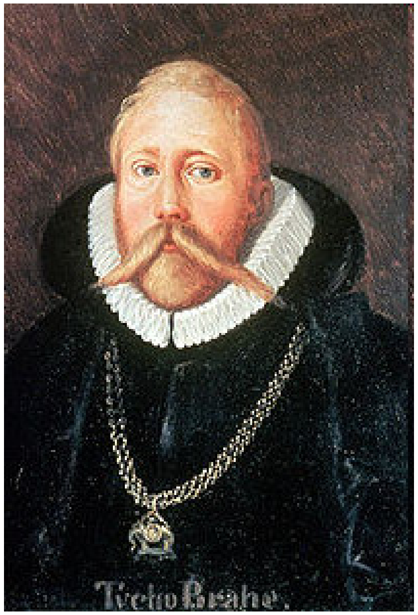
Near Neighbor Search in High Dimensional Data (1)

Motivation
Distance Measures
Shingling
Min-Hashing

Anand Rajaraman

Tycho Brahe



RUDOLPHINÆ,

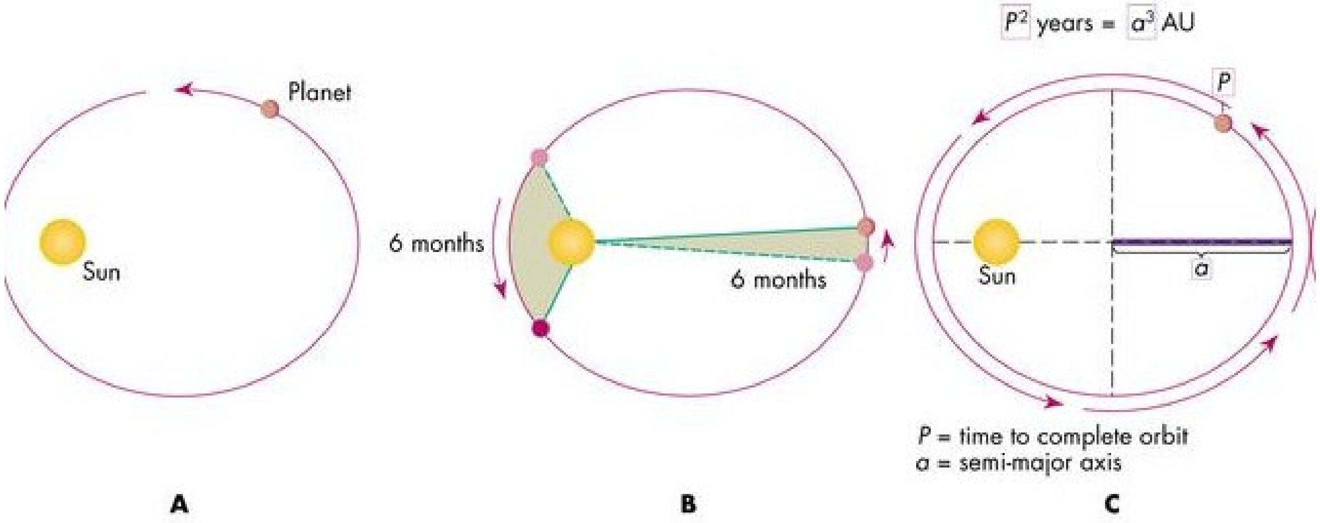
TYCHONE,

PRIMUM ANIMO CONCEPTA ET DESTINA-

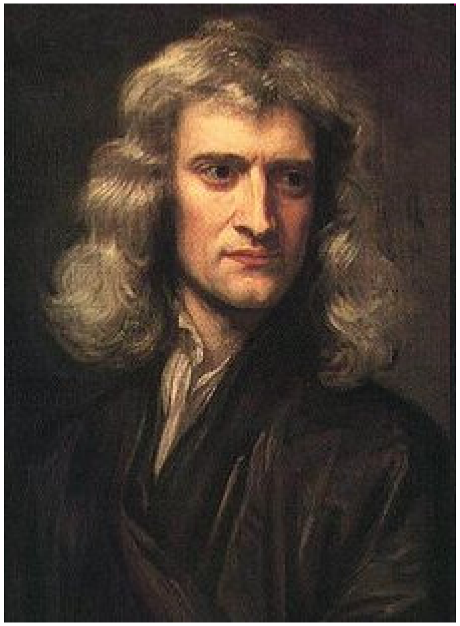
RUDOLPHI, MATTHIÆ, FERDINANDI,

IOANNES KEPLERUS,

Johannes Kepler



… and Isaac Newton



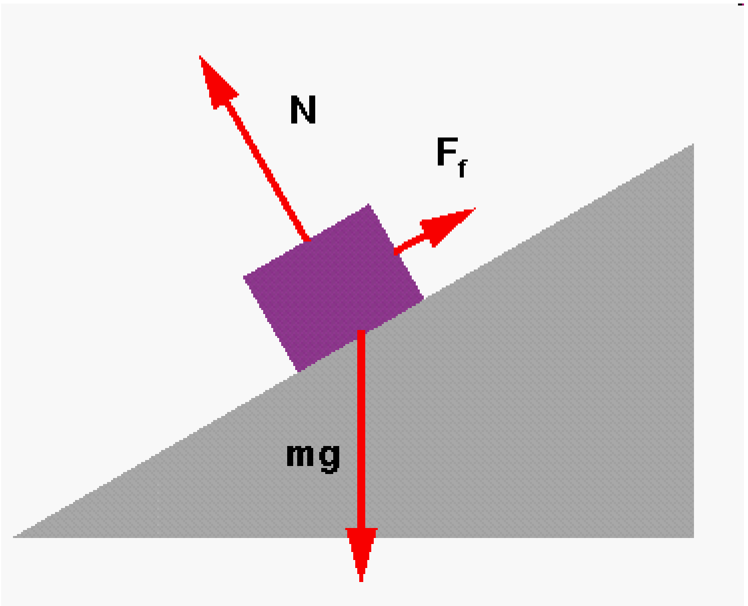
Newton's Law of Universal Gravitation

$$\vec{F} = \frac{-GMm\hat{r}}{r^2}$$

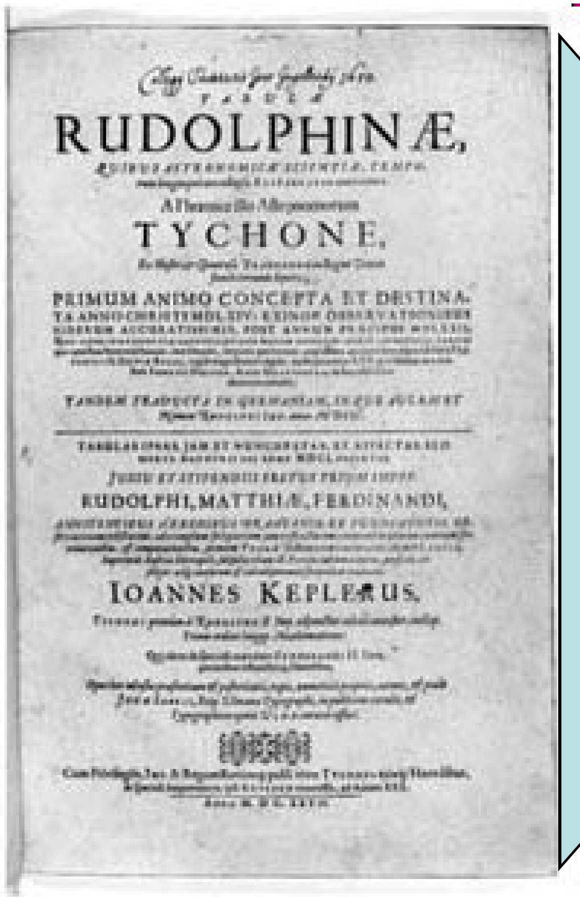
Newton's 2nd Law

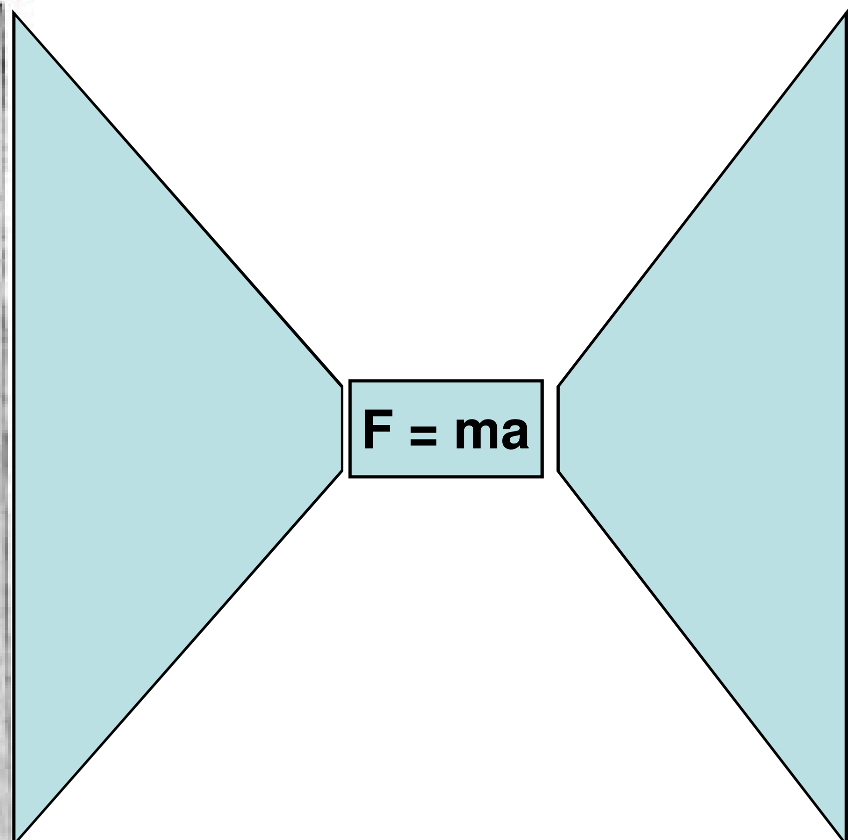
$$\vec{F} = d/dt(m\vec{v})$$

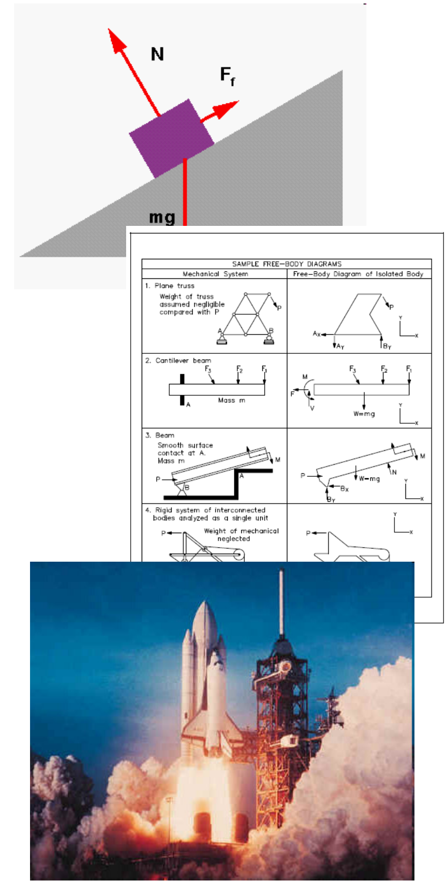
Figure 11.0



The Classical Model


TABULÆ
RUDOLPHINÆ,
TYCHONE,
PRIMUM ANIMO CONCEPTA ET DESTINA-
RUDOLPHI, MATTHIÆ, FERDINANDI,
IOANNES KEPLERUS,



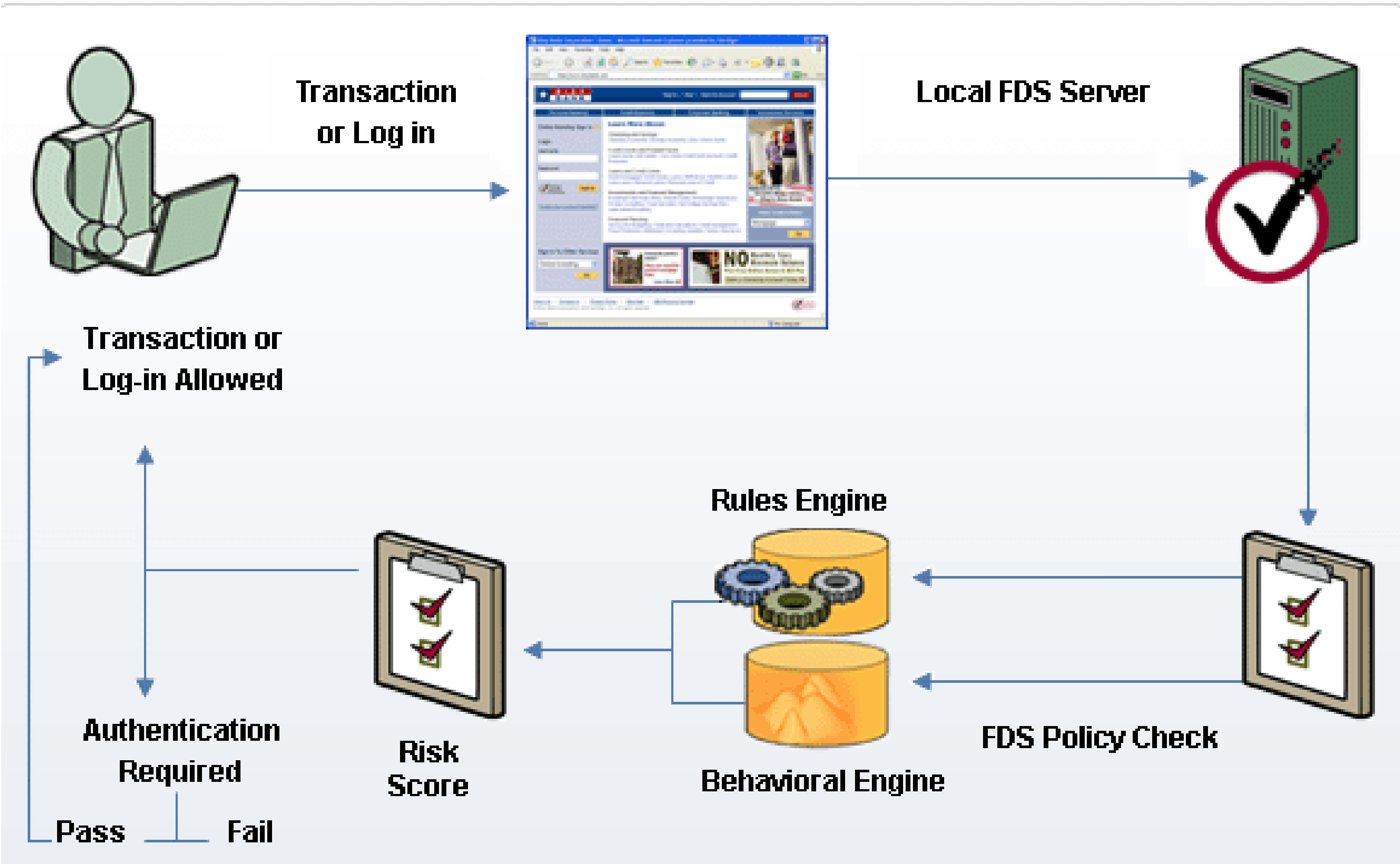


Data

Theory

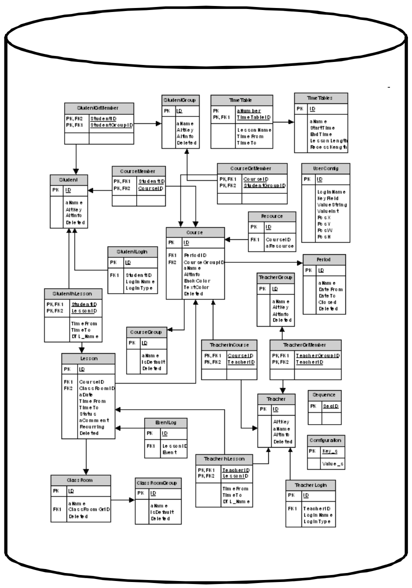
Applications

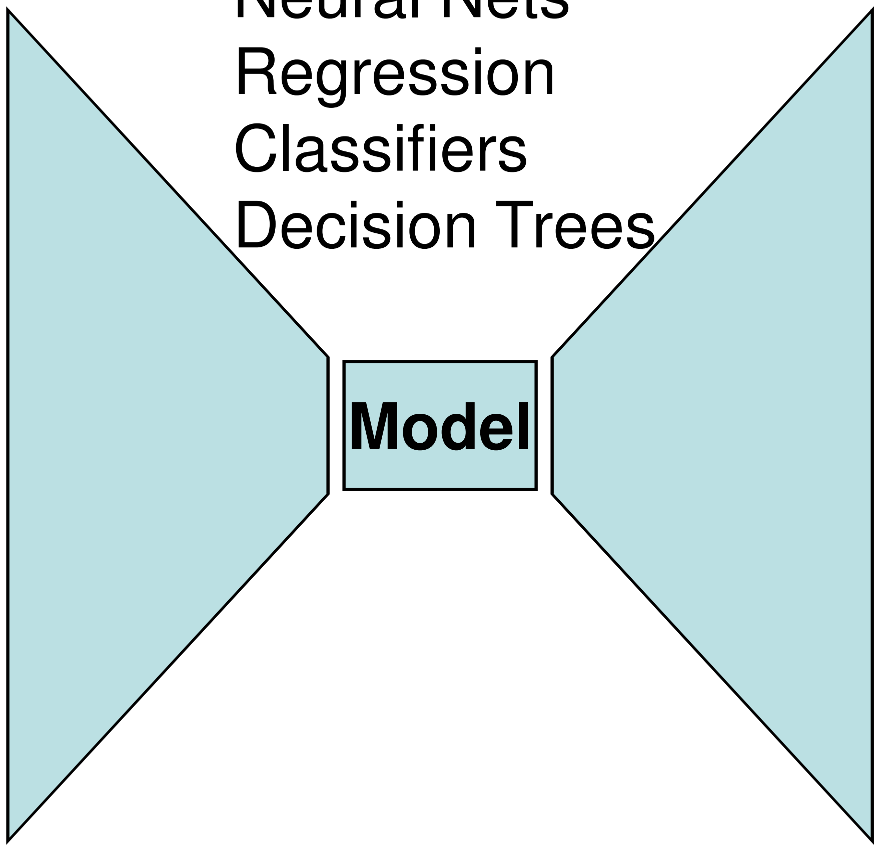
Fraud Detection

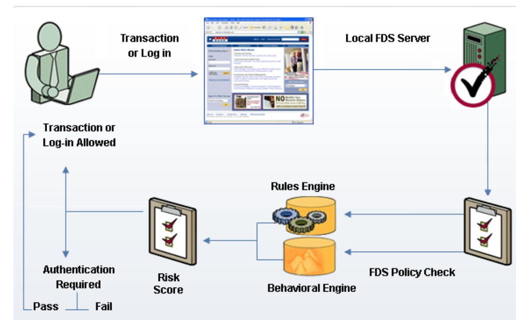


Model-based decision making

Neural Nets
Regression
Classifiers
Decision Trees

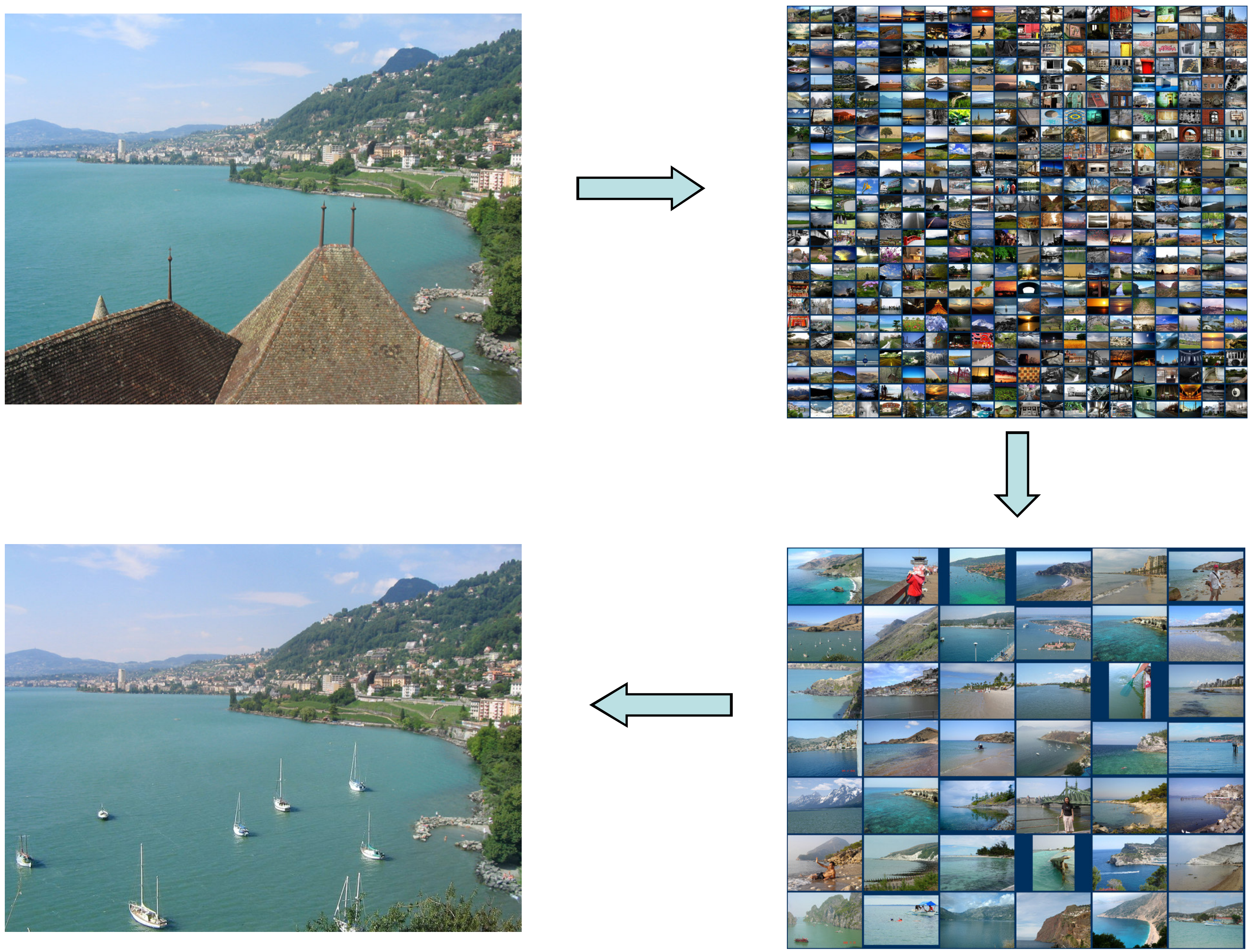






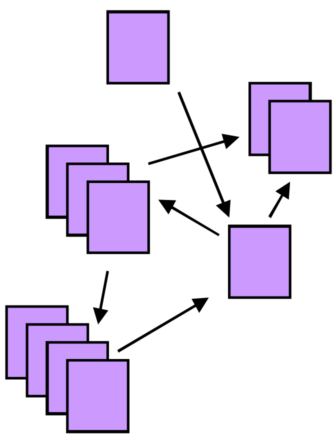
Data **Model** **Predictions**

Scene Completion Problem



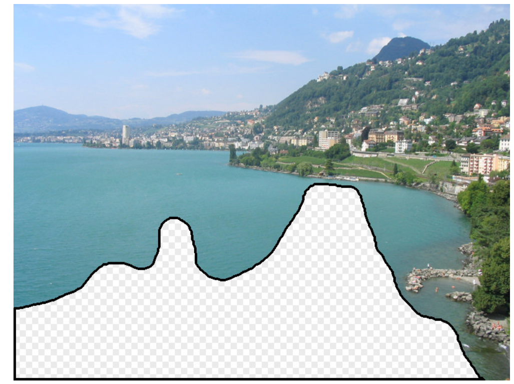
Hays and Efros, SIGGRAPH 2007

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

- Find near-neighbors in high-D space
  - documents closely matching query terms
  - customers who purchased similar products
  - 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 result from attributes of near-neighbors

Example: Question Answering

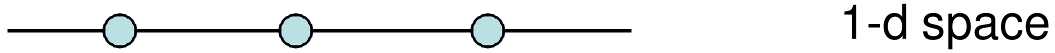
- Who killed Abraham Lincoln?
- What is the height of Mount Everest?
- Naïve algorithm
  - Find all web pages containing the terms “killed” and “Abraham Lincoln” in close proximity
  - Extract k-grams from a small window around the terms
  - Find the most commonly occuring k-grams

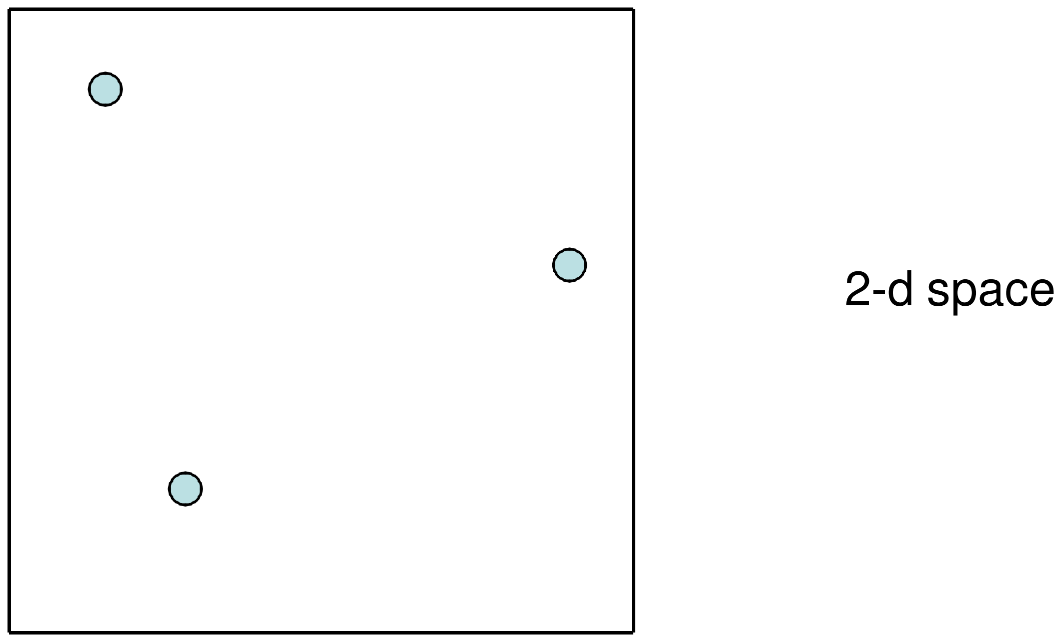
Example: Question Answering

- 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





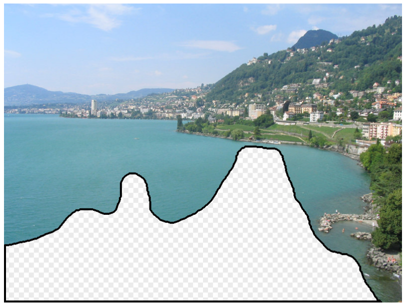
The Curse of Dimensionality

- 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!

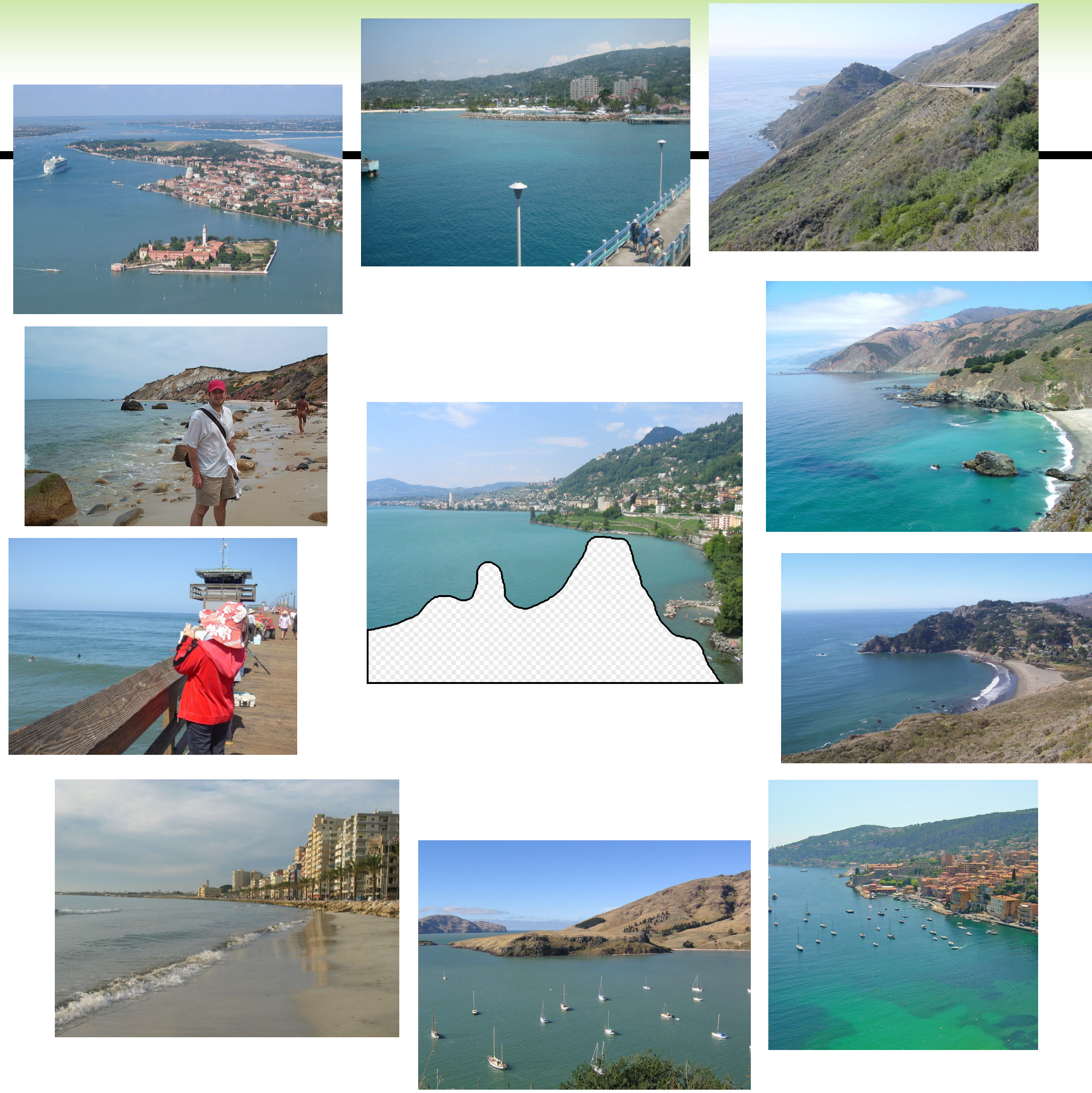
Example: Scene Completion



Hays and Efros, SIGGRAPH 2007

10 nearest neighbors from a collection of 20,000 images

Hays and Efros, SIGGRAPH 2007



10 nearest neighbors from a collection of 2 million images

Hays and Efros, SIGGRAPH 2007

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.

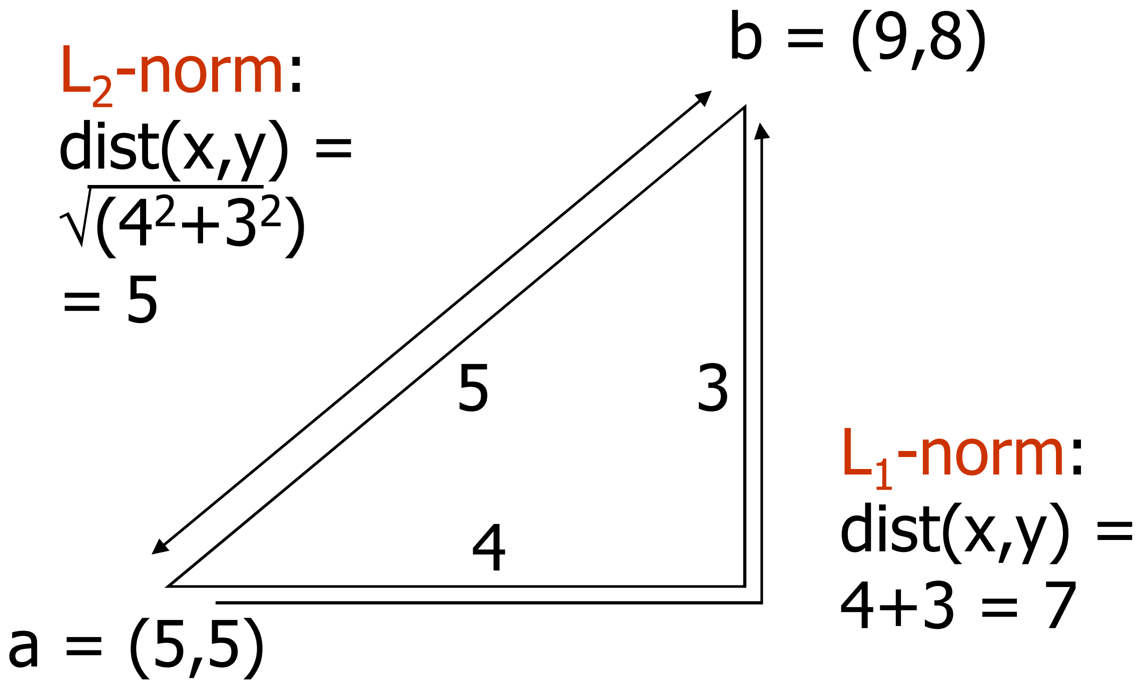
Axioms of a Distance Measure

- *d* is a *distance measure* if it is a function from pairs of points to real numbers such that:
  1. $d(x,y) \geq 0$.
  2. $d(x,y) = 0$ iff $x = y$.
  3. $d(x,y) = d(y,x)$.
  4. $d(x,y) \leq d(x,z) + d(z,y)$ (*triangle inequality*).

Some Euclidean Distances

- *L_2 norm* : d(x,y) = square root of the sum of the squares of the differences between *x* and *y* in each dimension.
  - The most common notion of "distance."
- *L_1 norm* : sum of the differences in each dimension.
  - *Manhattan distance* = distance if you had to travel along coordinates only.

Examples of Euclidean Distances



Another Euclidean Distance

- *L_∞ norm* : d(x,y) = the maximum of the differences between *x* and *y* in any dimension.
- Note: the maximum is the limit as *n* goes to ∞ of the L_n norm

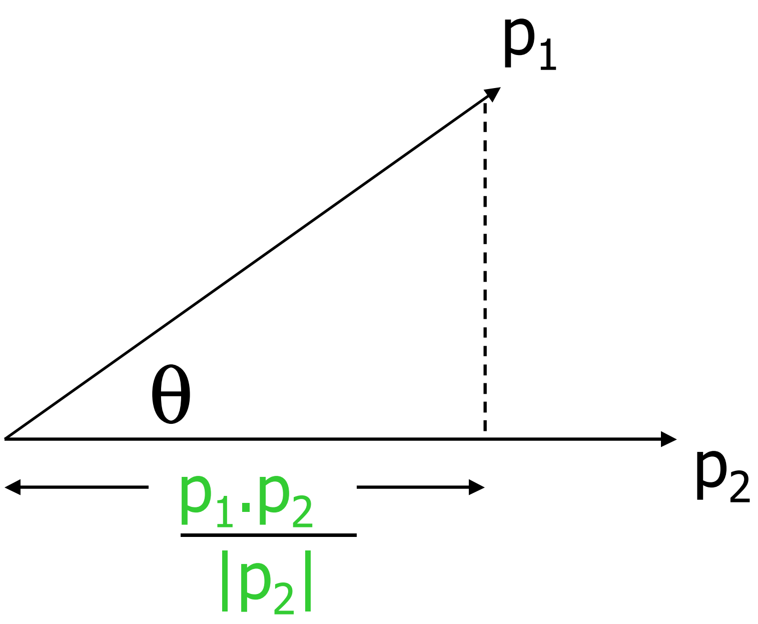
Non-Euclidean Distances

- *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

- 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: $p_1.p_2/|p_2||p_1|$.
  - Example: p_1 = 00111; p_2 = 10011.
  - $p_1.p_2 = 2$; $|p_1| = |p_2| = \sqrt{3}$.
  - $\cos(\theta) = 2/3$; θ is about 48 degrees.

Cosine-Measure Diagram



$d\ (p_1, p_2) = \theta = \arccos(p_1.p_2/|p_2||p_1|)$

Why C.D. Is a Distance Measure

- 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: physical reasoning. If I rotate an angle from *x* to *z* and then from *z* to *y*, I 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 Distance Measure

- 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*.

Variant Edit Distances

- 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

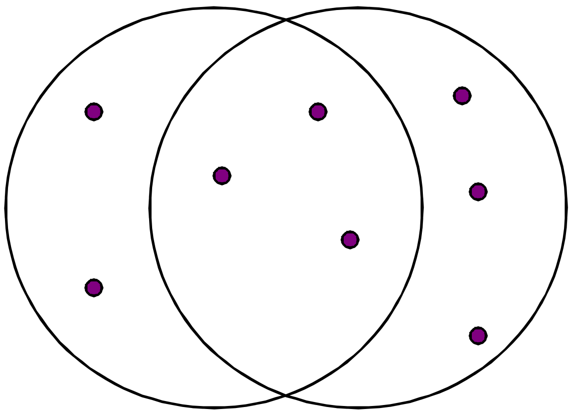
Hamming Distance

- *Hamming distance* is the number of positions in which bit-vectors differ.
- Example: $p_1 = 10101$; $p_2 = 10011$.
- $d(p_1, p_2) = 2$ because the bit-vectors differ in the 3rd and 4th positions.

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

Encoding sets as bit vectors

- 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: p_1 = 10111; p_2 = 10011.
- Size of intersection = 3; size of union = 4, Jaccard similarity (not distance) = 3/4.
- d(x,y) = 1 – (Jaccard similarity) = 1/4.

Finding Similar Documents

- Locality-Sensitive Hashing (LSH) is a general method to find near-neighbors in high-dimensional data
- We'll introduce LSH by considering a specific case: finding similar text documents
  - Also introduces additional techniques: shingling, minhashing
- Then we'll discuss the generalized theory behind LSH

Problem Statement

- 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."

Near Duplicate Documents

- 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
- We first need to formally define "near duplicates"

Documents as High Dimensional 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; doc = abcab. Set of 2-shingles = {ab, bc, ca}.
  - Option: shingles as a bag, count ab twice.
- Represent a doc by its set of *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.
- 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?

Similarity metric

- Document = set of k-shingles
- 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
  - $Sim\,(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$

Motivation for LSH

- Suppose we need to find near-duplicate documents among N=1 million documents
- Naively, we'd have to compute pairwise 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…

Key idea behind LSH

- Given documents (i.e., shingle sets) D1 and D2
- If we can find a hash function *h* such that:
  - if *sim*(D1,D2) is high, then with high probability *h*(D1) = *h*(D2)
  - if *sim*(D1,D2) is low, then with high probability *h*(D1) ≠ *h*(D2)
- Then we could hash documents into buckets, and expect that "most" pairs of near duplicate documents would hash into the same bucket
  - Compare pairs of docs in each bucket to see if they are really near-duplicates

Min-hashing

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

The shingle matrix

- Matrix where each document vector is a column

documents

shingles

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

Min-hashing

- Define a hash function *h* as follows:
  - Permute the rows of the matrix randomly
    - Important: same permutation for all the vectors!
  - Let *C* be a column (= a document)
  - *h*(*C*) = the number of the first (in the permuted order) row in which column *C* has 1

Minhashing Example

Input matrix

3	1	0	1	0
4	1	0	0	1
7	0	1	0	1
6	0	1	0	1
1	0	1	0	1
2	1	0	1	0
5	1	0	1	0

h →

2	1	2	1

Surprising Property

- The probability (over all permutations of the rows) that $h(C_1) = h(C_2)$ is the same as $Sim(C_1, C_2)$
- That is:
  - $\mathbf{Pr}[h(C_1) = h(C_2)] = Sim(C_1, C_2)$

- Let's prove it!

Proof (1) : Four Types of Rows

- Given columns 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 $Sim(C_1, C_2) = a/(a + b + c)$.

Proof (2): The Clincher

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

- Now apply a permutation
  - Look down the permuted columns 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.
  - So $\mathbf{Pr}[h(C_1) = h(C_2)] = a/(a + b + c) = Sim(C_1, C_2)$

LSH: First Cut

- Hash each document using min-hashing
- Each pair of documents that hashes into the same bucket is a candidate pair
- Assume we want to find pairs with similarity at least 0.8.
  - We'll miss 20% of the real near-duplicates
  - Many false-positive candidate pairs
    - e.g., We'll find 60% of pairs with similarity 0.6.

Minhash Signatures

- Fixup: Use several (e.g., 100) independent min-hash functions to create a signature Sig(C) for each column C
- The *similarity of signatures* is the fraction of the hash functions in which they agree.
- Because of the minhash property, the similarity of columns is the same as the expected similarity of their signatures.

Minhash Signatures 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

Implementation (1)

- Suppose N = 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
  - $h_1, h_2, \ldots$
  - For rows *r* and *s*, if $h_i(r) < h_i(s)$, then *r* appears before *s* in permutation *i*.
  - We will use the same name for the hash function and the corresponding min-hash function

Example

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

$h(x) = x \bmod 5$

$h(1)=1$, $h(2)=2$, $h(3)=3$, $h(4)=4$, $h(5)=0$

$h(C1) = 1$

$h(C2) = 0$

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

$g(1)=3$, $g(2)=0$, $g(3)=2$, $g(4)=4$, $g(5)=1$

$g(C1) = 2$

$g(C2) = 0$

$Sig(C1) = [1,2]$

$Sig(C2) = [0,0]$

Implementation (3)

- For each column *c* and each hash function h_i, keep a "slot" $M(i, c)$.
  - $M(i, c)$ will become the smallest value of $h_i(r)$ for which column *c* has 1 in row *r*
  - Initialize to infinity
- Sort the input matrix so it is ordered by rows
  - So can iterate by reading rows sequentially from disk

Implementation (4)

for each row r

for each column c

if c has 1 in row r

for each hash function h_i **do**

if $h_i(r) < M(i, c)$ **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.
  - This way we compute $h_i(r)$ only once for each row
- Questions for thought:
  - What's a good way to generate hundreds of independent hash functions?
  - How to implement min-hashing using MapReduce?

The Big Picture