

Sparse Multidimensional Data

- ❑ Cases and Applications
 - Text databases
 - ❑ Sparse binary vectors
 - ❑ Word frequency vectors
 - Recommender systems
- ❑ Containment and similarity queries
- ❑ Indexing for containment and similarity
- ❑ Computing recommendations

	team	coach	play	ball	score	game	win	lost	timeout	season
Document 1	3	0	5	0	2	6	0	2	0	2
Document 2	0	7	0	2	1	0	0	3	0	0
Document 3	0	1	0	0	1	2	2	0	3	0

Text Databases



Application: Text Databases

- A **text database** is a collection of documents.
- Each document is considered as a **list of words**.
- Example:
 - A database of abstracts, legal transcripts, newspaper articles, the WWW.

Search in a Text Database

- ❑ We typically search in a text database, using interesting **keywords**.
 - E.g., find all documents containing “Shakespeare” and “Othello”
- ❑ Two popular forms of queries
 - Containment queries (a.k.a. Boolean retrieval)
 - ❑ E.g., find documents containing “Shakespeare” and “Othello”
 - Ranking queries
 - ❑ E.g., rank documents **based on their similarity** to a list of terms or a sample text
- ❑ Searching and managing text is a main subject of **Information Retrieval**

Containment queries

- ❑ Search terms are expressed in a formula that connects them with **AND** (\wedge), **OR** (\vee), **NOT** (\neg) predicates.
- ❑ A document qualifies the query or does not qualify the query (Boolean answer).
- ❑ Example:
 - (“car” \vee “automobile”) \wedge (“convertible” \vee “cabriolet”)

....John arrived in
a convertible
automobile....

- ❑ **Stemming** is used to unify different endings of the same word (e.g., remove -ing, -ed, etc.)

Evaluation of containment queries

- ❑ Given a text database, how do we evaluate containment queries on it?
- ❑ Naive approach:
 - Scan each document in the database and verify the query
 - **Expensive**: Each document must be parsed and each word must be compared with the query keywords
- ❑ Better:
 - Preprocess the documents and extract the words from them.
 - **Index** the documents based on the words they contain.
 - Use the index to evaluate containment queries.

Set comparison

- ❑ Assume containment queries with only AND predicates (e.g., $s = \text{"A"} \wedge \text{"D"} \wedge \text{"M"}$)
- ❑ Each query is a set of keywords s
- ❑ Each document is a set of words t
- ❑ Problem: check if $s \subseteq t$
- ❑ Algorithm (similar to merging sorted lists)
 - Sort s and t
 - Scan the words of s and t concurrently until some word of s is not found in t , or all words are found.
- ❑ Expensive (sorting + 1 pass)

$s = A \textcircled{D} M$
 $t = A B \textcircled{E} F N M O P$

stop here! return false

Signature-based Indexing



Signatures

- ❑ Sets can be approximated by **signatures**.
- ❑ Signatures are bitmaps generated by some hash function.
- ❑ Let W be the set of keywords that appear in any document of the database (e.g., the dictionary of terms). In other words, W is the **domain** of the set elements. Formally, each document/query is a subset of W .
- ❑ Let b be a **signature-length** and H a hash function that maps each word e in W to a number from 0 to $b-1$.
- ❑ The **signature** of a set s is a bitmap of length b formed by setting bit $H(e)$ for each $e \in s$.

Signatures

□ Example:

- $W = \{1, 2, \dots, 100\}$ (words encoded by numeric identifiers)
- $b = 10$.
- $H(e) = e \text{ modulo } 10$.
- $x = \{38, 67, 83, 90, 97\}$, $\text{sig}(x) = 1001000110$.
- $y = \{18, 67, 70\}$, $\text{sig}(y) = 1000000110$.

□ Signatures are **lossy** approximations; they do not represent the sets **exactly**. Two different sets may have the same signature.

- E.g., $z = \{18, 27, 33, 60\}$: $\text{sig}(z) = 1001000110 = \text{sig}(x)$.

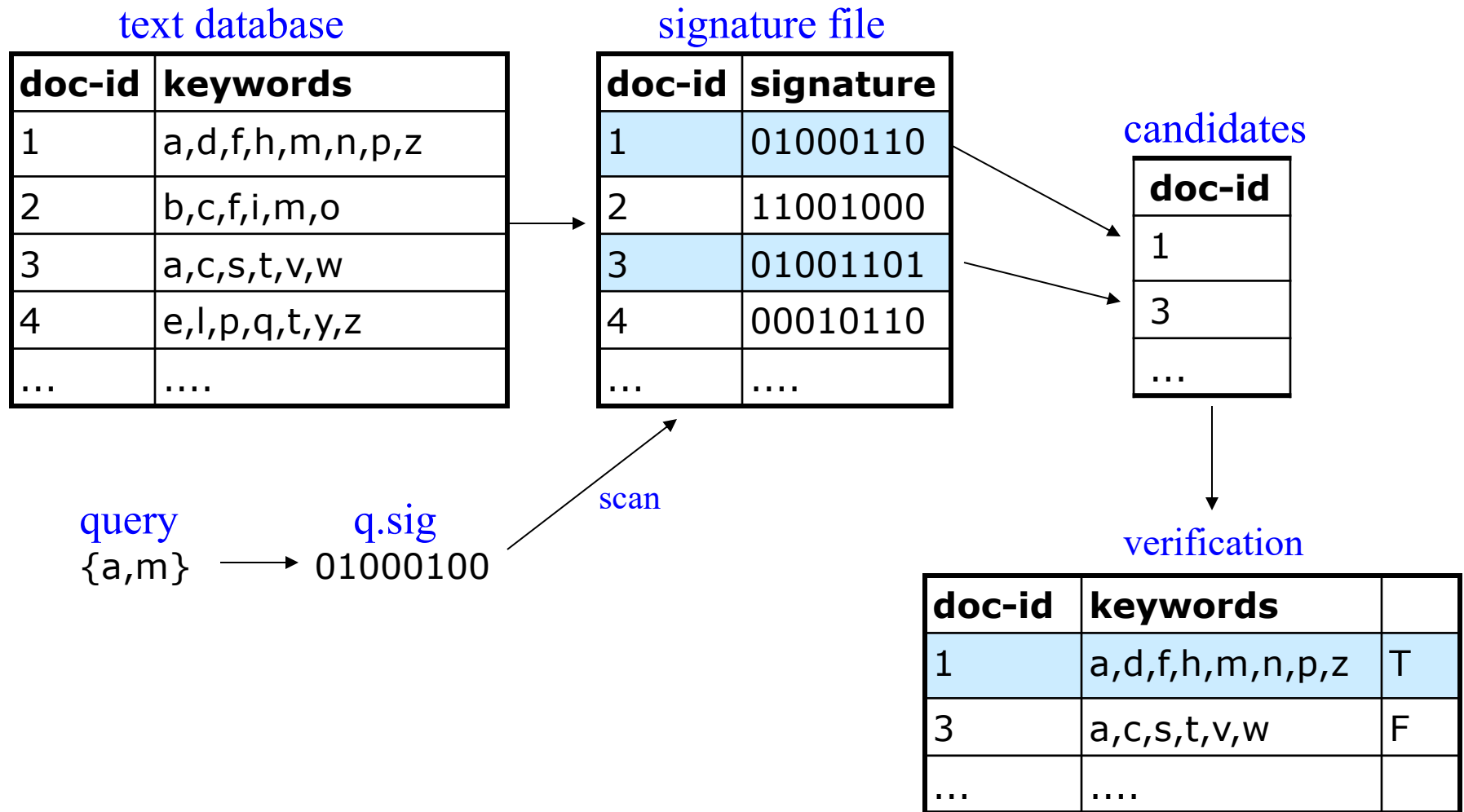
Query processing using signatures

- if $x \subseteq y$ then $\text{sig}(x) \subseteq \text{sig}(y)$
- $\text{sig}(x) \subseteq \text{sig}(y) \Leftrightarrow \text{sig}(x) \& \neg \text{sig}(y) = 0$
 - e.g., $0100010000 \subseteq 0110010010$
- Similarly:
 - if $x = y$ then $\text{sig}(x) = \text{sig}(y)$
 - if $x \cap y \neq \emptyset$ then $\text{sig}(x) \& \text{sig}(y) \neq 0$
- Using bitwise operations on signatures we can determine fast if the sets **may** qualify the predicate.
- Signatures are used as fast **filters** for set-operations.

Query processing using signatures

- ❑ Given a text database D , we can create a *signature file* that stores sequentially the signatures of all documents in D .
- ❑ A containment query q is then processed in two steps:
 - $sig(q)$ is computed and the signature file is scanned to find the document-ids which qualify the selection predicate on the signature level (*drops*)
 - for each drop, the document is compared to q . If a drop does not pass the refinement step it is called a *false drop*.
- ❑ several selection query types exist:
 - *set containment selection* (or *subset query*): Find all documents in D that contain q . The inverse *superset query* is also useful.
 - Also: *set equality selection*, *set overlap selection*

Example



Updating signature files

- ❑ Similar to updating of relations. When a document is inserted to the text database a new signature is inserted to the signature file. Deletions are similar.
- ❑ Each signature has fixed-length. Managing signature files is similar to managing relations of fixed-length records.

Signature-based indexes

- ❑ The signatures can be indexed in order to avoid accessing all of them for a single query.
- ❑ Several methods exist, among which the **bit-sliced signature** file dominates.
- ❑ Given a query signature $q.sig$, only the columns where $q.sig$ has 1 are accessed and bitwise ANDed.
- ❑ Example: if $q.sig = 0010000110$, only columns S_2, S_7, S_8 are accessed.

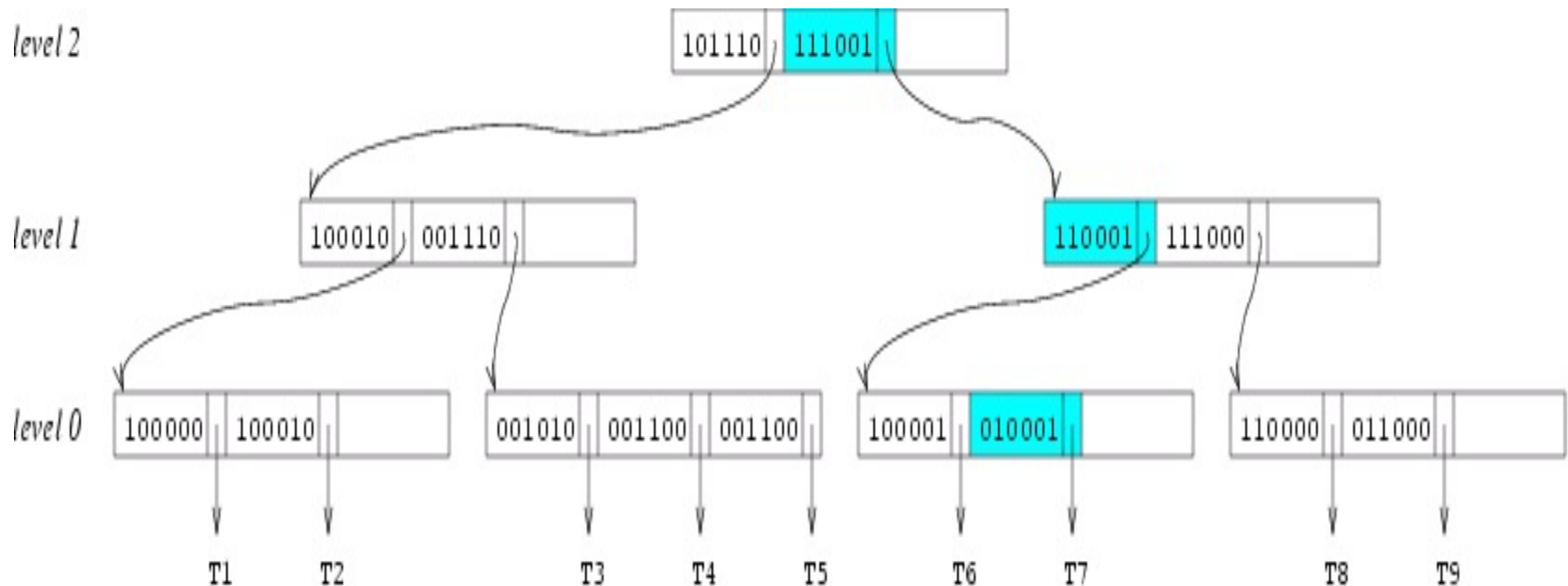
	$b=10$									
	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
1	1	0	1	0	0	0	1	0	1	0
2	0	1	0	1	0	1	0	1	0	1
3	1	0	0	0	1	0	1	1	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
N	0	1	0	0	0	0	1	1	0	1

bit-sliced signature file

Signature-based indexes

- ❑ Another method is the *signature-tree* (good for dynamic data).
- ❑ Similar to the R-tree, but instead of MBRs it uses signatures.
- ❑ Leaf nodes contain document signatures
- ❑ Directory nodes contain entries, whose signature is the logical OR of all signatures in the sub-tree pointed by them.
- ❑ Heuristics similar to R-tree construction are used to insert/update signatures in the tree.

The SG-tree (example)



Example of a **set-containment** query:

$W = \{a,b,c,d,e,f\}$

$q = \{b,f\}$ (i.e., find all documents that contain b and f)

$q.sig = 010001$

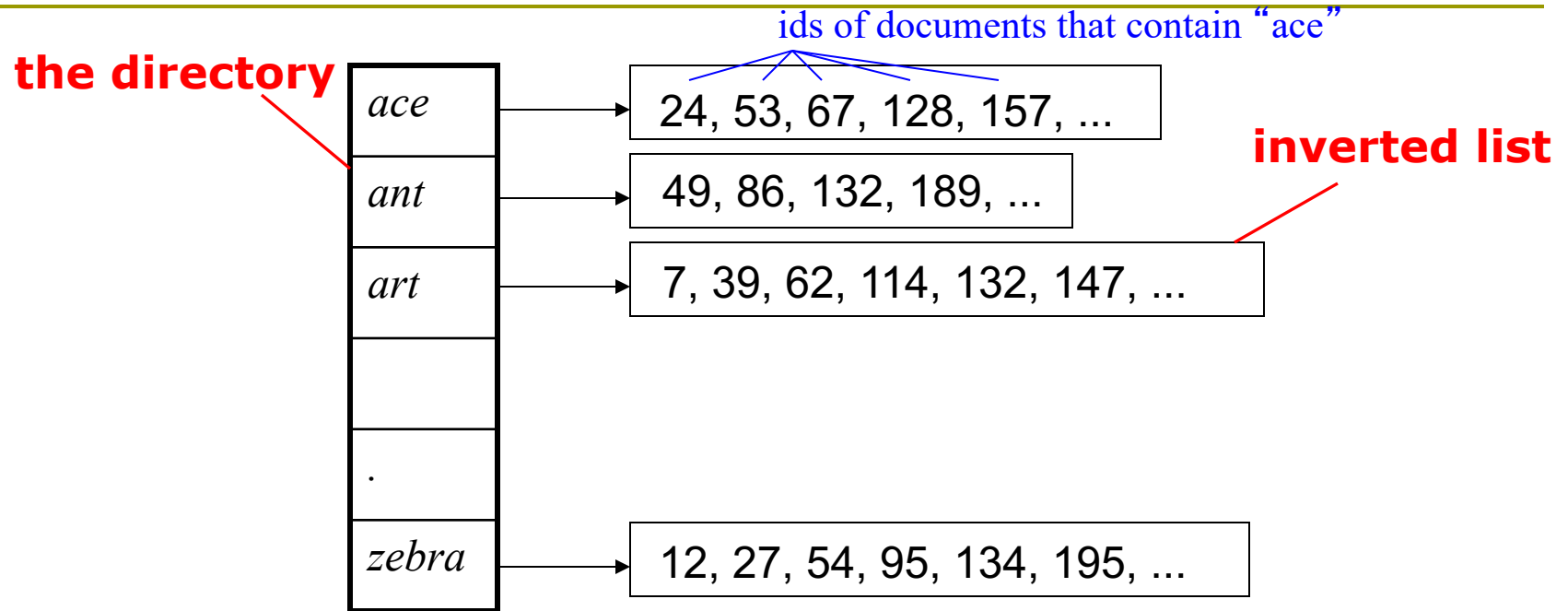
The Inverted File



The Inverted File

- A popular indexing method for **sparse sets** is to build an *inverted file* for the set elements in W .
- For each word e , the index has an *inverted list*, which stores the ids of the documents that contain the word e .
 - Lists are sorted
- A containment query is processed by **intersecting the sorted lists** that correspond to the elements of the query.

The Inverted File - Example



- ❑ Query: Find the documents that contain $\{ant, art\}$
 - Method: join the inverted lists that correspond to *ant* and *art*. This results in $\{132\}$.
- ❑ List intersection is cheap (just one pass) because the ids are sorted

The Inverted File - Comments

- ❑ The inverted file was shown superior to signature-based techniques in several studies
 - Signatures are good only if the sets are dense
- ❑ It is best for **static data**. Updates are expensive, thus they are performed in batches
- ❑ It can be easily compressed, so it can be much smaller than the indexed document collection

The Inverted File - Compression

- ❑ An inverted list is compressed by converting it to a list with the *run-length encoding* of ids (i.e., the gaps between consecutive ids)
 - Example: {11,24,57,102,145,173,...}
becomes: {11,13,33,45,43,28,...}
- ❑ A variable-length encoding is used for the gaps, e.g., Golomb coding
- ❑ Few bits, e.g. 10-11 suffice to encode each id

number	code
11	01011
13	01101
33	1100000
45	1101100
43	1101010
28	101011

Document Similarity and Information Retrieval



Document ranking

- ❑ So far, we have considered a document as a sparse, $|W|$ -length **bit-vector**, indicating which terms it contains from W .
 - E.g. $W=\{a,b,c,d,e,f\}$. A document with keywords a,c,f is represented by 101001
- ❑ Alternatively, a document may be represented by a $|W|$ -length vector, indicating the relevance of each term in the document. The relevance (weight) of a keyword e to a document could be e 's frequency in the document (**bag of words model**).
 - E.g. $W=\{a,b,c,d,e,f\}$. A document with vector $\{3,0,2,0,0,1\}$ contains keyword a 3 times, c 2 times, and f 1 time.
 - Vector is **sparse** (most entries are zeros)

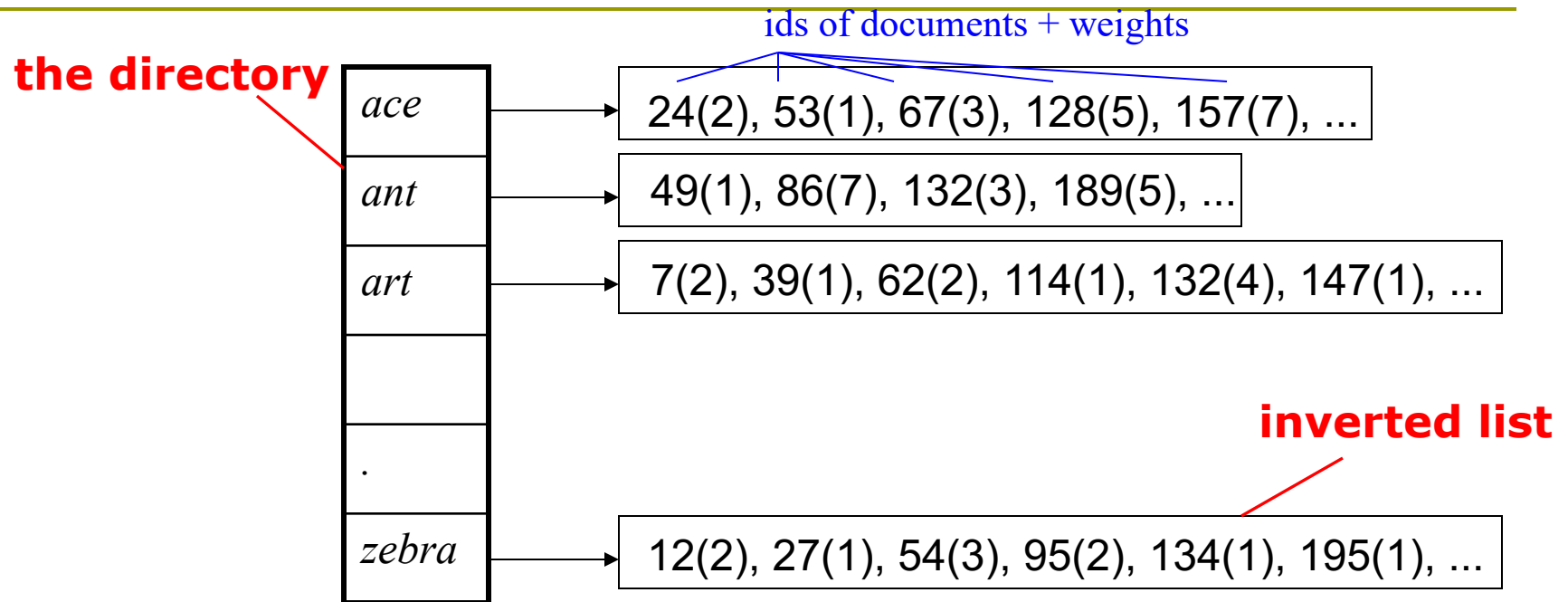
Document ranking

- Consider a query text q , with relevance vector to the keywords $w_q = w_{q,1}, w_{q,2}, \dots, w_{q,N}$. The **cosine similarity** of q to a document d is defined by:

$$\text{sim}(d, q) = \frac{\vec{w}_d \bullet \vec{w}_q}{|\vec{w}_d| \times |\vec{w}_q|} = \frac{\sum_{i=1}^N w_{d,i} \times w_{q,i}}{\sqrt{\sum_{i=1}^N w_{d,i}^2} \times \sqrt{\sum_{i=1}^N w_{q,i}^2}}$$

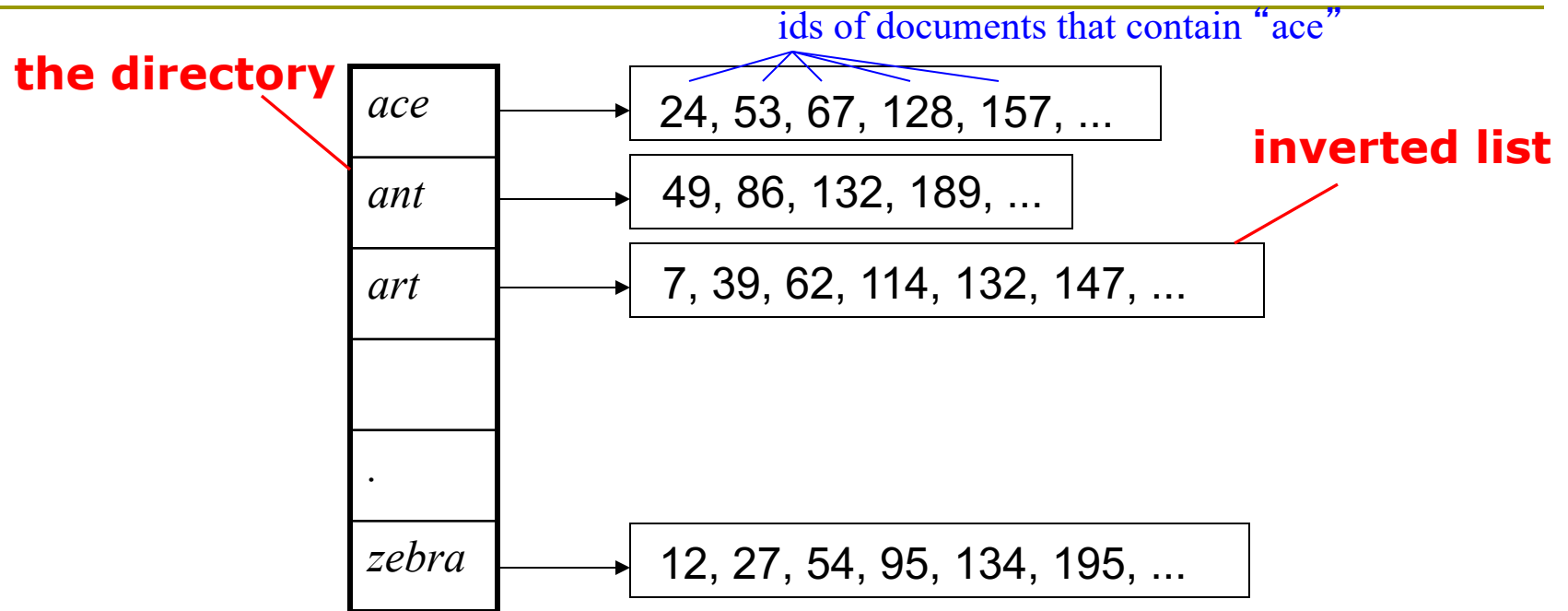
- Given a database of documents and a query q , we can define a **ranking** of the documents based on their similarity to q .

Using the Inverted File for ranking



- Query: Find the documents similar to $\{ant, art\}$
 - Method: Accumulate the weights of documents for each term, by accessing and merging the inverted lists.
 - While measuring the similarity of documents, maintain a set of the K most similar documents to the query.

Special case: binary vectors



- ❑ Ignore multiplicity information
- ❑ Query: Find the documents similar to $\{ant, art, zebra\}$
 - Method: merge corresponding inverted lists and measure number of appearances for each document
 - If all documents are large, we can ignore denominator
 - ❑ Similarity measure reduces to a dot-product between binary vectors

Ranking measures

- ❑ So far, we have not considered the **different frequencies of terms** (words) in the document collection
 - **Rare terms** in a collection are more informative than frequent terms
- ❑ In addition, we measure the relevance of a term t to a document d by using simply the frequency of t in d
 - But in reality relevance is not linear to frequency

Log-frequency weighting

- The log frequency weight of term t in d is

$$w_{t,d} = \begin{cases} 1 + \log_{10} \text{tf}_{t,d}, & \text{if } \text{tf}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- $0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 1.3, 10 \rightarrow 2, 1000 \rightarrow 4$, etc.
- Score for a document-query pair: sum over terms t in both q and d :
- $\text{score} = \sum_{t \in q \cap d} (1 + \log \text{tf}_{t,d})$
- The score is 0 if none of the query terms is present in the document.

Document frequency and idf weight

- ❑ Rare terms are more informative than frequent terms
 - **Stop words** (e.g., 'the', 'and', etc.) are not informative at all
- ❑ df_t is the document frequency of t : the number of documents that contain t
 - df_t is an inverse measure of the informativeness of t
- ❑ For each term t define the inverse document frequency $idf_t = \log (N/df_t)$, where N is the number of documents
 - idf_t captures how informative t is in the whole document collection
- ❑ idf_t gives different importance to different keywords in a query

tf-idf weighting

- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$w_{t,d} = \log(1 + \text{tf}_{t,d}) \times \log_{10}(N / \text{df}_t)$$

- Best known weighting scheme in information retrieval
 - Note: the “-” in tf-idf is a hyphen, not a minus sign!
 - Alternative names: tf.idf, tf x idf
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection

Score for a document given a query

$$\text{Score}(q, d) = \sum_{t \in q \cap d} \text{tf.idf}_{t,d}$$

- Inverted files can readily be used to compute document scores and rank them
 - Each document is represented by a vector of tf-idf weights, one for each term
 - Inverted list for term t contains the documents d that include t and their $\text{tf.idf}_{t,d}$
- There are many variants
 - How “tf” is computed (with/without logs)
 - Whether the terms in the query are also weighted
 - ...

Improving efficiency

- ❑ Ranking requires accessing multiple inverted lists **at their entirety**
- ❑ Inverted lists can be very long and should be **merged**
- ❑ Internet-scale search requires instant answers
- ❑ **Remedy:**
 - Smarter indexing that avoids accessing entire inverted lists
 - For queries of many keywords, apply **soft conjunction**

Basic Approaches

- ❑ Only consider high-idf query terms
 - Ignore query terms that are not important (e.g., stop-words)
- ❑ For multi-term queries, only consider documents that contain **several** of the query terms
 - e.g. at least 3 out of 4 terms
 - Imposes a soft conjunction of inverted lists

Champion lists

- ❑ Precompute for each term t , the r docs of highest weight in t 's postings
 - Call this the champion list for t
 - (a.k.a. fancy list or top docs for t)
- ❑ Note that r has to be chosen at index build time
 - Thus, it's possible that $r < K$
- ❑ At query time, only compute scores for docs in the champion list of some query term
 - Pick the K top-scoring docs from amongst these

Authority of a document

- ❑ When searching collections of millions or billions of documents, we want the results to be both relevant to our query and **authoritative**
- ❑ We have seen how to measure relevance to query terms
- ❑ **Authority** is a query-independent property of a document that captures its importance
- ❑ Examples of authority signals:
 - Wikipedia pages vs other webpages
 - Articles in famous newspapers
 - Papers with many citations
 - Webpages with high **Pagerank**

Ranking considering authority

- ❑ Need to preprocess the documents and give each of them an **authority score** $g(d)$ in $[0,1]$
- ❑ For any query q , the **net score** of a document d can be computed as:
 - $\text{score}(q,d) = g(d) + \text{similarity}(q,d)$
 - Any other linear combination can be used
 - Cosine similarity, or other similarity measures between q and d can be used
- ❑ Search objective then becomes to find the top K documents by net score

Top K by net score – fast methods

- First idea: Order all inverted lists by $g(d)$
- This is a common ordering for all lists
 - For example, documents can be given IDs by descending $g(d)$
- Thus, we can concurrently traverse query terms' postings for
 - List intersection
 - Cosine similarity computation
- We can apply early termination of list scans by deriving bounds from the top K documents so far
- Time-bounded search can also be applied
 - E.g. return the top K documents found within 50ms

Champion lists in $g(d)$ -ordering

- Can combine champion lists with $g(d)$ -ordering
- Maintain for each term a champion list of the r docs with highest $g(d) + \text{tf-idf}_{td}$
- Seek top- K results from only the docs in these champion lists
- High and low lists:
 - For each term maintain a high and a low inverted list
 - High list is the champion list
 - Use high lists first for query evaluation
 - If you get less than K results, only then proceed to merge low lists

Impact-based ordering

- Order each inverted list by tf-idf_{td} (or net score)
- Now the order of documents in different lists can be different
- Perform top K aggregation (as in multi-criteria top-k queries)

Phrase queries

- ❑ We want to be able to answer queries such as ***"iPhone case"*** – as a phrase
- ❑ Thus the sentence *"My iPhone has a soft case"* is not a match.
 - one of the few **"advanced search"** ideas that works
 - Many more queries are *implicit phrase queries*
- ❑ For phrase queries, simple inverted files do not work

Approach 1: Biword indexes

- ❑ **Idea**: index every **consecutive pair** of terms in the text as a phrase (**special word**)
- ❑ For example the text "Friends, Romans, Countrymen" would generate the biwords
 - ***friends romans***
 - ***romans countrymen***
- ❑ Each of these **biwords** is now a dictionary term
- ❑ Two-word phrase query-processing is now immediate.

Longer phrase queries

- ❑ Longer phrases can be processed by breaking them down
- ❑ ***black iPhone case*** can be broken into a containment query with biwords:
 - ***black iPhone AND iPhone case***
- ❑ However, search is **not accurate**
 - can have false positives
 - ❑ E.g. "I have a black iPhone and I want to use Kim's iPhone case"
- ❑ Must verify the query for each retrieved document

Issues for biword indexes

- ❑ False positives
- ❑ Index blowup due to bigger dictionary
 - Infeasible for more than biwords, big even for them (wise to use only for popular biwords)
- ❑ Biword indexes are not the standard solution (for all biwords) but can be part of a compound strategy

Approach 2: Positional indexes

- In the inverted lists, store, for each ***term*** the position(s) in which tokens of it appear:

<***term***, number of docs containing ***term***;

doc1: position1, position2 ... ;

doc2: position1, position2 ... ;

etc.>

Positional index example

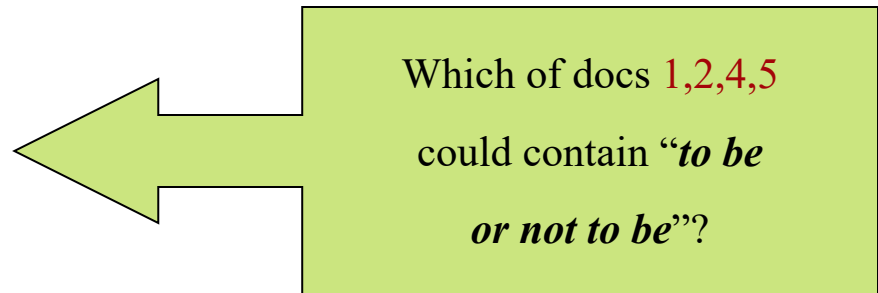
<*be*: 993427;

1: 7, 18, 33, 72, 86, 231;

2: 3, 149;

4: 17, 191, 291, 430, 434;

5: 363, 367, ...>



- For phrase queries, we use a merge algorithm recursively at the document level
- But we now need to deal with more than just equality

Processing a phrase query

- ❑ Extract inverted index entries for each distinct term: ***to, be, or, not.***
- ❑ Merge their *doc:position* lists to enumerate all positions with “***to be or not to be***”.
 - ***to:***
 - ❑ 2:1,17,74,222,551; 4:8,16,190,**429,433**; 7:13,23,191;
...
 - ***be:***
 - ❑ 1:17,19; 4:17,191,291,**430,434**; 5:14,19,101; ...
- ❑ Same general method for proximity searches

Proximity queries

- LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
 - here, / k means “within k words of”.
- Clearly, positional indexes can be used for such queries; biword indexes cannot.

Positional index size

- ❑ A positional index expands postings storage *substantially*
 - Even though indices can be compressed
- ❑ Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries ... whether used explicitly or implicitly in a ranking retrieval system.

Rules of thumb

- A positional index is 2–4 as large as a non-positional index
- Positional index size 35–50% of volume of original text
 - Caveat: all of this holds for “English-like” languages

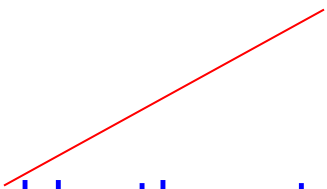
Combination schemes

- These two approaches can be profitably combined
 - For particular phrases ("**Michael Jackson**", "**Britney Spears**") it is inefficient to keep on merging positional postings lists
 - Even more so for phrases like "**The Who**"
 - Use biword inverted lists only for popular biwords
 - For other phrase searches and for proximity queries use positional index

Quality of Information Retrieval

- A search engine is good if
 - It indexes fast new documents and performs efficient updates
 - It searches fast
 - It has an expressive query language
 - It has a good UI
 - It is free
 - ...
 - The users are satisfied by the returned results

hard to measure
user satisfaction!



Measuring quality

- ❑ Quality of results depends on **intended use** (e.g., web search vs. e-commerce search)
- ❑ Relevance of results by a search engine based on benchmarks:
 - A document collection
 - A set of queries
 - Feedback from a set of users
 - ❑ **Binary assessment**: usually users mark query results as **relevant** or **not relevant**

Standard relevance benchmarks

- ❑ TREC - National Institute of Standards and Technology (NIST) has run a large IR test bed for many years
- ❑ Reuters and other benchmark doc collections used
- ❑ specified
 - sometimes as “Retrieval tasks” queries
- ❑ Human experts mark, for each query and for each doc, Relevant or Nonrelevant
 - or at least for subset of docs that some system returned for that query

Unranked retrieval evaluation:

Precision and Recall

- ▣ **Precision**: fraction of retrieved docs that are relevant = $P(\text{relevant}|\text{retrieved})$
- ▣ **Recall**: fraction of relevant docs that are retrieved
= $P(\text{retrieved}|\text{relevant})$

	Relevant	Nonrelevant
Retrieved	tp	fp
Not Retrieved	fn	tn

- ▣ Precision $P = \text{tp}/(\text{tp} + \text{fp})$
- ▣ Recall $R = \text{tp}/(\text{tp} + \text{fn})$

Precision, Recall and F-score

- ▣ You can get high recall (but low precision) by retrieving all docs for all queries!
- ▣ Recall is a non-decreasing function of the number of docs retrieved
- ▣ Combined measure that assesses precision/recall tradeoff is **F measure** (weighted harmonic mean):

$$F = \frac{1}{\alpha \frac{1}{P} + (1-\alpha) \frac{1}{R}} = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}$$

- ▣ People usually use balanced F_1 measure
 - i.e., with $\beta = 1$ or $\alpha = 1/2$

Substring Search



String Matching

- ❑ We have discussed how to search for documents containing a **set of words**.
- ❑ We will now discuss searching for **sub-strings** in a **long string**.
- ❑ Queries on string sequences often allow errors in matches. Therefore an interesting and challenging subject is **approximate string matching**.

Applications

□ Approximate keyword search

- text collections digitized by OCR contain a large percentage of errors (7-16%)
- typing or spelling errors in documents.

□ Computational Biology

- find subsequences in a long DNA sequence that approximately match a query sequence

□ Speech Recognition

- word identification in audio signals that have errors (lossy compression, bad pronunciation)

Problem 1: Search for substrings

- A popular data structure for indexing long strings in order to search for substrings in them is the **suffix tree**.
- The suffix tree is a compressed form of a **trie** that indexes all **suffixes** of the string.
- Algorithms for performing approximate search on suffix trees are also available.

Example of a suffix trie

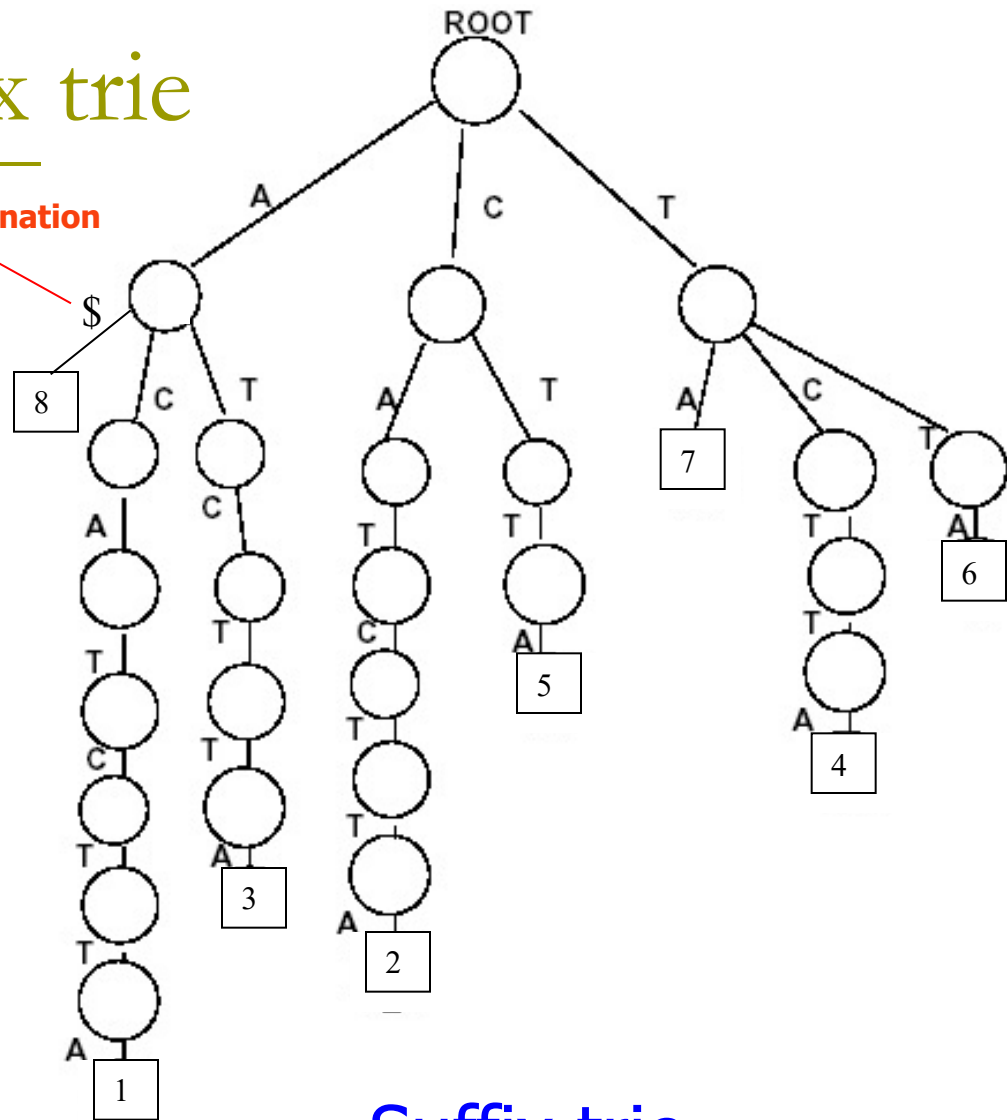
String

1 2 3 4 5 6 7 8
S=A C A T C T T A

suffixes:

A C A T C T T A
C A T C T T A
A T C T T A
T C T T A
C T T A
T T A
T A
A

Special termination
character



Suffix trie

Note!! This is an example of a string of characters, but it could be a string of words (document)

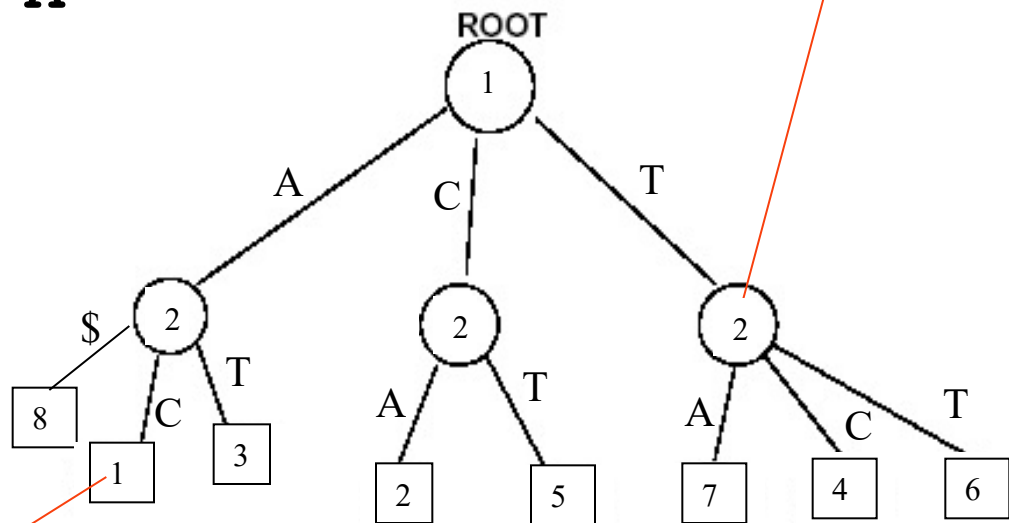
Example of a suffix tree

String

1 2 3 4 5 6 7 8
S=A C A T C T T A

The *suffix tree* is
a suffix trie,
where unary
paths are
compressed

Suffix tree



Each internal
node
represents a
unique
substring

Each leaf node is the
position of the suffix
compressed in the path

Suffix link connects node
indexing xS to node
indexing S (used for fast
construction)

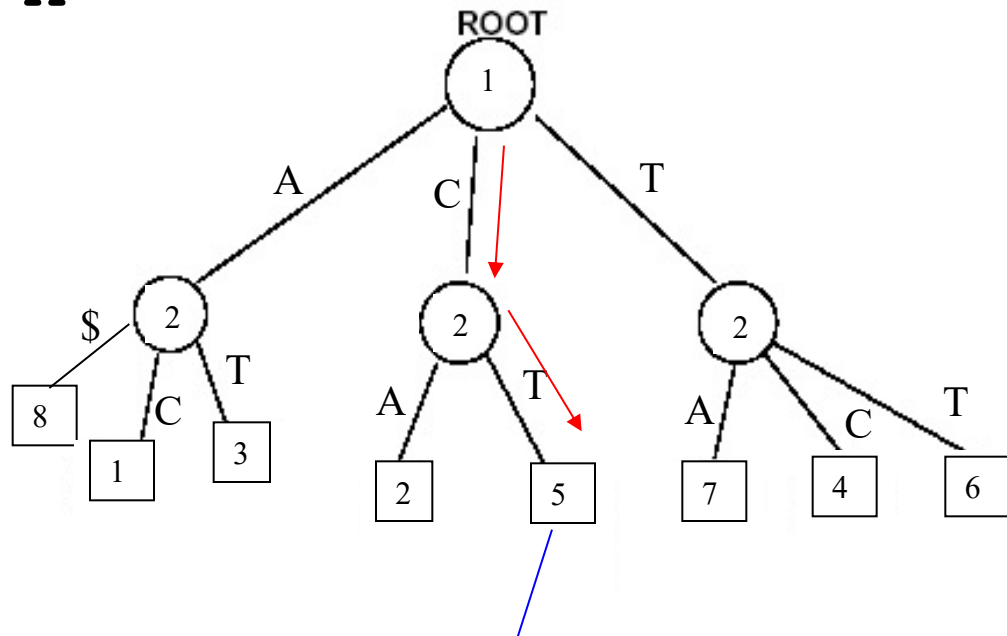
Example of a suffix tree search

String

1 2 3 4 5 6 7 8
S=A C A T C T T A

Search for “CTT”

Suffix tree



- a) we know that there is a substring CT*, starting at position 5
- b) go to that position and scan the string to check for occurrence of “CTT”

Suffix trees are good, but ...

- ❑ Main memory structures; how to construct and maintain **very large trees**?
 - ❑ Create **suffix arrays** instead of suffix trees
 - ❑ suffix array: a lexicographically sorted array of all suffixes and their positions in the string (could be secondary-memory array)
 - ❑ E.g., $\langle A, 8 \rangle, \langle AC^*, 1 \rangle, \langle AT^*, 3 \rangle$, etc.
 - ❑ use binary search or **a sparse index** to find fast a suffix in the array
 - ❑ Keys for suffix array could be created:
 - ❑ By creating a suffix tree and accessing the leaves
 - ❑ Difficult if the tree cannot fit in memory
 - ❑ Linear scan of the string, hashing of prefixes of suffixes, construction of suffixes, then external sorting of suffixes
 - ❑ Details are out of the scope of this course

Problem 2: Approximate search

- How do we define approximate search?
 - Must first define a similarity function between strings
- The similarity measure between two strings is typically dependent on the application and does not allow for general-purpose solutions
- The most widely accepted similarity metric is the **edit distance**.
 - the number of primitive operations (insert, delete, replace) necessary to transform one string to the other

Example of edit distance

- What is the edit distance between “survey” and “surgery”?

s u r **v** e y

s u r **g** e y

replace (+1)

s u r g e **r** y

insert (+1)

Edit distance = 2

General Edit Distance

- In the **general version of edit distance**, different operations may have different costs, or the costs depend on the characters involved.
 - replacement could be more expensive than insertion, or replacing “a” with “o” could be less expensive than replacing “a” with “k”.
- The general edit distance is powerful enough for a wide range of applications therefore most query processing algorithms consider it as a standard.
- The general edit distance **does not satisfy the triangular inequality** and thus it is *not* a metric.

Example of generic edit distance: String Alignment

- In biological string matching the similarity metric is often called “string alignment”
- Let S and T be strings. An alignment maps S and T into string S' and T' by inserting spaces into S and T , such that $|S'| = |T'|$.

Example:

$S = \text{acgcaggctc}$
 $T = \text{agcgtc}$

acgcaggctc
| | | | | | | |
ag_cg__tc

acgcaggctc
| | | | | | | |
a_gc__gtc

optimal alignment=distance

Other Similarity Metrics

(for reference only)

- ❑ Edit distance **with transpositions** (e.g., $ab \rightarrow ba$)
- ❑ **LCS** distance (allows only insertions/deletions)
- ❑ **Hamming** distance (allows only substitutions – only when $|s1| = |s2|$)
- ❑ **Episode** distance (allows only insertions – not symmetric)
- ❑ **Reversals** (allows reversing substrings)
- ❑ **Block distance** (allows rearrangement and permutation of substrings)
- ❑ **q -gram distance** (based on finding common substrings of fixed length q).

Definition of the basic approximate search problem

- Given a query substring q and a long sequence t , find all substrings in t which are similar to q given a distance metric.
- Two versions of the problem:
 - We are given a similarity threshold ϵ and ask for all substrings within this distance from q
 - We are given a number k and we ask for the k most similar substrings
- In a more general problem version, the long sequences t could be more than one.

A dynamic programming algorithm for computing the edit distance

- Problem: find the edit distance between strings x and y .
- Create a $(|x|+1) \times (|y|+1)$ matrix C , where $C_{i,j}$ represents the minimum number of operations to match $x_{1..i}$ with $y_{1..j}$. The matrix is constructed as follows.
 - $C_{i,0} = i$
 - $C_{0,j} = j$
 - $C_{i,j} =$
 - $C_{i-1,j-1}$ if $x_i = y_j$,
 - $1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$, else.

Example:

Optimal alignment

The diagram illustrates an optimal alignment between the words "survey" and "survey". A point above the word "survey" has lines connecting it to the 's', 'u', 'r', 'g', 'e', 'r', and 'y' characters in the second word. The alignment is represented by the following table, where red numbers indicate the cost of the alignment and yellow circles highlight the optimal path.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	0	1	2	3	4	5	6
u	2	1	0	1	2	3	4	5
r	3	2	1	0	1	2	3	4
v	4	3	2	1	1	2	3	4
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

How do we perform substring search?

- The same dynamic programming algorithm can be used to find the most similar **substrings** of a query string q .
- The difference is that we set $C_{0,j}=0$ for all j , since any text position could be the potential start of a match.
- If the similarity distance bound is k , we report all positions, where $C_m \leq k$ (m is the last row – $m = |q|$).

Example: q=survey, t=surgery, k=2

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

Comments on the dynamic programming algorithm

- It is very flexible, since it can be used to compute most distance metrics under the “generic edit distance” definition. It also requires only $O(|q|)$ space (only the previous column is necessary to compute the next one). Thus a single scan of t suffices.
- However, it is not efficient. The worst-case complexity is $O(|q||t|)$.
- Several variations have been proposed to reduce its complexity (out of the scope of this course).

Recommender Systems



Recommender systems

- Goal: **predict** the response of a user to options
 - Recommend products to e-commerce users
 - Recommend news articles to readers
 - Recommend movies to watch (e.g. see IMDb)
- They use past knowledge about the interests of users
- Two main categories of recommender systems
 - **Content-based systems**
 - **Classify items to categories**
 - Recommend to a user items in the categories s/he prefers
 - **Collaborative filtering systems**
 - Define **similarity measures** between users and/or items
 - Recommend to a user items preferred by similar users

The Utility Matrix

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

- ❑ The matrix is **sparse**
 - ❑ Ratings are **explicit** or **implicit**
 - ❑ **Goal of recommender system:** predict unknown values of the matrix **which are high**
- 1 if user bought/viewed item
0 otherwise

Content-Based Recommendations

- General approach:
 - Construct a **profile** for each item
 - Recommend to a user **items with similar profile** compared to the ones that s/he prefers
- Example of movie profile
 - The set of actors of the movie
 - The director
 - The year
 - The genre of movie
 - IMDB assigns a genre to every movie



Description-based recommendation

- ❑ **Documents** are often used to describe items
 - Product descriptions and reviews, news items
- ❑ Use **dominant words as features**
 - Words with the highest tf-idf scores
 - Use set-similarity measures
 - ❑ Use Jaccard similarity
 - ❑ If the same number of words is used for each document, this is equivalent to the **size of intersection**
- ❑ **Item similarity can be defined based on document similarity**
 - Additional features can be added (prev. slide)

Pros/Cons: Content-based Approach

- +: No need for data from other users
- +: Able to recommend to users with unique tastes
- +: Able to recommend new & unpopular items
- +: Able to provide explanations
- -: Finding appropriate features can be hard
- -: Recommendations for new users
 - How to build a user profile?
- -: Overspecialization
 - Never recommends items outside user's profile
 - Unable to exploit quality judgments of other users

Collaborative filtering

- ❑ Instead of using item-item similarity, use **similarity between users**
 - Similarity between rows of the Utility Matrix
- ❑ For a given user, find most similar users and recommend items that they like
- ❑ Simple approaches
 - Jaccard similarity (assume only 1's and 0's)
 - ❑ **lose ratings data**
 - Cosine similarity
 - ❑ **some users tend to rate higher compared to others**

Pearson correlation coefficient

□ S_{xy} = items rated by both users x and y

$$sim(x, y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)(r_{ys} - \bar{r}_y)}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)^2} \sqrt{\sum_{s \in S_{xy}} (r_{ys} - \bar{r}_y)^2}}$$

$\bar{r}_x, \bar{r}_y \dots$ avg. rating of x, y

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3



subtract the (row) mean and use Cosine similarity

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	2/3			5/3	-7/3		
B	1/3	1/3	-2/3				
C				-5/3	1/3	4/3	
D		0					0

Rating Predictions

- Let \mathbf{r}_x be the vector of user \mathbf{x} 's ratings
- Let \mathbf{N} be the set of k users most similar to \mathbf{x} who have rated item i
- Prediction for rating of item i by user \mathbf{x} :

- $r_{xi} = \frac{1}{k} \sum_{y \in N} r_{yi}$ (simple average)

- $r_{xi} = \frac{\sum_{y \in N} s_{xy} \cdot r_{yi}}{\sum_{y \in N} s_{xy}}$ (weighted average)

Shorthand:
 $s_{xy} = \text{sim}(\mathbf{x}, \mathbf{y})$

- Other options?
 - Clustering items based on ratings before prediction

	HP	TW	SW
A	4	5	1
B	4.67		
C		2	4.5
D	3		3

Item-Item Collaborative Filtering

□ **So far: User-user collaborative filtering**

□ **Another view: Item-item**

- For item i , find similar items rated by x
- Estimate rating r_{xi} based on ratings by x to similar items to i
- Can use same similarity metrics and prediction functions as in user-user model

$$r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}$$

s_{ij} ... similarity of items i and j

r_{xj} ... rating of user u on item j

$N(i;x)$... set of items rated by x similar to i

Item-Item CF ($|N|=2$)

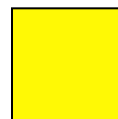
users

movies

	1	2	3	4	5	6	7	8	9	10	11	12
1	1		3			5			5		4	
2			5	4			4			2	1	3
3	2	4		1	2		3		4	3	5	
4		2	4		5			4			2	
5			4	3	4	2					2	5
6	1		3		3			2			4	



- unknown rating



- rating between 1 to 5

Item-Item CF ($|N|=2$)

users

movies

	1	2	3	4	5	6	7	8	9	10	11	12
1	1		3		?	5			5		4	
2			5	4			4			2	1	3
3	2	4		1	2		3		4	3	5	
4		2	4		5			4			2	
5			4	3	4	2					2	5
6	1		3		3			2			4	



- estimate rating of movie 1 by user 5

Item-Item CF ($|N|=2$)

	users												
	1	2	3	4	5	6	7	8	9	10	11	12	sim(1,m)
1	1		3		?	5			5		4		1.00
2			5	4			4			2	1	3	
<u>3</u>	2	4		1	2		3		4	3	5		<u>0.41</u>
4		2	4		5			4			2		-0.10
5			4	3	4	2					2	5	-0.31
<u>6</u>	1		3		3			2			4		<u>0.59</u>

Neighbor selection:

Identify movies similar to movie 1, rated by user 5

Here we use Pearson correlation as similarity:

1) Subtract mean rating m_i from each movie i

$$m_1 = (1+3+5+5+4)/5 = 3.6$$

row 1: [-2.6, 0, -0.6, 0, 0, 1.4, 0, 0, 1.4, 0, 0.4, 0]

2) Compute cosine similarities between rows

Item-Item CF ($|N|=2$)

		users												
		1	2	3	4	5	6	7	8	9	10	11	12	sim(1,m)
movies	1	1		3		?	5			5		4		1.00
	2			5	4			4			2	1	3	
	<u>3</u>	2	4		1	2		3		4	3	5		<u>0.41</u>
	4		2	4		5			4			2		-0.10
	5			4	3	4	2					2	5	-0.31
	<u>6</u>	1		3		3			2			4		<u>0.59</u>

Find N highest similarity scores:

$s_{1,3}=0.41$, $s_{1,6}=0.59$

Item-Item CF ($|N|=2$)

users

movies

	1	2	3	4	5	6	7	8	9	10	11	12
1	1		3		2.6	5			5		4	
2			5	4			4			2	1	3
<u>3</u>	2	4		1	2		3		4	3	5	
4		2	4		5			4			2	
5			4	3	4	2					2	5
<u>6</u>	1		3		3			2			4	

Predict by taking weighted average:

$$r_{1.5} = (0.41 \cdot 2 + 0.59 \cdot 3) / (0.41 + 0.59) = 2.6$$

$$r_{ix} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{jx}}{\sum s_{ij}}$$

Before:

$$r_{xi} = \frac{\sum_{j \in N(i; x)} s_{ij} r_{xj}}{\sum_{j \in N(i; x)} s_{ij}}$$

CF: Using baseline estimates

- Define **similarity** s_{ij} of items i and j
- Select k nearest neighbors $N(i; x)$
 - Items most similar to i , that were rated by x
- Estimate rating r_{xi} as the weighted average:

$$r_{xi} = b_{xi} + \frac{\sum_{j \in N(i; x)} s_{ij} \cdot (r_{xj} - b_{xj})}{\sum_{j \in N(i; x)} s_{ij}}$$

baseline estimate for r_{xi}

$$b_{xi} = \mu + b_x + b_i$$

- μ = overall mean of all ratings
- b_x = rating deviation of user x
= (avg. rating of user x) - μ
- b_i = rating deviation of movie i

Pros/Cons of Collaborative Filtering

- + Works for any kind of item
 - No feature selection needed
- - Cold Start:
 - Need enough users in the system to find a match
- - Sparsity:
 - The user/ratings matrix is sparse
 - Hard to find users that have rated the same items
- - First rater:
 - Cannot recommend an item that has not been previously rated
- - Popularity bias:
 - Cannot recommend items to someone with unique taste
 - Tends to recommend popular items

Matrix factorization (MF) for CF

- Assumption: the ratings matrix R is actually the product of two long, thin matrices

users												factors			users												factors																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																			
1		3			5			5		4			.1	-.4	.2																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																															</

- Objective: identify Q and P , such that the RSME is minimized

$$\text{RMSE} = \frac{1}{|R|} \sqrt{\sum_{(i,x) \in R} (\hat{r}_{xi} - r_{xi})^2}$$

(predicted) rating of user x on item i in $Q \cdot P^T$
 (actual) rating of user x on item i in R
 non-blank elements only

Ratings as Products of Factors

- How to predict the missing rating of user x for item i ?

				users								
	1		3		5		5		4			
		5	4	?	4		2	1	3			
items	2	4		1	2	3	4	3	5			
		2	4		5		4		2			
		4	3	4	2				2	5		
	1		3		3		2		4			

	.1	-.4	.2
	-.5	.6	.5
	-.2	.3	.5
items	1.1	2.1	.3
	-.7	2.1	-.2
	-.1	.7	.3
		factors	

Q

\approx

$$\hat{r}_{xi} = q_i \cdot p_x$$

$$= \sum_f q_{if} \cdot p_{xf}$$

q_i = row i of Q

p_x = column x of P^T

					users							
	1.1	-.2	.3	.5	-.2	-.5	.8	-.4	.3	1.4	2.4	-.9
	-.8	.7	.5	1.4	.3	-1	1.4	2.9	-.7	1.2	-.1	1.3
factors	2.1	-.4	.6	1.7	2.4	.9	-.3	.4	.8	.7	-.6	.1
							P^T					

Ratings as Products of Factors

- How to predict the missing rating of user x for item i ?

users

items

1		3			5			5		4	
		5	4	?		4			2	1	3
2	4		1	2		3		4	3	5	
	2	4		5			4			2	
		4	3	4	2					2	5
1		3		3			2			4	

≈

items

.1	-.4	.2
-.5	.6	.5
-.2	.3	.5
1.1	2.1	.3
-.7	2.1	-2
-1	.7	.3

factors

Q

factors

users

1.1	-.2	.3	.5	-.2	-.5	.8	-.4	.3	1.4	2.4	-.9
-.8	.7	.5	1.4	.3	-1	1.4	2.9	-.7	1.2	-.1	1.3
2.1	-.4	.6	1.7	2.4	.9	-.3	.4	.8	.7	-.6	.1

P^T

$$\hat{r}_{xi} = q_i \cdot p_x$$

$$= \sum_f q_{if} \cdot p_{xf}$$

q_i = row i of Q

p_x = column x of P^T

Ratings as Products of Factors

- How to predict the missing rating of user x for item i ?

users

items

1		3			5			5		4	
		5	4	2.4	4			2	1	3	
2	4		1	2		3		4	3	5	
	2	4		5			4			2	
		4	3	4	2					2	5
1		3		3			2			4	

≈

items

.1	-.4	.2
-.5	.6	.5
-.2	.3	.5
1.1	2.1	.3
-.7	2.1	-2
-1	.7	.3

factors

Q

factors

users

1.1	-.2	.3	.5	-2	-.5	.8	-.4	.3	1.4	2.4	-.9
-.8	.7	.5	1.4	.3	-1	1.4	2.9	-.7	1.2	-.1	1.3
2.1	-.4	.6	1.7	2.4	.9	-.3	.4	.8	.7	-.6	.1

P^T

$$\hat{r}_{xi} = q_i \cdot p_x$$

$$= \sum_f q_{if} \cdot p_{xf}$$

q_i = row i of Q

p_x = column x of P^T

A general MF algorithm

1. Preprocess initial Utility (Ratings) matrix
 - Subtract from each nonblank element the average rating of the corresponding user
 - (!) Must undo any normalization to the predicted ratings
2. Initialize a random solution (Q, P)
 - Give all elements a random value (e.g. the average rating)
3. For each element in Q and P (in some order) change its value such that the difference in the RMSE after the change is maximized
 - Repeat step 3 until the RMSE converges to a small value
4. Repeat steps 2 and 3 for different initial random solutions and keep the best factorization (best local minimum)

Example of step 3

$$\mathbf{R} = \begin{bmatrix} 5 & 2 & 4 & 4 & 3 \\ 3 & 1 & 2 & 4 & 1 \\ 2 & & 3 & 1 & 4 \\ 2 & 5 & 4 & 3 & 5 \\ 4 & 4 & 5 & 4 & \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} 2.6 & 1 \\ 1 & 1 \\ z & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}}_{\mathbf{Q}} \times \underbrace{\begin{bmatrix} 1.617 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}}_{\mathbf{P}^T} = \underbrace{\begin{bmatrix} 5.204 & 3.6 & 3.6 & 3.6 & 3.6 \\ 2.617 & 2 & 2 & 2 & 2 \\ 1.617z + 1 & z + 1 & z + 1 & z + 1 & z + 1 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.617 & 2 & 2 & 2 & 2 \end{bmatrix}}_{\hat{\mathbf{R}}}$$

try to change

affect RMSE

Objective: find value of z that minimizes:

$$(2 - (1.617z + 1))^2 + (3 - (z + 1))^2 + (1 - (z + 1))^2 + (4 - (z + 1))^2$$

Solution: set derivative to 0, gives $z = 1.178$

Gradient Descent

- The Matrix-Factorization algorithm is an example of gradient descent
 - given some data points (nonblank elements of M), for each point we find the direction of change that most decreases the error function
- **Improvement**: look at only a randomly chosen fraction of the data when seeking to minimize the error
 - Stochastic Gradient Descent

The Netflix Prize (2009)

□ Training data

- 100 million ratings, 480000 users, 17770 movies
- 6 years of data: 2000-2005

□ Test data

- 2.8 million ratings
- **Evaluation criterion:** Root Mean Square Error

$$(\text{RMSE}) = \frac{1}{|R|} \sqrt{\sum_{(i,x) \in R} (\hat{r}_{xi} - r_{xi})^2}$$

- **Netflix's system RMSE: 0.9514**

□ Competition

- 2,700+ teams
- **\$1 million** prize for 10% improvement on Netflix

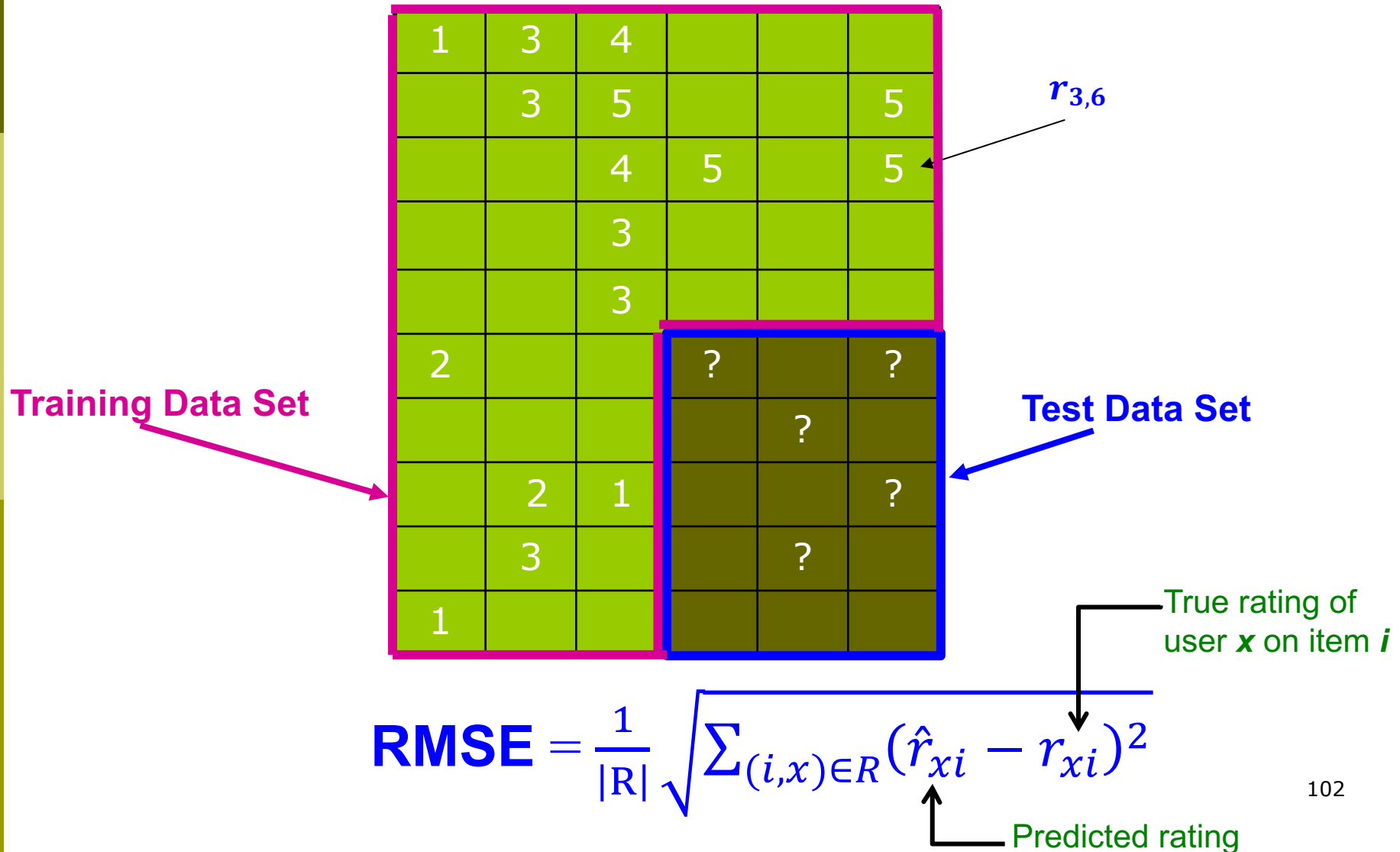
The Netflix Utility Matrix R

480,000 users

17,700 movies

1	3	4			
	3	5			5
		4	5		5
		3			
		3			
2			2		2
				5	
	2	1			1
	3			3	
1					

Utility Matrix R : Evaluation



BellKor Recommender System

- The winner of the Netflix Challenge!

- Multi-scale modeling of the data:

Combine top level, “regional” modeling of the data, with a refined, local view:

- **Global:**

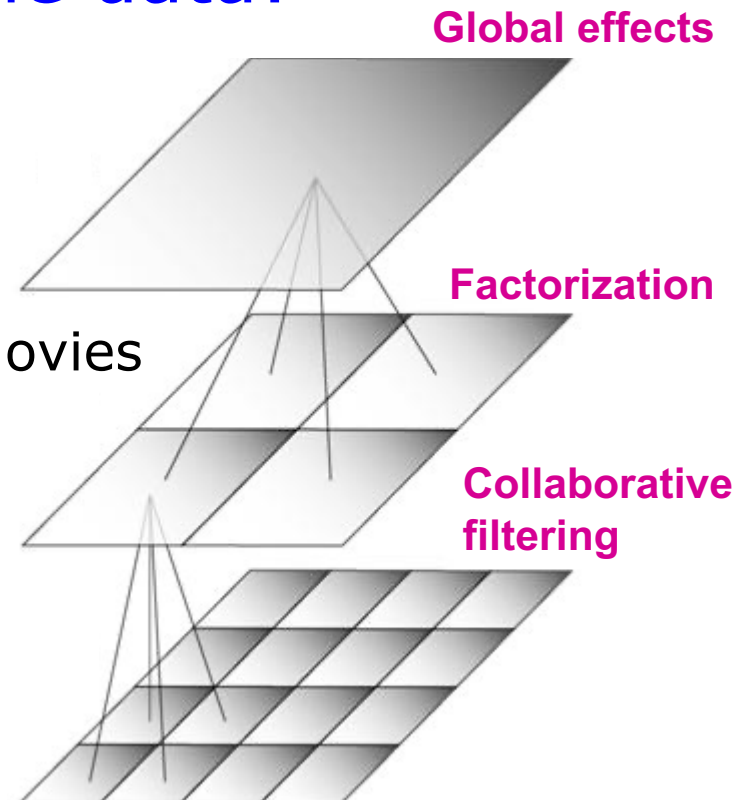
- Overall deviations of users/movies

- **Factorization:**

- Addressing “regional” effects

- **Collaborative filtering:**

- Extract local patterns



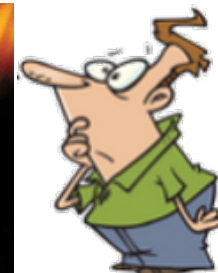
Modeling Local & Global Effects

□ Global:

- Mean movie rating: 3.7 stars
- *The Sixth Sense*: 0.5 stars above avg
- Joe rates 0.2 stars below avg.

⇒ Baseline estimation:

Joe will rate The Sixth Sense 4 stars



□ Local neighborhood (CF/NN):

- Joe didn't like related movie *Signs*
- ⇒ Final estimate:

Joe will rate The Sixth Sense 3.8 stars



Summary

- ❑ Sparse multidimensional data are found in many applications
 - Text documents can be modeled as sparse keyword containment vectors
 - Users in recommender systems are modeled as sparse vectors with rated items
- ❑ Similarity between sparse vectors finds application in
 - Document retrieval
 - Recommender systems
- ❑ Indexing: signature-based indexes, inverted files
- ❑ Fast and accurate recommendation: use matrix factorization