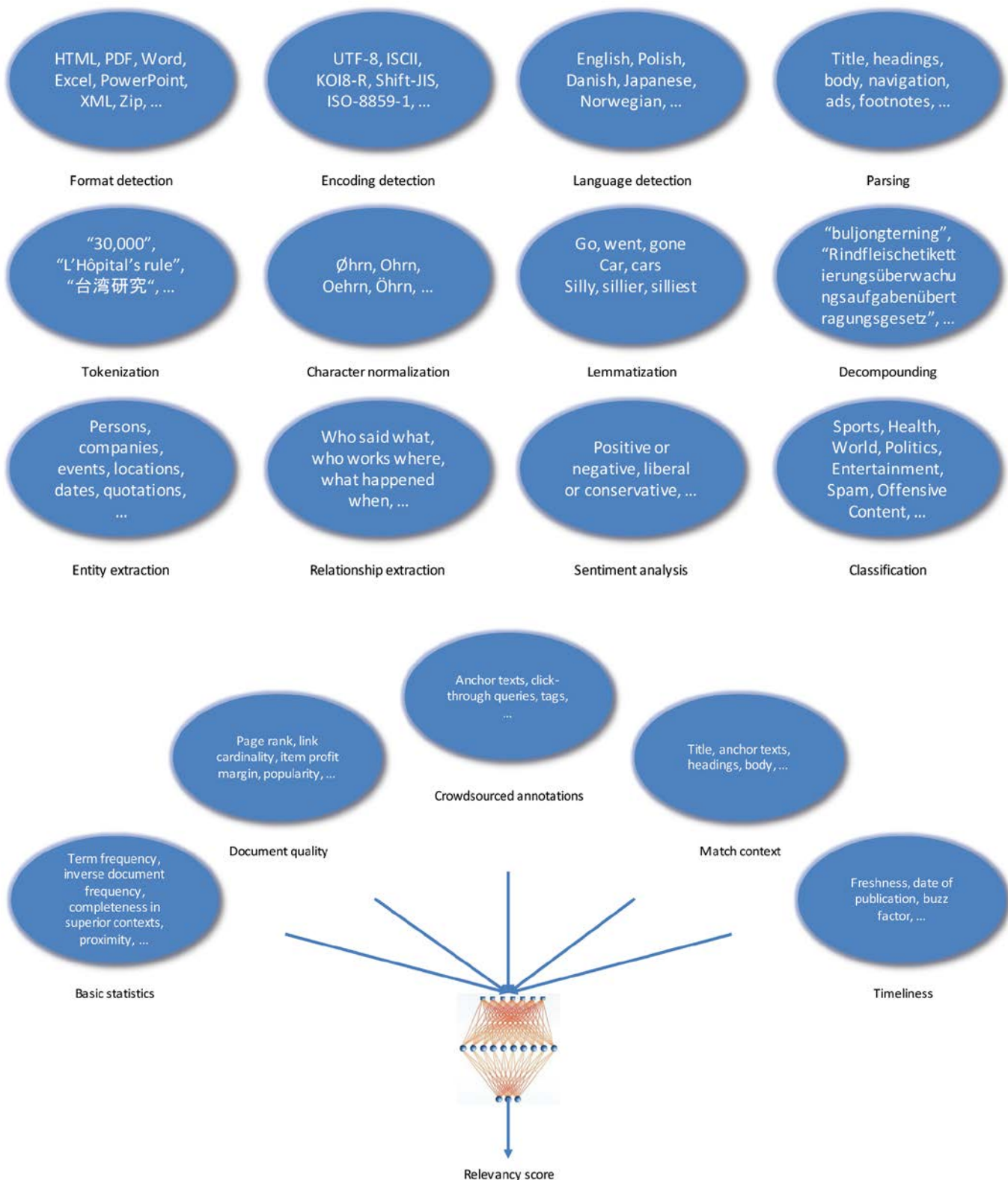


Contents

Information Retrieval (IR)	1
Inverted Index	2
Postings Lists	2
Preprocessing and Linguistic Modules	3
Dictionary Data Structure	5
Tolerant Retrieval	6
Index Construction	9
Index Compression	10
Dictionary Compression	11
Postings Compression	11
Ranked Retrieval Model	13
Vector Space Model	13
TFIDF Scoring	14
Optimizations for Efficient Ranking	15
Techniques for Improving Results	16
Relevance Feedback (RF)	16
Query Expansion (QE)	18
Evaluation	18
Text Classification	20
ML Classification	20
Naïve Bayes (NB)	21
Vector Space Classification	22
Web Search	24
Distributed Indexing	24
Link Analysis	25
XML Retrieval	26
Supplementary	28

Information Retrieval (IR)

Information Retrieval (IR) is the retrieval of *unstructured* materials (texts), which is to be contrasted with the retrieval of *structured* materials (databases). There are 2 main types of IR, either boolean retrieval model or ranked retrieval model.



Boolean Retrieval Model: *Complex* queries are in the form of a boolean expression (AND, OR, NOT). But since the results are exact, it's good for people with precise understanding of their needs as well as of the collection in general. Note that boolean queries often result in either too few (e.g. 0) or too many (e.g. thousands) results. For an arbitrary query, we can evaluate and store the answers for intermediate expressions in a complex expression.

Term-Document Incidence Matrix is a boolean table of term vs. documents (i.e. 1 if the doc contains the term, 0 otherwise).

- Very easy to search with this technique (simply do the bitwise AND, OR, NOT between different row/column **vectors**).
- Requires too much space. For $N = 10^6$ docs (10^3 words/doc and 6 bytes/word), we need 6GB. If vocabulary size is 10^5 terms, then we have $10^5 \times 10^6 = 10^{11}$ bits of data in the matrix, more than the data stored in the docs themselves.
- ➔ Since the matrix is very sparse, it's better to record only positions that have bit 1, which can be done with inverted index.

Inverted Index

Inverted Index is made up of a *dictionary* (in RAM) which has the vocabulary, statistics info, and pointers to the *postings lists* (on disk).

Brutus	→	1	2	4	11	31	45	173	174
Caesar	→	1	2	4	5	6	16	57	132 ...
Calpurnia	→	2	31	54	101				

- Posting maps to a docID, which is the ID of a document unit.
- Index granularity defines how small a document unit is. One unit can represent an email (attachment), a file or a collection of files, a chapter, a paragraph, or a line. The smaller the unit is, the easier it is for the user to find relevant materials.
- Postings lists are usually sorted by the docID. They can be implemented as:
 - o Linked list → Fast sorted insertion (due to the use of pointers), so it's *only* good for frequently updated index.
 - o ArrayList → Fast traversal (due to contiguous memory, so good for caches), uses less space (due to lack of pointers).

Postings Lists

Operations on Postings Lists:

- Intersection (or merging) is the most central operation that one can do on postings lists. The algorithm shown here is only for merging 2 terms (i.e. $p_1 \wedge p_2$), but it can easily be modified for other boolean queries as well, such as $(p_1 \vee \neg p_2)$, $(p_1 \wedge \neg p_2)$, etc.
- Let A and B be 2 postings lists of length m and n . Then queries of the forms $A \wedge B$ (merge/intersect) or $A \vee B$ might take $\mathcal{O}(m + n)$ time. Other queries (e.g. $\neg A \wedge \neg B$) might take much more than $\mathcal{O}(m + n)$.
- **Optimization:** For a large/complex boolean queries, break it up into several intermediate queries.
 - o E.g. $(A \wedge B \wedge C) \rightarrow$ Process them in increasing order of the postings lists' sizes (i.e. start with the shortest list first).
 - o E.g. $(A \vee B) \wedge (C \vee D) \wedge (E \vee F) \rightarrow$ Estimate the size of each \vee individually (the conservative way is just to add up the doc frequency of the 2 postings lists in question), and then process the \wedge in increasing order of sizes.

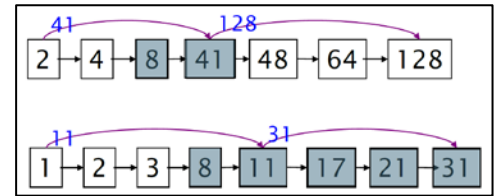
```

INTERSECT( $p_1, p_2$ )
1   $answer \leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4      then  $\text{ADD}(answer, \text{docID}(p_1))$ 
5           $p_1 \leftarrow \text{next}(p_1)$ 
6           $p_2 \leftarrow \text{next}(p_2)$ 
7  else if  $\text{docID}(p_1) < \text{docID}(p_2)$ 
8      then  $p_1 \leftarrow \text{next}(p_1)$ 
9      else  $p_2 \leftarrow \text{next}(p_2)$ 
10 return  $answer$ 

```

Skip Lists:

- Suppose we have done processing with [8]. Now increment the pointers to [41] and [11]. Look at skip pointer of [11], which is the *lower* one. Since the skip pointer points to [31] which is still smaller than [41], so we jump directly to [31]. Then we repeat by comparing [41] with skip pointer of [31].
- Trade-off for the locations of the skip pointers:
 - o Too many skips → Shorter skip spans (i.e. a skip covers less area) → More likely to skip → But lots of comparisons.
 - o Too few skips → Long skip spans → Few successful skips → But few pointer comparisons.
 - o For a list of length L , we generally use \sqrt{L} evenly-spaced skip pointers. Even spacing ignores the distribution of query terms among the docs, which is a good assumption since we assume that the docIDs are random.
- Skiplists are not always beneficial for performance. This happens in many situations:
 - o Too few or too many skip pointers can hurt the performance of skip pointers.
 - o If the 2 postings lists that are very similar in content, then skip pointers are useless because there is nothing to skip.
 - o If postings lists keep changing due to updates, then it would be very difficult to place the skip pointers.
 - o Skip lists are not useful for complex compound boolean queries. This is because only the original postings lists have the skip pointers. The intermediate postings lists have no skip pointers.



Preprocessing and Linguistic Modules

Main Challenges:

- *Parsing*: Followings are some of the classification problems faced when parsing a document.
 - o Mixture of document formats (PDF/Word/Excel/XML/HTML).
 - o Mixture of languages (English, Norwegian, Chinese, etc.).
 - o Mixture of character sets (UTF-8, CP1252, etc.).
- *Tokenization*: What's a token? Dashes/apostrophe (e.g. John's, or state-of-the-art)? Numbers (e.g. dates, IP addresses, computer-generated keys, postal tracking numbers, phone numbers, area codes, etc.)? Compound token (e.g. New York)?
- *Languages*: Followings are some of the potential language-specific problems.
 - o Many words in German are compound words. Using a compound splitter module gives 15% performance boost.
 - o Chinese and Japanese often have no spaces between words.
 - o Arabic (read from *right* → *left* for texts, but *left* → *right* for numbers).
- *Case folding*: Many words can only be distinguished by capitalization. E.g. (cat ↔ C.A.T), (bush ↔ Bush), (fed ↔ Fed), (window ↔ Windows). Rule of thumb is to convert to lowercase only the words appearing at the beginning of a sentence.
- *Soundex*: Should words that sound similar be mapped to each other? This is mostly for proper nouns (i.e. names).
- *Thesaurus*: Should synonyms be mapped to each other? E.g. (phone ↔ cellphone), (computer ↔ laptop).
- *Stop words*: Ignoring frequent words (the, a, to, of) might be a good idea. But in reality, we shouldn't do that because:
 - o These words are important in many cases (e.g. "to be or not to be", or "flights to Oslo").
 - o Term weighting techniques that are deployed can often reduce the impact of common words on document rankings.

Fields and Zones: Semi structured data are simply mixtures of free texts with structured *metadata* (fields/zones).

- *Field* is a piece of metadata with values of specific formats (e.g. publication date, language, file type, file size).
 - *Zone* is a piece of metadata that contains an arbitrary amount of text (e.g. author, title, abstract, references).
- ➔ Parse and build inverted indexes for both fields and zones, such that we can search on these metadata as well.

Normalization is the process of normalizing tokens so that matches occur despite superficial differences.

- E.g. (USA ↔ U.S.A), (Åland ↔ Aaland), (color ↔ colour), (anti-virus ↔ antivirus), (authorize ↔ authorization).
- Standard normalization method is to implicitly create *equivalent classes* (done by *stemming* or *lemmatization*). Note that for this to work, both the text and the query must both identically be normalized.
- Alternative method is to use *asymmetric expansion* (done either during index construction or at query time).
 - o Less efficient than equivalent classes, because there are more postings to be stored and merged.
 - o More flexible than equivalent classes, because expansion lists can overlap while not being identical (i.e. asymmetric).
 - o E.g. (QUERY → RESULT), (windows → MS Windows, windows, window), (window → window, windows).

Stemming and Lemmatization: Aim to reduce a word to its base form.

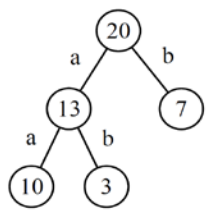
- E.g. (*be*, am, are, is), (*car*, car's, cars', cars), (*fast*, faster, fastest), (*rank*, ranked, ranking).
 - Stemming is mostly about crude affix chopping. Common stemmers for English are Lovins, Snowball, and Porters.
 - o Porter's algorithm has 5 phases of word reductions.

Sample text: Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

Porter stemmer: such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to a pictur of express that is more biolog transpar and access to interpret
 - Each phase consists of a set of rules on how to chop off the affix or to transform the word in some ways.
 - o E.g. (ational → ate), (tional → tion), (sses → ss).
 - Lemmatization performs linguistic/grammatical analysis to reduce a term to its base form. This might improve the IR, but not by much. Using a stemmer (which uses language-specific rules) is often good enough.
- ➔ Note that both stemming and lemmatization might increase recall while harming precision.

total count	20
a	13
aa	10
ab	3
b	7

(a) Count table



(b) Trie representation

field	32-bit	64-bit
index entry: token ID	4	4
index entry: pointer	4	8
start of index (pointer)	4	8
overhead of index structure	x	y
node value		
<i>total (in bytes)</i>	$12 + x$	$20 + y$

(c) Memory footprint per node in an implementation using memory pointers

0	13	offset of root node
1	10	node value of 'aa'
2	0	size of index to child nodes of 'aa' in bytes
3	3	node value of 'ab'
4	0	size of index to child nodes of 'ab' in bytes
5	13	node value of 'a'
6	4	size of index to child nodes of 'a' in bytes
7	a	index key for 'aa' coming from 'a'
8	4	relative offset of node 'aa' ($5 - 4 = 1$)
9	b	index key for 'ab' coming from 'a'
10	2	relative offset of node 'ab' ($5 - 2 = 3$)
11	7	node value of 'b'
12	0	size of index to child nodes of 'b' in bytes
13	20	root node value
14	4	size of index to child nodes of root in bytes
15	a	index key for 'a' coming from root
16	8	relative offset of node 'a' ($13 - 8 = 5$)
17	b	index key for 'b' coming from root
18	2	relative offset of node 'b' ($13 - 2 = 11$)

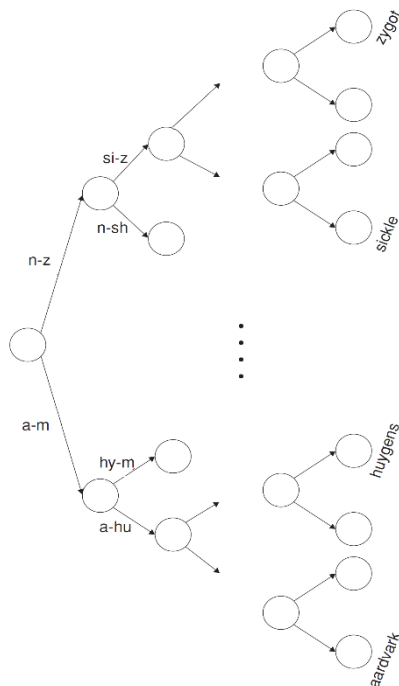
(d) Trie representation in a contiguous byte array. In practice, each field may vary in length.

Trie Packing with Byte Array

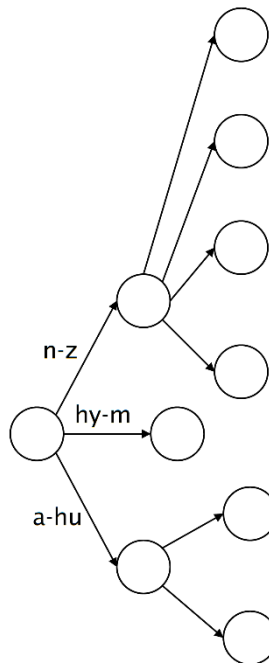
Dictionary Data Structure

Choices for Data Structures:

- **Naïve**: An array of objects, where each object represents a dictionary term. Each object requires approx. 28 bytes, which consists of a string for the term itself (20 bytes), doc frequency (4 bytes), and pointer to posting list (4 bytes).
- **Hashtables**: $\mathcal{O}(1)$ lookup, but no prefix search, and if the vocabulary grows, it might be very expensive to rehash *everything* in order to add new terms to the dictionary.
- **Trees**: Either binary trees, or *b*-trees (where no. of children in every internal node is between the interval $[a, b]$).
 - o Useful for prefix/suffix search or wildcard queries, e.g. terms starting with *hyp*.
 - o Cons: Slower compared to hashtables. Takes $\mathcal{O}(\log M)$ time for balanced tree (where M is the vocabulary size).
 - o Cons: Rebalancing the binary trees is expensive (note that *b*-tree doesn't need rebalancing because it has the interval).
- **Suffix array**: A suffix is basically a string standing at the "end" of some kind of text. Thus a suffix object can be represented 2 integers (one for the docID and the other one for the offset).
 - o With suffix array, we can either perform *phrase query searches* (token-wise), or handle *wildcards* (character-wise).
 - o To create suffixes, we split the text either token-wise or character-wise. Then sort the suffixes alphabetically.
 - o Sorting with quicksort takes $\mathcal{O}(n \log n)$ time. Searching with binary search takes $\mathcal{O}(\log n)$ time.
- **Trie**: Basically like a finite state automata that looks like a tree. Nodes are states that represent dictionary terms (but the terms are stored in transition arrows, not the nodes themselves). Each node also stores the corresponding doc frequency.
 - o Searching is to perform a trie-walk (i.e. Aho-Corasick algorithm), which is simply a traversal through the automaton.
 - o Trie allows for tolerant retrieval (or more specifically, for prefix searching).
 - o We can use a **recognizer** (minimal automaton) or a **byte array** to pack a trie. See the pictures!



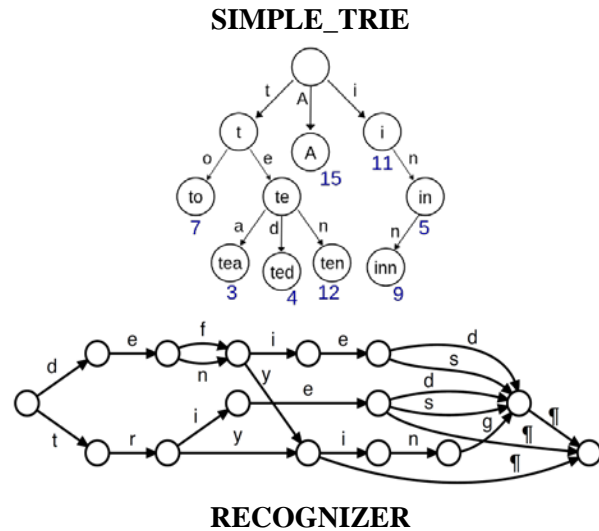
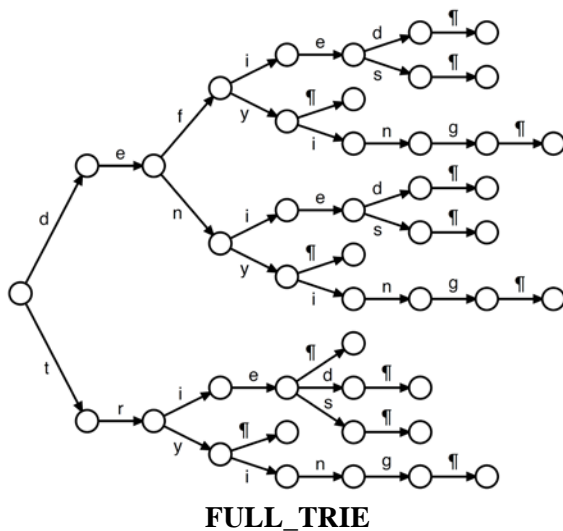
BINARY_TREE



B_TREE [2, 4]

Index	Suffix
10	i
7	ippi
4	issippi
1	ississippi
0	mississippi
9	pi
8	ppi
6	sippi
3	sissippi
5	ssippi
2	Ssissippi

SUFFIX_ARRAY



Tolerant Retrieval

Tolerant Retrieval (incl. proximity search, phrase search, soundex, wildcards, and spelling correction):

- Requires some inverted indexes (e.g. n -grams indexes) in addition to the normal inverted index.
- E.g. "(SPELL(moriset) /3 toron*to) OR SoundDex(chaikofski)" is a compound query which can trigger different modules of the tolerant retrieval system in order to perform tolerant retrieval.

Proximity Search is a way of specifying the distance between 2 terms in a query. Examples of proximity operators:

- Space means *disjunction* (i.e. OR). Double quotes means phrase search. Exclamation (!) means wildcard. & means AND, and /s, /p, and /k ask for matches in the same sentence, same paragraph or within k words.
- ➔ E.g. Query "disab! /p access! /s work-site work-place (employment /3 place)" may be used to retrieve information about the requirements for disabled people to be able to access a workplace.

Phrase Queries (e.g. "Oslo University" will not match with "University of Oslo").

- **Biwords:** Index every consecutive pair of terms in the text as a phrase.
 - o E.g. Query (ABCDE) becomes dictionary terms (AB), (BC), (CD), (DE) → Then apply merging on these biwords.
 - o Cons: Blowup dictionary size. Create false positives, e.g. (AB) and (BC) might not actually be next to each other.
- **Positional Indexes:** For each postings, also store the positions of the term in the doc.
 - o Requires lots of memory (the index size will depend on the average document size). But very useful for phrase and proximity search (note that phrase queries are the special case of proximity queries, where $k = 1$).
 - o For k proximity searches, intersect 2 postings lists like normal. But for every hit (i.e. every time we have a match for 2 postings, we intersect the positional lists of the postings as well.
 - o Let p_1 and p_2 be some positions in the 2 positional lists. For each pair (p_1, p_2) , we have a hit if $|p_1 - p_2| \leq k$. It means we need to use a double loop (for/while). So merging 2 positional lists of size m and n takes $\mathcal{O}(mn)$ time.
- **Combination Schemes:** Mostly use positional indexes. But biwords can be used for extremely common phrases, like names (e.g. Bill Gates, Palo Alto, Ford Credit, The Walking Dead, etc.)

Wildcards may result in the execution of many merging operations, since there are now lots and lots of postings lists to be intersected against each other (e.g. $se*ate \wedge fil*er$). Thus we want to discourage users from using this functionality.

- If data-structure for the dictionary is a hash table, then we can use ***n*-gram** to handle wildcards, which is very efficient.
- If data structure for the dictionary is something other than the hash table (e.g. a trie, a binary tree, a *b*-tree, or a suffix array) which supports prefix searches, then ***permuterms*** can be used to handle wildcards more efficiently.

Wildcards for Hash Tables → ***n*-gram**:

- Enumerate all sequences of *n* characters occurring in any terms. These sequences are called ***n*-grams**, and will be stored on a ***second*** inverted index that maps each ***n*-gram** to the dictionary ***terms*** that contain that particular *n*-gram.
- Let ***mon**** be the query. On the ***second*** inverted index, merge the 3 lists: ***\$m \wedge mo \wedge on***. This will find all dictionary terms that have that prefix, and also other terms as well (e.g. ***moon***). These other terms will be post-filtered out, such that only terms that have ***mon*** as prefix will be looked up on the normal dictionary stored on the hash-table.

Wildcard for Other Data Structures → ***Permuterms***:

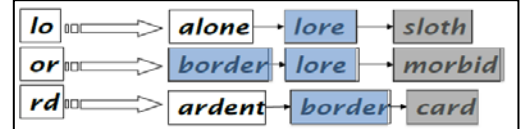
- ***Prefix search***: E.g. ***mon**** retrieves all words in range ***mon ≤ w < moo***.
 - o Easy with binary tree, *b*-tree, or a trie, because words of the same prefix all share the same branch.
 - o Easy with suffix array, because suffixes are sorted alphabetically, so words of the same prefix are next to each other.
 - ***Suffix search***: E.g. ****mon*** retrieves all words in range ***nom ≤ w < non***. Strategy is to maintain an additional dictionary (using the same data structure, whatever that may be) in order to store all the terms in the ***backwards*** order.
 - ***Infix search***: Perform a prefix search and filter out the results by checking if the suffix is of the right form.
- ➔ Performing suffix and infix search like above is very ***inefficient*** in both time and space. Solution is to use ***permuterms*** (which will be stored in the dictionary) in order to transform all the suffix/infix searches into prefix searches.
- o For a dictionary term, ***rotate*** it to generate all the permuterms → E.g. *yes* gives *yes\$, es\$y, s\$ye*.
 - o For a wildcard query, rotate it such that ******* is the last symbol (i.e. transforming any query into a prefix search). Some examples of the rotations are $\{X \rightarrow X\$ \}, \{ *X \rightarrow X\$ * \}, \{ X * Y \rightarrow Y \$ X * \}, \{ * X * \rightarrow X * \}, \{ X * Y * Z \rightarrow Z \$ X * \}$ and then filter.
 - o For English, storing all the permuterms (instead of just the terms) blow up the dictionary size by approx. 4 times.

Spelling Correction:

- Either correct typos in the docs (especially for OCR documents) or in the queries.
 - o Use domain-specific knowledge (e.g. *O/P* confusion on qwerty keyboard, and *O/D* confusion on OCR documents).
 - o Correct typos in queries. Then either retrieve all the correct docs (which is time consuming), or use "Did you mean?".
- ***Isolated Word Correction***: Correct misspelling without considering contexts (e.g. *'form'* & *'from'* are not misspelled).
 - o Relies on some kind of hand-maintained standard dictionaries (e.g. Cambridge), or an industry-specific dictionary.
 - o Goal: Given a lexicon and a string *Q*, return the words in the lexicon closest to *Q*. Edit distance and *n*-gram overlap (or the combination of them) are the main ways to do this.
- ***Context Sensitive Correction***: Suggest corrections for each of the query terms, then permute ***all*** the possible phrases.
 - o E.g. For query *"flew form Oslo"*, examples of possible phrases are *"flea from Osli"*, or *"fled form Oslo"*, etc.
 - o Present to the users (with "did you mean?") only the phrases with lots of hits.

Spelling Correction (n -gram):

- E.g. Let S_q & S_t be the sets of trigrams for the query and dictionary term (e.g. "november" and "december").
 - o Then $S_q = \{nov, ove, vem, \mathbf{emb}, \mathbf{mbe}, \mathbf{ber}\}$ and $S_t = \{dec, ece, cem, \mathbf{emb}, \mathbf{mbe}, \mathbf{ber}\}$.
 - o *Jaccard coefficient (JC)* gives a measure for the degree of overlapping $\rightarrow JC = |S_1 \cap S_2| / |S_1 \cup S_2| = 3/9 = 1/3$.
 - o Usually, we also have a threshold for JC, e.g. declare a match if $JC \geq 0.75$.
- Splitting both S_q & S_t into n -grams during query time is inefficient. Instead, keep a second inverted index that maps each n -gram to the dictionary terms that contain that particular n -gram (exactly like for wildcard queries).
 - o This allows us to use the usual merging algorithm to declare a match if the threshold is, e.g. $JC \geq 2/3 = 0.67$.
 - o E.g. Let the query term be "lord". Thus $S_q = \{lo, or, rd\}$. Intersect these 3 lists such that a term is accepted if it appears in at least 2 out of the 3 lists (in order to satisfy $JC \geq 2/3$).



Spelling Correction (Edit Distance): An edit distance is the minimum no. of operations to convert a string S_1 to S_2 .

- **Levenshtein** distance: There are 3 operations which are Insert, Delete, Replace.
 - **Damerau** distance: There are 4 operations which are Insert, Delete, Replace, Transposition (i.e. swapping operation).
 - **Weighted** distance: Same as either of the previous 2, but an operation is given a weight that depends on the characters involved (e.g. for OCR docs, replacing m by n should give smaller edit distance compared to replacing m by p).
- ➔ All types of edit distance can be computed by **dynamic programming**, implemented on an *edit table*. Here we assume that both the query term ($s = s_0 \dots s_m$) and the dictionary term ($t = t_0 \dots t_n$) are known beforehand.

	$\varepsilon(t_0)$	$C(t_1)$	$A(t_2)$	$T(t_3)$
$\varepsilon(s_0)$	0	1	2	3
$A(s_1)$	1	(1,0,1) 1	(1,1,2) 1	(1,2,3) 2
$C(s_2)$	2	(2,1,1) 1	(1,1,1) 2	(2,1,2) 2
$T(s_3)$	3	(3,2,1) 2	(2,1,2) 2	(2,2,2) 2

	$\varepsilon(t_0)$	$C(t_1)$	$A(t_2)$	$T(t_3)$
$\varepsilon(s_0)$	0	1	2	3
$A(s_1)$	1	(1,0,1) 1	(1,1,2) 1	(1,2,3) 2
$C(s_2)$	2	(2,1,1) 1	(1,1,1) 1*	(2,1,2) 2
$T(s_3)$	3	(3,2,1) 2	(2,1,2) 2	(2,2,2) 1*

LHS is Levenshtein distance. RHS is Damerau distance. Bold number in cell $[i, j]$ is the edit distance $D[i, j]$ for that cell. And (x, y, z) for cell $[i, j]$ are edit distances of the (left, diagonal, top) cell with respect to $[i, j]$.

- For LHS, we have $D[i, j] = \min((x + 1), (y + Q[i, j]), (z + 1))$.
- For RHS, we have $D[i, j] = \min((x + 1), (y + Q[i, j]), (z + 1), (R[i, j]))$.
- If $s_i = t_j$, then $Q[i, j] = 0$. But if $s_i \neq t_j$, then $Q[i, j] = 1$.
- If $s_i = t_{j-1} \wedge s_{i-1} = t_j$, then $R[i, j] = D[(i - 2), (j - 2)] + 1$. Otherwise, $R[i, j] = \infty$.

- ➔ Alternatively, we can also use **bit parallelism** to compute the Levenshtein/Damerau edit distance. The algorithm is to represent the edit table as a set of horizontal/vertical bit vectors (assuming unit edit costs, not weighted edit costs), such that the computations become fancy and very fast bit masking/shifting operations.

➔ But how do we use the edit distance to handle a *misspelled query*? Here we have several methods.

- **Method 1:** (Very slow). Compute edit distance between the query with every dictionary term.
- **Method 2:** (Better than 1). Here we use n -gram (with threshold) to filter out most of the dictionary terms. Then find the edit distance between query and the filtered terms. Present to users only the terms with smallest edit distances.
- **Method 3:** (Very slow). Enumerate all strings that are within a preset edit distance (k). Out of these strings, present to users only those that are actually in the dictionary (assuming that the dictionary only has correctly spelled words).
- **Method 4:** (Better than 3). If dictionary is stored as a *trie*, we can use *dynamic programming* and *Ukkonen cutoff* to efficiently compute all dictionary terms that are within a preset edit distance (k) of the query term. The idea is to traverse the trie, while computing the edit table (column by column) at the same time. Since all strings below a node in the trie share the same prefix, so it must share the same column in the edit table. So every time we go down a node, we have to compute the edit distances for the next column in the table. If at any time the edit distances grow bigger than k , reject that computation branch (i.e. node/column) and move on to the next branch. Otherwise, continue with that branch until we reach the leaf node (where the corresponding dictionary term will be presented to users as the correctly spelled word). So basically, the Ukkonen's strategy is to cut off the search space as soon as k is reached.

Soundex: Words (mostly names) that sound the same but spelled differently should be hashed to the same value for searching.

- Common algorithm is to hash both dictionary terms and query terms into 4-character codes. This is not always accurate.
 - (1) Retain the first character of the word. For all other character, we map as follow: $\{0 \leftarrow A, E, I, O, U, H, W, Y\}$, $\{1 \leftarrow B, F, P, V\}$, $\{2 \leftarrow C, G, J, K, Q, S, X, Z\}$, $\{3 \leftarrow D, T\}$, $\{4 \leftarrow L\}$, $\{5 \leftarrow M, N\}$, $\{6 \leftarrow R\}$.
 - (2) For each pair of consecutive identical digits, remove one of them. Then remove all zeros.
 - (3) Pad the resulting string with trailing zeros and return the first 4 positions.

➔ E.g. "Emelie" and "Emily" both hash to the same value of E540.

Index Construction

Indexer: Simplest algorithm is to split the docs into term-docID pairs, sort the pairs (first by term, then by docID), and then make a posting list for each term (by collapsing the similar occurrences of the same term). However, the main memory might not big enough for the sort, so the solution is to use BSBI or SPIMI. Both will index the documents in chunks (i.e. create an index for each chunk and store the intermediate results on the disk), and hence will require a merging algorithm in the end.

BSBI (block sort-based indexing):

- (1) Divide all documents into several blocks. Block size is chosen carefully to permit a fast in-memory sort.
 - (2) For each block, parse each document into termID-docID pairs.
 - (3) *Inversion:* Sort the termID-docID pairs (first by termID, then by docID). Then make a postings list for each termID.
 - (4) Write the entire index for this block into a file on the disk.
- ➔ If T is the total no. of termID-docID pairs, then BSBI takes $\mathcal{O}(T \log T)$ time, which is the complexity of quick sort. But due to slow disk seeks, the actual indexing time is dominated by the parsing of the documents and the final merge.
- ➔ BSBI needs extra data structure for term-termID mapping, which might not fit in memory → SPIMI to the rescue!

SPIMI (single pass in-memory indexing):

- (1) Divide all documents into several blocks. Block size is chosen carefully to store the postings lists for the entire block.
 - (2) For each block, parse each document into a *stream* of term-docID pairs. So while parsing a document, we will make a posting for each pair and add that posting directly to the appropriate posting list.
 - (3) Sort the inverted index (i.e. postings lists) by terms in lexicographical order. Write the entire index onto disk.
- ➔ Difference between BSBI and SPIMI is: While BSBI sorts all postings first before creating the postings lists for the inverted index, SPIMI updates the postings lists as it parses the documents (by adding the postings directly to the lists).
- SPIMI is faster because there is no sorting required.
 - SPIMI saves memory. It doesn't need to keep term-termID mapping, because it works directly with term-docID.
- ➔ SPIMI takes $\mathcal{O}(T)$ because sorting of term-docID pairs is not required, but we still need to parse them.

Simultaneous merging after performing BSBI or SPIMI:

- Merging is done by *simultaneously* maintaining the read buffers for all the block files (which store indexes to be merged), and a write buffer for output file. Refill each read buffer if needed. The buffers are to reduce no. of disk seeks.
- In each iteration, select the lowest term (or termID) that has not yet been processed. All postings lists for this term are read and merged. The merged list will then be written back to disk.

Index Compression

Heap's Law is an empirical law which states that $|V| = kT^b$ (where $|V|$ is the vocabulary's size, T is the no. of tokens in the collection, $30 \leq k \leq 100$, and $b \approx 0.5$). Thus, $\log|V| = \log k + b \log T \rightarrow$ the log-log plot gives the slope of $b \approx 0.5$. So the Heaps' law basically gives the relationship between the vocabulary size and the no. of tokens.

Zipf's Law is an empirical law about relative frequencies of terms (indirectly related to distribution of terms in a collection).

- Define collection frequency of a term as the total no. of occurrences of that term in the collection (counting multiple occurrences). Let cf_i be the collection frequency of the i th most frequent term (note that i is basically like a ranking).
- In words, this law states that for a typical collection, *frequent* terms are few, but *rare* terms are many.
- Mathematically, this law states that $cf_i \propto 1/i \Leftrightarrow cf_i = k(1/i)$ for some constant k . Thus $\log(cf_i) = \log(k) - \log(i)$.

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, $k = 4$	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ -encoded	101.0

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	$\Delta\%$	cumul %	Size (K)	$\Delta\%$	cumul %	Size (K)	$\Delta\%$	cumul %
Unfiltered	484			109,971			197,879		
No numbers	474	-2	-2	100,680	-8	-8	179,158	-9	-9
Case folding	392	-17	-19	96,969	-3	-12	179,158	0	-9
30 stopwords	391	-0	-19	83,390	-14	-24	121,858	-31	-38
150 stopwords	391	-0	-19	67,002	-30	-39	94,517	-47	-52
stemming	322	-17	-33	63,812	-4	-42	94,517	0	-52

Note that preprocessing of documents (e.g. numbers elimination, case folding, stop words, stemming) will give us a *lossy* compression.

Dictionary Compression

Fixed-width Terms vs. Dictionary-as-a-string:

- Simplest way to store a dictionary is to allocate a fixed no. of bytes (e.g. 20 characters) for each term in the dictionary.
 - o Each term needs 20 bytes for the string, 4 bytes for the frequency, 4 bytes for postings list pointer → **28 bytes/term**.
 - o Bad as it can't handle very long terms, and bad for compression because average word length is only ≈ 8 characters.
- ➔ Main technique for dictionary compression is to store all the terms as a very long continuous string S . To find a term stored in S , we will use pointers in order to locate the position of that term in the string.

Without Blocking:

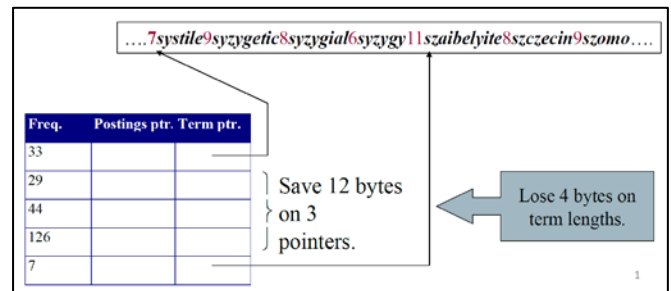
Associate the location of each *term* with a unique pointer.

- Since S has $|S| \approx 8T$ unique positions (where 8 is average word length, and T is the vocabulary size), thus we need to use pointers of $\log_2(|S|)$ bits. So each term needs 8 bytes (for the term string), 4 bytes (for the frequency), 4 bytes (for postings list pointer), and $\log_2(|S|) \approx 4$ bytes (for the pointer to S) → **20 bytes/term**.
- Without blocking, we can simply do a binary search to search for a term, because each term has its own pointer.
- ➔ *Searching*: Since each term has its own pointer, we can simply do a binary search to find a term (which is very fast).

Blocking:

Associate the location of each *block* (of size k) with a unique pointer. In addition, store the term length in front of every *term* in S . These lengths will then be used to search for a term in a particular block.

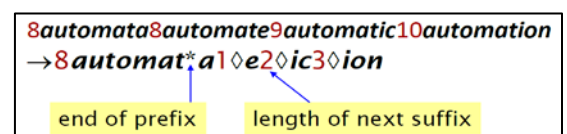
- Storing the term length requires **one** extra byte for each term.
- Let $k = 4$ and $\log_2(|S|) \approx 4$ bytes. Then a block of k terms need $8k$ bytes (for the term string), $4k$ bytes (for the frequency), $4k$ bytes (for the postings lists pointers), 4 bytes (for the pointer to S), and k bytes (for the term lengths) → $17k + 4$ bytes in total, which gives **18 bytes/term**.



- ➔ *Searching*: Do a binary search to find the block, and then do a linear search through the terms within the block. So if the block size k is too big, it might hurt the time performance even though it might save some memory space.

Front Coding:

Best to be done in blocks (i.e. combining with the blocking method). Sort the words in the dictionary such that terms of each block share the same prefix (so that the prefix is stored only once).



Postings Compression

Compression Schemes:

VB is the simplest and often good enough. Rice/Golomb gives better compression but slower than VB. Gamma is only good for small numbers and also very slow. S9 is complex but it's the best (both for space and speed).

Compression with Gaps:

Assume a posting is just a docID. Let N be no. of docs in the collection. Then a docID may need $\log_2(N)$ bits. Our goal is to use *less* than that. This can be achieved by only storing the **gaps** (since the gaps tend to be smaller, and more compression can be done on the gaps as well) → E.g. {34,47,160,322} becomes {34,13,113,162}.

VB Compression (variable bytes):

- Idea: Storing gaps with fix-sized integers would waste a lot of space, because the gaps are small for frequent terms but big for rare terms (e.g. "pout" may use ~ 20 bits/gap, while "the" may use ~ 1 bit/gap). So our goal is to use the technique similar to UTF8 and use variable no. of bytes to store different gaps.
 - Algorithm: For each and every byte, dedicate the first bit as the *continuation* bit c (so that $c = 1$ for the last byte, and $c = 0$ for all other bytes). Fill the rest of the bits with the binary codes of the gap G .
 - o E.g. $33549 = 2 \times 128^2 + 6 \times 128^1 + 13 \times 128^0 = (00000010\ 00000110\ 10001101)_2$.
- ➔ If $128^{k-1} \leq G < 128^k$ where G is the gap in decimal, then VB coding will use k bytes for G .
- ➔ VB wastes space if there are many small gaps. But the byte-alignment makes it very efficient/fast to process.
- ➔ Note that we can also use a different "*unit of alignment*", e.g. 32 bits (words), 16 bits, or 4 bits (nibbles).

Gamma (γ) Compression:

- To encode number N in unary, use N occurrences of 1's followed by 0 as the final digit. E.g. 3 can be encoded as 1110.
 - Each γ code is made of 2 parts (*offset* and *length*). The offset is simply the gap G in binary without the most significant bit. The length is the length of the offset. So the total length of a γ code is equal to $2 \log_2(G) + 1$.
- ➔ Since γ coding works at the bit level, it's good for small gaps but very inefficient because it doesn't have byte alignment.
- ➔ Note that γ coding is a parameter-less coding scheme (because it has no parameters like b as in Rice/Golomb coding).

Rice/Golomb Compression:

- Compute g as the *average* of the numbers/gaps to be encoded. E.g. $g = \text{average}(34, 13, 113, 162) = 81$
 - Compute b as the *parameter* of the coding, which is dependent on the average g .
 - o Rice ➔ Round g down to the closest power of 2. So for $g = 81$, we have $b = 64$.
 - o Golomb ➔ Choose $b \approx 0.69g$. So for $g = 81$, we have $b = 56$.
 - For each number/gap X , encode it as (X/b) in unary followed by $(X \bmod b)$ in binary.
 - o Rice ➔ $162 = 2 \times 64 + 34 = (110\ 100010)$
 - o Golomb ➔ $162 = 2 \times 56 + 50 = (110\ 110010)$
- ➔ Simple to implement with bitwise operations, but inefficient due to alignment at bit levels.
- ➔ Parameter b can either be chosen *locally* (i.e. for each postings list) or *globally* (i.e. for the entire index).

Simple9 (S9) Compression: Pack as many numbers/gaps as possible into a single word-aligned code (i.e. 32 bits alignment).

- Algorithm: Each word is split into 4 control bits and 28 data bits. For each word, there are 9 cases to choose from.
 - o Cases: (1 number 28-bit), (2 numbers 14-bit), (3 numbers 9-bit and wasted 1), (4 numbers 7-bit), (5 numbers 5-bit and wasted 3), (7 numbers 4-bit), (9 numbers 3-bit and wasted 1), (14 numbers 2-bit), (28 numbers 1-bit).
 - o The 4 control bits are used to distinguish the 9 cases above, assuming that every number/gap uses less than 28 bits.
 - o Note that the control bits are always stored in the beginning of each word, before any data bits.
 - o E.g. $\{34, 13, 113, 125\} \rightarrow \{0100\ 0100010\ 0001101\ 1110001\ 1111101\} \rightarrow 4$ numbers 7-bit.
- ➔ S9 is very fast because of word-alignment and efficient bit masking operations. It also gives good compression.

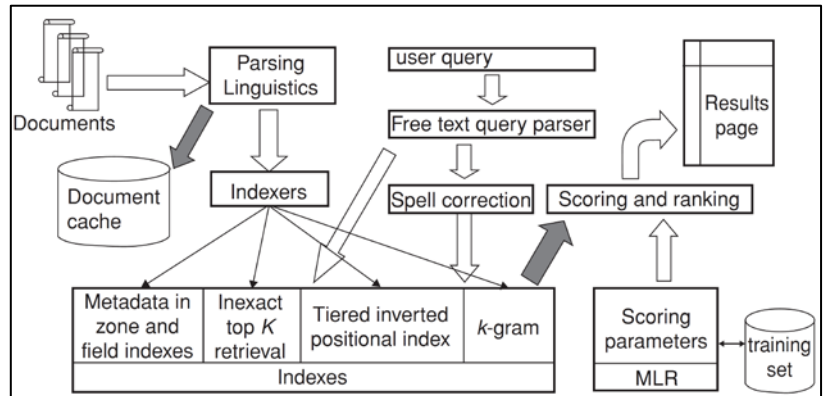
Example: Compress {34, 13, 113, 162}.

- {10100010, 10001101, 11110001, 00000001 10100010} → VB coding.
- {111110 00010, 1110 101, 1111110 110001, 11111110 0100010} → Gamma coding.
- {0 100010, 0 1101, 10 110001, 110 100010} → Rice coding.
- {0 100010, 0 1101, 110 1, 110 110010} → Golomb coding.

Ranked Retrieval Model

Benefits of ranking over boolean retrieval are:

- Most people can't write boolean queries.
 - We often want highly relevant documents first, so we don't want exact matching.
 - Large result sets are not an issue, because we can just show the top k results.
- ➔ MLR (machine learned ranking) can be used to learn which scoring function is the best.



Vector Space Model

Basic Scoring on Vector Space:

- Represent each document and each query as a vector of dimension $|V|$ (where V is the vocabulary). Depending on the scoring function, weights of these vectors can be computed accordingly → E.g. 1 & 0 as weights (as in incidence matrix), or term frequency as weights (as in count matrix), or TFIDF as weights (most common).
- Note that instead of using a matrix (in which most of the weights are 0), we often use postings lists to represent the vectors such that only nonzero weights are stored in memory, thus saving some space.
- Scoring function $\text{Score}(q, d)$ aims to compute the query-doc similarity. This is done either by JC or by cosine scores.
- Jaccard coefficient can be used to measure query-doc overlaps → $\text{Score}(A, B) = \text{JC}(A, B) = |A \cap B| / |A \cup B|$.
 - Compute JC between the query and each of the docs. Then use those to compare/rank the docs against each other.
 - JC computation is simple and fast, but it completely ignores term frequency and document frequency.
- Cosine scores can be used to measure the angle between the 2 vectors, which reflect the proximity between them.
 - Note that *angles* are preferred over *absolute distance* between 2 vectors, because absolute distance is very sensitive to the lengths of the vectors rather than just the directions. Let d_1 and d_2 be the 2 documents, so that d_2 has the exact same content as d_1 but the content is repeated multiple times. Then d_2 is much longer than d_1 even though they point to the same direction, i.e. $|d_2| > |d_1|$ but $\text{cosine}(d_1, d_2) = 1$.
 - For range $[0^\circ, 180^\circ]$, the smaller the angle, the larger the cosine, the closer the 2 vectors, and the higher the relevance.

$$\text{Score}(q, d) = \text{cosine}(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{|\mathbf{q}| |\mathbf{d}|}$$

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

Term-Doc Count Matrix: Each of the weights represents the no. of time a term occurs in a doc. Each column is a **vector** in $\mathbb{N}^{|V|}$ space (where the vocabulary size $|V|$ gives us the dimension of the space).

Generic Algorithm for Cosine Computation:

- This algorithm uses TAAT (term-at-a-time), and a double for-loop (line 3 to 6) to compute all the dot products.
- Fetching the postings list (line 4) is necessary because we only need to consider docs with at least 1 of the query terms.
- $Length[N]$ is an array of N elements, with each representing the length of the vector describing the corresponding doc.
- $w_{t,d}$ and $w_{t,q}$ are the weights for the term t on vectors \mathbf{d} and \mathbf{q} .

```

COSINESCORE(q)
1  float Scores[N] = 0
2  float Length[N]
3  for each query term t
4  do calculate  $w_{t,q}$  and fetch postings list for t
5      for each pair( $d, tf_{t,d}$ ) in postings list
6      do Scores[d] +=  $w_{t,d} \times w_{t,q}$ 
7  Read the array Length
8  for each d
9  do Scores[d] = Scores[d] / Length[d]
10 return Top K components of Scores[]

```

TFIDF Scoring

Term Frequency (TF): Let $tf_{t,d}$ be the term frequency (i.e. no. of times term t occurs in document d).

- Relevancy generally doesn't increase proportionally with term frequency (e.g. a doc with 100 occurrences of a query term is more relevant than a doc with 1 occurrence, but not 100 times more relevant). Solution is to use logarithms.

$$Score(q, d) = \sum_{t \in (q \cap d)} (w_{t,d}) \quad \text{where} \quad w_{t,d} = \begin{cases} 1 + \log(tf_{t,d}), & \text{if } tf_{t,d} > 0 \\ 0, & \text{if } tf_{t,d} \leq 0 \end{cases}$$

Inverse Document Frequency (IDF): Since rare terms are more informative than frequent terms (e.g. stop words), we need to use IDF to put more weights on the rare query terms and less weights on frequent query terms.

- Let df_t be the document frequency (i.e. no. of docs containing t), which is simply the length of the postings list for t .
- Define $idf_t = \log(N/df_t)$, where N is the total no. of documents and the logarithm is used to "dampen" the effect. Note that idf_t measures the informativeness of the query term t (i.e. rare terms tend to have high IDF values).

➔ Since IDF compares the informativeness of the query terms against each other, thus IDF has no effect on *one*-term queries.

➔ E.g. For the query "good phone", the IDF weighting will make occurrences of "phone" much more important than "good".

TFIDF Weighting is the best weighting scheme in IR. It's basically the combination between TF and IDF weighting schemes. The base of the logarithms doesn't matter, because it's just a constant and thus doesn't affect the ranking between the docs.

$$Score(q, d) = \sum_{t \in (q \cap d)} (tfidf_{t,d}) \quad \text{where} \quad tfidf_{t,d} = \log(1 + tf_{t,d}) \times \log\left(\frac{N}{df_t}\right)$$

SMART Notation: Since TFIDF scoring has many variants, this notation uses the acronyms given in the table to indicate which combinations we want to use. Combination *ddd.qqq* has the 1st part dedicated for the document vector, and the 2nd part dedicated for the query vector. The variant *lnc.ltc* is the most common in IR.

Term frequency		Document frequency		Normalization	
n (natural)	$tf_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(tf_{t,d})$	t (idf)	$\log \frac{N}{df_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \dots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-df_t}{df_t}\}$	u (pivoted unique)	$1/u$ (Section 6.4.4)
b (boolean)	$\begin{cases} 1 & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^\alpha$, $\alpha < 1$
L (log ave)	$\frac{1 + \log(tf_{t,d})}{1 + \log(\text{ave}_{t \in d}(tf_{t,d}))}$				

Optimizations for Efficient Ranking

Static Quality Scores:

- Cosine scoring is a query-dependent *dynamic* scoring scheme that measures *relevancy* of a doc with respect to a query.
 - Quality scoring (e.g. PageRank, no. of citations, reputations, endorsement, etc.) is a query-independent *static* scoring scheme that measures the *authoritativeness* of the documents. Static score for document d is denoted by $g(d)$.
- ➔ Remember to combine the static and the dynamic scores together, i.e. $NetScore(q, d) = g(d) + \text{cosine}(q, d)$.

DAAT (document at a time) vs. TAAT (term at a time):

- Given a query, DAAT computes the cosine score (i.e. the dot product) for one document at a time.
 - o Cosine computation needs the TFIDF weights for all the terms in document. This requires a concurrent traversal through all postings lists, so that cosine score for a document can be completely computed before going to the next.
 - o Due to the concurrent traversal, postings lists must have some *common* ordering, either by static score $g(d)$ or by docID. Common ordering means for any $d_1 < d_2$ on a postings list, we also have $d_1 < d_2$ on any other postings list.
- Given a query, TAAT computes the cosine contribution from one *query term* at a time for all the docs in the collection, before moving on to the next query term. Values accumulated at the very end will be the cosine scores for the docs.
 - o No concurrent traversal is needed, since evaluating 1 term at a time also means processing 1 postings list at a time.
 - o TAAT can thus handle any types of ordering, even non-common ones like ordering by TFIDF weights.

Main Optimization Techniques for Efficient Ranking:

- Use a heap (binary tree where every node is \geq than the children) to quickly choose the top K docs with the largest scores.
- Compute cosine scores for as few documents as possible (note that cosine computation is very expensive). This can be done by *index elimination* (during query time) and/or *champion lists* (during index build time).
 - o As cosine scores are not computed for all the docs, this may give us an *imprecise* top K selection (i.e. inaccurate).
 - o Generic approach is to find set A of *contenders* (where $K < |A| \ll N$), which consists only of docs that we will compute the cosine scores for. Note that A might not contain the top K (hence the "imprecise" selection).
- Do cluster pruning during preprocessing ➔ Pick \sqrt{N} docs at random to be used as *leaders*. For every other doc (*follower*), use the doc-doc cosine scores to precompute the nearest leader. Each cluster has 1 leader and $\approx \sqrt{N}$ followers. Given a query, compute the cosine scores with the leaders to find the nearest leader L . Then return K nearest docs from L 's cluster.

Index Elimination: Generic algorithm for cosine computation already considers only docs that has at least 1 query term (as all other terms have 0 as weights anyway). However, we can still eliminate a little further by doing the following:

- For multi-term queries, consider only query terms with high IDF scores. This is because terms with low IDF scores will have lots of documents, thus forcing us to compute the cosine scores for more documents than necessary.
- For multi-term queries, only compute cosine scores for docs containing several of these query terms (e.g. at least 3 out of 4). To implement this, perform a conjunction for all the postings lists. If any of the docs appears in at least 3 of the postings lists, then we compute the cosine score for it.

Champion Lists: Order the docs in postings lists by some kind of weights $\omega_{t,d}$. Use DAAT if $\omega_{t,d} = g(d)$. Use TAAT if $\omega_{t,d} = \text{tfidf}_{t,d}$. Then for each dictionary term t , precompute a champion list by picking out r docs of the highest weights.

- It's possible that $r < K$ because r is chosen at index build time. Solution is to use a tiered index, where each postings list is broken up into a *high* list (i.e. champion list) and a *low* list (which contains the rest of the docs in the postings list).
 - During query time, only use the top tier (i.e. high list). If that fails to yield K docs, we drop down to the low list.
- ➔ The 2 techniques (champion lists & index elimination) can be combined such that at query time, we use index elimination to efficiently compute scores only for the docs in the champion lists, thus finding the top K docs from the champion lists.

Impact Ordered Postings and Optimization for TAAT:

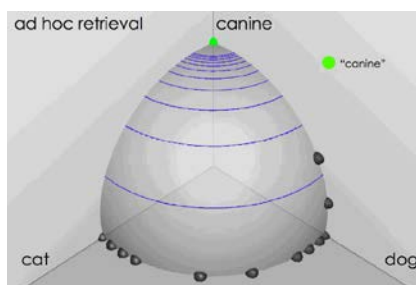
- Impact ordering is when postings are arranged in decreasing order by some sort of dynamic weights (e.g. $\omega_{t,d} = \text{tfidf}_{t,d}$).
- With impact ordering, there are 2 *optimizations* that can be done when accumulating cosine scores with TAAT.
 - o For each postings list, stop either after we have already got K docs, or after $\omega_{t,d}$ has dropped below some threshold.
 - o Accumulate scores for query terms with the highest IDF values first, because these terms will contribute more to the final scores. As we update score contribution from each query term, stop if the scores stay relatively unchanged.

Techniques for Improving Results

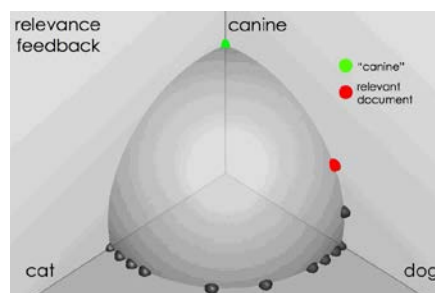
There are 2 main types of analysis that can be done at *query* time in order to improve results and achieve high recall.

- *Local* dynamic analysis (i.e. analysis of only the results set): Relevance feedback (RF), and pseudo RF.
- *Global* static analysis (i.e. analysis of the entire collection): Query expansion (QE) with thesauri.

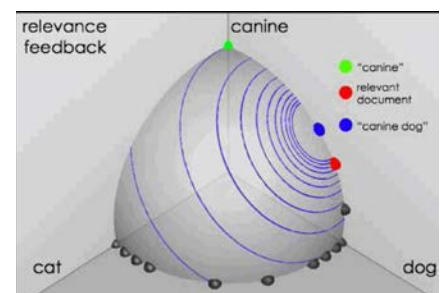
Relevance Feedback (RF)



Ad-hoc results for query "canine"
(i.e. regular retrieval without RF)



User feedback



Results after RF

Relevance Feedback is to use user feedback about relevancy (relevant/nonrelevant) of the docs in the results set in order to compute a better results set, thus improving recall/precision (especially if RF is allowed to go through many iterations).

- RF *assumes* that users have sufficient knowledge about the collection to provide the initial query. However:
 - o Users may use different spellings (e.g. Traikoski ↔ Tchaikovsky) → Solution is to use spelling correction.
 - o Users may use different vocabulary (e.g. cosmonaut ↔ astronaut) → Solution is to use asymmetric expansion.
 - o Users may use the wrong language, and vectors for different languages can be very different from one another.
- RF *assumes* that term distributions of the relevant docs will be similar to each other. It means that relevant docs should cluster around a centroid prototype, which is far away from the centroid prototype for nonrelevant docs. This may not work if relevant docs are in *multimodal* class (i.e. docs belonging to several clusters), which can occur when:
 - o There are synonyms (e.g. relevant docs belonging to both "Burma" cluster and "Myanmar" cluster).
 - o Results are taken from disjunctive sets (e.g. docs about "Phone" may be in both "Sony" cluster and "Nokia" cluster).
 - o Solution → Using document tags that link to other similar clusters' prototypes may help addressing this issue.
- Cons: Most users hate having to judge the relevancy. They often choose *queries reformulation* as the alternative.
- Cons: RF increases response time, because interactions with the users means that RF must take place during the search.
- ➔ E.g. Features like "similar/related page" or "more like this" are examples of RF.
- ➔ *Pseudo RF* is the automated variant of RF. It assumes that some top K documents are the relevant ones rather than having to gather feedback from users. Works well for most cases, but may go horribly wrong for some other cases.
- ➔ *Implicit RF* is a variant of RF, which works a bit better than pseudo RF. It assumes that clicks indicate relevancy.
 - o Based on the assumption that the document summaries displayed to users are indicative of relevancy.
 - o Based on statistics (user clicks over time), which are gathered globally rather than being query-specific.
- ➔ *Centroid $\mu(S)$* is a vector/point representing the center of mass for all document vectors/points (\mathbf{d}) in collection S .

$$\mu(S) = \frac{1}{|S|} \sum_{\mathbf{d} \in S} \mathbf{d}$$

Rocchio Algorithm: Users judge relevancy of docs that were returned for the initial query \mathbf{q}_0 . The feedback will give us S_r (the set of known relevant docs) and S_{nr} (the set of known nonrelevant docs). The algorithm then seeks a new query \mathbf{q}_m that theoretically moves towards relevant docs and away from nonrelevant docs. Finally, new results set will be returned for \mathbf{q}_m .

- Computation of \mathbf{q}_m is done by the formula below. Note that if all negative entries of \mathbf{q}_m will be ignored (i.e. set to 0).

$$\mathbf{q}_m = \alpha \mathbf{q}_0 + \beta \mu(S_r) - \gamma \mu(S_{nr})$$

- Coefficients α, β, γ can either be hand-chosen or set empirically by ML. For most cases, the higher β/γ ratio, the better.
 - o Negative RF occurs when \mathbf{q}_m gets closer to $\mu(S_{nr})$ and away from $\mu(S_r)$.
 - o Positive RF occurs when \mathbf{q}_m gets closer to $\mu(S_r)$ and away from $\mu(S_{nr})$. Use $\gamma = 0$ if we want only positive RF.

Evaluation of RF Effectiveness:

- *Method 1* (best): Measure how fast users find relevant docs with RF. Then compare that to QE or to query reformulation.
- *Method 2*: Perform RF on 2 collections. Compute precision/recall of \mathbf{q}_0 for the 1st collection, and \mathbf{q}_m for the 2nd one.
- *Method 3*: Compute precision/recall for both \mathbf{q}_0 and \mathbf{q}_m to see if \mathbf{q}_m gives better results. But even if \mathbf{q}_m improves the precision/recall, it may be due to known relevant docs (judged by users) are now ranked higher.

- *Method 4*: Address the weakness of method 3 by using the residual collection, such that docs that have been judged will not be included in precision/recall computation. But this method can also be unfair, especially if the residual collection contains very few relevant results, leading to an even lower precision/recall compared to q_0 .

Query Expansion (QE)

Query Expansion (QE): Assist users in expanding/improving a query (through the use of a global thesaurus). In RF, users give feedback (relevant/nonrelevant) on the docs. But in QE, the system possibly suggests additional query terms such that the users can then give feedback (good/bad) on words/phrases to tell the system which expanded query is the best for retrieval.

- System should give additional info (e.g. which are stop words, which are stemmed to what, no. of hits on a query term).
 - Global thesaurus can be used to suggest synonyms to users, which can then be used to expand the original query. Note that use of thesaurus can be combined with term weighting (e.g. weight the added terms less than the original terms)
- ➔ E.g. Features like "also try" or query suggestions are examples of *interactive* QE. There is also an *automatic* QE (in which the expansion is done implicitly without the users being aware of it).

Methods for Building Thesaurus:

- (1) Use query log mining to look at the manual query reformulations of other users to make suggestions for new users.
- (2) Use a manual thesaurus maintained by human editors, who will build up a set of synonyms for each concept, with or without designating a canonical term. E.g. list of medical terminologies and synonyms for medicine.
- (3) Use an automatic thesaurus, which is generated either by using *word co-occurrence* or by doing a *grammatical analysis*.
 - *Method 1*: Assuming that similar words are likely to co-occur in the same document, then we can exploit word co-occurrence statistics. Let $A = [A_{t,d}]$ be the term-doc matrix with length-normalized rows. Then $C = AA^T$, where cell $C_{u,v} = \text{cosine}(\mathbf{u}, \mathbf{v})$ is the similarity score between the 2 terms u and v . Note that word ambiguity can lead to false positives (words deemed similar that aren't) or false negative (words deemed dissimilar that are indeed similar), thus lowering the accuracy (e.g. the query "apple laptop" may be expanded to "apple fruit laptop").
 - *Method 2*: Assume similar words are likely to appear in similar grammatical contexts (e.g. all food items are similar to each other because they can be grown, cooked, eaten, and digested).

Evaluation

Precision (P) and Recall (R) have an inverse relationship, so **F-Measure** is often used as a combined measure. Parameter α specifies the trade-off between P and R . People often want a balanced trade-off, which is when $\alpha = 0.5$ and $\beta = 1$.

$$P = \mathbb{P}(\text{relevant} \mid \text{retrieved}) = tp/(tp + fp)$$

$$R = \mathbb{P}(\text{retrieved} \mid \text{relevant}) = tp/(tp + fn)$$

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

	Relevant	Nonrelevant
Retrieved	True positive (tp)	False positive (fp)
Not retrieved	False negative (fn)	True negative (tn)

A/B Testing: Evaluate the effect of a change in the system, by running a bunch of *single*-variable tests. For each test, only 1 parameter is varied from the current system, so that we can judge if the proposed system gives a positive or a negative effect.

- Only small proportion of traffic is diverted to the proposed system during testing. Current system will still be available.
- ➔ Since A/B testing is easy to deploy, it's often preferred over powerful statistical analysis that tests *multiple* things at once.

User Happiness Scores (UHS):

- For e-commerce, user's buy rates after using the search engine can be used as UHS. Do users leave soon after searching?
- Use relevancy judgments from human raters as UHS. Relevancy can be binary (relevant/nonrelevant) or multi-level. But note that human judgments are often unreliable and inconsistent over time between raters, not to mention the cost of hiring these human raters. Another challenge is how to actually come up with the test queries.
- User clicks can also be used as UHS. However, click rates depend on many other things than just the true relevance of the docs. Misleading titles and result summaries can trick people to click more.

Benchmarking: Evaluation of a search engine requires a test collection, a query set, and an assessment methodology which quantifies UHS into some kind of *goodness* measure (e.g. P@K or NDCG).

Binary Relevance (for benchmarking): Apply P@K or R@K or MAP (mean average precision) only on the top results.

- P@K and R@K are the precision and recall at rank K (i.e. considering only documents from rank 1 up to rank K). Since we have a binary relevance assessment (relevant/nonrelevant), it's relatively easy to calculate the precision and recall.
- AP (average precision) at rank K is defined as $AP_{(K)} = \text{average}(p_1, \dots, p_K)$ where $p_k = P@k$ for all $1 \leq k \leq K$. Note that we only consider relevant documents (i.e. d_k must be marked as relevant for p_k to be taken into account).
- MAP at rank K is defined as the average of all the $AP_{(K)}$ of many different rankings. Note that a ranking is defined as an ordering (i.e. arrangement) of some documents with respect to a particular query.

Multi-Level Relevance (for benchmarking): Relevance score for each document is taken from the range $[0, k]$ for some k .

- *Idea:* Highly relevant docs are more useful than marginally relevant ones. The lower the rank, the less useful it is (since it will less likely be seen by users). So the *gain* (i.e. relevance scores) for lower-ranked docs should be discounted.
- CG (cumulated gain) at rank k is defined as the gains (relevance scores) accumulated from rank 1 up to k .
- DG (discounted gain) of document at rank k is defined as $r/\log(k)$, where r is the relevance score for that document.
- DCG (discounted CG) at rank k is defined as the total DG accumulated from rank 1 up to k . Use any base for the log!

$$DCG(k) = r_1 + \sum_{i=2}^k \frac{r_i}{\log(i)}$$

- NDCG (normalized DCG) of a ranking algorithm is $NDCG = DCG/\overline{DCG}$, where \overline{DCG} is of an ideal ranking algorithm.
 - o An ideal ranking algorithm would rank the documents in the accurate order of decreasing relevance score. This means \overline{DCG} has the maximum value that one could get, hence the word *ideal*.
 - o By the definition given, the normalization will always give NDCG to be within the range $[0,1]$. The higher NDCG is, the closer it is to 1, and the closer the our ranking is to the ideal ranking.

Text Classification

Text Classification Tasks:

- *Spam filtering* and *standing queries* are not about ranking, but they are classification problems (relevant vs. nonrelevant).
- Standing queries are queries that are periodically executed on a collection to which new documents are added over time (thus automatically serve users' ongoing needs) → E.g. Google Alerts (alerting users if it finds new relevant results).

Classification Methods:

- *Manual classification* → Accurate and consistent when done by experts. But difficult/expensive to scale.
 - *Coding a rule-based classifier by hands* → Accurate (if the rules are carefully refined over time by experts), easy to scale for large problem size. But building/maintaining the rules is expensive and requires lots of programming resources.
 - *Machine learning (supervised/unsupervised)* → Create a ML algorithm, which can then be used to establish the rules.
- ➔ E.g. Consider a collection of articles. For 2nd method, we give rules like if an article has the term "proton", then that article is about "science". For 3rd method, machines learn from the training dataset to establish these rules automatically.

Evaluation of a Classifier:

- Low accuracy on testing set means *underfitting*. High accuracy may indicate *overfitting* or *appropriate-fitting*.
- Evaluation metrics (incl. precision and recall) for a single class c :

$P(c) = \frac{\text{no. of docs correctly classified as } c}{\text{no. of docs classified as } c} = \frac{tp}{tp + fp}$	$\text{Accuracy} = \frac{\text{no. of docs correctly classified}}{\text{total no. of docs}} = \frac{tp + tn}{tp + tn + fp + fn}$
$R(c) = \frac{\text{no. of docs correctly classified as } c}{\text{no. of docs belonging to } c \text{ for real}} = \frac{tp}{tp + fn}$	$F\text{-Score}(c) = \frac{2 \times P(c) \times R(c)}{P(c) + R(c)} = \frac{2 \times \text{Precision}(c) \times \text{Recall}(c)}{\text{Precision}(c) + \text{Recall}(c)}$

- Evaluation metrics for the entire classifier (i.e. considering all classes).
 - o *Macro-averaging* → Compute the evaluation values for each class (as in above), and then take the average.
 - o *Micro-averaging* → Sum all evaluation values of all the classes together to make a "pool", and then just compute as if we have the evaluation metrics for only one class (i.e. after summing into a pool, use the formulae above).

		Gold standard	
		Positive	Negative
Our own classification	Positive	True positive (tp)	False positive (fp)
	Negative	False negative (fn)	True negative (tn)

ML Classification

Supervised Learning Classification:

- Let $\mathcal{C} = \{c_1, \dots, c_k\}$ be the set of all available classes/categories. F is called a classifier if $F(d) = c$ (for some $c \in \mathcal{C}$).
- To keep things simple, we mostly consider only *one-of* classification (i.e. a document can only belong to 1 category).
- Training set (hand-classified) is a set of tuples of the form $\langle d, c \rangle$, where document d belongs to class $c \in \mathcal{C}$. It might be difficult to gather enough training data. It takes time to collect, organize, and quality-check the data.
- Our task is to write a learning method that enables the machine to automatically come up with the classifier F . Examples of such learning algorithms are: NB (most common), Rocchio, kNN, and SVM (most powerful).

Feature Selection:

- *Concept drift* refers to the fact that the usefulness of a particular feature may change over time.
- Most common features are the vocabulary terms themselves. But we can also consider other features (such as doc length, domain-specific features, zones, fields, etc.) that can potentially contribute to dividing the data into different regions.
- Some features (e.g. title, abstract) may contribute more to the zoning of the docs, and thus should be *upweighted*.
- Too few features can lead to underfitting (or information loss). Too many features can lead to overfitting (or inaccuracy).
- We need trade-offs between too few and too many features. Examples of feature selection algorithms are:
 - o *Hill-climbing technique* (iterative selection) can be used to automatically select the features.
 - o *Mutual information scores* (similar to NB) measures how much the feature contributes to a good classifier.

Properties of ML Classification:

- Decision boundary can be *linear* (i.e. a hyperplane) or *nonlinear*. Linear classifier gives the result as a linear combination of features, i.e. $F(\mathbf{w}^T \mathbf{x})$ where \mathbf{w} is the weight vector which specifies the coefficients for different features.
 - o Linear classifiers will fail if the data aren't linearly separable. Nonlinear classifiers are often prone to overfitting.
 - o kNN is nonlinear. Rocchio and NB are both linear. As for SVM, there exists both a linear and a nonlinear variant.
- High *bias* means the classifier often gives the wrong predictions. High *variance* means different training datasets may lead to very different classifiers. We often need a trade-off between bias and variance.
 - o kNN is prone to have *high variance* and *low bias*, which may lead to overfitting.
 - o NB is prone to have *low variance* and *high bias*, which may lead to underfitting.
- Time complexity and amount of available training data are important factors when choosing ML classifying algorithm.
 - o kNN requires no training time but a lot of time for classifying. On the other hand, SVM requires lots of training time but not so much for classifying. NB is like a compromise of the 2 extremes, thus is usually preferred.
 - o If there is no data, then design hand-crafted rules. If there is fairly little data, then choose a high-bias linear classifier (like NB). If there are lots of good data, then SVM is probably the best choice.

Naïve Bayes (NB)

Main Idea: Classify document d to class c^* if $P(c^*|d)$ gives the highest probability. Combine that with Bayes rule as well.

$$c^* = \underset{c \in \mathcal{C}}{\operatorname{argmax}}(P(c|d)) \quad \text{where} \quad P(c|d) = \frac{P(d|c) \times P(c)}{P(d)} = \frac{1}{P(d)} P(c) \cdot P(d|c)$$

- Bag-of-words representation represents any document d as a set of terms, i.e. $d = \{w_1, \dots, w_n\}$. Assume that the words w_i are all conditionally independent from one another (hence the *naïve*). Then:

$$P(d|c) = P(\{w_1, \dots, w_n\}|c) \approx P(w_1|c) \dots P(w_n|c) \Leftrightarrow P(c|d) \propto P(c)P(w_1|c) \dots P(w_n|c)$$

- Using logarithms can simplify the computation because addition is easier and less prone to floating-point errors compared to multiplication. So we have $c^* = \underset{c \in \mathcal{C}}{\operatorname{argmax}}(L_c)$, where L_c is the logarithm of the probability that d belongs to c :

$$L_c = \log(P(c|d)) = \log(P(c)P(w_1|c) \dots P(w_n|c)) = \log(P(c)) + \sum_{i=1}^n \log(P(w_i|c))$$

NB Training: Split available training data into 2 parts (80% for training set, and 20% for testing set). Training set can then be used to empirically/statistically estimate each $P(w_i|c)$ and $P(c)$, while the testing set is for testing our classifier.

- $P(c) = N_c/N$ (where N_c is no. of docs belonging to c in the training set, and N is total no. of docs in the training set).
- $P(w_i|c) = M_{c,w_i} / M_c$ (where M_{c,w_i} is no. of occurrences of w_i in all the documents belonging to c in the training set, and M_c is total no. of words in all the documents belonging to c in the training set).

➔ Formulas above give maximum-likelihood estimation. But if w_i never occurs in c , then $P(w_i|c) = 0$ which is not really reasonable, especially with limited training data. We can fix this with Laplace smoothing to achieve multinomial NB model. Let V the vocabulary which contains all the unique terms in the entire training set.

$$P(w_i|c) = \frac{M_{c,w_i} + 1}{\sum_{t \in V} (M_{c,t} + 1)} = \frac{M_{c,w_i} + 1}{\sum_{t \in V} (M_{c,t}) + \sum_{t \in V} (1)} = \frac{M_{c,w_i} + 1}{M_c + |V|}$$

Vector Space Classification

Vector Space Classification aims to find boundaries that can separate different classes into different regions of space.

- Assume that documents from different classes don't overlap too much.
- Assume that docs (represented by points/vectors on the planes) of the same class form a contiguous region of space.

Rocchio Classification:

- During the training, compute the **centroid** (i.e. prototype) for each and every class in the training set.
- To classify document d , find the closest centroid, and classify d to the corresponding class.

K Nearest Neighbor (kNN) requires no training nor feature selection, but needs a lot of time during the classification.

- To avoid ties, we often choose k to be odd (3 or 5 are most common).
- To classify document d , find the k nearest neighbors to d in the vector space. Classify d to the majority class.

➔ E.g. Let $k = 5$, and the neighbors are $\{\langle d_1, c_1 \rangle, \langle d_2, c_2 \rangle, \langle d_3, c_2 \rangle, \langle d_4, c_3 \rangle, \langle d_5, c_4 \rangle\}$. Then classify d to c_2 .

Linear SVM (Support Vector Machine):

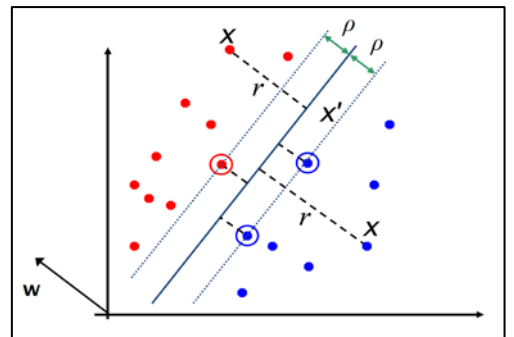
- The goal is to choose a hyperplane that separates data between 2 classes $\{+1, -1\}$ with the maximum margin.
 - o Support vectors are points that lie on the 2 boundaries of the margin. SVM classifier depends only on these vectors.
 - o Small margin means high variance (i.e. the classifier becomes very sensitive to variations in the training data).
- Let \mathbf{w} normal to the chosen plane. Let $\rho = 1/|\mathbf{w}|$ be the margin, so that our goal is to maximize ρ by minimizing $|\mathbf{w}|$.
- Let $y \in \{+1, -1\}$ which depends on which side \mathbf{x} is on. Then by geometry:

$$\mathbf{x} - \mathbf{x}' = y \frac{r\mathbf{w}}{|\mathbf{w}|} \Leftrightarrow \mathbf{x}' = \mathbf{x} - y \frac{r\mathbf{w}}{|\mathbf{w}|} = \mathbf{x} - y \frac{r\mathbf{w}}{\sqrt{\mathbf{w}^T \mathbf{w}}}$$

- Since \mathbf{x}' is on the plane, so $\mathbf{w}^T \mathbf{x}' + b = 0$ (with b as the bias term). Thus:

$$\mathbf{w}^T \mathbf{x}' + b = \mathbf{w}^T \left(\mathbf{x} - y \frac{r\mathbf{w}}{\sqrt{\mathbf{w}^T \mathbf{w}}} \right) + b = \mathbf{w}^T \mathbf{x} + b - yr \sqrt{\mathbf{w}^T \mathbf{w}} = 0$$

$$\Leftrightarrow \mathbf{w}^T \mathbf{x} + b = yr |\mathbf{w}| \Leftrightarrow r = y \frac{\mathbf{w}^T \mathbf{x} + b}{|\mathbf{w}|}$$



- For each and every training data point $\langle \mathbf{x}_i, y_i \rangle$, it's clear that $r_i \geq \rho$. Hence:

$$r_i = y_i \frac{\mathbf{w}^T \mathbf{x}_i + b}{|\mathbf{w}|} \geq \rho = \frac{1}{|\mathbf{w}|} \Leftrightarrow y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1$$

- **Problem:** Solve \mathbf{w} and b , so that $|\mathbf{w}|^2 = \mathbf{w}^T \mathbf{w}$ is minimized, and $y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1$ for all $\langle \mathbf{x}_i, y_i \rangle$ in the training set. In other words, $(\mathbf{w}^T \mathbf{x}_i + b) \geq 1$ if $y_i = 1$, and $(\mathbf{w}^T \mathbf{x}_i + b) \leq -1$ if $y_i = -1$.
- **Solution:** Use quadratic optimization to compute Lagrangian multipliers α_i so that $|\mathbf{w}| = |\sum_i (\alpha_i y_i \mathbf{x}_i)|$ is minimized and $y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1$ for all $\langle \mathbf{x}_i, y_i \rangle$ in the training set. This gives $\alpha_i \neq 0$ for support vectors, and $\alpha_i = 0$ for non-support vectors. Note that for any support vector \mathbf{x}_k , we have $y_k(\mathbf{w}^T \mathbf{x}_k + b) = 1$, which gives $b = y_k - \mathbf{w}^T \mathbf{x}_k$.
- Once we have learned the best values for \mathbf{w} and b , we can use our classifier to classify document \mathbf{x} by doing the following:

$$F(\mathbf{x}) = \text{sign}(\mathbf{w}^T \mathbf{x} + b) = \text{sign}\left(\sum_i (\alpha_i y_i \mathbf{x}_i^T \mathbf{x}) + b\right) \quad \text{where} \quad \text{sign}(z) = \begin{cases} 1, & \text{if } z > t \\ -1, & \text{if } z < -t \\ 0, & \text{otherwise} \end{cases}$$

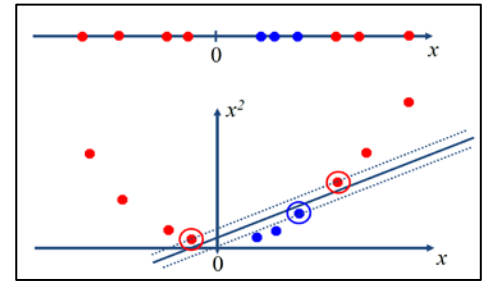
- Note that if $F(\mathbf{x}) = 0$, then the classifier cannot classify \mathbf{x} to any of the 2 classes. Here, we use t to specify the tolerance level. If $t = 0$, then \mathbf{x} can always be classified to one of the 2 classes unless it lies exactly on the chosen plane.

Soft Margin SVM: Real word classification problems are not always 100% linearly separable (due to noise in the data or outliers). So we might want to extend SVM with *soft* margin by allowing a few training points to be misclassified.

- Introduce a slack variable ξ_i for each training data point \mathbf{x}_i . This slack is the distance the \mathbf{x}_i must be moved such that it will lie on the margin of the side that we want \mathbf{x}_i to be.
- **Problem:** Solve \mathbf{w} and b , such that $\mathbf{w}^T \mathbf{w} + C \sum_i (\xi_i)$ is minimized, and $y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq (1 - \xi_i)$ where $\xi_i \geq 0$ for all i . Note that the higher C is, the softer the margin. In other words, C is like a parameter that prevents overfitting.
- **Solution** is of the form $\mathbf{w} = \sum_i (\alpha_i y_i \mathbf{x}_i)$, and $b = y_m(1 - \xi_m) - \mathbf{w}^T \mathbf{x}_m$ where $m = \underset{k}{\text{argmax}}(\alpha_k)$.

Non Linear SVM (Support Vector Machine):

- The idea is to expand the vector space, by mapping each point from the initial space to a *higher* dimensional space in which the training data can be linearly separable. Then do the linear SVM classification in this expanded hyperspace. If we use $\Phi(\mathbf{x})$ to map \mathbf{x} to a higher dimension, then classification becomes $F(\mathbf{x}) = f(\Phi(\mathbf{x})) = \text{sign}(\mathbf{w}^T \Phi(\mathbf{x}) + b)$.



- **Kernel trick:** Let $K(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j)$ be the kernel function (i.e. dot product in expanded vector space). Then:

$$F(\mathbf{x}) = F(\Phi(\mathbf{x})) = \text{sign}(\mathbf{w}^T \Phi(\mathbf{x}) + b) = \text{sign}\left(\sum_i (\alpha_i y_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x})) + b\right) = \text{sign}\left(\sum_i (\alpha_i y_i K(\mathbf{x}_i, \mathbf{x})) + b\right)$$

- So the learning involves finding an appropriate Kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$ which depends on how we want to map the data points to a higher dimensional space. The learning also involves computing α_i and b .
- Once the learning is done, we can use the formula to $F(\mathbf{x}) = F(\Phi(\mathbf{x}))$ above to classify the document \mathbf{x} .
- Some popular kernels are polynomial kernel $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T \mathbf{x}_j + 1)^d$ or Gaussian kernel $K(\mathbf{x}_i, \mathbf{x}_j) = e^{-|\mathbf{x}_i - \mathbf{x}_j|^2 / 2\sigma^2}$.

Web Search

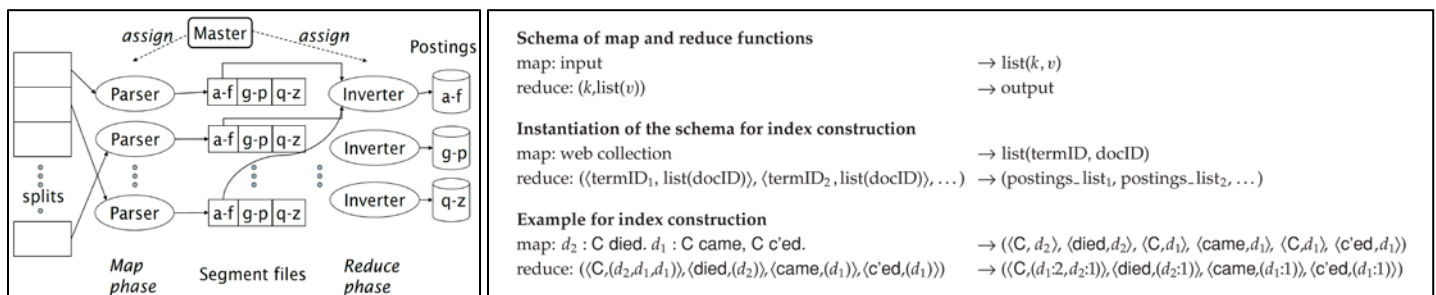
Challenges to Web Search are scale (billions of web pages), heterogeneous content (video, audio, picture, file types, XML, unstructured text, or databases), spams and website trusts, business aspects (e.g. sponsored search), etc.

- Precision is more important than recall (especially on top results). Recall only matters if no. of matches is very small.
- Necessary to filter out untrusted pages (spam). Common spamming techniques are invisible text, cloaking (server returns fake content to web crawlers), etc. Solution is to use ML to classify spam, blacklists, quality signal from users, etc.

Distributed Indexing

Distributed Indexing is to create an index distributed on several machines, either by terms or by documents. We focus on term-partitioned index here, since it can easily be converted to document-partition index (which is actually preferred by large search engines). We will use MapReduce here, which is a general architecture for distributed computing. It's designed to be run on a computer cluster (i.e. a set of cheap standard computers) as opposed to a specialized supercomputer.

- Master machine (*master node*) divides the docs into *splits* (or blocks). Each will be assigned to a machine (or *node*). For simplicity, we assume that all nodes share the same term-termID mapping.
- **Parser** nodes (for *map* phase): Parse each assigned document to a stream of termID-docID pairs. Each pair will be written to an appropriate disk partition (i.e. which corresponds to a file on disk), where each partition is for a range of terms' first letters (e.g. a-f, g-p, q-z). It's also more efficient if the parsers write to some local files first, before writing onto the disk.
- **Inverter** nodes (for *reduce* phase): Each inverter grabs one disk partition. Sort all termID-docID pairs in that partition, and create a postings list for each term. Then write the postings lists back into disk.



Dynamic Index is an index for a document collection that changes over time (due to documents being added, deleted, or modified). To construct a dynamic index, we can use the dynamic indexing algorithm (which is less preferred). An alternative is to periodically reconstruct the index from scratch (i.e. make a new index while the old one is still available for querying).

Dynamic Indexing Algorithm: Maintain a large main index on disk, and a small auxiliary index in main memory. New postings from new documents will be put into the auxiliary index. Searches will run across both indexes.

- Use invalidation bit vector to mark which documents have been deleted. Periodically clean up these deletions.
 - Every time the auxiliary index is full, merge it with the main index.
- ➔ Note that dynamic indexing is usually not preferred because having multiple indexes (not only Z_0 but all the other I_i indexes as well) makes the maintenance to be a very difficult job.

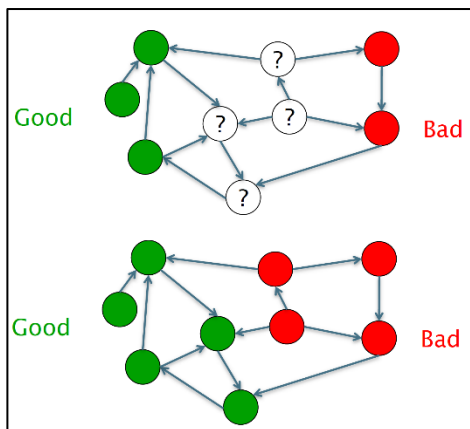
Merging Indexes for Dynamic Indexing Algorithm:

- Let Z_0 be the auxiliary index in main memory, and n be the max no. of postings that Z_0 can store. Let T be total no. of postings to be indexed. Then the algorithm will perform $\lceil T/n \rceil$ merges in total.
- *Standard way* is to simply merge Z_0 with the main index. Each posting is touched in each merge. So it's $\mathcal{O}(T^2/n)$.
- *Logarithmic merge* is to merge Z_0 with a series of indexes (I_i) on disk in an logarithmic manner.
 - o Let I_i be an index of size $2^i n$, where $i = [0, \infty]$. All indexes I_i will be stored on the disk as single files.
 - o Postings *percolate up* the sequence I_i . When up to n postings have been accumulated in Z_0 , the $2^0 n$ postings in Z_0 are transferred to a new index I_0 that is created on disk. The next time Z_0 is full, it's merged with I_0 to create an index Z_1 of size $2^1 n$. then Z_1 is either stored as I_1 (if I_1 doesn't exist) or merged with I_1 into Z_2 (if I_1 exists), and so on.
 - o For each search request, we query Z_0 and all currently valid indexes I_i on disk.
 - o Let I_k be the largest index (such that k is the deepest level of the merge), then $|I_k| = 2^k n = T \Leftrightarrow k = \log(T/n)$. Since each posting must percolate up the entire sequence, so each posting is processed $\log(T/n)$ time. And because the total no. of postings is T , so the total time taken is $T \times \log(T/n) = \mathcal{O}(T \log(T/n))$.

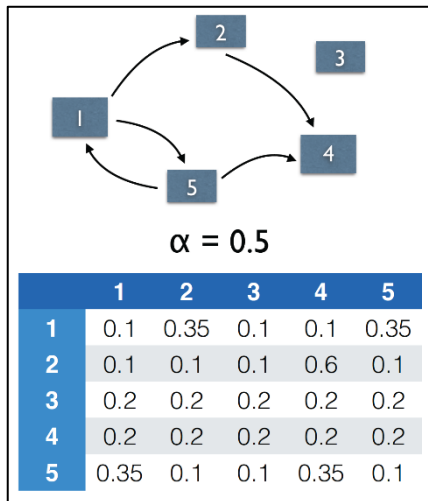
Link Analysis

PageRank: Imagine a web server that randomly surfs the web for eternity. To move from A to B , the surfer can either follow a *direct* link (all links are equally likely), or he can *teleport* to B by typing in the URL (all possible web pages are equally likely). Define α as the relative probability of teleporting versus using the direct link.

- Use transition matrix P , where $P[i, k]$ holds the probability of going from node i to node k (i.e. row to column). Note that the stochastic matrix $P' = P^T$ is the one that will be used for the computation. Let \mathbf{x}_0 be some initial vector, then $\lim_{n \rightarrow \infty} (P')^n \mathbf{x}_0 = P' \mathbf{x}_\infty = \mathbf{x}_\infty$ is the eigenvector that gives the PageRank score in the range $[0, 1]$ for each of the web nodes.
- To compute P , let N be the total no. of pages available and D_i be the no. of direct links from node i .
 - o If there is no direct link from i to k , then $P[i, k] = \alpha/N$.
 - o If there is a direct link from i to k , then $P[i, k] = \alpha/N + (1 - \alpha)/D_i$.
 - o We must also make sure that $\sum_k P[i, k] = 1$.



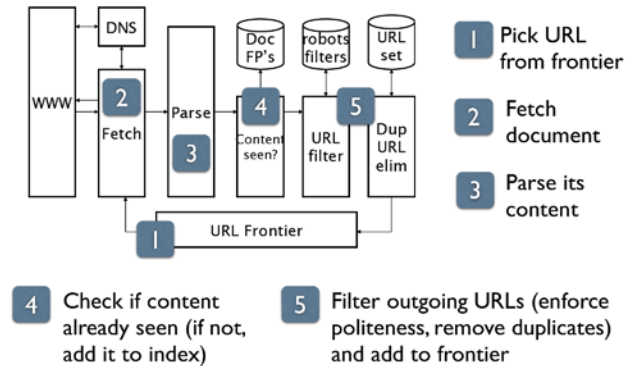
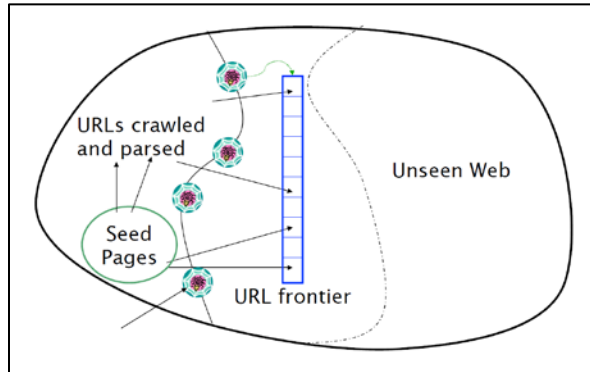
If you point to a *bad* node, you are *bad*.
If a *good* node points to you, you are *good*.



$$\mathbf{x}_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ and } \mathbf{x}_\infty = \begin{bmatrix} 0.190 \\ 0.190 \\ 0.144 \\ 0.286 \\ 0.190 \end{bmatrix}$$

Web Crawlers should be distributed/scalable/efficient, polite (e.g. not flooding the servers, or only crawl the allowed pages), robust to all types of content (e.g. ill-constructed pages or dynamically generated pages), and be able to prioritize.

- URL frontier is the data structure storing the set of URLs that have been detected but not yet crawled.
- URL frontier system must be able to enforce prioritization and politeness → Use 2 FIFO queues. *Front* queues are for prioritization (each queue is for a priority level). *Back* queues are for politeness (each queue is for a specific host).



XML Retrieval

3 Types of Retrieval are RDB (relational databases), unstructured retrieval (web search), and structured retrieval (XML).

- XML is more flexible than databases, and more structured than a free-text IR.
- XML represents semi-structured data (i.e. structured data that do not fit into relational model), and thus we want to be able to run queries that combine textual criteria with structural criteria.

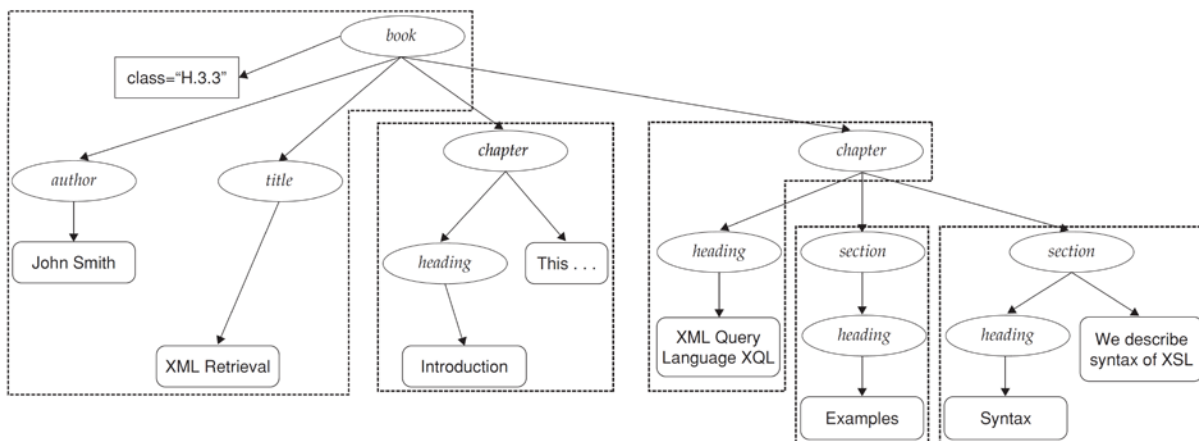
➔ *Structured retrieval principle*: System should always retrieve the most specific part of a document answering a query.

XML Basics: An XML document is an ordered and labeled tree. Leaf nodes contain the actual data (i.e. text strings). Each element node has a name (i.e. element *type*) and a set of *attributes*. Each attribute has a name and a value.

- XML schema puts constraints on the structure of allowable XML documents for a particular application.
- Most query languages for XML (e.g. XQuery or XIRQL) can handle numerical attributes, joins, and ordering constraints.
- NEXI (Narrowed Extended XPath I) is a common format for XML queries.
 - o E.g. "`article [./yr = 2001 or ./yr = 2002] //section [about (., summer holidays)]`"
 - o The query above searches for articles between 2001 and 2002 about summer holidays. Here, `./yr` is for relational constraints (i.e. only consider exact matches), while `about` is for ranking constraint.
- *Data-centric* XML is used for messaging between enterprise applications. It's usually filled with numerical and non-text data, thus are usually used in analytics program (to e.g. produce a graph, etc.). Note that most data-centric XML is stored in databases, and not in inverted index (as for text-centric XML).
- *Text-centric* XML is used for annotating content that is rich in text, thus demands a good integration of text retrieval functionality → E.g. find the ISBN's of books with at least 3 chapters discussing cocoa production, ranked by price.

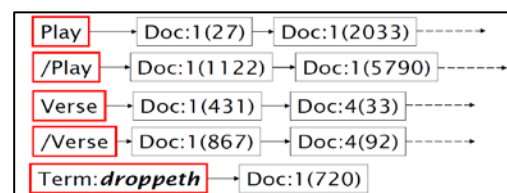
Challenges in XML Retrieval:

- There are usually many schemas in an application, which users don't know about. Thus users often are not able to compose good structured queries. Solution is to provide a good UI layer (with helps) for the users.
- Which parts of the XML documents to return? Deciding which level in the XML tree to return is quite difficult.
- How do we determine indexing unit that defines the index granularity? Here are some approaches:
 - o *Non-overlapping*: Divide the XML nodes into non-overlapping units, e.g. use *book*, *chapter*, and *section* as indexing units (see the pic). But this is quite counter-intuitive to the users because the units are not coherent.
 - o *Top-down*: Use the largest elements (e.g. *book*) as indexing units, and then post-process the results to find the parts (e.g. a chapter) that best match the query. However, this might fail to return the best-match for the queries because the relevance of a whole book is often not a good predictor of the relevance of small sub-elements within it.
 - o *Bottom-up*: Search in the leaves, select the most relevant ones and then extend them to larger units in post-processing. However, the relevance of a leaf is often not a good predictor of the relevance of elements it's contained in.
- The easiest approach for indexing granularity is to index all elements. But then we will get redundancy caused by the nested elements, making the search unnecessarily lengthy and complicated.
 - o Some restriction strategies include (1) discard all small elements, (2) discard all elements that users don't usually want to look at, (3) discard all elements that system designer deemed to be not relevant or useful.
 - o We can also do some post-processing to remove the redundancy in the results set caused by nesting.
 - o Alternatively, we can collapse nested elements into 1 result, and then *highlight* the query terms in relevant passages.



XML Retrieval Implementation:

- SimNoMerge is a variant of cosine scores. It will be used to compute the scores for XML retrieval. The algorithm takes into account the *context resemblance* function $C_R(c_q, c_d)$, which measures similarity between a path c_q in a query, and a path c_d in a document.
- For dictionary, number each element to maintain the ordering of elements and the nesting (i.e. parent-child relationship). Then build a positional index for each *element* (by marking the beginning and end) and each *term*. A term contained within an element can be viewed as merging of the postings lists.



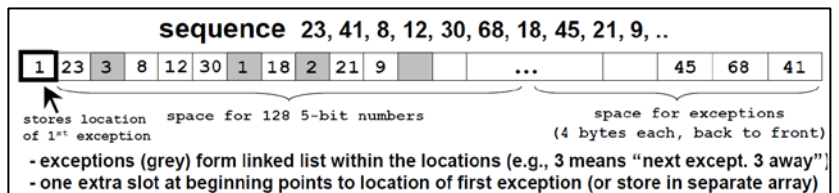
Supplementary

Evaluation: F_1 is F measure when $\alpha = 0.5$ and $\beta = 1$. This defines the *break-even point* (i.e. *balanced measure*). Another measurement is R -precision which is defined as $F_1(P', R') = P' = R'$.

- Let S_r be the set of all relevant documents in the collection for a particular query, and let $K = |S_r|$. Then we can define R -precision as $R' = R@K$. But this is exactly the same as $P' = P@K$. In other words $P' = R'$.
- Plugging $P' = R'$ into equation of F_1 gives $F_1 = P' = R'$. This shows the relationship between F_1 and R -precision.
- Interpolated precision at the recall level r is defined as the highest precision found for any recall level $r' > r$. In other words, it is defined as the highest $P@K$ considering only the ranks K where $R@K > r$.

PFOR (patched frame of reference) is a compression algorithm. The idea is choose a fixed no. of bits to encode 90% of the gaps, and we will try to use as few bits per gap as possible here. For the other 10% of the gaps, we can use a fixed-sized integer to encode (i.e. 32 bits). These 10% of the gaps will act as *exceptions*.

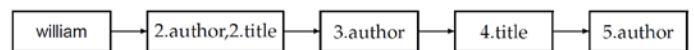
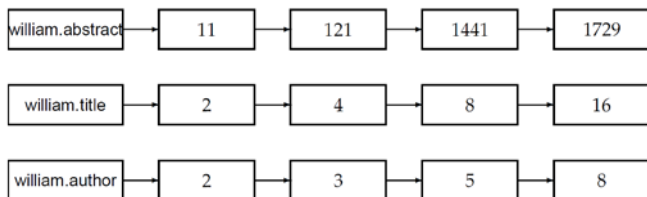
- E.g. If for the next 128 gaps, 90% of them are < 32 , then we choose $b = 5$. Allocate $128b$ bits for those 90% plus some space at the end for the exceptions, which will be stored as integers.



- Note that we often use a block of 128 gaps because that will usually fit into caches, thus increasing the efficiency.

Zones and Fields:

- Each field/zone has a data structure that represents the inverted index (for fields, that's called parametric index).
 - o Dictionary for a parametric index comes from a fixed preset vocabulary (e.g. a set of languages, or a set of dates).
 - o Dictionary for a zone index doesn't have a fixed vocabulary, but it depends on what the text of that zones contains.
- Weighted zone scoring (for zones index): Let g_i be the "zone importance score" which tells how important it is (relative to other zones) to find a match in zone i . Hence we require $\sum_i(g_i) = 1$. We also let $s_i = 1$ if the query term occurs in zone i , and vice versa for $s_i = 0$. Therefore, weighted zone score between the query term and the document is $\sum_i(g_i s_i)$.
 - o The weights g_i can be machine-learned from training examples.
 - o For the boolean query $p_1 \wedge p_2$ (where p_1 and p_2 are zone terms), then we can simply use the algorithms intersecting 2 postings lists in order to calculate the weighted zone scores for the documents where both terms must appear in the zones of the documents. This is best achieved if we encode the zone indexes as extensions of the postings.



- ➔ Zones indexes as extensions of dictionary entries (thus can act as a separate index).
- ➔ Cons: Blow up the dictionary size by a lot.
- ➔ Zones indexes as extensions of postings (thus can't act as a separate index).
- ➔ Cons: Raise the complexity of the postings by a lot.