Information Retrieval

Some sources:

- Stanford's NLP Book
- Related course in ETH Zurich

Terms:

1. Precision: percentage of relevant documents amongst all retrieved documents
2. Recall: percentage of retrieved documents amongst all relevant documents

Lecture 1 (N-grams)

Language model → Created based on a collection of text, and we assign a probability to a sequence of words.

Ngram LM → remembers sequences of n tokens

Probability-based LM with **smoothing** (e.g. add 1 smoothing: add 1 count to all entries in the LM, including those that are not seen). Smoothing is used to take into account to a probable word that has not been seen yet, so that we reduce the number of 0 probability that we get.

Example:

Q: paint my love

Unigram: P(paint) x P(my) x P(love)
Bigram: P(paint) x P(my | paint) x P(love | my).

Lecture 2 (Boolean retrieval)

Assumption: We have a **fixed** set of documents and we want to find documents that is **relevant** to a given task.

Boolean Retrieval

- Document is the unit that we decide to build our model over
- Corpus / Collection is the group of documents over which we perform retrieval
- The **Boolean retrieval model** is a model for information retrieval in which we can pose any query which is in the form of a Boolean expression of terms (AND OR NOT). It views each document as a set of words.

Term-Document Incidence Matrix

Think adjacency matrix. Term on the rows, Document on the column. $x_{i,j} = 1$ iff term i exists in document j.

We can now take the rows as vectors (or its complement) of 0 and 1 → called **incidence vectors**, and do a bitwise AND to get the result we wanted.

Might not be scalable on a huge corpus → resulting in a sparse index.

Inverted Index

Think adjacency list. Improves from the previous approach. Something like:
`term (number of docs containing term) : docId1 -> docId2 ->docId3 -> ...`

We keep a **dictionary** of these **terms** and for each term, a **posting list**: a list that records which documents the term occurs in. Each item on the list is called a **posting**.

The dictionary is sorted alphabetically, and the posting list is sorted by document ID.

Lecture 3 (Posting lists and Choosing Terms)

Posting lists enhancement

- faster merges by using skip lists: improve AND queries, but are only available for original posting list

Handling Phrase queries:

Some techniques:

1. extended biwords (bucketing terms based on nouns and articles). E.g. "catcher in the rye" → "catcher rye".
    - However, this technique cannot verify whether the document contains the original phrase for 3 or more words.
    - It also significantly expands the size of vocabulary
2. Positional indexes: can handle explicit proximity search
3. Hybrid: Augment positional indexes with biword indexes, e.g. store common queries / common individual words but rare phrase queries as biwords index.

Positional Indices

We also store the position(s) in which the tokens appear

- <term, number of docs containing term; < docId1: position1, position2, ... >; < docId2: position1, position2, ... >; etc. >
- We can further compress this
- index size depends on average document size, since we need an entry for each occurrence (not just once per document)

Usually this index is ~2 - 4 times bigger compared to non-positional index.

Considerations

1. Text extraction: different text format / language result in different encoding on the disk. We should be able to extract the text from all these documents. Another problem that might arise is a single document might contain multiple languages.
2. Granularity of indexing: What should the unit document be? 1 file? 1 email? a group of files? This is usually down to tradeoff between precision and recall. The problems with large document units can be alleviated with (explicit / implicit) proximity search.
3. Tokenization: What are the correct tokens to use? Hard to get it consistent, really...
   - English tokenization: How to handle apostrophe, hyphen, spaces?
   - Numbers, dates, other non-standard formats
   - Language issues: German noun compounds, French (L'ensemble), Chinese and Japanese (no spaces between words), Japanese (multiple writing systems), Arabic or Hebrew (inconsistent RTL or LTR)
4. Stop word removal?
   - Remove "the", "a", "and", "to", "be"
   - Trend is moving away from removing stop words, since we now have good compression techniques and good query optmizations.
5. Normalizing tokens to terms
   - Equivalence classing:
     - deleting periods, hyphens, accents, e.g. U.S.A → USA
     - case folding (reduce all letters to lower case)
     - synonyms, spelling variations, transliteration variations ...
   - Asymmetric expansion
     - search for different keywords e.g. different search for window vs windows vs Windows
6. Lemmatization: reduce inflectional/variant forms to base form.
   - e.g. am, are, is → be
7. Stemming: reduce terms to their "roots" before indexing (usually by chopping off end of word)
   - compressed, compression → compress

Lecture 4 (Dictionaries and Tolerant Retrieval)

Search structure for dictionaries

1. Hash table:
   - Fast lookup
   - However, there is no support for prefix search, no easy way to find minor variants.
   - We might need to use linear hashing or rehash everything as the vocab grows.
2. B-Tree
   - Solves prefix problem, runs in O(log M)
   - However, search trees required that the characters used have a prescribed ordering, which might not exist in some language, e.g. Chinese and Japanese. Note: They now have standard ordering for their character set.

Wildcard queries (*)

Prefix search is easy using B-tree. For postfix search, we can use a reversed B-tree.

Some solutions to handle general "X*Y" search:

1. For single * : Do prefix search, postfix search then intersect!
2. Permuterm index
3. 2-step query processing with additional k-gram index

Permuterm Index

Idea: We add a special character (`$`) to the end of a term, then we index all possible rotation of the modified term.

For any query,

1. Rotate the query s.t. it become a prefix search
2. Use the permuterm to lookup original vocabulary terms matching a wild card query.
3. Lookup these matching terms in the original inverted index, to retrieve matching documents.

For cases where there are more than 1 wildcards, e.g. "X*Y*Z", we can find all matching terms for "X*Z" and then exhaustively enumerate each candidate to check whether it matches the original query.

Problem: number of indexed terms grows proportional to the average word length.

Using k-gram indexes

Idea: We maintain a second inverted index mapping k-grams to dictionary terms

For any query,

1. Find matching (dictionary) terms by issuing a Boolean Query on the second inverted index.
2. Surviving dictionary terms are then looked up in the term-document inverted index.
3. Post-filter false positives

Advantage: Fast and space efficient compared to permuterm
Disadvantage: Query processing can be quite expensive

Spelling Correction

Two main flavors: isolated (check each word on its own for misspell), context-sensitive (look at surrounding words)

There are 2 sides of spelling correction: query correction, and document correction (often in this case, we don't change the document but we fix the query-document mapping)

Isolated word correction

We need a source for correct spellings, e.g. Merriam-Webster's English Dictionary.

"Closest" definition is a combination of:

1. (Weighted) edit distance:
    - weight can be assigned by considering the likelihood of a typo
    - Question: For which string should we compute the edit distance? Exhaustive search is expensive...
2. k-gram overlap with normalization: compare the number of n-gram overlap
    - By using k-gram index, we can limit the set of vocabulary term for which we compute the edit distance
    - Use Jaccard coefficient: $\frac{|X \cap Y|}{|X \cup Y|}$ as the measure of overlap, where X is the set of all k-grams in query q and Y is the set of k-grams in a matching vocabulary term.
        - Note: Although we can easily compute X, it is not the case for Y. Luckily, what we actually need is only $|X|$, $|Y|$ and $|X \cap Y|$ which are all readily available.

Context-sensitive spelling correction

Possible approach:

1. Retrieve possible "corrections" to each term in the query, and enumerate all possible resulting phrase with one term "corrected" at a time
2. Break phrase queries into a conjunction of biwords, look for biwords that only need one term to be corrected, and enumerate phrases with common biwords

General issues

We have enumerate several ways to generate corrections, but how do we choose what to present to users? Some heuristics:

1. Hit-based spelling correction: based on the popularity of the correction
2. Query log analysis: based on previous queries

Phonetic Correction (Soundex)

The main idea here is to generate, for each term, a “phonetic hash” so that similar-sounding terms hash to the same value, e.g. Chebyshev and Tchebysheff

Overall scheme:

1. Turn every term to be indexed into a 4-character reduced form. Build an inverted index from these reduced forms to the original terms; call this the soundex index.
2. Do the same with query terms.
3. When the query calls for a soundex match, search this soundex index

How do we generate the reduced form? See wikipedia.

When might this algorithm not work? It rests on the following observations which might not be true for non-European languages:

1. Vowels are viewed as interchangeable, in transcribing names;
2. Consonants with similar sounds (e.g., D and T) are put in equivalence classes

Lecture 5 (Index Construction)

BSBI (Blocked Sort-Based Indexing):

Very similar to External Merge Sort.
Difference: Instead of indexing on term, we index on termID

1. segments the collection into parts of equal size
2. sorts the termID–docID pairs of each part in memory
3. stores intermediate sorted results on disk
4. merges all intermediate results into the final index

Constraint: Need data structure to map terms to termID which must fit in-memory.

SPIMI (Single Pass In Memory Indexing):

Similar to BSBI with improvements:

1. Instead of first collecting all termID–docID pairs and then sorting them (as we did in BSBI), postings are directly added to the postings list. As we stream the document from smaller ID to larger ID, the postings list itself is already sorted. However, we still need to sort the terms before writing to disk to facilitate the final merge.
2. There is no more term-termID mapping held in memory, because we directly store the term in the inverted index.

Distributed Indexing

Pain points: web-scale indexing, fault-tolerance

An application of the MapReduce algorithm where

- Map is the Parser, i.e. parsing documents → list of [termID, docID] pairs
- Reduce is the Inverter, i.e. transforming list of [termID, list(docID)] → list of [termID, posting lists]

Dynamic indexing

So far, our assumption is that the document collecton is static. Re-indexing every time a new document is added/removed is expensive (incur too much random disk seeks).

Instead, we maintain a (big) main index and a (smaller, in-memory) auxiliary index where new docs are indexed. Over time, the auxiliary index is merged with the main index.

- Advantages of this approach is to reduce the number of random disk seeks in the naive approach.
- To search, we query both the main index and the auxiliary index and merge the results.
- We also maintain a invalidation vector bit for deleted docs and filter all results that comes from deleted docs.

We can further extend the idea from using just 2 indexes to $\log_2(T/n)$ indexes $I_0, I_1, \ldots$ of size $2^0 \times n, 2^1 \times n, \ldots$. This technique is called logarithmic merge

- Sacrifices a bit of querying complexity (because we now needs to merge $\log_2(T/n)$ results instead of 2 previously) for a much faster indexing complexity (from $O(T^2/n)$ down to $O(T \cdot log(T/n)))$
- Disadvantage: Harder to maintain collection-wide statistics

Other indexing problems

- positional indexes : much larger
- user access rights : whether a user has access to a certain doc

Lecture 6 (Index Compression)

book chapter

Why compress?

1. Less disk space
2. Improved cache utilization → because more data can be fit into memory
3. Faster disk-to-memory data transfer: read compressed data + decompressing is faster than reading uncompressed data.

Case folding, stemming, stop word elimination, and vector space model are forms of lossy compression.

Statistical Properties

Heap's Law

Heap's Law **estimates the number of vocabulary terms** (M) as a function of the collection size: $M = kT^b$ where T is the number of tokens in the collection, and typical values for k, b is $30 \leq k \leq 100$ and $b \approx 0.5$

Heaps' law suggests that the dictionary size continues to increase with more documents in the collection and therefore, the size of the dictionary is quite large for large collections.

Zipf's Law

The i-th most frequent term has frequency proportional to $\frac{1}{i}$ that of the most frequent term

Dictionary Compression

Why?

- Search begins with the dictionary, and hence we want to keep it in memory.
- Main memory might be limited in phones / embedded hardware
- We want it to be small enough so that search is fast.

The easiest way to store a dictionary is to use a fixed-width entry for terms

1. Dictionary as a string: Store the terms in a string, and use term pointers (and the next term pointers) to indicate the start (and end) of the term in the string.
   - e.g. automataautomateautomaticautomation
2. Blocked storage: Group terms in the string into k-sized blocks and keeping a term pointer only for the first term of each block. Additionally, we add an extra byte before each term to indicate the length of the corresponding term.
   - By increasing k, (compressed) dictionary size decreases, but term lookup becomes increasingly slow due to linear search.
   - e.g. 8automata8automate9automatic10automation
3. Front coding: Leveraging the fact that consecutive entries in an alphabetically sorted list share common prefixes.
   - e.g. 8automat$*$a1$\diamond$e2$\diamond$ic3$\diamond$ion

Other schemes: minimal perfect hashing (not suitable for dynamic environment)

In some cases, it might not be feasible to store the entire dictionary in main memory. If that's the case, we can partition the dictionary onto pages stored on disk, and index the first term of each page using a B-tree.

Postings file compression

Observation: Postings for frequent terms are close together.
Idea: Instead of storing docID, we can store gaps between consecutive postings which takes much less space than storing the docID.

To encode small numbers in less space than large numbers, we look at 2 methods of compression: bytewise compression and bitwise compression.

Variable Byte(VB) Encoding

The last 7 bits of a byte are “payload” and encode part of the gap. The first bit of the byte is a continuation bit. It is set to 1 for the last byte of the encoded gap and to 0 otherwise.

To decode, we read a sequence of bytes with continuation bit 0 terminated by a byte with continuation bit 1. We then extract and concatenate the 7-bit parts.

The idea of VB encoding can also be applied to larger or smaller units than bytes: 32-bit words, 16-bit words, and 4-bit words or *nibbles*. In general, bytes offer a good compromise between compression ratio and speed of decompression, and between time and space.

γ encoding

γ codes implement variable-length encoding by splitting the representation of a gap G into the pair (length, offset).

- Offset is G in binary, but with the leading 1 removed.
- Length encodes the length of offset in unary-code (unary code of n is a string of n 1s followed by a 0)
- e.g. the γ code of 13 is therefore 1110101, the concatenation of length 1110 and offset 101.

Define entropy $H(P)$ of a discrete probability distribution P as $H(P) = -\sum_{x \in X} P(x) \log_2 P(x)$. Note that we have $\sum_{x \in X} P(x) = 1$. It can be shown that γ encoding is within a factor of $\approx$ 2 of the optimal code for distributions with large entropy.

γ encoding has several desirable properties:

1. universality: The code is within a factor of optimal for an arbitrary distribution P of the collection frequencies
2. prefix-free: There is a unique decoding of a sequence of γ codes
3. parameter-free: It doesn't need to fit the parameters of a model to the distribution of gaps in the index. This simplifies implementation of compression / decompression and completely removes the possibility of the original parameters being no longer appropriate in dynamic indexing situations.

γ codes achieve great compression ratios, but they are expensive to decode because many bit-level operations - shifts and masks - are necessary to decode a sequence of γ codes as the boundaries between codes will usually be somewhere in the middle of a machine word.

δ encoding

γ encoding is relatively inefficient for large numbers. So, δ codes differ by encoding the first part of the code (*length*) in γ code instead of unary code.

Lecture 7 (Scoring, Weighting, VSM)

book chapter

Problem with boolean retrieval: either too few or too many results are returned.
Solution: Instead of a set of documents satisfying a query, a ranked retrieval system returns an **ordering** over the (top) documents in the collection with respect to a query. The query is usually free text queries.

Weighting, Scoring

We would like to compute a score between a query term t and a document d. Several ways of scoring:

1. Term frequency(tf): score based on the number of occurrences of term t in document d
   - Observation: documents that mentions a query term more often is more relevant for that particular query.
   - We might not want to use raw term frequency though, since a document with 10 occurrences of the term is not 10 times more relevant than that of 1 occurrence.
   - Instead, use a log-frequency weighting scheme such as $\mathrm{wf}_{t,d} = 1 + \log \mathrm{tf}_{t,d}$ for tf > 0 and 0 otherwise.
2. Inverse document frequency (idf): $\mathrm{idf}_t = \log \frac{N}{\mathrm{df}_t}$ where df_t is the number of documents that containt t and N is the collection size.
   - Observation: frequent terms are less informative than rare terms
   - We define idf this way instead of $\frac{1}{\mathrm{df}_t}$ to dampen the effect of idf (using log) and to keep the value non-negative because $\mathrm{df}_t \leq N$

- idf weighting only affects ranking for queries with at least 2 terms.
  - Note: We use document frequency instead of collection frequency (total number of occurrences of a term across the entire collection)
3. tf-idf weighting: $w_{t,d} = \mathrm{wf}_{t,d} \times \mathrm{idf}_t$.
   - best known weighting scheme in IR
   - highest when t occurs many times within a small number of documents
   - lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal)
   - lowest when the term occurs in virtually all documents, e.g. stop words

The simplest way to assign a score to a document d for a particular query q is $\sum_{t \in q} \mathrm{tf.idf}_{t,d}$

Storing tf-idf weight for each posting is not space-efficient (floating number storage). Instead, we maintain an idf value for each dictionary term, and a tf value for each postings entry.

Vector Space Model

This is paradigm for free text queries.

Documents as vectors

View document d as a vector in a $|T|$-dimension space where $|T|$ is the number of vocabulary terms in the collection. The set of documents in a collection then may be viewed as a set of vectors in a vector space, in which there is one axis for each term.

- One can use any kind of weighting (some are described above) to determine the elements of the vectors. These are very sparse vectors as most of the entries are 0.
- VSM is a *bag-of-words* model. It doesn't retain the relative ordering information of terms.

Queries as vectors

Represent queries as vectors in the space, and rank documents according to their proximity to the query in this space.

Formalizing proximity

We use the **cosine similarity**, i.e. $\cos(v_1, v_2)$, as the proximity metric between two vectors v_1 and v_2. For length-normalized vectors, the cosine similarity can be computed as the dot product between the two vectors.

Variant tf-idf functions

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

► **Figure 6.15** SMART notation for tf-idf variants. Here *CharLength* is the number of characters in the document.

SMART Notation denotes combination used with the notation $ddd.qqq$

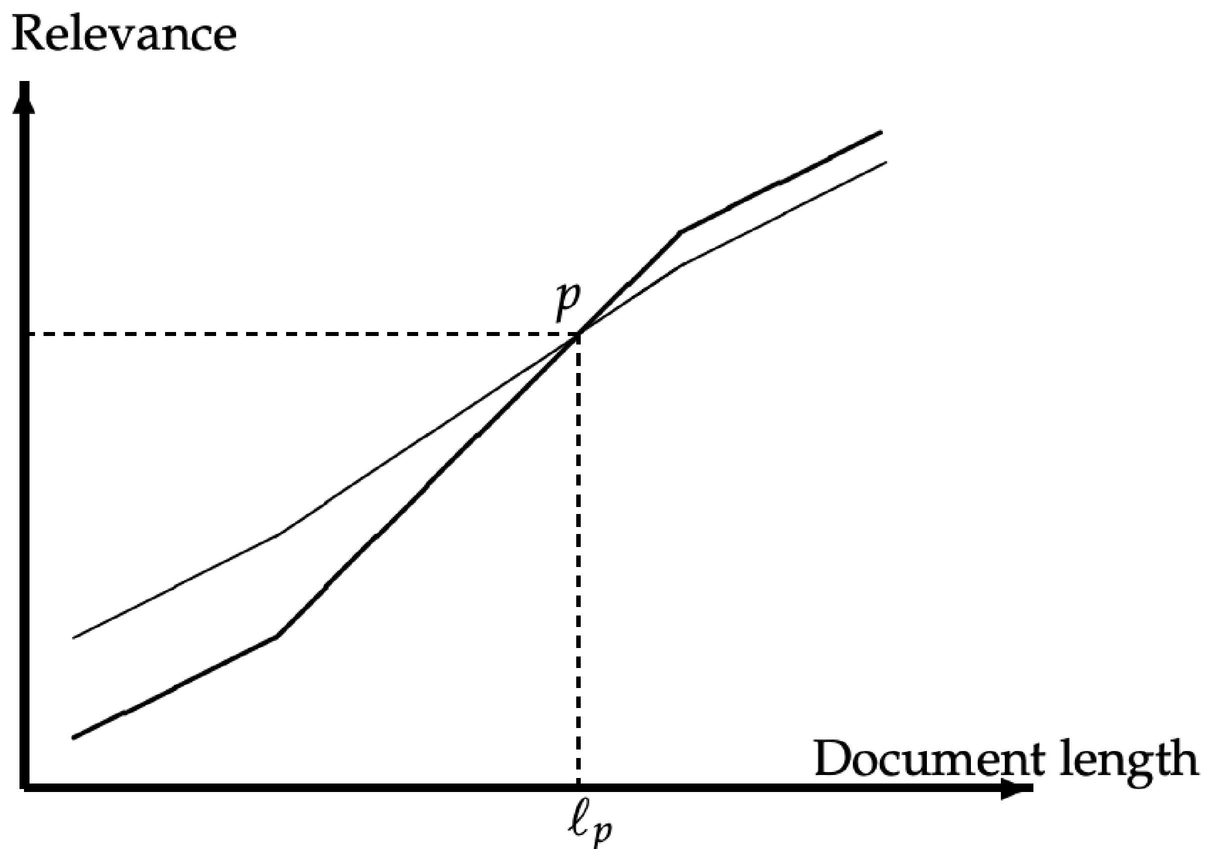
A very standard weighting is **lnc.ltc**

Why we don't normally use idf in the document vector?
If the index is dynamic, then we would need to update each document vector such that it reflects the correct idf whenever a document is added/removed/updated, which is very expensive.

Pivoted normalized document length

Previously, we normalized each document vector by the Euclidean length of the vector, but that eliminated all information on the length of the original document. We do not account that longer documents will have higher tf values (as it contains more terms), and contain more distinct terms.

We can compensate for document length by using pivoted document length normalization.



► **Figure 6.16** Pivoted document length normalization.

Suppose we graph the probability of relevance as prescribed by the cosine normalization (thin line) and the actual probability of relevance (thick line) as functions of document length. Define *pivot* as the document length for which this two curve cross. We want to tweak the normalization length s.t. the cosine probability relevance resembles more closely to the actual relevance.

The simplest idea is to rotate the cosine relevance around the pivot, that is by changing the normalization length from euclidean length $|\overrightarrow{V(d)}|$ to $a|\overrightarrow{V(d)}| + (1-a)l_p$ for some normalization factor a. In practice, this equation is well approximated by $au_d + (1-a)l_p$ where u_d is the number of unique terms in document d.

Summary

Algorithm for vector space ranking

1. Represent each document as a weighted tf-idf vector.
2. Represent the query as a weighted tf-idf vector
3. Compute the cosine similarity score for the query vector and each document vector
4. Rank documents by score and return the top K documents to the user.

Lecture 8 (Complete Search System)

Efficient Scoring

To improve scoring efficiency, we can let $w_{t,q} = 1$ for $t \in q$

```
FASTCOSINESCORE(q)
 1   float Scores[N] = 0
 2   for each d
 3   do Initialize Length[d] to the length of doc d
 4   for each query term t
 5   do calculate w_{t,q} and fetch postings list for t
 6       for each pair(d, tf_{t,d}) in postings list
 7       do add wf_{t,d} to Scores[d]
 8   Read the array Length[d]
 9   for each d
10   do Divide Scores[d] by Length[d]
11   return Top K components of Scores[]
```

► **Figure 7.1** A faster algorithm for vector space scores.

Big problem: calculating cosine similarities between the query and a large number of documents is really expensive

Generic solution:

1. Find a set A of contenders s.t. $K < |A| \ll N$. It doesn't necessarily contain the K top-scoring documents for the query, but is likely to contain many of them.
2. Return the K top-scoring documents in A.

In the following, we discuss some heuristics to improve scoring speed, while tolerating a bit of sloppiness

Index Elimination

1. Only consider terms with high idf (above preset threshold)
   - similar in spirit to stop word removal
2. Only consider documents that contain many (or all) of the query terms
   - can be accomplished during postings traversal
   - DANGER: might end up with fewer than K candidate documents

Champion Lists

Also called *fancy lists* or *top docs*.

Idea: precompute for each term t in the dictionary, the set of the r documents with the heighest weights for t.

Note: For tf-idf weighting, this just means taking the r documents with the highest tf values for term t.

At query time, only compute scores for docs that is part of some query term's champion list. Then, pick the K top-scoring docs from amongst these.

Danger: As r is set at the time of index construction, it may be less than K (specified at query time)

Static Quality Scoring and Ordering

Suppose we have a measure of authority $g(d)$ for each document d that is static, i.e. query-independent. E.g. number of favorable reviews of a webpage, citations, views etc.

Idea: We want our results to be both relevant and authoritative. So, we can use a function of $g(d)$ and cosine scores (measure of relevance) to order our postings list instead of docID.

A simple function combining cosine relevance and authority is $\text{net-score}(q, d) = g(d) + cos(q, d)$.

Since $g(d)$ is static, this is a common ordering for all postings.

Two applications:

1. In time-bound applications, static quality ordering allows us to stop postings traversal early, as top-scoring docs are likely to appear early in postings traversal.
2. We can extend the idea of champion lists by picking r docs with highest $f(g(d), \text{tf-idf}_{t,d})$

Tiered Indexes

A common solution for having less than K documents to accumulate score.

Idea: Break inverted index into tiers of decreasing importance. At query time, use only top tier index. If it has less than K docs, drop to lower tiers.

Impact Ordering

Key idea: Order the postings list based on a function that do not necessarily preserves ordering across terms, also known as **term-at-a-time scoring**. The simplest would be: decreasing order of $\text{tf}_{t,d}$ for each term t.

So far, we always have **document-at-a-time scoring**. Ordering our postings list by docID or other common ordering such as static quality scores, allow us to support concurrent traversal of all of the query terms' postings list, computing the score of each doc as we encounter them.

Although we lost the ability to traverse postings list concurrently, we can significantly lower the number of docs for which we accumulate scores:

1. Pick terms to traverse in decreasing order of idf
   - query terms likely to contribute to final scores are considered first
   - we can adapt, i.e. not process / process shorter prefix, how we query terms with lower idf by considering the impact that the previous query term had towards document scores.
2. Early termination while traversing t's postings
   - can be after a fixed number of r docs or after $\mathrm{tf}_{t,d}$ drops below some threshold.

Note: this is can be thought as a generalization of index elimination and champion lists

Cluster Pruning

Construction phase:

1. Pick $\sqrt{N}$ documents at random from the collection and call these *leaders*
   - random is good because it is fast, and adequately represents the data distribution
2. For each doc that is not a leader, compute b_1 nearest leaders

Query phase:

1. Given a query q, find b_2 leaders L closest to q by computing cosine similarities form q to each of the $\sqrt{N}$ leaders.
2. The candidate set A is all b_2 leaders in L together with their followers.

IR Components

Parametric and Zone Indexes

Fields: A region with finitely-many values, e.g. date of creation, doc format

We build **parametric indexes** for fields. Query processing then consists (as usual) of postings intersections, except that we may merge postings from standard inverted as well as parametric indexes.

Zone: A region of the doc that contain **an arbitrary amount of text**, e.g. title, abstract, references

For zone indexing, zones can be encoded:

1. as extensions of dictionary terms, e.g. william.abstract, william.title, william.author
2. in the postings, e.g. 2.author, 2.title

We could also implement a weighted zone scoring, i.e. title is more important than abstract, etc.

Query-term proximity

In free text queries, users prefer docs in which most or all of the query terms appear close to each other. We can quantify this by letting ω be the number of words in the smallest window in a document d that contains all the query terms.

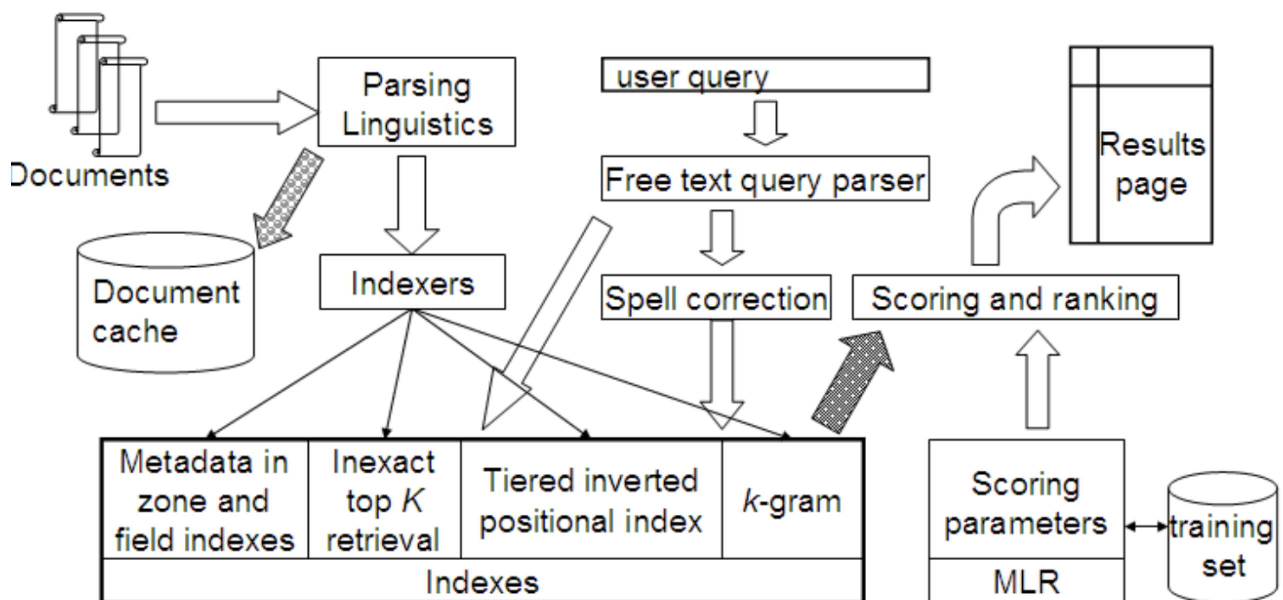
Idea: run one or more queries to the indexes generated by the *query parser*

For example, a query parser may issue a stream of queries:

1. Run the user-generated query string as a phrase query
2. If there are too few docs matching, run 2-term phrase queries
3. Run vector space query consisting of all individual query terms
4. Rank matching docs by combining contributions from vector space scoring, static quality, proximity weighting ω and potentially other factors.

Complete Search System

supports free text queries, boolean, zone and field queries.



► **Figure 7.5** A complete search system. Data paths are shown primarily for a free text query.

document cache: stores copy of each parsed document to generate results snippets (snippets of text accompanying each document)

Vector space scoring and query operator interaction

How does vector space model interacts with boolean, wildcard or phrase queries studied earlier?

Boolean queries

Vector space index can be used to answer boolean queries.

Wildcard queries

Wildcard and vector space queries require different indexes. But we can also interpret wildcard component as spawning multiple terms in the vector space. Documents matching more terms are likely to be scored higher.

Phrase queries

The representation of documents as vectors is fundamentally lossy (do not store the relative order of terms).

Vector space retrieval can identify documents heavy in these 2 terms, with no way of prescribing if they occur consecutively. Otoh, phrase retrieval tells us of the existence of the exact phrase, without any indication of the relative frequency / weight of the phrase.

In general, an index built for vector space retrieval cannot be used for phrase queries, but combining these 2 retrieval paradigms can be useful (See query-term proximity example).

Lecture 9 (Evaluation)

[book chapter]

We'd like to empirically measure the impact of the techniques to the relevance of the result.

Relevance measurement requires 3 elements:

1. A document collection
2. A test suite of information needs, expressible as queries
3. A set of relevance judgements, usually binary assessment of either relevant or non-relevant for each query-document pair, also known as *gold standard / golden truth*

Relevance is assessed relative to an information need, not a query.

Information need vs query

Information need: Information on whether drinking red wine is more effective at reducing your risk of heart attacks than white wine.

Query: wine AND red AND white AND heart AND attack AND effective

Unranked retrieval evaluation

Computed using unordered sets of documents

Basic measures for IR effectiveness:

1. Precision: percentage of relevant documents amongst all retrieved documents
2. Recall: percentage of retrieved documents amongst all relevant documents

It is usually a tradeoff between precision and recall. In most cases, one is more important than the other. E.g. web surfers would want high precision, but do not care about recall rates, while paralegals and intelligence analysts are concerned with high recall albeit lower precision.

As the number of documents retrieved increases, usually precision decreases while recall increases.

It is easy to get high precision (just get 1 relevant results) or high recall (return all documents). Obviously that is not what we want. A single measure that trades off precision vs recall is the **F measure**, which is the weighted HM of precision and recall.

A common choice is to calculate the **balanced F measure** $F_{\beta=1} = \frac{2PR}{P+R}$. Values of β can be tweaked to emphasize precision / recall.

Ranked retrieval evaluation

Evaluation measures

Interpolated precision at a certain recall level r is defined as the highest precision found for any recall level $r' \geq r$.

- Justification: Almost anyone would be prepared to look at a few more docs if it would increase the precision of the larget set

11-point Interpolated Average Precision

For each information need, the interpolated precision is measured at the 11 recall levels of 0.0, 0.1, 0.2, ..., 1.0. For each recall level, calculate the AM of the interpolated precision of each information need.

Mean Average Precision (MAP)

If the set of relevant documents (ordered by relevance) for an information need $q_j \in Q$ is $\{d_1, \ldots, d_{m_j}\}$ and $R_{jk} = \{d_1, d_2, \ldots, d_k\}$ then the **Average Precision** for q_j is defined as $\frac{1}{m_j} \sum_{k=1}^{m_j} \text{Precision}(R_{jk})$.

- The average precision approximates the area under the uninterpolated precision-recall curve.

MAP is an average of Average Precision over all information needs.

- MAP scores normally vary widely across information needs
- MAP weights each information need equally, even if some queries returns much more / much less relevant documents than that of the others.

Precision at k

Web search, in particular, do not care about precision at all recall levels. Instead, it only cares about measuring precision at fixed low levels of retrieved results, e.g. the 10 docs shown in the first page. This is referred to as "Precision at 10"

R-precision

Assuming we know a set of relevant documents R (can be incomplete), we calculate the precision for the top-R documents. Interestingly, the precision and recall rate for this top-R

documents are the same and hence R-precision is identical to the **break-even point**

Although it is unclear why you should be interested in the break-even point rather than either the best point on the curve (the point with maximal F-measure) or a retrieval level of interest to a particular application (Precision at k), R-precision turns out to be highly correlated with MAP empirically, despite measuring only a single point.

ROC curve

An ROC curve plots the true positive rate, i.e. recall, against the false positive rate (the percentage of retrieved doc amongst the unrelevant).

- for a good system, the graph climbs steeply on the left side
- This notion is not useful for unranked retrieval, since the false positive rate would be almost 0 as the number of unrelevant document is always so large.

NDCG

NDCG (normalized discounted cumulative gain) is designed for situations of non-binary notions of relevance and is increasingly used with ML approaches to ranking.

For a query $j \in Q$, let $R(j,d)$ be the relevance score assessors gave to document d and Z_{kj} a normalization factor s.t. a perfect ranking's NDCG at k for query j is 1. Then

$$\text{NDCG}(Q,k) = \frac{1}{|Q|}\sum_{j=1}^{|Q|} Z_{kj} \sum_{m=1}^{k} \frac{2^{R(j,m)}-1}{\log_2(1+m)}$$

Creating Test Collections For Evaluation

Aside from the document collections, we also need

1. a collection of information needs: usually best designed by domain experts
2. relevance assessment: time-consuming process using human judges

A human is not a device that reliably reports a gold standard judgement of relevance. However, it is interesting to consider and measure how much agreement there is between judges.

A common measure for agreement is the **kappa statistic** $\frac{P(A)-P(E)}{1-P(E)}$ where $P(A)$ is the proportion of the times the judges agreed and $P(E)$ is the expected proportion of the times they would agree (usually calculated using marginal statistics)

- As a rule of thumb, a kappa value >0.8 (good), 0.67 - 0.68 (fair), <0.67 (data cannot be used as a basis for evaluation)
- For >2 judges, we can use pairwise kappas or ANOVA.

Once we have the set of relevant / nonrelevant docs, we can vary IR systems and parameters to carry out comparative experiments. However, some problems with this approach are:

1. The relevance of 1 doc is assumed to be independent of the relevance of other doc

2. Assumption that users' information needs do not change once they start looking at retrieval results
   - No distinction between relevance and **marginal relevance**, i.e. whether a document still has distinctive usefulness after the user has looked at some other documents
3. Assessments are binary (no nuance)
   - Relevance can be divided into 3 or 4 classes
4. Relevance of a doc to an information need is treated as an absolute, objective decision.
5. Results may not translate across domains, as it is highly dependent on our training set.

A/B Testing

Previously we have discussed benchmark testing using precision / recall / F-measures etc. Once we have built an IR system which is used by a large number of users, we can refine the system by employing **A/B Testing**. In A/B testing we direct a small proportion of trafic to a variant system, which has exactly 1 thing changed from the current system. We then measure to see whether the change has positive / negative effect.

Results snippets

We want to make the results list informative enough for the user. We can do this by providing a **snippet**, a short summary of the document designed to allow the user to decide its relevance.

Static summary

Key point: the summary is extracted and cached at indexing time.

Static summary is generally made of a subset of document and its metadata, e.g. the first 2 sentence, title, author etc. It can also be generated by NLP techniques such as text summarization.

Dynamic summary

Key point: the summary is generated at query time

Display one or more "windows" on the document containing one or several of the query terms, also known as **keyword-in-context (KWIC)** snippets. Given a variety of keyword occurrences, the goal is to choose fragments which are maximally informative, self-contained, and short enough.

Usually we need to locally cache a fixed large-enough prefix (say 10k chars) of documents at index time, to be able to quickly reconstruct the context surrounding the query words.

Lecture 10a (Query Expansion, Relevance Feedback)

Problem: in most collections, the same concept may be referred to using different words, e.g. "aircraft" match "plane". This issue is known as **synonymy**.

Two major class of tackling this problem

1. global methods (**query expansion**) : reformulating terms independent of the query and results returned from it s.t. the new query match other semantically similar terms
2. local methods (**relevance feedback**) : adjusts a query relative to the docs that initially appear to match the query.

Relevance Feedback

Involves user in the retrieval process so as to improve the final result set. Generally the flow is similar to:

1. Retrieve an initial set of most relevant documents based on user query
2. Let the user mark some returned docs as relevant/nonrelevant
3. The IR system computes a better representation of user's information need and return a revised set of retrieval results.

Underlying idea: We want to find a query vector that maximizes similarity with relevant docs while minimizing similarity with nonrelevant docs.

Explicit RF

In light of partial knowledge of known relevant and nonrelevant documents, **Rocchio algorithm** proposes using the modified query $\overrightarrow{q_m} = \alpha\overrightarrow{q_0} + \beta \cdot \text{Centroid}(D_r) - \gamma \cdot \text{Centroid}(D_{nr})$ where q_0 be the original query vector, D_r and D_{nr} are the set of known relevant and nonrelevant documents respectively, and $\text{Centroid}(S) = \frac{1}{|S|}\sum_{\mathbf{s}\in S} \mathbf{s}$.

However, since it is easy for the query to leave the positive quadrant (which is not desirable), the Rocchio algorithm sets $\gamma = 0$

In practice, relevance feedback is **most useful to increase recall** and positive feedback are much more valuable than negative feedback, so most IR systems set $\gamma < \beta$.

When does RF work?

RF relies on the following assumptions

1. User's initial query at least partially works
   - Can be violated when there are misspellings, cross-language IR, mismatch of searcher's vocab with collection vocab (e.g. laptop vs notebook computer)
2. Require relevant documents to be tightly clustered
   - Can be violated if subsets of the docs uses different vocab (e.g. Burma vs Myanmar), disjunctive query, or instances of a general concept (e.g. felines)

However, RF is not necessarily popular with users, since it requires an extra interaction, it is hard to explain to users why a certain doc is retrieved, and long queries generated by RF are inefficient.

RF is also not popular in web search, because users care less about recall rate.

Pseudo RF

Also known as **blind relevance feedback** provides an automatic (without user interaction) technique to apply RF by assuming that the top k docs returned by the initial query as relevant, and continue to apply RF.

This technique mostly works, but can be affected by query drift.

Indirect / Implicit RF

Indirect RF is less reliable than explicit feedback, but is more useful than pseudo RF, which contains no user judgements. Moreover, it is easy to collect implicit feedback in large qualities in a high volume system like web search.

E.g. clicks on links were assumed to indicate that the page was likely relevant to the query, and thus the system ranks those documents more highly. This is one form of **clickstream mining**

Query Expansion

In RF, additional input (relevant/nonrelevant judgement) is given on **docs**, which is used to reweight terms in the documents.
In Query Expansion, additional input (good/bad search term) is given on **words or phrases**

Problem: How to generate alternative / expanded queries?
Common solution: use some form of thesaurus to generate synonyms and related words.

In general query expansion increases recall, but may decrease precision when terms are ambiguous, e.g. "interest rate" → "interest rate fascinate evaluate". Despite being less successful than RF overall, it might be as good as pseudo RF. It also has the advantage of being much more understandable to the user.

Methods for building thesaurus:

1. Use of a controlled vocab maintained by human editors, where there is a canonical term for each concept. It is common in well-resourced domain such as the medical domain.
2. Manual thesaurus. Human editors build up sets of synonymous names for concepts without designating a canonical term
3. Automatically derived thesaurus
4. Query reformulations based on query log mining : exploit manual query reformulations of other users to make suggestions to new users. Generally used in web search.

Automatically derived thesaurus

1. word co-occurrence statistics
2. shallow grammatical analysis of the text, e.g. we say entities that are grown, cooked, eaten and digested are most likely food items.

The simplest way to compute cooccurrence is based on term-term similarities in $C = AA^T$ where A is the term-document matrix.

Problems:

- term ambiguity introduces irrelevant statistically correlated terms, e.g. "apple computer" -> "apple red fruit computer"
- false positive and false negative
- terms in the automatic thesaurus are highly correlated in documents → may not retrieve many additional documents.

Lecture 10b (Structured Retrieval)

Queries in **structured retrieval** can be either structured or unstructured, but we will assume in this chapter that the collection consists only of structured documents, e.g. digital libraries, patent DBs, HTMLs, other document stored as markedup text.

Three main problems for relational DB

1. An unranked system (like a DB) can return a large set leading to information overload
2. Users often don't precisely state structural constraints – may not know possible structure elements are supported
   - tours AND (COUNTRY: Vatican OR LANDMARK: Coliseum)?
   - tours AND (STATE: Vatican OR BUILDING: Coliseum)?
3. Users may be unfamiliar with structured search and the necessary advanced search interfaces or syntax

Solution: adapt ranked retrieval to structured documents

Challenges in XML retrieval

1. Return parts of documents (but which part?) instead of full documents
2. What is the appropriate indexing unit?
3. Redundancy caused by nested elements
4. Schema diversity / structural mismatch

Vector space model for XML Retrieval

XML DOM can be visualized as a tree with text at leaves.

A simple vector space model for XML retrieval is based on **lexicalized subtrees**, subtrees of the XML DOM with at least one vocab term.

Each dimensions in the vector space are lexicalized subtrees, which encode the term and the position within XML tree. Compare this to VSM in unstructured retrieval where the dimensions are vocab terms.

There is a tradeoff between the dimensionality of the space and accuracy of query results. If we restrict dimensions to only vocab terms, then precision can suffer, whereas if we create a separate dimension for each lexicalized subtrees, the dimensionality of the space become too large (slow search).

As a compromise, we use **structural term**: a pair of <XML-context c (path to term), term t> as the dimension.

As user cannot remember the details of the schema, we interpret all queries as extended queries, i.e. there can be any number of intervening nodes between any parent-child node pair in the query.

Context Resemblance

A simple measure of similarity of a path c_q in a query and a path c_d in a document is defined as $\mathrm{CR}(c_q, c_d) = \frac{1+|c_q|}{1+|c_d|}$ if c_q matches c_d and 0 otherwise.

A variant of cosine similarity for structured retrieval

SimNoMerge(q,d) = Context Resemblance x Query Term Weight x Normalized Document Term Weight

- Not a true cosine measurement as it can exceed 1.

```
SCOREDOCUMENTSWITHSIMNOMERGE(q, B, V, N, normalizer)
 1  for n ← 1 to N
 2  do score[n] ← 0
 3  for each ⟨c_q, t⟩ ∈ q
 4  do w_q ← WEIGHT(q, t, c_q)
 5      for each c ∈ B
 6      do if CR(c_q, c) > 0
 7            then postings ← GETPOSTINGS(⟨c, t⟩)
 8                 for each posting ∈ postings
 9                 do x ← CR(c_q, c) * w_q * weight(posting)
10                    score[docID(posting)] += x
11  for n ← 1 to N
12  do score[n] ← score[n]/normalizer[n]
13  return score
```

Note: B is the set of all XML contexts

An alternative similarity function is **SimMerge** which relaxes the conditions of query and document by:

1. Collecting the statistics for computing query term weight and document term weight from all contexts that have a non-zero resemblance to the context c in line 5.
2. Merging all structural terms in the document that have a non-zero context resemblance to a given query structural term c_q.
3. Relaxing the conext resemblance function

Evaluation of XML Retrieval

To evaluate relevance for CAS (content-and-structure) queries, we define component coverage and topical relevance as orthogonal dimensions of relevance.

Component coverage: covers whether the element retrieved is "structurally" correct, i.e. neither too low nor too high in the tree. We distinguish into 4 cases:

1. Exact coverage (E) → main topic and self-contained
2. too small (S) → main topic but not self-contained (too detailed)
3. too large (L) → self contained, but not main topic (too broad)
4. no coverage (N)

Topical relevance: highly relevant (3), fairly relevant (2), marginally relevant (1), nonrelevant(0)

The relevance-coverage combinations $Q(rel, cov)$ are quantized as follows

1. 1.00 if (rel, cov) = 3E
2. 0.75 if (rel, cov) = 2E / 3L / 3S
3. 0.5 if (rel, cov) = 1E / 2L / 2S
4. 0.25 if (rel, cov) = 1S / 1L
5. 0 otherwise

This evaluation scheme takes account of the fact that binary relevance judgments, which are standard in unstructured information retrieval, are not appropriate for XML retrieval. With this quantization function Q, we are able to grade a component as **partially relevant.** The number of relevant items retrieved can be calculated as the sum of the relevance-coverage combinations.

Using structure in retrieval helps increase precision at the top of results list, although recall may suffer.