Search Engine Basics
- Web Search Engine Elements
    - Spider: Builds Corpus (Collects web pages recursively)
    - Indexer: Creates Inverted Indexes
    - Query Processor: Servers query results

Characterizing the Web
- Can Measure the web by
    - Number of web sites
    - Languages of Web Pages
    - Rate of Change of Pages
    - Document Content Type
    - Linkage
    - Graph
    - Content
    - Statistics

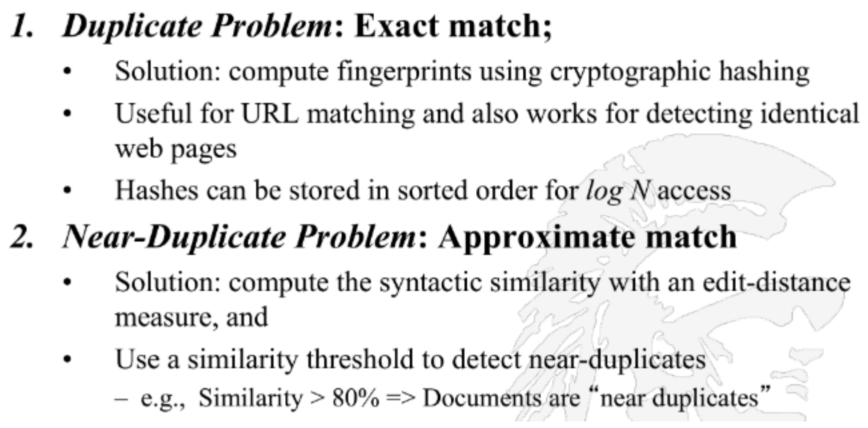
Search Engine Evaluation
- Precision and Recall
    - Precision: # of Relevant Items / # of all Retrieved Items
    - Recall: # of Relevant items retrieved / # of all relevant items
    - Precision: True Positive / (True Positive + False Positive)
    - Recall = True Positive / (True Positive + False Negative)
    - Accuracy: (True Positive + True Negative) / (True Positive + False Positive + False Negative + True Negative)
- F score and MAP and DCG
    - F-score: 2RP/(R+P)
    - Mean Average Precision: Average of P(q)/Q
    - Discounts Cumulative Gain
- Google's Process
    1. Precision Evaluation
    2. Side by Side Experiments
    3. Live traffic Experiments
    4. Full Launch
- A/B Testing: 2 Versions of the page to see which does better
- User clicks for evaluation
    - # of clicks per link
    - User Click Sequence

Web Crawling
- Robots.txt
- BFS vs DFS
- LIFO vs FIFO
- Avoid Duplicates
- Normalize URLS
    - Convert scheme and host to lower case
    - Cap letters in escape sequences

    - Decode Parent-Encode Octets of unreserved characters
    - Remove default port
- Spider traps
- Spam Web Pages
- DNS Caching, Pre-fetching, Multithreading
- Distrubuted Crawling

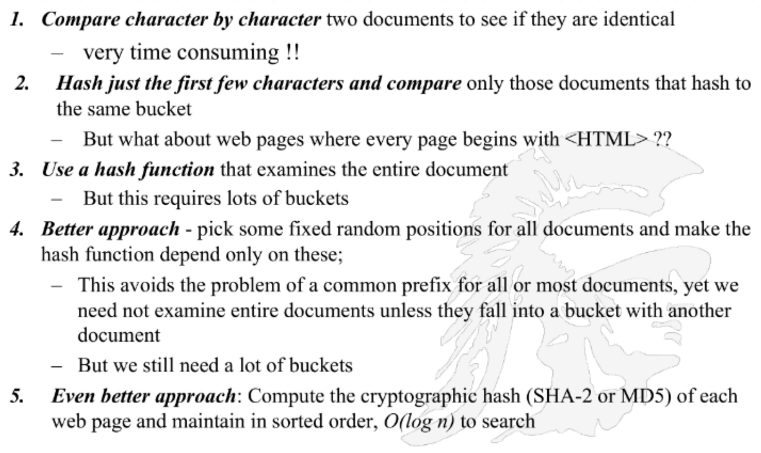
DeDuplication

- Same page, different links or near duplicate page (Mirroring)
- Why worry about duplicates
    - Redundnacy if bad link
    - Reduce crawl time
    - Better connectivity analysis
- Near Duplicates
    - Spam Detection
    - Plagiarism
    - Clustering (Related articles)



1. ***Duplicate Problem*: Exact match;**
   - Solution: compute fingerprints using cryptographic hashing
   - Useful for URL matching and also works for detecting identical web pages
   - Hashes can be stored in sorted order for *log N* access
2. ***Near-Duplicate Problem*: Approximate match**
   - Solution: compute the syntactic similarity with an edit-distance measure, and
   - Use a similarity threshold to detect near-duplicates
     – e.g., Similarity > 80% => Documents are "near duplicates"

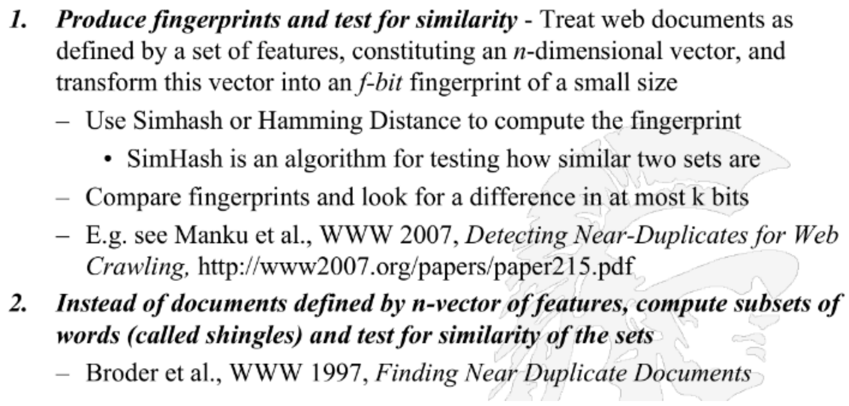
- Hash Function
    - Fast and easy
    - Small change to text yields completely different hash
    - Ex: MD-5, Sha-1, Sha-2
    - Identifying Identicals



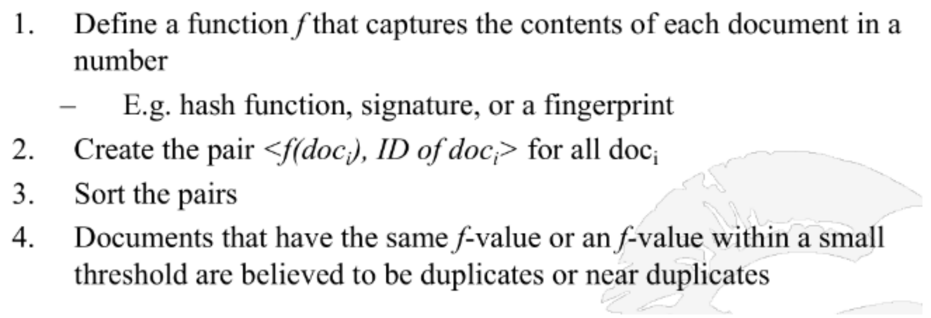
1. ***Compare character by character*** two documents to see if they are identical
   – very time consuming !!
2. ***Hash just the first few characters and compare*** only those documents that hash to the same bucket
   – But what about web pages where every page begins with <HTML> ??
3. ***Use a hash function*** that examines the entire document
   – But this requires lots of buckets
4. ***Better approach*** - pick some fixed random positions for all documents and make the hash function depend only on these;
   – This avoids the problem of a common prefix for all or most documents, yet we need not examine entire documents unless they fall into a bucket with another document
   – But we still need a lot of buckets
5. ***Even better approach***: Compute the cryptographic hash (SHA-2 or MD5) of each web page and maintain in sorted order, *O(log n)* to search

    -

- Identifying Near Identicals

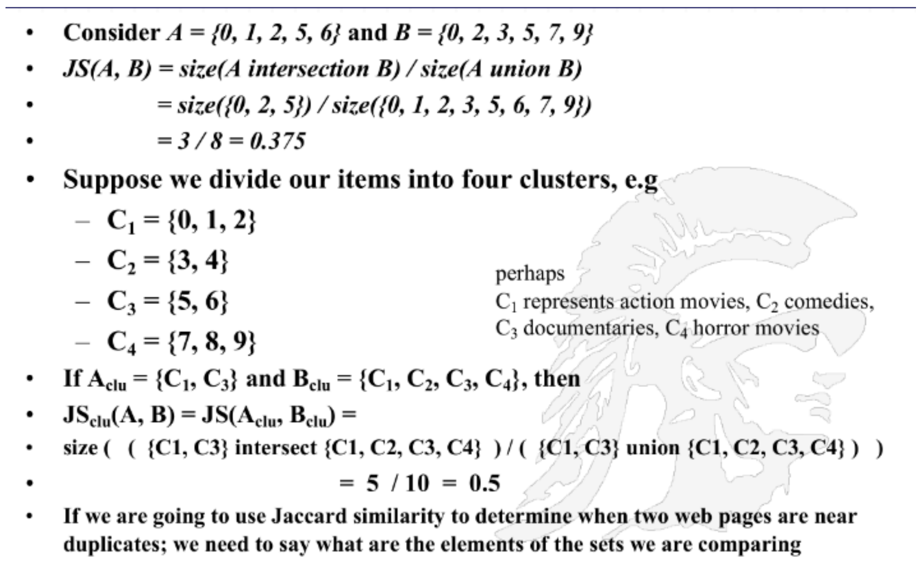


1. ***Produce fingerprints and test for similarity*** - Treat web documents as defined by a set of features, constituting an *n*-dimensional vector, and transform this vector into an *f-bit* fingerprint of a small size
   - Use Simhash or Hamming Distance to compute the fingerprint
     - SimHash is an algorithm for testing how similar two sets are
   - Compare fingerprints and look for a difference in at most k bits
   - E.g. see Manku et al., WWW 2007, *Detecting Near-Duplicates for Web Crawling,* http://www2007.org/papers/paper215.pdf
2. ***Instead of documents defined by n-vector of features, compute subsets of words (called shingles) and test for similarity of the sets***
   - Broder et al., WWW 1997, *Finding Near Duplicate Documents*



1. Define a function *f* that captures the contents of each document in a number
   - E.g. hash function, signature, or a fingerprint
2. Create the pair <$f(doc_i)$, *ID of* doc_i> for all doc_i
3. Sort the pairs
4. Documents that have the same *f*-value or an *f*-value within a small threshold are believed to be duplicates or near duplicates

- Jaccard Similarity

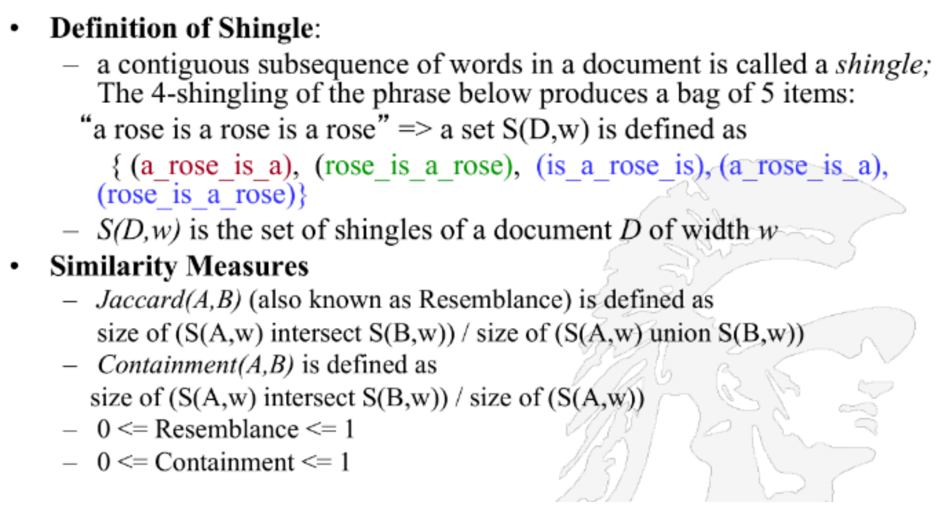


- **Consider *A = {0, 1, 2, 5, 6}* and *B = {0, 2, 3, 5, 7, 9}***
- ***JS(A, B) = size(A intersection B) / size(A union B)***
- ***= size({0, 2, 5}) / size({0, 1, 2, 3, 5, 6, 7, 9})***
- ***= 3 / 8 = 0.375***
- **Suppose we divide our items into four clusters, e.g**
  - **C_1 = {0, 1, 2}**
  - **C_2 = {3, 4}**
  - **C_3 = {5, 6}**
  - **C_4 = {7, 8, 9}**

perhaps
C_1 represents action movies, C_2 comedies, C_3 documentaries, C_4 horror movies

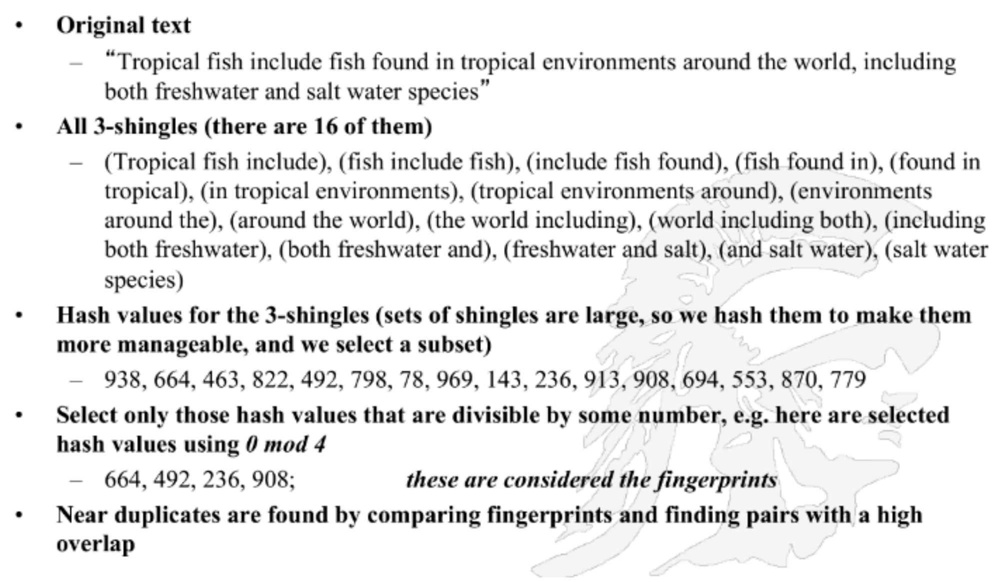
- **If A_{clu} = {C_1, C_3} and B_{clu} = {C_1, C_2, C_3, C_4}, then**
- **JS_{clu}(A, B) = JS(A_{clu}, B_{clu}) =**
- **size (({C1, C3} intersect {C1, C2, C3, C4}) / ({C1, C3} union {C1, C2, C3, C4}))**
- **= 5 / 10 = 0.5**
- **If we are going to use Jaccard similarity to determine when two web pages are near duplicates; we need to say what are the elements of the sets we are comparing**

-

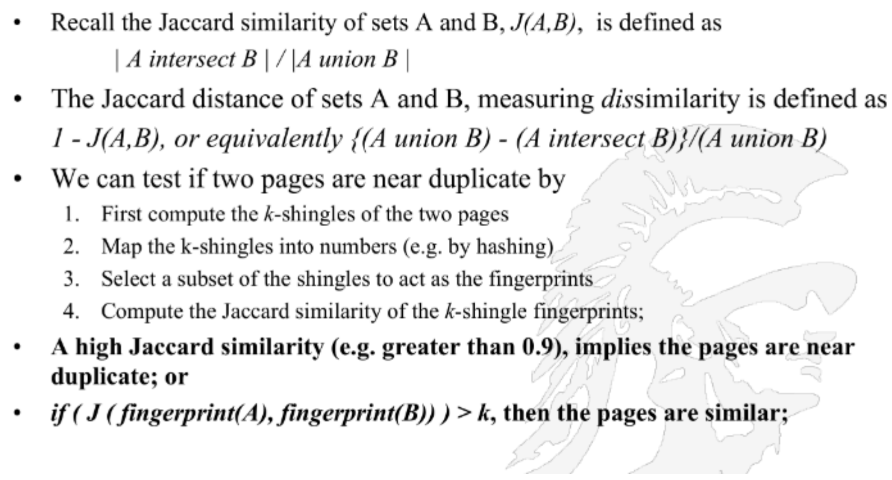


- **Definition of Shingle**:
  - a contiguous subsequence of words in a document is called a *shingle;* The 4-shingling of the phrase below produces a bag of 5 items:
    "a rose is a rose is a rose" => a set S(D,w) is defined as
    { (a_rose_is_a), (rose_is_a_rose), (is_a_rose_is), (a_rose_is_a), (rose_is_a_rose)}
  - *S(D,w)* is the set of shingles of a document *D* of width *w*
- **Similarity Measures**
  - *Jaccard(A,B)* (also known as Resemblance) is defined as
    size of (S(A,w) intersect S(B,w)) / size of (S(A,w) union S(B,w))
  - *Containment(A,B)* is defined as
    size of (S(A,w) intersect S(B,w)) / size of (S(A,w))
  - 0 <= Resemblance <= 1
  - 0 <= Containment <= 1

- Can make many choices with singles (Spaces or no spaces, How large k should be, etc.)

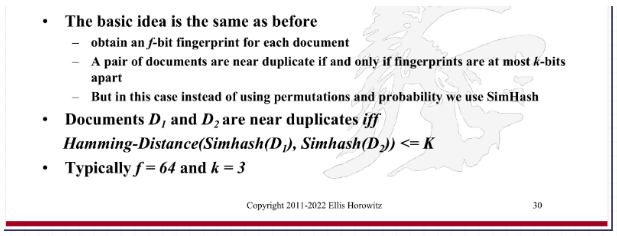


- **Original text**
  - "Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species"
- **All 3-shingles (there are 16 of them)**
  - (Tropical fish include), (fish include fish), (include fish found), (fish found in), (found in tropical), (in tropical environments), (tropical environments around), (environments around the), (around the world), (the world including), (world including both), (including both freshwater), (both freshwater and), (freshwater and salt), (and salt water), (salt water species)
- **Hash values for the 3-shingles (sets of shingles are large, so we hash them to make them more manageable, and we select a subset)**
  - 938, 664, 463, 822, 492, 798, 78, 969, 143, 236, 913, 908, 694, 553, 870, 779
- **Select only those hash values that are divisible by some number, e.g. here are selected hash values using *0 mod 4***
  - 664, 492, 236, 908; ***these are considered the fingerprints***
- **Near duplicates are found by comparing fingerprints and finding pairs with a high overlap**



- Recall the Jaccard similarity of sets A and B, *J(A,B)*, is defined as
  | *A intersect B* | / |*A union B* |
- The Jaccard distance of sets A and B, measuring *dis*similarity is defined as
  1 - J(A,B), or equivalently {(A union B) - (A intersect B)}/(A union B)
- We can test if two pages are near duplicate by
  1. First compute the *k*-shingles of the two pages
  2. Map the k-shingles into numbers (e.g. by hashing)
  3. Select a subset of the shingles to act as the fingerprints
  4. Compute the Jaccard similarity of the *k*-shingle fingerprints;
- **A high Jaccard similarity (e.g. greater than 0.9), implies the pages are near duplicate; or**
- ***if (J (fingerprint(A), fingerprint(B))) > k*, then the pages are similar;**

Simhash

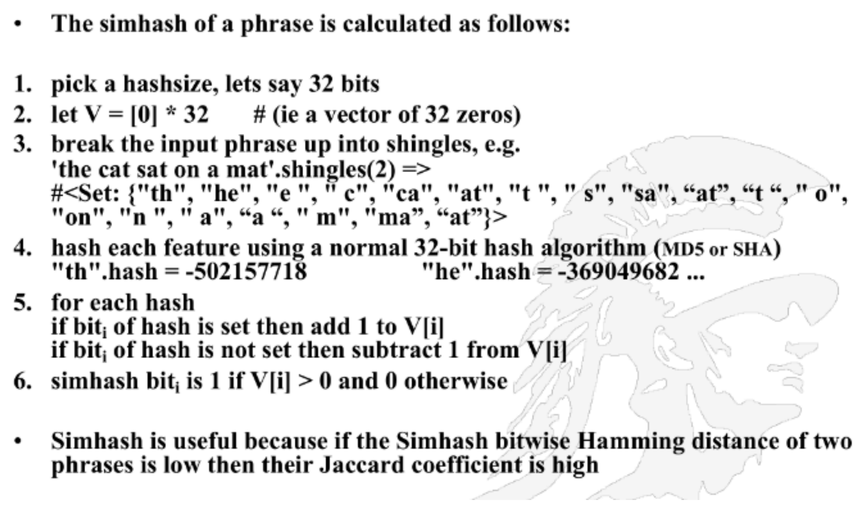


- The basic idea is the same as before
  - obtain an *f*-bit fingerprint for each document
  - A pair of documents are near duplicate if and only if fingerprints are at most *k*-bits apart
  - But in this case instead of using permutations and probability we use SimHash
- Documents D_1 and D_2 are near duplicates *iff* *Hamming-Distance(Simhash(D_1), Simhash(D_2))* <= *K*
- Typically $f = 64$ and $k = 3$

Copyright 2011-2022 Ellis Horowitz 30

SimHash (aka Charikar Similarity) is essentially a dimension reduction technique - it maps a set of weighted features (contents of a document) to a low dimensional fingerprint, eg. a 64-bit word.

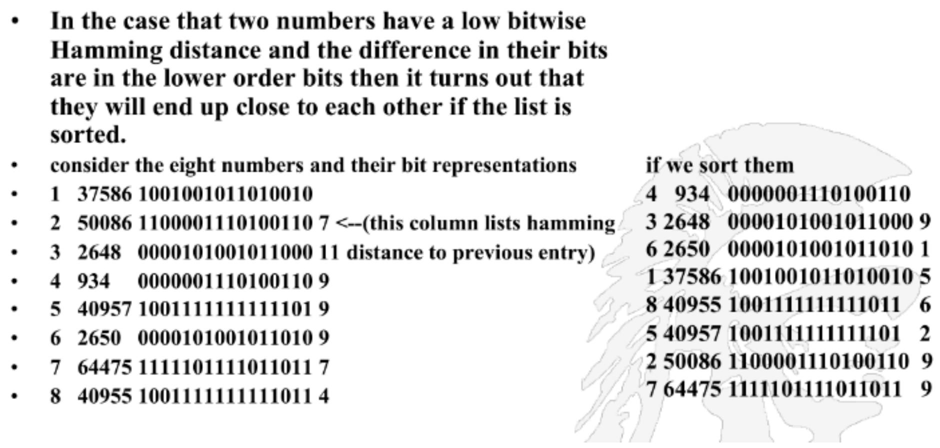
And, **documents that are nearly identical have nearly similar fingerprints that differ only in a small # of bits.** In other words, similar inputs lead to similar outputs (hash values), hence 'Sim'Hash; other hashing techniques, eg. MD5, do not have this property (in other words, even a tiny change in the input leads to a huge change in the output). This similarity property is what makes SimHash, an excellent tool for similarity detection of documents.



- The simhash of a phrase is calculated as follows:

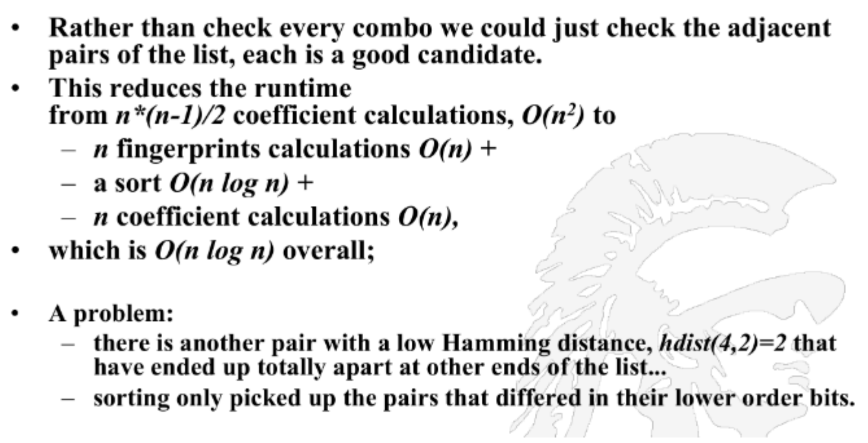
1. pick a hashsize, lets say 32 bits
2. let V = [0] * 32   # (ie a vector of 32 zeros)
3. break the input phrase up into shingles, e.g.
   'the cat sat on a mat'.shingles(2) =>
   #<Set: {"th", "he", "e ", " c", "ca", "at", "t ", " s", "sa", "at", "t ", " o", "on", "n ", " a", "a ", " m", "ma", "at"}>
4. hash each feature using a normal 32-bit hash algorithm (MD5 or SHA)
   "th".hash = -502157718   "he".hash = -369049682 ...
5. for each hash
   if bit$_i$ of hash is set then add 1 to V[i]
   if bit$_i$ of hash is not set then subtract 1 from V[i]
6. simhash bit$_i$ is 1 if V[i] > 0 and 0 otherwise

- Simhash is useful because if the Simhash bitwise Hamming distance of two phrases is low then their Jaccard coefficient is high



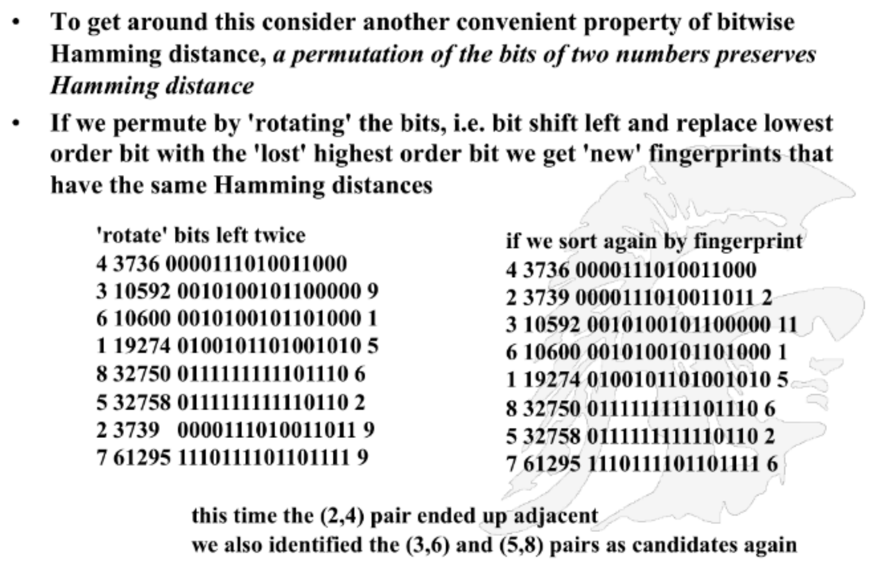
- In the case that two numbers have a low bitwise Hamming distance and the difference in their bits are in the lower order bits then it turns out that they will end up close to each other if the list is sorted.

consider the eight numbers and their bit representations	if we sort them
1 37586 1001001011010010	4 934 0000001110100110
2 50086 1100001110100110 7 <--(this column lists hamming	3 2648 0000101001011000 9
3 2648 0000101001011000 11 distance to previous entry)	6 2650 0000101001011010 1
4 934 0000001110100110 9	1 37586 1001001011010010 5
5 40957 1001111111111101 9	8 40955 1001111111111011 6
6 2650 0000101001011010 9	5 40957 1001111111111101 2
7 64475 1111101111011011 7	2 50086 1100001110100110 9
8 40955 1001111111111011 4	7 64475 1111101111011011 9



- Rather than check every combo we could just check the adjacent pairs of the list, each is a good candidate.
- This reduces the runtime from *n*(n-1)/2* coefficient calculations, $O(n^2)$ to
  - *n* fingerprints calculations $O(n)$ +
  - a sort $O(n \log n)$ +
  - *n* coefficient calculations $O(n)$,
- which is $O(n \log n)$ overall;

- A problem:
  - there is another pair with a low Hamming distance, *hdist(4,2)=2* that have ended up totally apart at other ends of the list...
  - sorting only picked up the pairs that differed in their lower order bits.



- To get around this consider another convenient property of bitwise Hamming distance, *a permutation of the bits of two numbers preserves Hamming distance*
- If we permute by 'rotating' the bits, i.e. bit shift left and replace lowest order bit with the 'lost' highest order bit we get 'new' fingerprints that have the same Hamming distances

'rotate' bits left twice

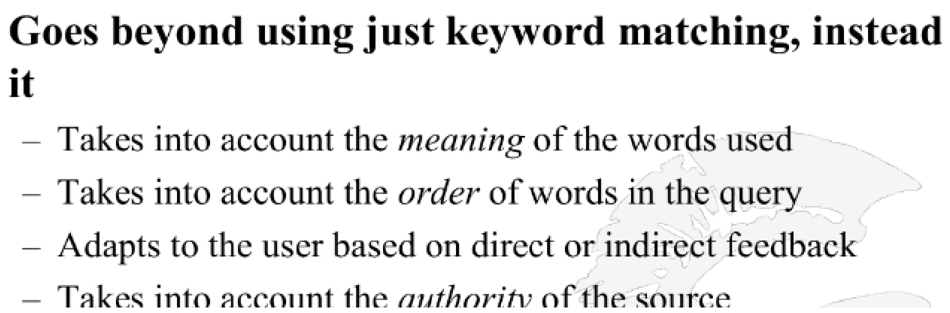
4	3736	0000111010011000	
3	10592	0010100101100000	9
6	10600	0010100101101000	1
1	19274	0100101101001010	5
8	32750	0111111111101110	6
5	32758	0111111111110110	2
2	3739	0000111010011011	9
7	61295	1110111101101111	9

if we sort again by fingerprint

4	3736	0000111010011000	
2	3739	0000111010011011	2
3	10592	0010100101100000	11
6	10600	0010100101101000	1
1	19274	0100101101001010	5
8	32750	0111111111101110	6
5	32758	0111111111110110	2
7	61295	1110111101101111	6

this time the (2,4) pair ended up adjacent
we also identified the (3,6) and (5,8) pairs as candidates again

Intro to IR

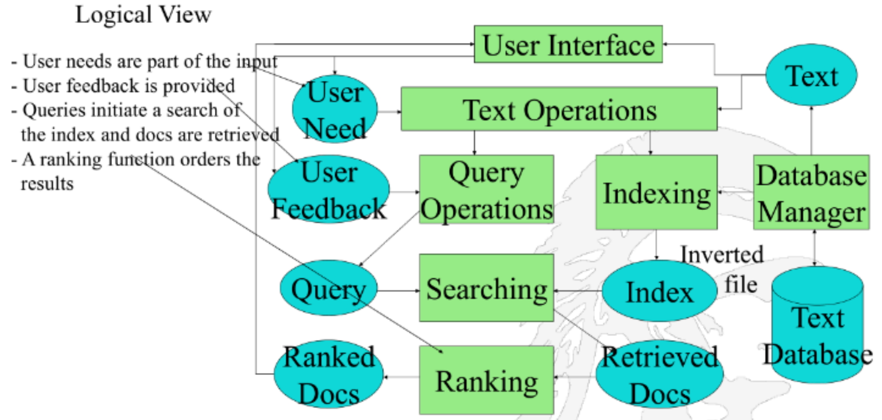
- 

Goes beyond using just keyword matching, instead it

- Takes into account the *meaning* of the words used
- Takes into account the *order* of words in the query
- Adapts to the user based on direct or indirect feedback
- Takes into account the *authority* of the source

Logical View

- User needs are part of the input
- User feedback is provided
- Queries initiate a search of the index and docs are retrieved
- A ranking function orders the results



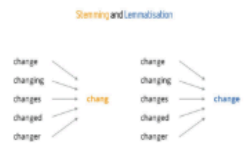
- Start with a text database; it is indexed; a user interface permits query operations which cause a search on the Index; matched documents are retrieved and ranked

- **Three major Information Retrieval Models are:**

1. Boolean models (set theoretic) (*Chapter 1 in Manning et al*)
2. Vector space models (statistical/algebraic) (*Chapter 2 in Manning et al*)
3. Probabilistic models (*Chapter 11 in Manning et al*)

- Preprocessing

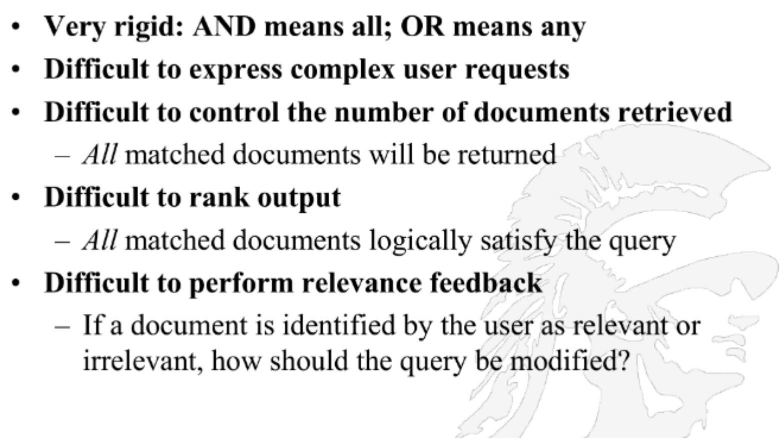
    1. **Strip unwanted characters/markup (e.g. HTML tags, punctuation, page numbers, etc.).**
    2. **Break into tokens (keywords) separating out whitespace.**
    3. **Stem tokens to "root" words**



    4. **Remove common stopwords (e.g. a, the, it, etc.).**
    5. **Detect common phrases (possibly using a domain specific dictionary).**
    6. **Build inverted index (keyword → list of docs containing it).**

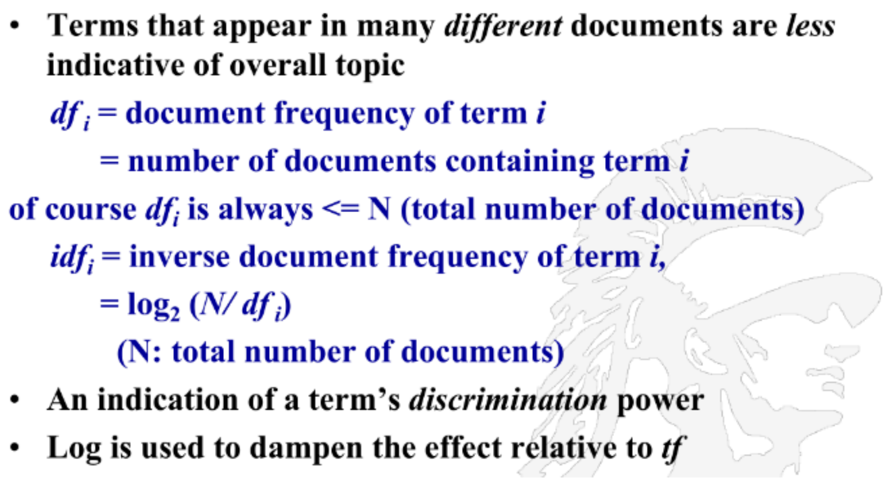
-

- Boolean Retrieval Model
    - Fast and Clean



- **Very rigid: AND means all; OR means any**
- **Difficult to express complex user requests**
- **Difficult to control the number of documents retrieved**
  - *All* matched documents will be returned
- **Difficult to rank output**
  - *All* matched documents logically satisfy the query
- **Difficult to perform relevance feedback**
  - If a document is identified by the user as relevant or irrelevant, how should the query be modified?

-
- Vector Space Model
  - Term Weight Frequency



- **Terms that appear in many *different* documents are *less* indicative of overall topic**

df_i = document frequency of term i

= number of documents containing term i

of course df_i is always <= N (total number of documents)

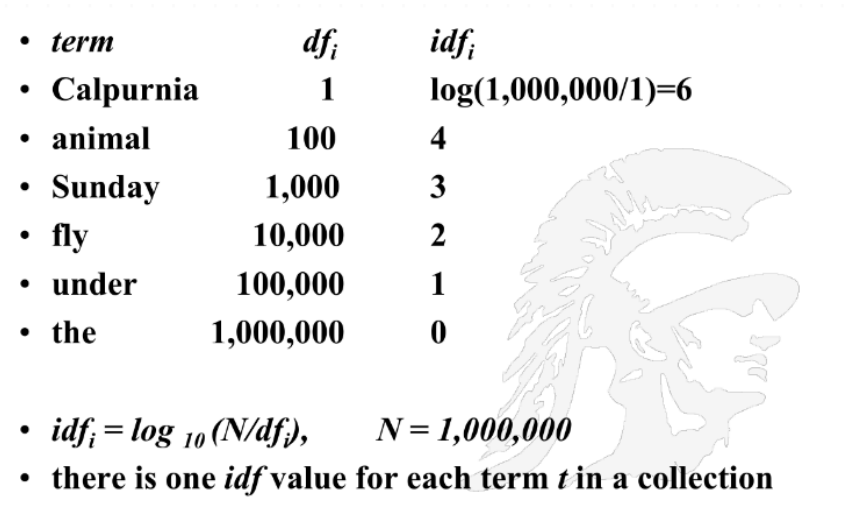
idf_i = inverse document frequency of term i,

$= \log_2 (N/df_i)$

(N: total number of documents)

- **An indication of a term's *discrimination* power**
- **Log is used to dampen the effect relative to *tf***

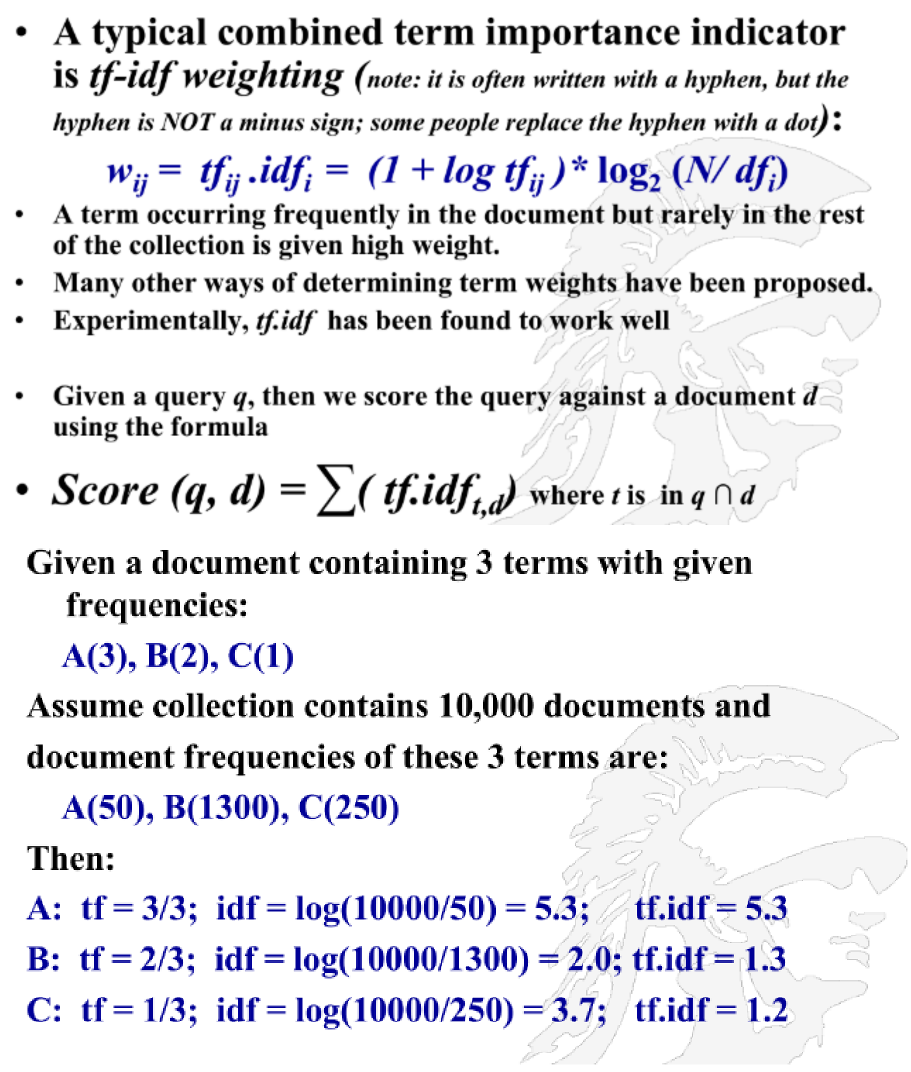
  -



• *term*	df_i	idf_i
• Calpurnia	1	log(1,000,000/1)=6
• animal	100	4
• Sunday	1,000	3
• fly	10,000	2
• under	100,000	1
• the	1,000,000	0

- $idf_i = \log_{10}(N/df_i)$, $N = 1{,}000{,}000$
- **there is one *idf* value for each term *t* in a collection**

  -
  - Tf * idf



- **A typical combined term importance indicator is *tf-idf weighting* (*note: it is often written with a hyphen, but the hyphen is NOT a minus sign; some people replace the hyphen with a dot*):**

$$w_{ij} = tf_{ij}\,.idf_i = (1 + \log tf_{ij}) * \log_2 (N/ df_i)$$

- **A term occurring frequently in the document but rarely in the rest of the collection is given high weight.**
- **Many other ways of determining term weights have been proposed.**
- **Experimentally, *tf.idf* has been found to work well**

- **Given a query *q*, then we score the query against a document *d* using the formula**
- ***Score (q, d)* = $\sum(tf.idf_{t,d})$ where *t* is in $q \cap d$**

Given a document containing 3 terms with given frequencies:

A(3), B(2), C(1)

Assume collection contains 10,000 documents and document frequencies of these 3 terms are:

A(50), B(1300), C(250)

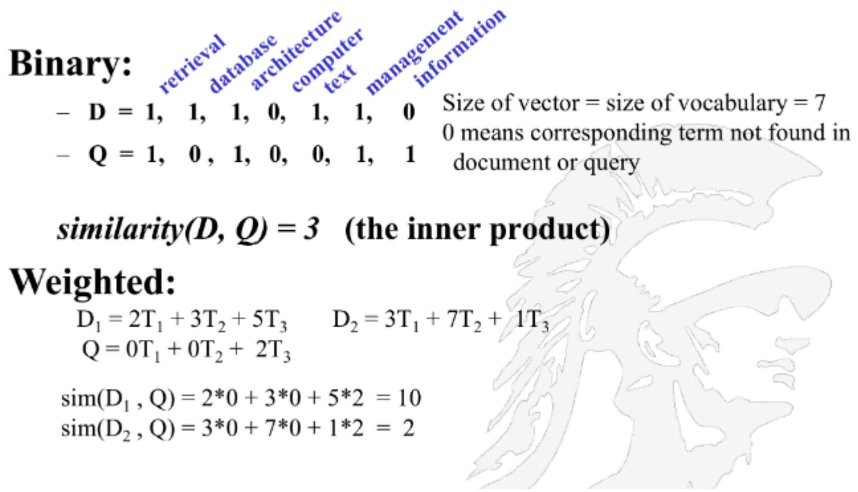
Then:

A: tf = 3/3; idf = log(10000/50) = 5.3; tf.idf = 5.3

B: tf = 2/3; idf = log(10000/1300) = 2.0; tf.idf = 1.3

C: tf = 1/3; idf = log(10000/250) = 3.7; tf.idf = 1.2

- Cosine Similarity



Binary:

	retrieval	database	architecture	computer	text	management	information
– D =	1,	1,	1,	0,	1,	1,	0
– Q =	1,	0 ,	1,	0,	0,	1,	1

Size of vector = size of vocabulary = 7
0 means corresponding term not found in document or query

similarity(D, Q) = 3 **(the inner product)**

Weighted:

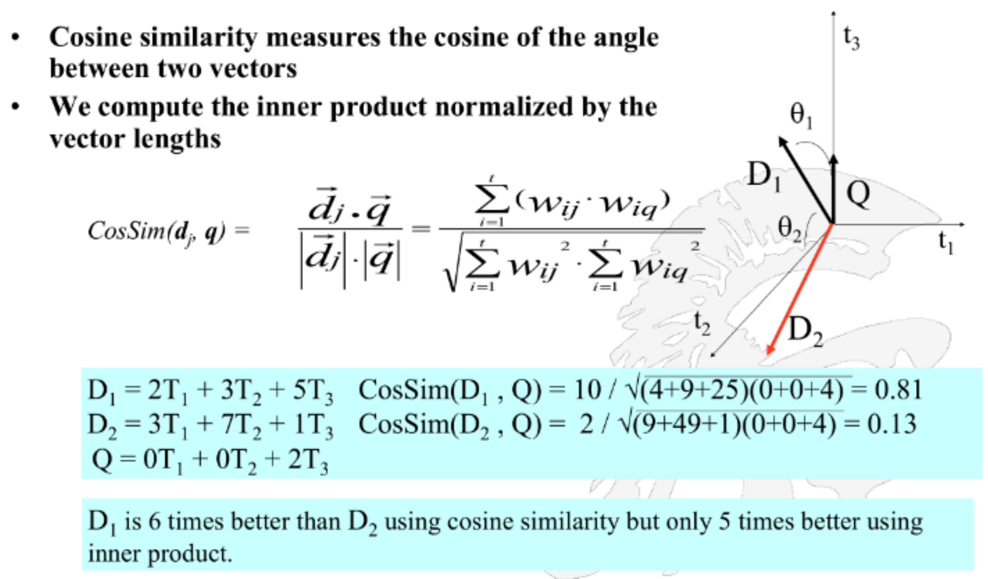
$D_1 = 2T_1 + 3T_2 + 5T_3$ $\quad D_2 = 3T_1 + 7T_2 + 1T_3$

$Q = 0T_1 + 0T_2 + 2T_3$

$\text{sim}(D_1 , Q) = 2*0 + 3*0 + 5*2 = 10$

$\text{sim}(D_2 , Q) = 3*0 + 7*0 + 1*2 = 2$

-


t3
θ1
D1
Q
θ2
t1
t2
D2

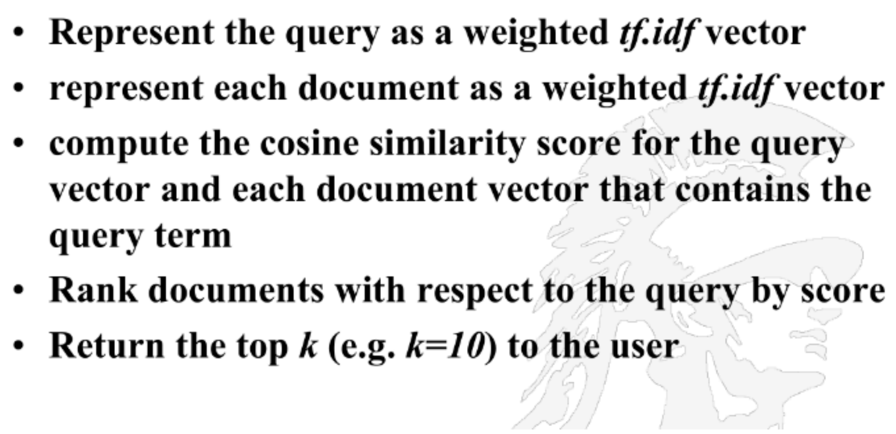
- Cosine similarity measures the cosine of the angle between two vectors
- We compute the inner product normalized by the vector lengths

$$CosSim(d_j, q) = \frac{\vec{d_j} \cdot \vec{q}}{|\vec{d_j}| \cdot |\vec{q}|} = \frac{\sum_{i=1}^{t}(w_{ij} \cdot w_{iq})}{\sqrt{\sum_{i=1}^{t} w_{ij}^2 \cdot \sum_{i=1}^{t} w_{iq}^2}}$$

$D_1 = 2T_1 + 3T_2 + 5T_3$ $\quad CosSim(D_1, Q) = 10 / \sqrt{(4+9+25)(0+0+4)} = 0.81$
$D_2 = 3T_1 + 7T_2 + 1T_3$ $\quad CosSim(D_2, Q) = 2 / \sqrt{(9+49+1)(0+0+4)} = 0.13$
$Q = 0T_1 + 0T_2 + 2T_3$

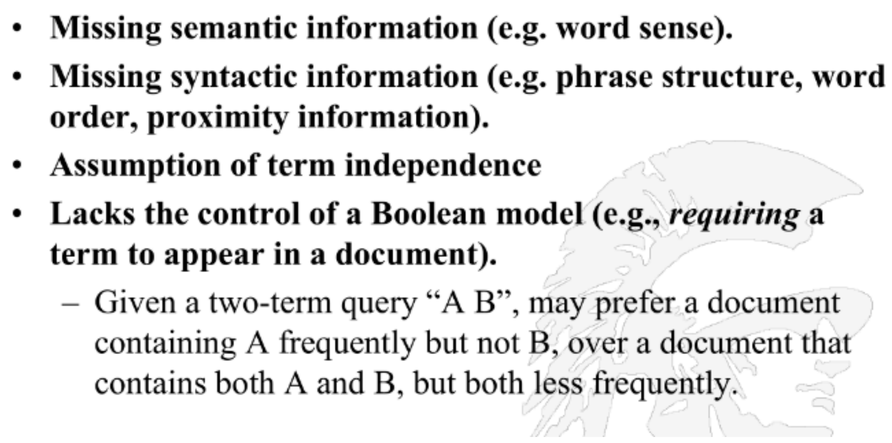
D_1 is 6 times better than D_2 using cosine similarity but only 5 times better using inner product.

-
- Vector Space ranking with Cosine Similarity



- **Represent the query as a weighted *tf.idf* vector**
- **represent each document as a weighted *tf.idf* vector**
- **compute the cosine similarity score for the query vector and each document vector that contains the query term**
- **Rank documents with respect to the query by score**
- **Return the top *k* (e.g. *k=10*) to the user**

-
- Works well but

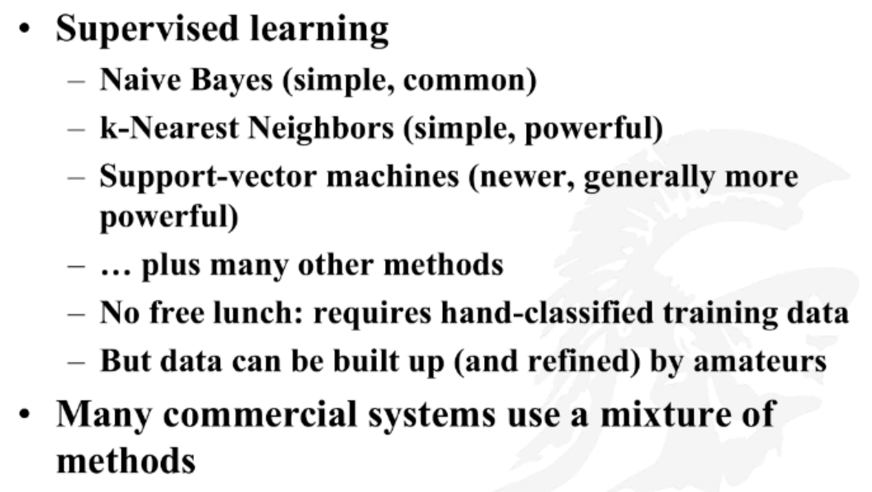


- **Missing semantic information (e.g. word sense).**
- **Missing syntactic information (e.g. phrase structure, word order, proximity information).**
- **Assumption of term independence**
- **Lacks the control of a Boolean model (e.g., *requiring* a term to appear in a document).**
  – Given a two-term query "A B", may prefer a document containing A frequently but not B, over a document that contains both A and B, but both less frequently.

Text Processing

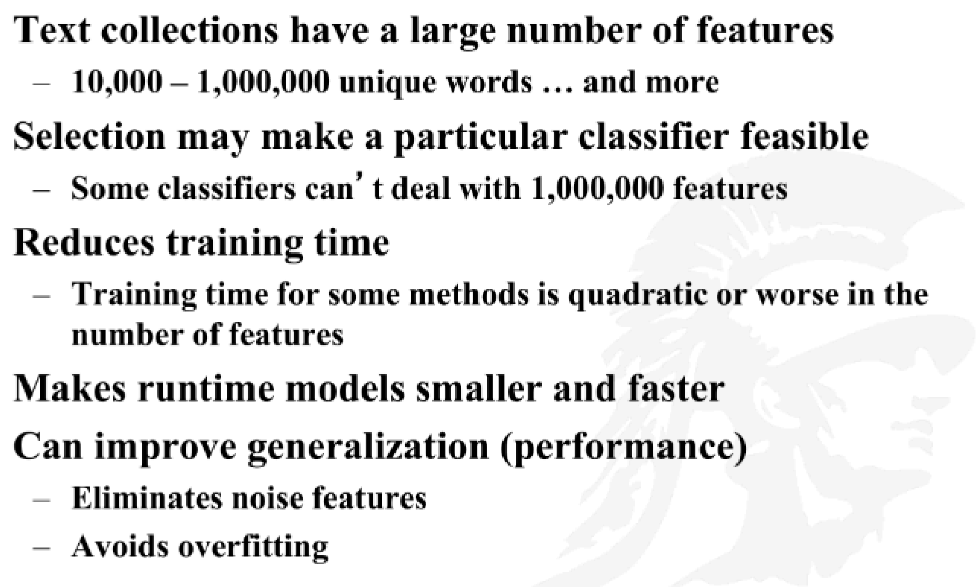
- Classification Methods
  - Manual

- Hand Coded Rule Based
- Supervised Learning



- Supervised learning
  - Naive Bayes (simple, common)
  - k-Nearest Neighbors (simple, powerful)
  - Support-vector machines (newer, generally more powerful)
  - … plus many other methods
  - No free lunch: requires hand-classified training data
  - But data can be built up (and refined) by amateurs
- Many commercial systems use a mixture of methods

-
- Feature Selection



Text collections have a large number of features
- 10,000 – 1,000,000 unique words … and more

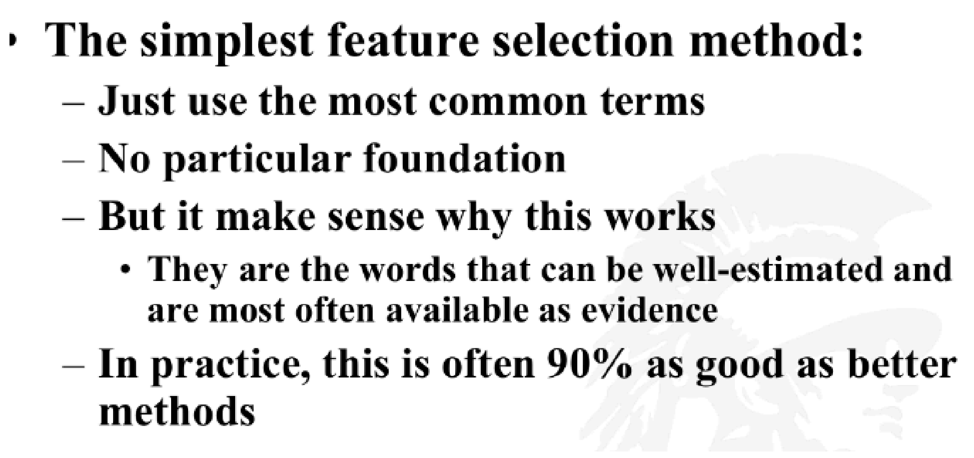
Selection may make a particular classifier feasible
- Some classifiers can't deal with 1,000,000 features

Reduces training time
- Training time for some methods is quadratic or worse in the number of features

Makes runtime models smaller and faster

Can improve generalization (performance)
- Eliminates noise features
- Avoids overfitting

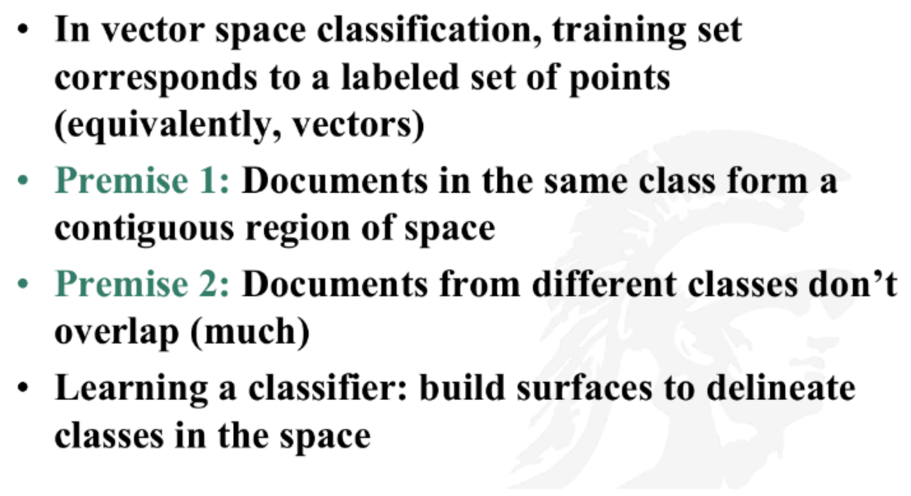
-



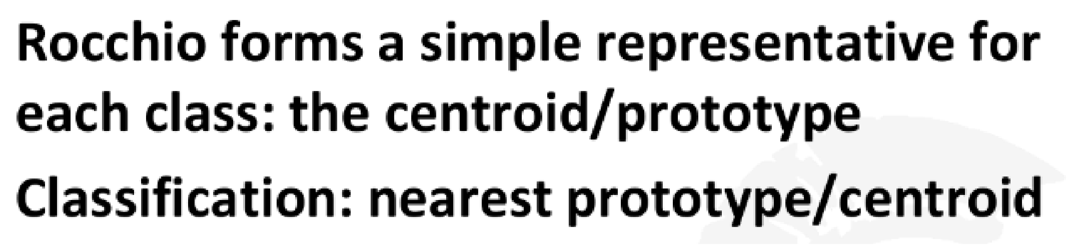
- The simplest feature selection method:
  - Just use the most common terms
  - No particular foundation
  - But it make sense why this works
    - They are the words that can be well-estimated and are most often available as evidence
  - In practice, this is often 90% as good as better methods

-

- Naive Bayes
    - Very fast learning and testing
    - Low storage
    - Very good in domains with equally important features
    - More robust to irrelevant features

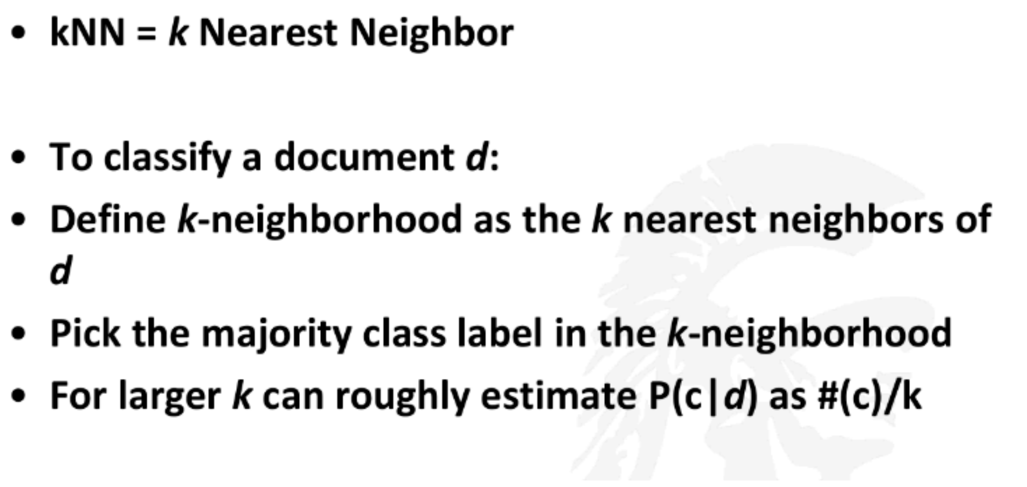


- **In vector space classification, training set corresponds to a labeled set of points (equivalently, vectors)**
- **Premise 1: Documents in the same class form a contiguous region of space**
- **Premise 2: Documents from different classes don't overlap (much)**
- **Learning a classifier: build surfaces to delineate classes in the space**



Rocchio forms a simple representative for each class: the centroid/prototype

Classification: nearest prototype/centroid



- **kNN = *k* Nearest Neighbor**

- **To classify a document *d*:**
- **Define *k*-neighborhood as the *k* nearest neighbors of *d***
- **Pick the majority class label in the *k*-neighborhood**
- **For larger *k* can roughly estimate P(c|*d*) as #(c)/k**