Near-Neighbor Search: Finding similar sets

CS246: Mining Massive Datasets

Jure Leskovec, Stanford University

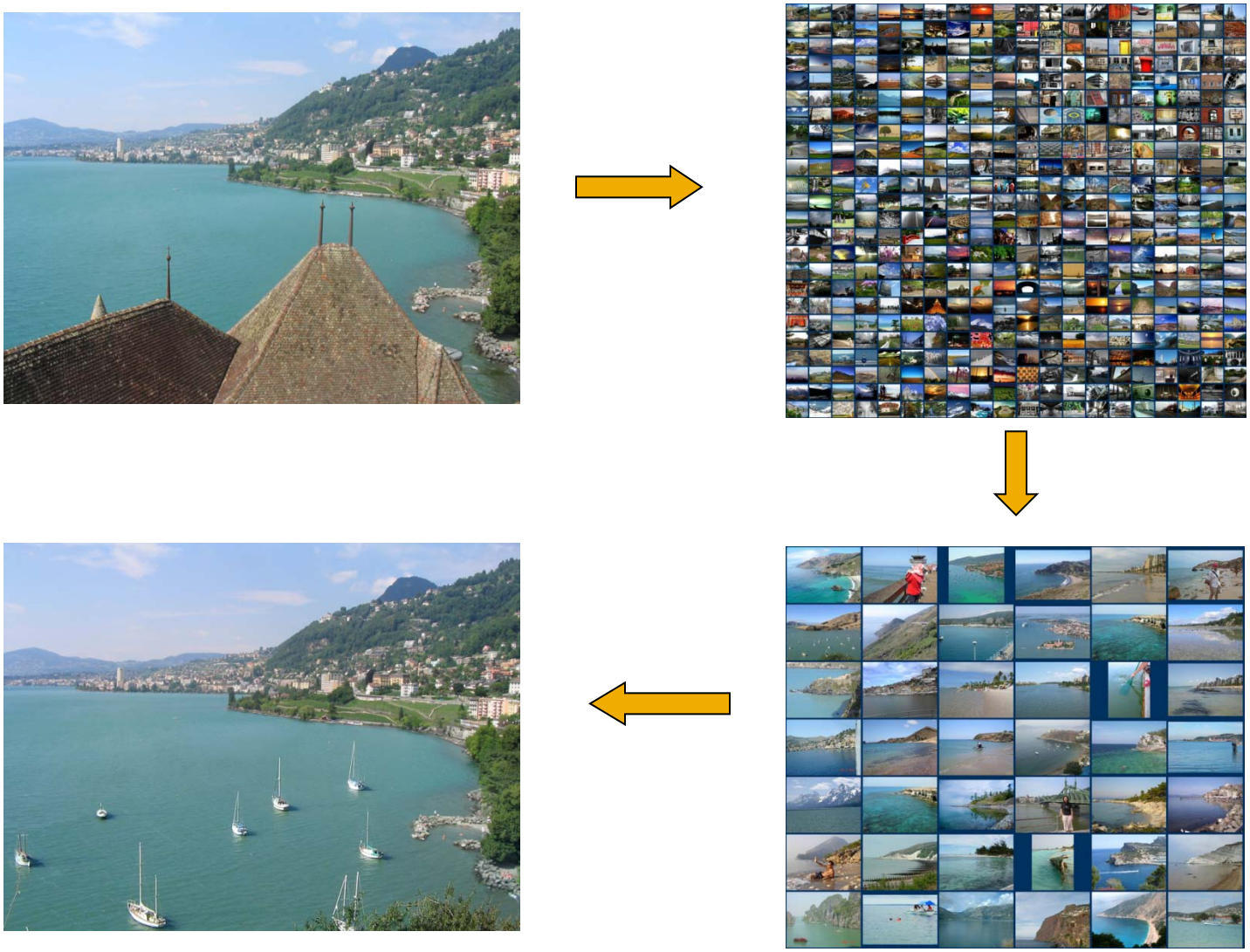
http://cs246.stanford.edu

Announcements

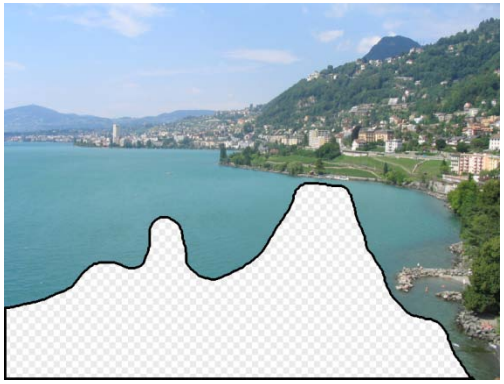
- **Tuesday 1/11 5-7pm Gates B12:** Hadoop Q&A session
- Association Rules Gradiance assignment is out!
  - Due: 2011-01-17 23:59

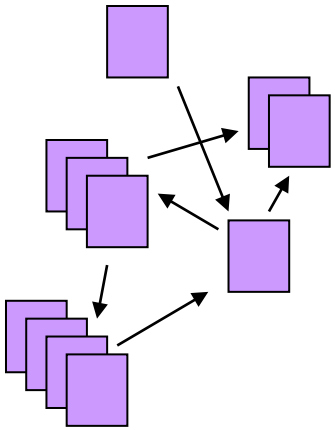
[Hays and Efros, SIGGRAPH 2007]

Scene Completion Problem



The Bare Data Approach





The Web

Simple algorithms with access to large datasets

High Dimensional Data

- Many real-world problems
  - Web Search and Text Mining
    - Billions of documents, millions of terms
  - Product Recommendations
    - Millions of customers, millions of products
  - Scene Completion, other graphics problems
    - Image features
  - Online Advertising, Behavioral Analysis
    - Customer actions e.g., websites visited, searches

A Common Metaphor

- Many problems can be expressed as finding "similar" sets:
  - Find near-neighbors in high-D space
- Examples:
  - Pages with similar words
    - For duplicate detection, classification by topic
  - Customers who purchased similar products
    - NetFlix users with similar tastes in movies
    - Products with similar customer sets
  - Images with similar features
  - Users who visited the same websites
- In some cases, result is set of nearest neighbors
- In other cases, extrapolate the result from attributes of near-neighbors

Example: Question Answering

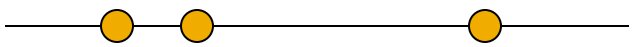
- Question Answering
  - Who killed Abraham Lincoln?
  - What is the height of Mount Everest?
- Naïve algorithm:
  - Find all web pages containing the terms “killed” and “Abraham Lincoln” in close proximity
  - Extract k-grams from a small window around the terms
  - Find the most commonly occuring k-grams

Example: Question Answering

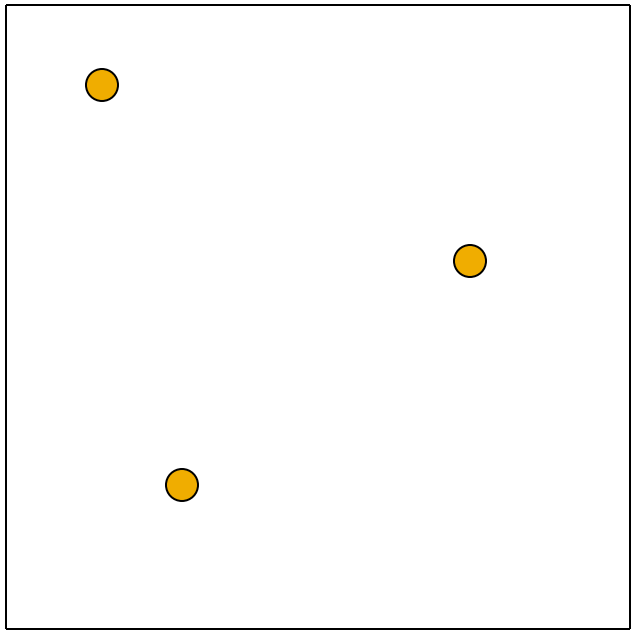
- Naïve algorithm works fairly well!
- Some improvements
  - Use sentence structure, e.g., restrict to noun phrases only
  - Rewrite questions before matching
    - "What is the height of Mt Everest" becomes "The height of Mt Everest is <blank>"
- The number of pages analyzed is more important than the sophistication of the NLP
  - For simple questions

Reference: Dumais et al.

The Curse of Dimesnsionality



1-d space



2-d space

The Curse of Dimensionality

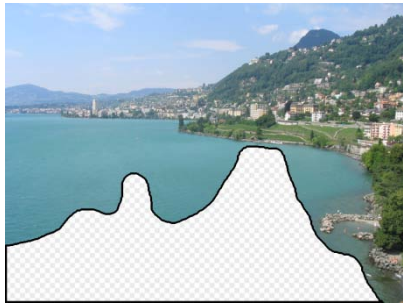
- Let's take a data set with a fixed number N of points
- As we increase the number of dimensions in which these points are embedded, the average distance between points keeps increasing
- Fewer "neighbors" on average within a certain radius of any given point

The Sparsity Problem

- Most customers have not purchased most products
- Most scenes don't have most features
- Most documents don't contain most terms
- Easy solution: Add more data!
  - More customers, longer purchase histories
  - More images
  - More documents
  - And there's more of it available every day!

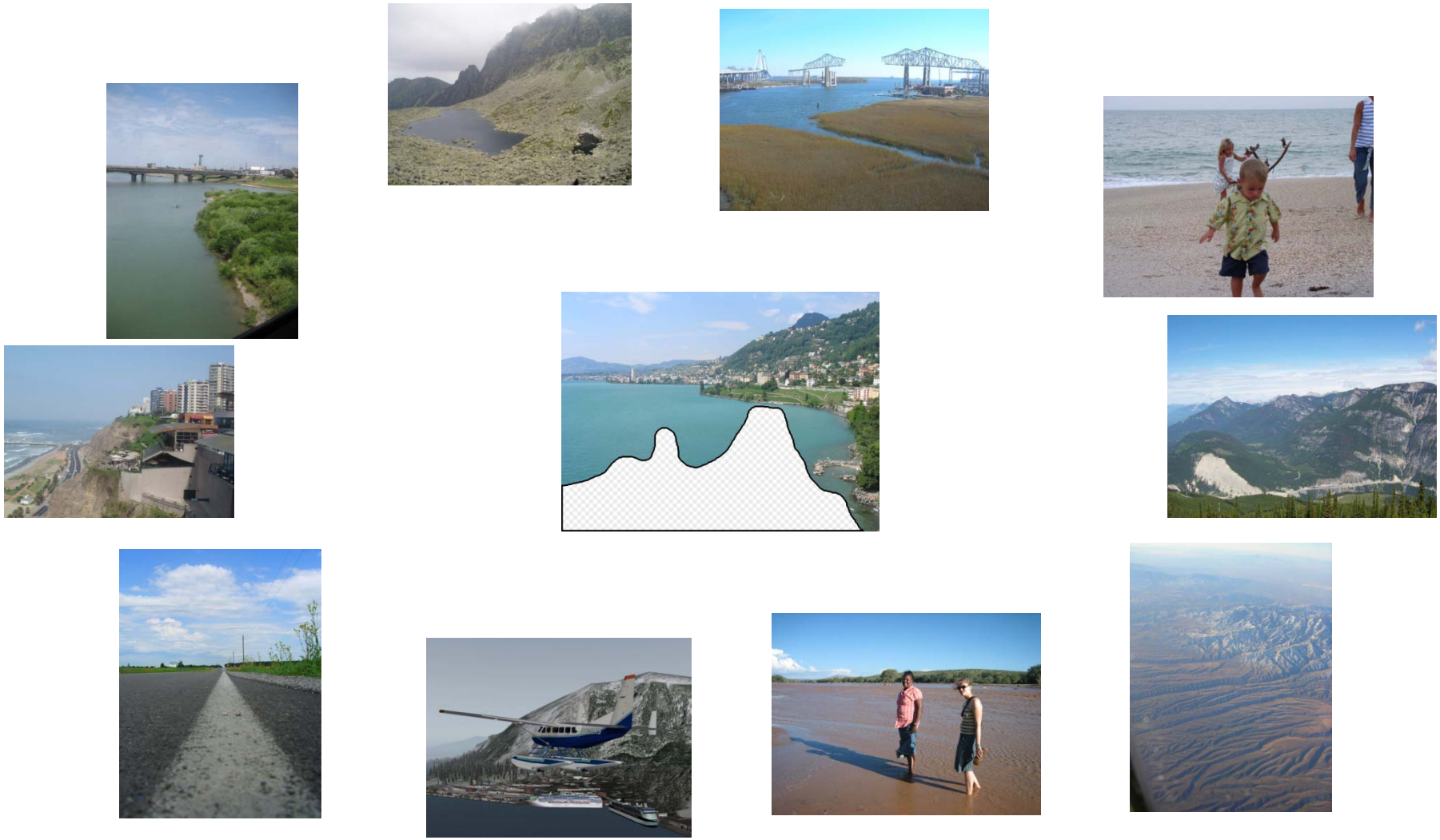
[Hays and Efros, SIGGRAPH 2007]

Example: Scene Completion



[Hays and Efros, SIGGRAPH 2007]

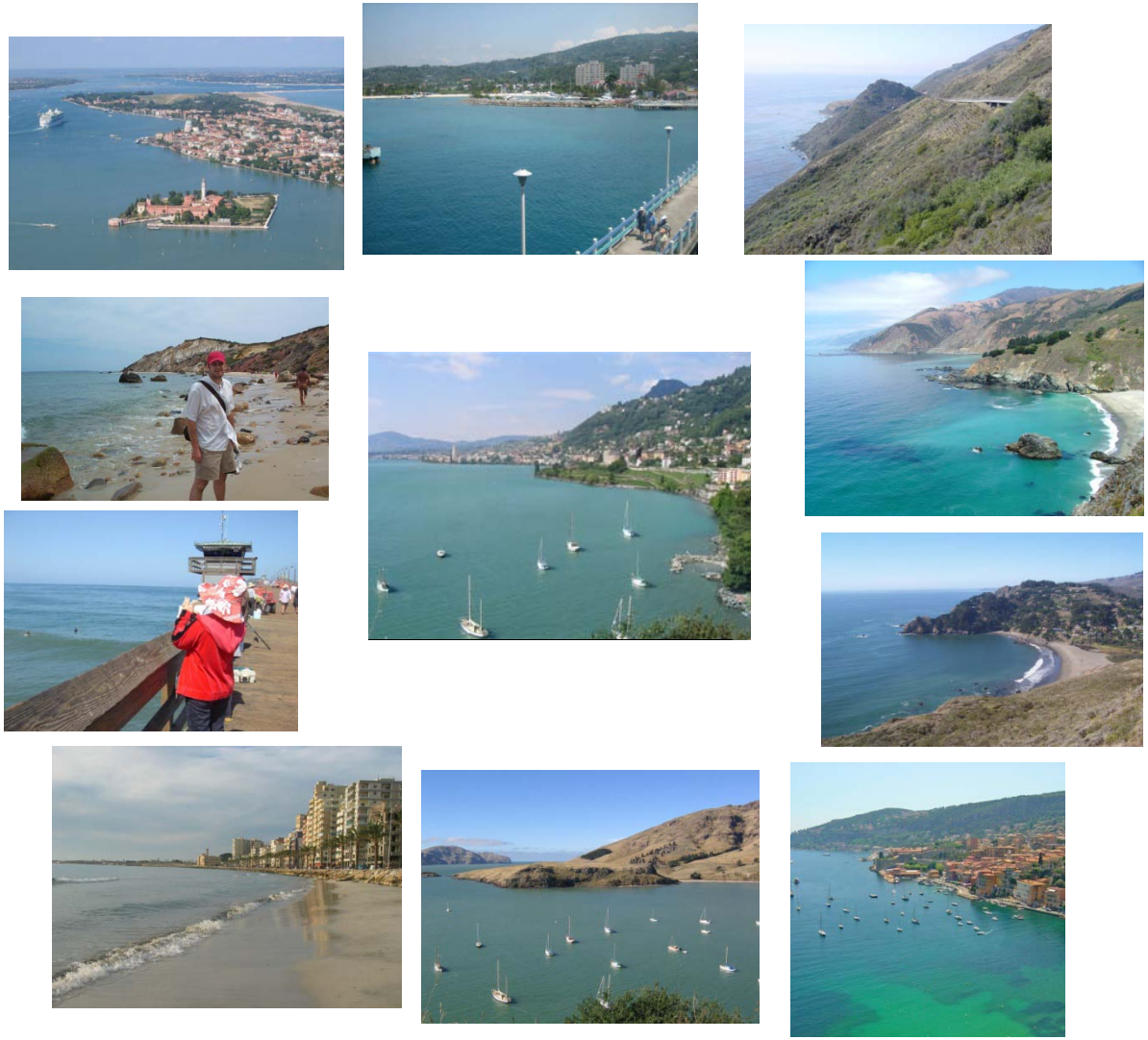
Example: Scene Completion



10 nearest neighbors from a collection of 20,000 images

[Hays and Efros, SIGGRAPH 2007]

Example: Scene Completion



10 nearest neighbors from a collection of 2 million images

Distance Measures

- We formally define "near neighbors" as points that are a "small distance" apart
- For each use case, we need to define what "distance" means
- Two major classes of distance measures:
  - Euclidean
  - Non-Euclidean

Euclidean Vs. Non-Euclidean

- A *Euclidean space* has some number of real-valued dimensions and "dense" points
  - There is a notion of "average" of two points
- A *Euclidean distance* is based on the locations of points in such a space
- A *Non-Euclidean distance* is based on properties of points, but not their "location" in a space

Axioms of a Distance Measure

- *d* is a *distance measure (i.e., metric)* if it is a function that maps from pairs of points to real numbers such that:
  1. $d(p,q) \geq 0$
  2. $d(p,q) = 0$ iff $p = q$
  3. $d(p,q) = d(q,p)$
  4. $d(p,q) \leq d(p,r) + d(r,q)$ (*triangle inequality*)

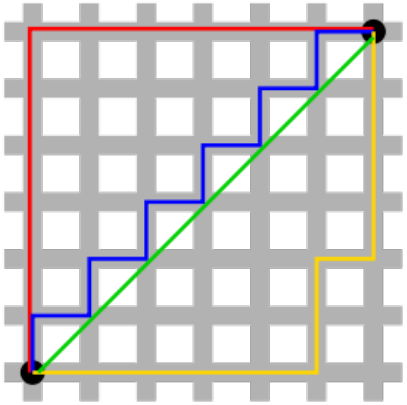
Some Euclidean Distances

- *L_2 norm*: d(p,q) = square root of the sum of the squares of the differences between p and q in each dimension:

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \mathrm{d}(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2+(q_2-p_2)^2+\cdots+(q_n-p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}.$$

  - The most common notion of "distance"
- *L_1 norm*: sum of the absolute differences in each dimension
  - *Manhattan distance* = distance if you had to travel along coordinates only

$$d_1(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_1 = \sum_{i=1}^{n}|p_i-q_i|,$$



Another Euclidean Distance

- *L_∞ norm*: d(x,y) = the maximum of the differences between *x* and *y* in any dimension.

$$D_{\text{Chebyshev}}(p,q) := \max_i(|p_i - q_i|).$$

- Note: the maximum is the limit as p goes to ∞ of the *L_p* norm:

$$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}$$

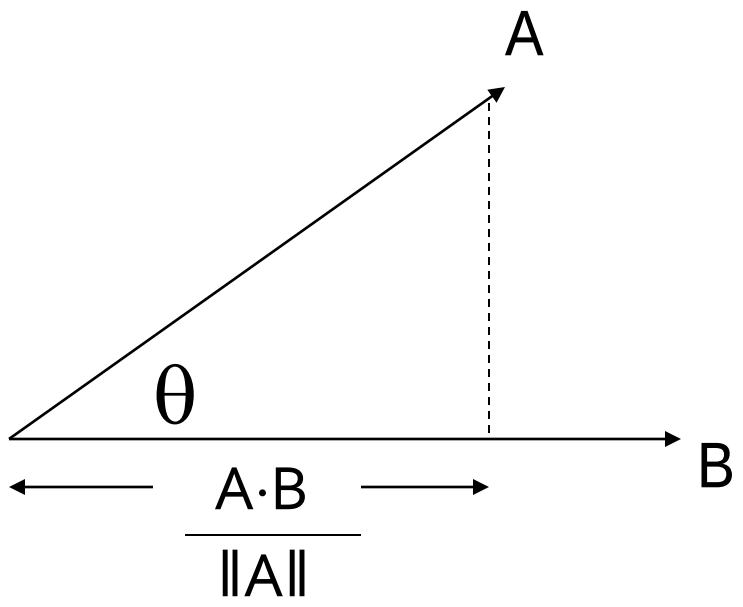
Non-Euclidean Distances

- *Cosine distance* = angle between vectors from the origin to the points in question
- *Edit distance* = number of inserts and deletes to change one string into another
- *Hamming Distance* = number of positions in which bit vectors differ

Cosine Distance/Similarity

- Think of a point as a vector from the origin (0,0,…,0) to its location
- Two points' vectors make an angle, whose cosine is the normalized dot-product of the vectors: $\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$
  - Example: A = 00111; B = 10011
  - $A \cdot B = 2$; $\|A\| = \|B\| = \sqrt{3}$
  - $\cos(\theta) = 2/3$; θ is about 48 degrees

Cosine Distance: Diagram



$$d(A, B) = \theta = \arccos(A \cdot B / \|A\| \, \|B\|)$$

Cosine Distance is a Metric

- d(x,x) = 0 because arccos(1) = 0
- d(x,y) = d(y,x) by symmetry
- d(x,y) $\geq$ 0 because angles are chosen to be in the range 0 to 180 degrees
- Triangle inequality: If we rotate an angle from *x* to *z* and then from *z* to *y*, we can't rotate less than from *x* to *y*

Edit Distance

- The *edit distance* of two strings is the number of inserts and deletes of characters needed to turn one into the other
  - Equivalently:

$$d(x,y) = |x| + |y| - 2|LCS(x,y)|$$

- LCS = *Longest Common Subsequence* = any longest string obtained both by deleting from *x* and deleting from *y*

Example: LCS

- x = *abcde* ; y = *bcduve*
- Turn x into y by deleting *a*, then inserting *u* and *v* after *d*
  - Edit distance = 3
- Or, LCS(x,y) = *bcde*
- Note that $d(x,y) = |x| + |y| - 2|LCS(x,y)|$

$$= 5 + 6 - 2*4 = 3$$

Edit Distance is a Metric

- d(x,x) = 0 because 0 edits suffice
- d(x,y) = d(y,x) because insert/delete are inverses of each other
- d(x,y) $\geq$ 0: no notion of negative edits
- Triangle inequality: changing *x* to *z* and then to *y* is one way to change *x* to *y*

Variants of the Edit Distance

- Allow insert, delete, and *mutate*
  - Change one character into another
- Minimum number of inserts, deletes, and mutates also forms a distance measure
- Ditto for any set of operations on strings
  - Example: substring reversal OK for DNA sequences

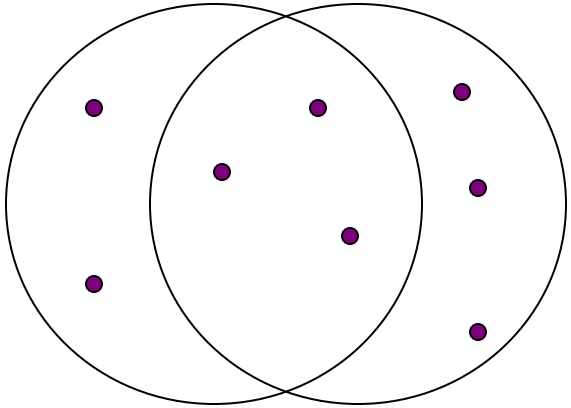
Hamming Distance

- *Hamming distance* is the number of positions in which bit-vectors differ
- Example: x = 10101; y = 10011
  d(x, y) = 2 because the bit-vectors x and y differ at 3rd and 4th position

Jaccard Similarity

- The *Jaccard Similarity* of two sets is the size of their intersection divided by the size of their union:
  - $Sim(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$
- The *Jaccard Distance* between sets is 1 minus their Jaccard similarity:
  - $d(C_1, C_2) = 1 - |C_1 \cap C_2| / |C_1 \cup C_2|$

Example: Jaccard Distance



3 in intersection
8 in union
Jaccard similarity= 3/8
Jaccard distance = 5/8

Finding Similar Items

The best techniques depend on whether you are looking for items that are very similar or only somewhat similar

We'll cover the "somewhat" case first, then talk about "very"

Problem: Finding Similar Documents

- Goal: Common text, not common topic
- Special cases are easy:
  - Identical documents
  - Pairs where one document is completely contained in another
- General case is hard:
  - Many small pieces of one doc can appear out of order in another

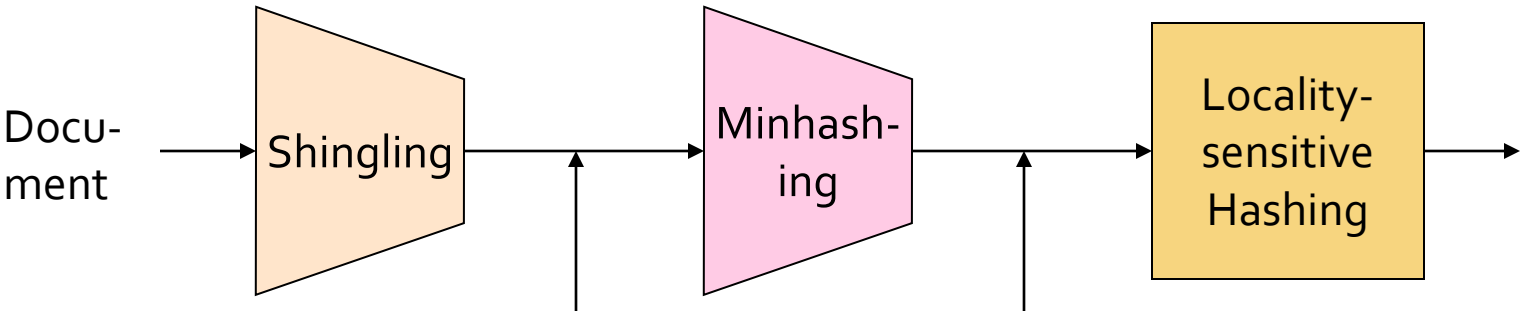
Problem Statement

- Goal: Given a large number (N in the millions or even billions) of text documents, find pairs that are "near duplicates"
- Applications:
  - Mirror websites, or approximate mirrors
    - Don't want to show both in a search
  - Plagiarism, including large quotations
  - Web spam detection
  - Similar news articles at many news sites
    - Cluster articles by "same story"

3 Essential Steps for Similar Docs

1. *Shingling*: convert documents, emails, etc., to sets
2. *Minhashing*: convert large sets to short signatures, while preserving similarity.
3. *Locality-sensitive hashing*: focus on pairs of signatures likely to be similar

The Big Picture



Candidate pairs : those pairs of signatures that we need to test for similarity.

The set of strings of length *k* that appear in the doc-ument

Signatures : short integer vectors that represent the sets, and reflect their similarity

Documents as High-Dim Data

- Simple approaches:
  - Document = set of words appearing in doc
  - Document = set of "important" words
  - Don't work well for this application. Why?
- Need to account for ordering of words
- A different way: Shingles

Shingles

- A *k*-shingle (or *k*-gram) for a document is a sequence of *k* tokens that appears in the document
  - Tokens can be characters, words or something else, depending on application
  - Assume tokens = characters for examples
- Example: k=2; D_1= abcab
  Set of 2-shingles: $S(D_1)$={ab, bc, ca}
  - Option: Shingles as a bag, count ab twice
- Represent a doc by a set of its *k*-shingles

Working Assumption

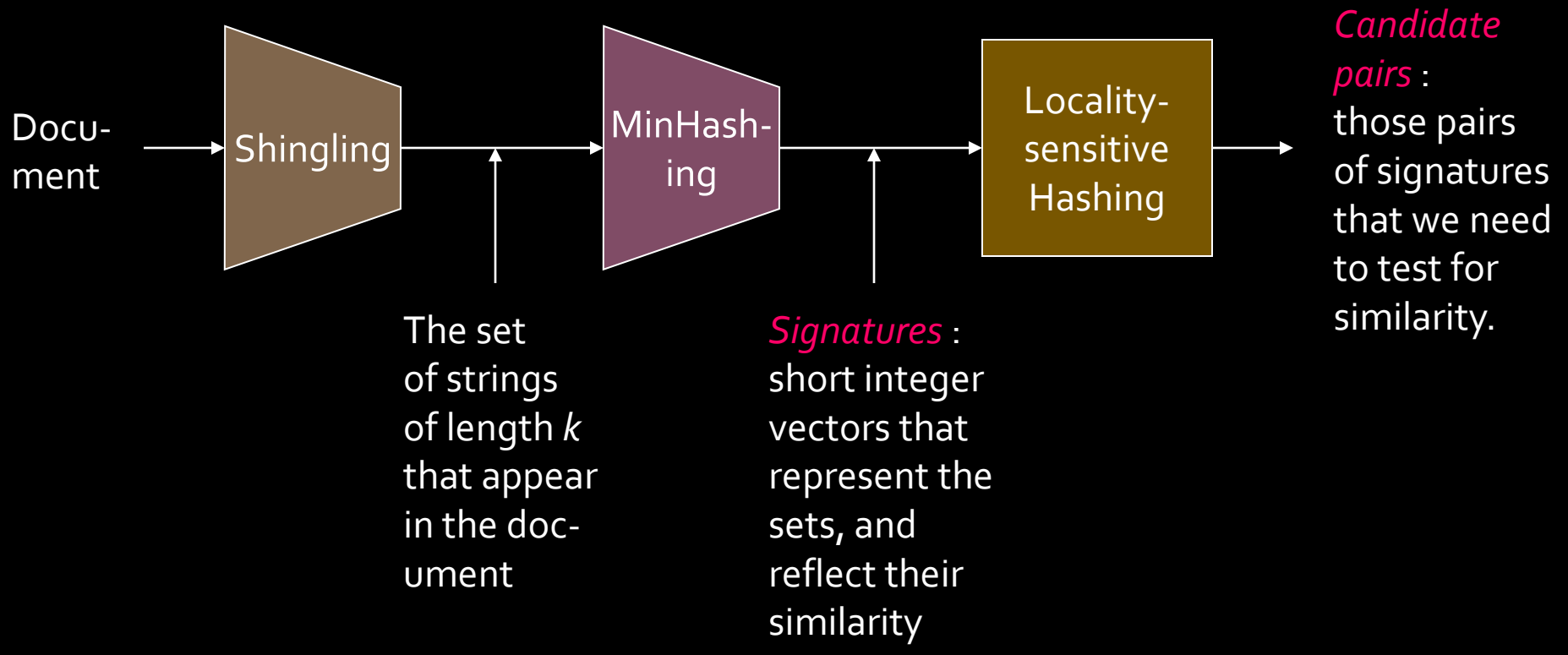
- Documents that have lots of shingles in common have similar text, even if the text appears in different order
- Careful: You must pick *k* large enough, or most documents will have most shingles
  - *k* = 5 is OK for short documents
  - *k* = 10 is better for long documents

Compressing Shingles

- To compress long shingles,
  we can hash them to (say) 4 bytes
- Represent a doc by the set of hash values of its *k*-shingles
- Idea: Two documents could (rarely) appear to have shingles in common, when in fact only the hash-values were shared

Thought Question

- Why is it better to hash 9-shingles (say) to 4 bytes than to use 4-shingles?
- Hint: How random are the 32-bit sequences that result from 4-shingling?


Docu-
ment
Shingling
MinHash-
ing
Locality-
sensitive
Hashing
Candidate
pairs :
those pairs
of signatures
that we need
to test for
similarity.
The set
of strings
of length k
that appear
in the doc-
ument
Signatures :
short integer
vectors that
represent the
sets, and
reflect their
similarity

MinHashing

Similarity Metric for Shingles

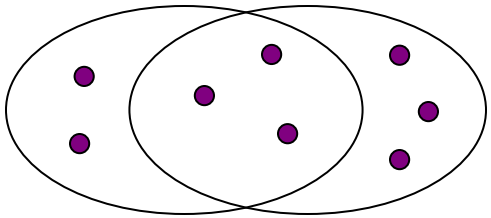
- Document D_1 = set of k-shingles $C_1=S(D_1)$
- Equivalently, each document is a 0/1 vector in the space of k-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- A natural similarity measure is the Jaccard similarity:

$$Sim(D_1, D_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$

Motivation for Minhash/LSH

- Suppose we need to find near-duplicate documents among N=1 million documents
- Naïvely, we'd have to compute pairwaise Jaccard similarites for every pair of docs
  - i.e, $N(N-1)/2 \approx 5*10^{11}$ comparisons
  - At 10^5 secs/day and 10^6 comparisons/sec, it would take 5 days
- For N = 10 million, it takes more than a year…

Encoding sets as bit vectors



- Many similarity problems can be formalized as finding subsets of some universal set that have significant intersection
- We can encode sets using 0/1 (bit, boolean) vectors
  - One dimension per element in the universal set
- Interpret set intersection as bitwise AND, and set union as bitwise OR

- Example: C_1 = 10111; C_2 = 10011
  - Size of intersection = 3; size of union = 4, Jaccard similarity (not distance) = 3/4
  - $d(C_1,C_2)$ = 1 – (Jaccard similarity) = 1/4

From Sets to Boolean Matrices

- Rows = elements of the universal set
- Columns = sets

- 1 in row *e* and column *s* if and only if *e* is a member of *s*

- Column similarity is the Jaccard similarity of the sets of their rows with 1

- Typical matrix is sparse

Example: Jaccard of Columns

- Each document is a column:

shingles

1	0	1	0
1	1	0	1
0	1	0	1
0	0	0	1
0	0	0	1
1	1	1	0
1	0	1	0

documents

$Sim(C_1,C_2)= 2/5 = 0.4$

Note:

- We might not really represent the data by a boolean matrix
- Sparse matrices are usually better represented by the list of places where there is a non-zero value

When Is Similarity Interesting?

1. When the sets are so large or so many that they cannot fit in main memory
2. Or, when there are so many sets that comparing all pairs of sets takes too much time
3. Or both

Outline: Finding Similar Columns

1. Compute signatures of columns = small summaries of columns

2. Examine pairs of signatures to find similar signatures
   - Essential: Similarities of signatures and columns are related

3. Optional: Check that columns with similar signatures are really similar.

Warnings

1. Comparing all pairs of signatures may take too much time, even if not too much space
   - A job for Locality-Sensitive Hashing
2. These methods can produce false negatives, and even false positives (if the optional check is not made)

Hashing Columns (Singatures)

- Key idea: "hash" each column *C* to a small *signature* $h(C)$, such that:
  1. $h(C)$ is small enough that we can fit a signature in main memory for each column
  2. $Sim(C_1, C_2)$ is the same as the "similarity" of $h(C_1)$ and $h(C_2)$
- Goal: Find a hash function *h* such that:
  - if $Sim(C_1,C_2)$ is high, then with high prob. $h(C_1) = h(C_2)$
  - if $Sim(C_1,C_2)$ is low, then with high prob. $h(C_1) \neq h(C_2)$
- Hash docs into buckets, and expect that "most" pairs of near duplicate docs hash into the same bucket

Min-hashing

- Clearly, the hash function depends on the similarity metric
  - Not all similarity metrics have a suitable hash function
- There is a suitable hash function for Jaccard similarity
  - **Min-hashing**

Minhashing

- Imagine the rows of the boolean matrix permuted under random permutation π
- Define "hash" function $h_\pi(C)$ = the number of the first (in the permuted order π) row in which column C has 1:

$$h_\pi(C)=\min \pi(C)$$

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

Minhashing Example

Permutation π

1	4	3
3	2	4
7	1	7
6	3	6
2	6	1
5	7	2
4	5	5

Input matrix

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

Signature matrix *M*

2	1	2	1
2	1	4	1
1	2	1	2

Surprising Property

- Choose a random permutation π
- Prob. that $h_\pi(C_1) = h_\pi(C_2)$ is the same as $Sim(C_1, C_2)$:

  $\mathbf{Pr}[h_\pi(C_1) = h_\pi(C_2)] = Sim(C_1, C_2)$
- Why?
  - Let X be a set of shingles, $X \subseteq [2^{64}]$, $x \in X$
  - Then: $\Pr[\pi(x) = \min(\pi(X))] = 1/|X|$
    - It is equally likely that any $x \in X$ is mapped to the min element
  - Let x be s.t. $\pi(x) = \min(\pi(C_1 \cup C_2))$
  - Then either: $\pi(x) = \min(\pi(C_1))$ if $x \in C_1$, or

    $\pi(x) = \min(\pi(C_2))$ if $x \in C_2$
  - So the prob. that both are true is the prob. $x \in C_1 \cap C_2$
  - $\Pr[\min(\pi(C_1))=\min(\pi(C_2))]=|C_1 \cap C_2|/|C_1 \cup C_2| = Sim(C_1, C_2)$

Four Types of Rows

- Given cols C_1 and C_2, rows may be classified as:

	C_1	C_2
a	1	1
b	1	0
c	0	1
d	0	0

- Also, a = # rows of type a , etc.
- Note $Sim(C_1, C_2) = a/(a + b + c)$
- **Then: Pr**$[h(C_1) = h(C_2)] = Sim(C_1, C_2)$
  - Look down the cols C_1 and C_2 until we see a 1
  - If it's a type-*a* row, then $h(C_1) = h(C_2)$
    If a type-*b* or type-*c* row, then not

Similarity for Signatures

- The *similarity of two signatures* is the fraction of the hash functions in which they agree

Min Hashing – Example

1	4	3
3	2	4
7	1	7
6	3	6
2	6	1
5	7	2
4	5	5

Input matrix

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

Signature matrix *M*

2	1	2	1
2	1	4	1
1	2	1	2

Similarities:

	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0
Sig/Sig	0.67	1.00	0	0

Minhash Signatures

- Pick (say) 100 random permutations of the rows
- Think of *Sig*(C) as a column vector
- Let *Sig*(C)[i] =

  according to the *i*-th permutation, the index of the first row that has a 1 in column *C*
- Note: We store the sketch of document C in ~100 bytes:

  $Sig(C)[i] = \min(\pi_i(C))$

Implementation – (1)

- Suppose 1 billion rows
- Hard to pick a random permutation from 1…billion
- Representing a random permutation requires 1 billion entries
- Accessing rows in permuted order leads to thrashing

Implementation – (2)

- A good approximation to permuting rows: pick 100 (?) hash functions
- For each column c and each hash function h_i, keep a "slot" $M(i, c)$
- Intent: $M(i, c)$ 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 permuation

Implementation – (3)

for each row r

  for each column c

    if c has 1 in row r

      for each hash function h_i **do**

        if $h_i(r)$ is a smaller value than $M(i, c)$ **then**

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

Example

Row	C1	C2
1	1	0
2	0	1
3	1	1
4	1	0
5	0	1

$h(x) = x \bmod 5$
$g(x) = 2x+1 \bmod 5$

	Sig1	Sig2
$h(1) = 1$	1	-
$g(1) = 3$	3	-
$h(2) = 2$	1	2
$g(2) = 0$	3	0
$h(3) = 3$	1	2
$g(3) = 2$	2	0
$h(4) = 4$	1	2
$g(4) = 4$	2	0
$h(5) = 0$	1	0
$g(5) = 1$	2	0

Implementation – (4)

- Often, data is given by column, not row
  - E.g., columns = documents, rows = shingles
- If so, sort matrix once so it is by row
- And *always* compute $h_i(r)$ only once for each row.