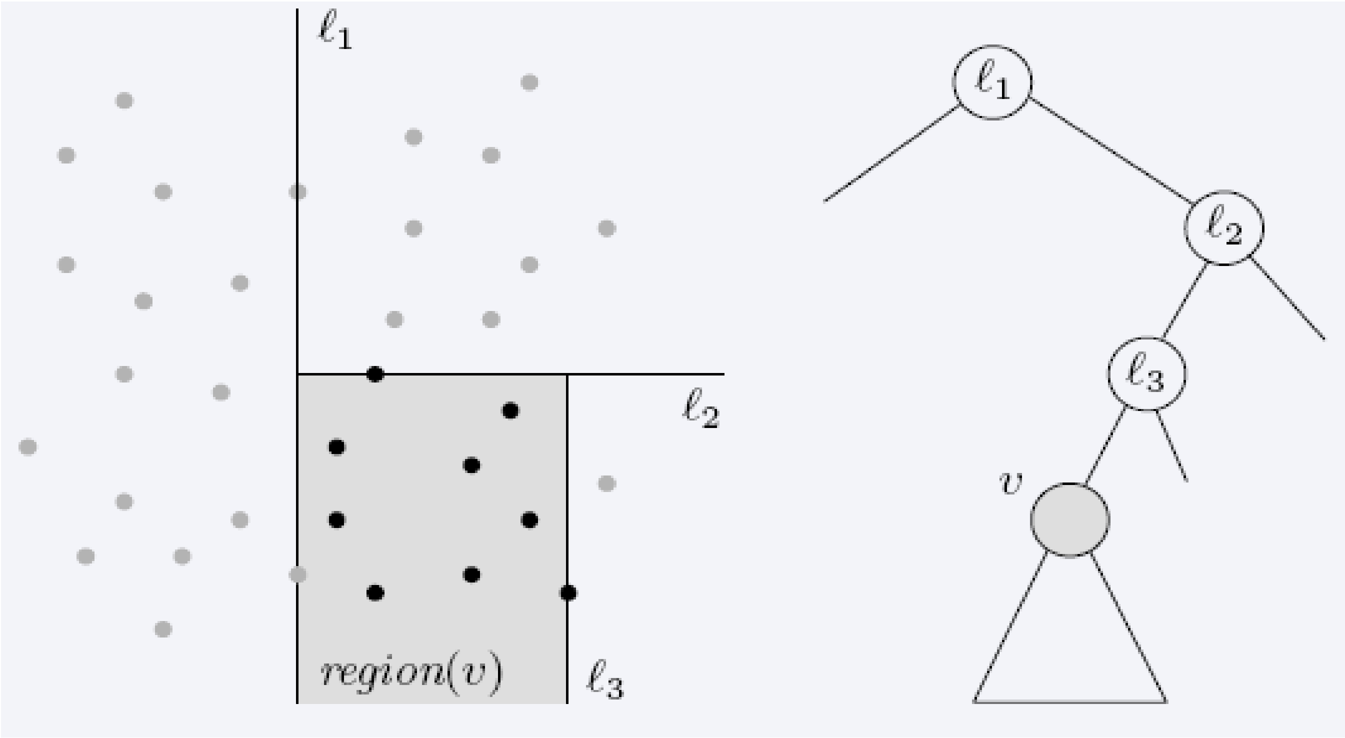


The complete kd-tree





Region of node v



Region(v) : the subtree rooted at **v** stores the points in black dots

Searching in kd-trees

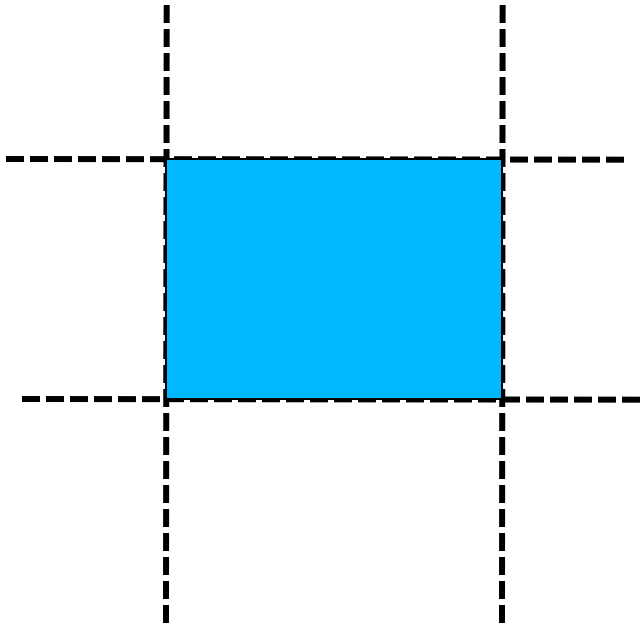
- Range-searching in **2-d**
  - Given a set of **n** points, build a data structure that for any query rectangle **R** reports all point in **R**

kd-tree: range queries

- Recursive procedure starting from **v = root**
- **Search (v,R)**
  - If **v** is a leaf, then report the point stored in **v** if it lies in **R**
  - Otherwise, if **Reg(v)** is contained in **R**, report all points in the **subtree(v)**
  - Otherwise:
    - If **Reg(left(v))** intersects **R**, then **Search(left(v),R)**
    - If **Reg(right(v))** intersects **R**, then **Search(right(v),R)**

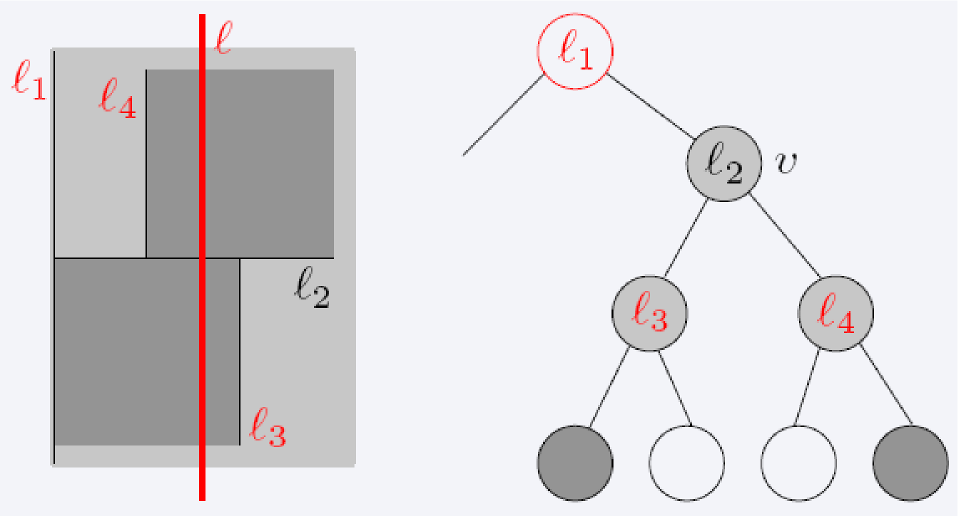
Query time analysis

- We will show that **Search** takes at most **$O(n^{1/2}+P)$** time, where **P** is the number of reported points
  - The total time needed to report all points in all sub-trees is **$O(P)$**
  - We just need to bound the number of nodes **v** such that **region(v)** intersects **R** but is not contained in **R** (i.e., boundary of R intersects the boundary of **region(v)**)
  - ***gross overestimation***: bound the number of **region(v)** which are crossed by any of the **4** horizontal/vertical lines



Query time (Cont’d)

- **Q(n):** max number of regions in an n-point kd-tree intersecting a (say, vertical) line?



- If **ℓ** intersects **region(v)** (due to vertical line splitting), then after two levels it intersects **2** regions (due to 2 vertical splitting lines)
- The number of regions intersecting **ℓ** is **$Q(n)=2+2Q(n/4)$** → **$Q(n)=(n^{1/2})$**

d-dimensional kd-trees

- A data structure to support range queries in $\mathbf{R}^d$

- Preprocessing time: **$O(n\log n)$**
- Space complexity: **$O(n)$**
- Query time: **$O(n^{1-1/d}+k)$**

Construction of the **d**-dimensional kd-trees

- The construction algorithm is similar as in **2-d**
- At the root we split the set of points into two subsets of same size by a hyperplane vertical to $\mathbf{x_1}$-axis
- At the children of the root, the partition is based on the second coordinate: $\mathbf{x_2}$-coordinate
- At depth **d**, we start all over again by partitioning on the first coordinate
- The recursion stops until there is only one point left, which is stored as a leaf

Locality-sensitive hashing (LSH)

- **Idea**: Construct hash functions **$h: R^d \rightarrow U$** such that for any pair of points **p,q**:
  - If **$D(p,q) \le r$**, then **$Pr[h(p)=h(q)]$** is high
  - If **$D(p,q) \ge cr$**, then **$Pr[h(p)=h(q)]$** is small
- Then, we can solve the "approximate NN" problem by hashing

- LSH is a general framework; for a given **D** we need to find the right **h**

Approximate Nearest Neighbor

- Given a set of points **X** in $\mathbf{R^d}$ and query point $\mathbf{q \epsilon R^d}$ **c**-Approximate **r**-Nearest Neighbor search returns:
  - Returns $\mathbf{p \in P, D(p,q) \leq r}$
  - Returns NO if there is no $\mathbf{p' \in X, D(p',q) \leq cr}$

Locality-Sensitive Hashing (LSH)

- A family **H** of functions **$h: R^d \rightarrow U$** is called **(P_1,P_2,r,cr)**-sensitive if for any p,q:
  - if **$D(p,q) \le r$**, then **$Pr[h(p)=h(q)] \ge P1$**
  - If **$D(p,q) \ge cr$**, then **$Pr[h(p)=h(q)] \le P2$**
- **P1 > P2**
- Example: **Hamming** distance
  - LSH functions: **$h(p)=p_i$**, i.e., the **i**-th bit of **p**
  - Probabilities: **$Pr[h(p)=h(q)]=1-D(p,q)/d$**

Algorithm -- preprocessing

- $\mathbf{g(p) = \langle h_1(p),h_2(p),\ldots,h_k(p)\rangle}$
- Preprocessing
  - Select $\mathbf{g_1,g_2,\ldots,g_L}$
  - For all $\mathbf{p \in X}$ hash $\mathbf{p}$ to buckets $\mathbf{g_1(p),\ldots,g_L(p)}$
  - Since the number of possible buckets might be large we only ***maintain the non empty ones***

- Running time?

Algorithm -- query

- Query **q**:
  - Retrieve the points from buckets **$g_1(q), g_2(q), \ldots, g_L(q)$** and let points retrieved be **$x_1, \ldots, x_L$**
    - If **$D(x_i, q) \leq r$** report it
    - Otherwise report that there does not exist such a NN
  - Answer the query based on the retrieved points
  - Time **$O(dL)$**

Applications of LSH in data mining

- Numerous….

Applications

- Find pages with similar sets of words (for clustering or classification)

- Find users in Netflix data that watch similar movies
- Find movies with similar sets of users

- Find images of related things

How would you do it?

- Finding very similar items might be computationally demanding task

- We can relax our requirement to finding ***somewhat similar*** items

Running example: comparing documents

- Documents have common text, but no common topic
- Easy special cases:
  - Identical documents
  - Fully contained documents (letter by letter)
- General case:
  - Many small pieces of one document appear out of order in another. What do we do then?

Finding similar documents

- Given a collection of documents, find pairs of documents that have lots of text in common
  - Identify mirror sites or web pages
  - Plagiarism
  - Similar news articles

Key steps

- **Shingling:** convert documents (news articles, emails, etc) to sets
- **LSH:** convert large sets to ***small signatures***, while preserving the similarity
- Compare the signatures instead of the actual documents

Shingles

- A **k-shingle** (or **k-gram**) is a sequence of k characters that appears in a document

- If doc = abcab and k=3, then 2-singles: {ab, bc, ca}

- **Represent a document by a set of k-shingles**

Assumption

- Documents that have similar sets of **k**-shingles are similar: same text appears in the two documents; the position of the text does not matter

- What should be the value of **k**?
  - What would large or small k mean?

Data model: sets

- Data points are represented as sets (i.e., sets of shingles)

- Similar data points have large intersections in their sets

  – Think of documents and shingles
  – Customers and products
  – Users and movies

Similarity measures for sets

- Now we have a set representation of the data
- Jaccard coefficient
- **A, B** sets (subsets of some, large, universe **U**)

$$sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Find similar objects using the Jaccard similarity

- Naïve method?

- Problems with the naïve method?
  - There are too many objects
  - Each object consists of too many sets

Speedingup the naïve method

- Represent every object by a signature (summary of the object)
- Examine pairs of signatures rather than pairs of objects
- Find all similar pairs of signatures
- **Check point:** check that objects with similar signatures are actually similar

Still problems

- Comparing large number of signatures with each other may take too much time (although it takes less space)
- The method can produce pairs of objects that might not be similar (false positives). The check point needs to be enforced

Creating signatures

- For object **x**, signature of **x (sign(x))** is much smaller (in space) than **x**

- For objects **x, y** it should hold that **sim(x,y)** is almost the same as **sim(sing(x),sign(y))**

Intuition behind Jaccard similarity

- Consider two objects: **x,y**

	x	**y**
a	1	1
b	1	0
c	0	1
d	0	0

- **a**: **#** of rows of form same as **a**
- **sim(x,y)= a /(a+b+c)**

A type of signatures -- minhashes

- Randomly ***permute*** the rows

	x	**y**
a	1	1
b	1	0
c	0	1
d	0	0

- **h(x):** first row (in permuted data) in which column **x** has an **1**

	x	**y**
a	0	1
b	0	0
c	1	1
d	1	0

- Use several (e.g., 100) independent hash functions to design a signature

“Surprising” property

- The probability (over all permutations of rows) that **h(x)=h(y)** is the same as **sim(x,y)**
- Both of them are **a/(a+b+c)**
- So?
  - **The similarity of signatures is the fraction of the hash functions on which they agree**

Minhash algorithm

- Pick **k** (e.g., 100) permutations of the rows
- Think of **sign(x)** as a new vector
- Let **sign(x)[i]**: in the **i**-th permutation, the index of the **first row that has** **1** for object **x**

Example of minhash signatures

- Input matrix

	x1	x2	x3	X4
1	1	0	1	0
2	1	0	0	1
3	0	1	0	1
4	0	1	0	1
5	0	1	0	1
6	1	0	1	0
7	1	0	1	0

→

1
3
7
6
2
5
4

→

	x1	x2	x3	X4
1	1	0	1	0
3	0	1	0	1
7	1	0	1	0
6	1	0	1	0
2	1	0	0	1
5	0	1	0	1
4	0	1	0	1

1	**2**	**1**	**2**

Example of minhash signatures

- Input matrix

	x1	x2	x3	X4
1	1	0	1	0
2	1	0	0	1
3	0	1	0	1
4	0	1	0	1
5	0	1	0	1
6	1	0	1	0
7	1	0	1	0

4
2
1
3
6
7
5

	x1	x2	x3	X4
4	0	1	0	1
2	1	0	0	1
1	1	0	1	0
3	0	1	0	1
6	1	0	1	0
7	1	0	1	0
5	0	1	0	1

2	**1**	**3**	**1**

Example of minhash signatures

- Input matrix

	x1	x2	x3	X4
1	1	0	1	0
2	1	0	0	1
3	0	1	0	1
4	0	1	0	1
5	0	1	0	1
6	1	0	1	0
7	1	0	1	0

3
4
7
6
1
2
5

	x1	x2	x3	X4
3	0	1	0	1
4	0	1	0	1
7	1	0	1	0
6	1	0	1	0
1	1	0	1	0
2	1	0	0	1
5	0	1	0	1

3	**1**	**3**	**1**

Example of minhash signatures

- Input matrix

	x1	x2	x3	X4
1	1	0	1	0
2	1	0	0	1
3	0	1	0	1
4	0	1	0	1
5	0	1	0	1
6	1	0	1	0
7	1	0	1	0

≈

x1	x2	x3	X4
1	2	1	2
2	1	3	1
3	1	3	1

	actual	signs
(x1,x2)	0	0
(x1,x3)	0.75	2/3
(x1,x4)	1/7	0
(x2,x3)	0	0
(x2,x4)	0.75	1
(x3,x4)	0	0

Is it now feasible?

- Assume a billion rows
- Hard to pick a random permutation of 1…billion
- **Even representing a random permutation requires 1 billion entries!!!**
- How about accessing rows in permuted order?
- ☹

Being more practical

- Approximating row permutations: pick **k=100** (?) hash functions **($h_1,\ldots,h_k$)**

for each row **r**

  for each column **c**

    if c has **1** in row **r**

      for each hash fun

        if $h_i(r)$ is a smalle

          $M(i,c) = h_i(r)$;

M(i,c) will become the smallest value of **$h_i(r)$** for which column **c** has **1** in row **r**; *i*.e., **$h_i(r)$** gives order of rows for **i-**th permutation.

Example of minhash signatures

- Input matrix

	x1	x2
1	1	0
2	0	1
3	1	1
4	1	0
5	0	1

	x1	x2
1	0	1
2	2	0

h(r) = r + 1 mod 5
g(r) = 2r + 1 mod 5