Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract Information Retrieval

Roman Böhringer

December 19, 2019

Department of Computer Science, ETH Zürich

Contents

Chapter 1

Introduction

Unstructured data (such as text, audio, video) grew much faster than structured data (graphs, trees, tables, cubes) during the last years. Information retrieval is about finding unstructured data that satisfies an information need from within large collections.

There is a semantic gap between the representation of information and the semantic description of information. Data becomes information with meaning, information becomes knowledge with a meaningful application.

The three paramount factors when dealing with data are capacity (how much data can we store?), throughput (how fast can we transmit data?) and latency (when do I start receiving data?). While capacity has increased a lot in the last decades, throughput and especially latency couldn't catch up.

1.1 History

To find books in libraries, the dewey decimal system was invented. It consists of three numbers, the first indicates the category, the second the subcategory and so on.

Before relational databases, there were file systems in the 1960s. In the 1970s, relational databases were introduced. The 1980s are known as the "object era" whereas the 2000s are the NoSQL era (key-value stores, triple stores, column stores, document stores, ...). Big data consists of the three Vs: Volume, variety and velocity.

1.2 Prefixes

The International System of Units defines these prefixes:

- **kilo (k)**: 1'000 (3 zeros)
- **Mega (M)**: 1'000'000 (6 zeros)
- **Giga (G)**: 1'000'000'000 (9 zeros)
- **Tera (T)**: 10^{12}
- **Peta (P)**: 10^{15}
- **Exa (E)**: 10^{18}
- **Zetta (Z)**: 10^{21}
- **Yotta (Y)**: 10^{24}

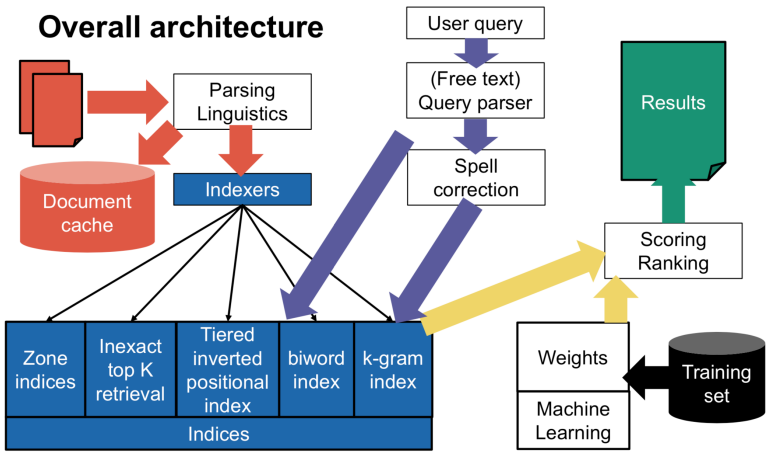
1.3 Data Model

We want to retrieve documents. Documents contain terms, where a term can be included in the document or not, can have a certain occurrence and can have an order. Documents can be represented as:

- List of terms (inclusion, occurrence and order)
- Bag of terms (inclusion, occurrence)
- Set of terms (inclusion), equivalent to a term-document bipartite graph

1.4 Overall Architecture

The overall architecture of an information retrieval system that is described in this summary can look like this:



Chapter 2

Boolean Retrieval

Boolean retrieval uses the set-of-terms data model, i.e. it is captured whether a document contains a term or not. A method to store this information is the incidence matrix where the columns are documents and the rows are terms. An entry $A_{i,j}$ is 1 if document j contains term i, 0 otherwise. The size of the matrix is therefore $\mathbb{B}^{\text{Terms} \times \text{Documents}}$.

2.1 Inverted Index

The incidence matrix is huge and has a lot of zeros. The inverted index therefore only stores the list of documents that contain the term for every term. We assume that each document has a document id, the combination $(\text{term}, \text{document id})$ is then called a posting. The list that is stored for each term is called a posting list, the combination of all terms and their posting list is the dictionary. Besides the posting list, the document frequency (number of documents a term appears in / length of the posting list) is usually also stored per-term.

The creation of an inverted index consists of:

1. Tokenization
2. Linguistic pre-processing
3. Assignment of document IDs (to get postings)
4. Sorting of the postings by term
5. Merging of the postings with the same term (to get the posting lists)
6. Addition of the document frequency

In step 3, we can use term IDs instead of the actual terms to save space (for the intermediate result).

For AND queries, the posting lists are intersected, for OR queries their union is calculated. For the intersection, two pointers are used. Whenever there's a match, both pointers are advanced and the document id is added to the result set. When there's no match, only the pointer that points to the smaller element is advanced.

When a query contains an AND NOT, the intersection algorithm can be adapted to "virtually" walk down the negated list, i.e. there's no need to materialize the negated list.

Because the gaps between document IDs can be very varying in different lists, skip lists can be introduced that allow the intersection algorithm to skip intermediate terms by storing additional pointers. The optimal skip distance is $\sqrt{\text{Number of postings}}$

When there is a query with multiple ANDs, the query terms can be sorted by increasing document frequency and the intersection of lists can be calculated according to this order (to keep intermediate results small).

2.2 Index construction

2.2.1 Collecting documents

Documents are a sequence of characters that are encoded in a certain character set. UTF-8 is a variable length character encoding. A character has a certain codepoint that is then encoded using a variable length encoding scheme.

The first challenge when retrieving documents is to identify the encoding which can be user-defined, annotated in document, learned with machine learning, ... There are also software-specific encodings (e.g. Microsoft Word) or data-specific encodings like XML special characters.

The second challenge is to define what constitutes a document because a file isn't always a good definition (e.g. for LaTeX source).

2.2.2 Tokenization

Punctuation is thrown away which isn't always trivial (e.g. e-mail addresses, isn't, etc...).

A token is a grouped sequence of characters. A type is an equivalence class, for instance "be" and "is" or "U.S.A" and "USA" could be the same type. There are certain stop words that occur very often and are often not considered when building the index.

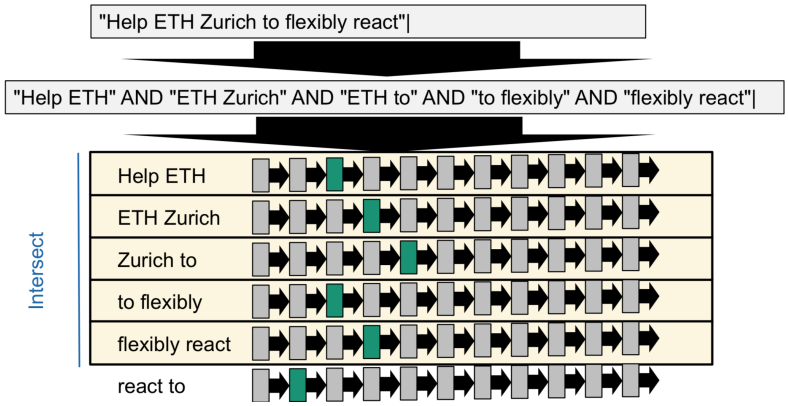
Normalization normalizes types to the same word, e.g. "U.S.A" is changed to "USA". Often times, accents / diacritics are removed, all words are low-

ercased (or truecased, i.e. the company ”Apple” is left as ”Apple” whereas the fruit is changed to ”apple”). Stemming is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form — generally a written word form. The Porter, Lovins and Paice stemmer are common stemmers. In contrast to Lemmatization, Stemming is very mechanical and rule-based.

It’s also possible to do expansion, e.g. change ”windows” to ”windows”, ”Windows” and ”window” but ”Windows” only to ”Windows”. This introduces asymmetry. Expansion can be done upon indexing (which increases the size of the dictionary) or upon querying (a query like ”lift” is changed to ”lift OR elevator”). Lemmatization builds equivalence classes / expands with natural language processing. Unlike stemming, lemmatization depends on correctly identifying the intended part of speech and meaning of a word in a sentence, as well as within the larger context surrounding that sentence, such as neighboring sentences or even an entire document.

2.3 Phrase Search

A simple posting list doesn’t allow phrase searches because we have no information on the proximity of terms. A possible solution is to build bi-word indices that index two words (i.e. in which documents the two words appear after another). The query can then be extended to bi-words:



This introduces false positives, the results have to be post-processed therefore.

More generally, one can build phrase indices (n) where n is the number of words that are indexed. This reduces the number of false positives, but the size of the vocabulary increases (exponentially).

2.3.1 Positional Index

An alternative is to store positional postings in the posting list. They contain the document id, the term frequency (the number of occurrences of the term in the document) and a list of positions (where does the term appear in the document). When querying, the intersection is first done according to the intersection algorithm and the lists with the positions are then also compared / intersected. With this approach, there are no false positives.

2.4 Storage

We can store the posting list as a hash table, where the terms are the keys and the lists the values. This enables constant time lookups, but there's no support for range queries, hash functions are not perfect in real life and there are space requirements for collision avoidance.

2.4.1 B+-tree

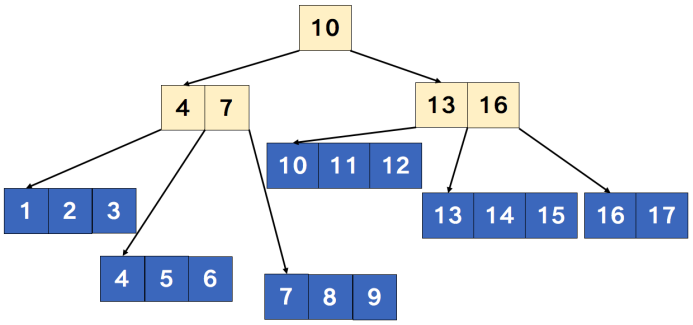
A $(d+1)-(2d+1)$ B+-tree is a tree where the number of children is between $d+1$ and $2d+1$ for any node (except the root node, it can only have two children) and all leaf nodes are at the same depth. A node with $d+1$ children has d keys, therefore the number of keys (per node) is between d and $2d$.

In a B+-tree, the terms (and posting lists) are only at the leaves, whereas in a B-tree, they are also stored at intermediate nodes.

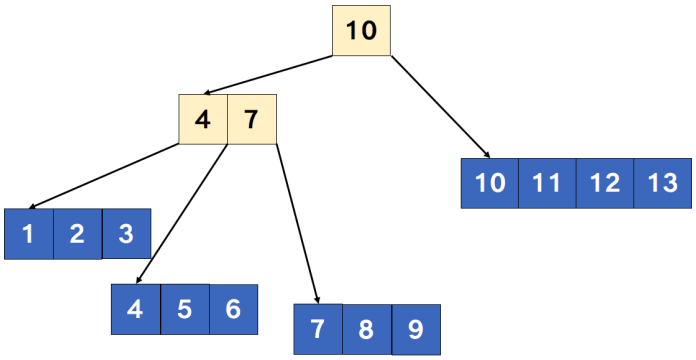
An advantage of B+-trees is that the tree without the leaf nodes can be kept in memory, whereas the terms and the posting lists are stored on disk. Because disks favor sequential accesses, this increases the read performance.

Example 1: 3-5 B+ tree

The following tree is a 3-5 B+ tree:



The following is NOT, because not all leaf nodes are at the same depth:



Example 2: Number of terms

How many terms can a 4-7 B+ tree of depth 3 contain at most?
For the first 3 levels, the nodes on each level can have at most 7 children. The leaf nodes can consist of 6 keys at most, we therefore have: $7^3 * 6 = 2058$
How many terms can a 4-7 B+ tree of depth 3 contain at least?
The root node can only have two children, the nodes on the second / third level have at least 4 children and the leaf nodes have to consist of at least 3 keys, therefore: $2 * 4 * 4 * 3 = 96$.

Chapter 3

Tolerant Retrieval

3.1 Wildcard queries

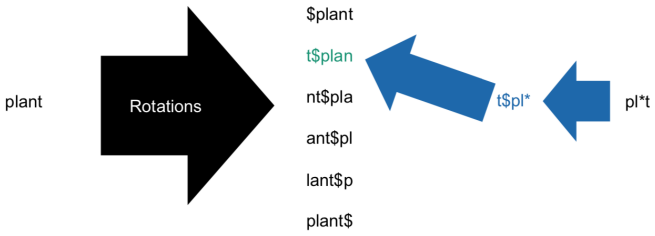
The wildcard symbol * matches any sequence of characters.

Trailing wildcard queries such as ”Math*” are very easy to handle with a B+-tree, we just do a range query. To handle leading wildcard queries like ”*ics”, we can build a B+-tree on the reversed terms and then do a range query (with the inverted query term, e.g. ”sci*”).

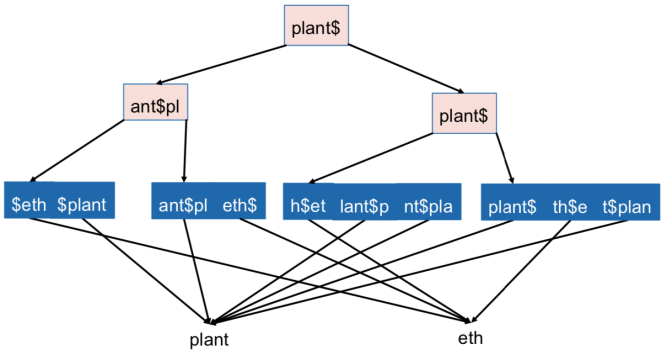
Both approaches can be combined to handle single-wildcard queries like ”Zu*ch” by rewriting the query to ”Zu* AND *ch” (i.e. doing a range query on the B+-tree, reverse B+-tree and intersecting the results). This gives false positives. For example, ”statics” would match ”stati*tics” with this approach, so the results need to be post-filtered.

3.1.1 Permuterm Index

Another option for single-wildcard queries is to build a permuterm index. A special character like ”$” is added in front of a term and every permutation of the term is mapped to the term. A wildcard-query is then also rotated such that it becomes a trailing wildcard query on the permuterm index:

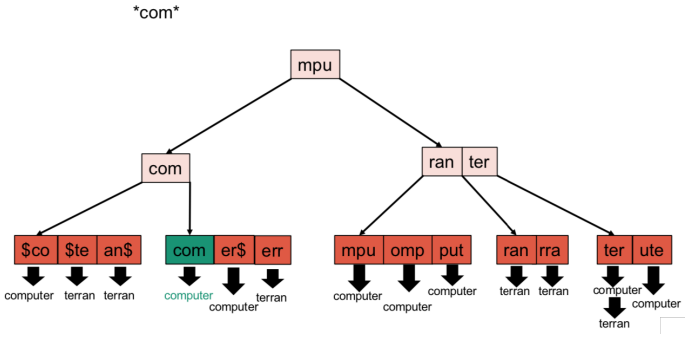


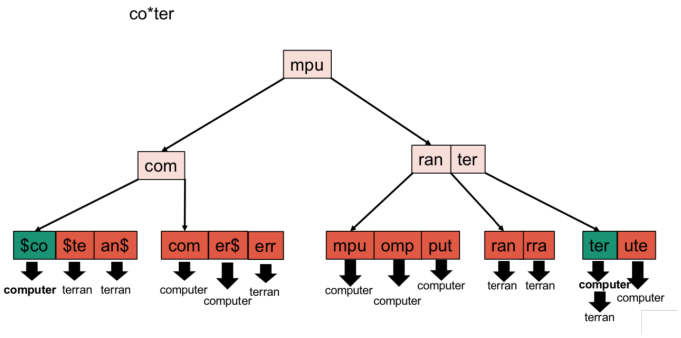
To allow this, a B+-tree is built from the permutations to the terms:



3.1.2 k-gram Index

A special character like "$" is added in front and at the end of every word. For instance, "computer" becomes "$computer$" and if we use 3-grams, we have "$co", "com", "omp", ... We can then build a B+-tree from the k-grams to terms and query this tree. This allows queries with multiple wildcards:





3.2 Spelling Correction

There are different approaches to spelling errors. One can always query also for corrected terms, only query for corrected terms if the query is not in the dictionary or if it isn't in the dictionary and when there aren't enough results. Besides changing the query, an engine can also make a spelling suggestion.

3.2.1 Edit Distance

The edit distance is the minimum number of operations required to transform one string into the other where (in the case of the Levenshtein distance) deletion, insertion and substitution is allowed. The problem can be solved with dynamic programming. Let $D[i,j]$ be the edit distance between length-i prefix of x and length-j prefix of y, $D[0,j] = j$ and $D[i,0] = i$. We can then update the distance according to:

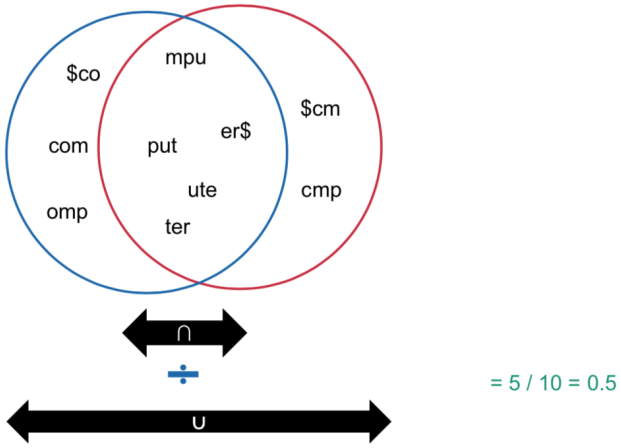
$$D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1]+\delta(x[i-1],y[j-1]) \end{cases}$$

Where $\delta(a,b)$ is 0 if $a = b$, 1 otherwise.

The first case is deletion (from x), the second case insertion (into x) and the third case replacement or matching / doing nothing.

3.2.2 Jaccard Coefficient

Because computing the edit distance for all terms is very expensive, we assume that two terms within a small edit distance have many k-grams in common. The Jaccard coefficient between two sets of k-grams is the size of the intersection divided by the size of the union. The sets of k-grams of two terms that are very similar have a high Jaccard coefficient:



The method for spelling correction therefore looks like this:

1. Get k-grams from the query term
2. Look them up in the *k*-gram index
3. Compute Jaccard coefficients
4. Keep terms with large Jaccard coefficients

The Jaccard coefficient can be computed efficiently because we already know the k-grams that the query term and the found term have in common (from the k-gram index) and can use the fact that $|A \cup B| = |A| + |B| - |A \cap B|$ to compute the denominator with just the found term's number of k-grams.

3.2.3 Context-sensitive Spelling Correction

Isolated-term correction fails to correct typographical errors such as "graduate form ETH". To mitigate this, one can enumerate the alternatives for each word and count the number of results. Because this is expensive, an alternative is to look up commonly searched biwords from other users, e.g. "graduate from" and "from ETH", combine those to an alternative query and count the number of results.

3.2.4 Phonetic Correction

The soundex algorithm maps every word to a 4 character fingerprint. The first character is kept, all other are changed to a number according to a table. Then, duplicates (numbers that appear consecutively) are removed / merged to only one number and zeros are removed. Finally, the string is trimmed to 4 characters (or padded with zeros if shorter than 4 characters).

Chapter 4

Index Construction

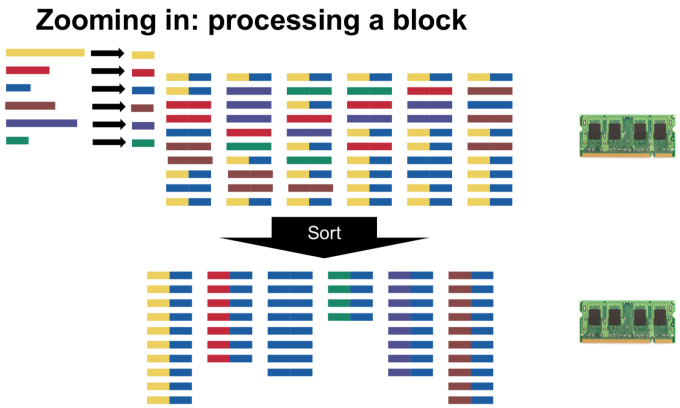
4.1 Hardware

Disks store data in blocks and data is sent to and from memory in blocks. Because seek latency is quite high, we want to avoid random accesses and read sequential data.

4.2 Blocked Sort-Based Indexing

With blocked sort-based indexing, we shard the collection of documents into different blocks and process the blocks one by one in memory. Intermediate results of a block (the posting list of this group) are written back to disk and in the end, the intermediate results are merged into the final index.

When processing a block, the (termID, docID) pairs are parsed and read into memory. They are then sorted and written back to disk.



For the merge step, a pointer is kept on every block (on disk) and the final posting list is constructed in memory.

The complexity for the first step is $O(T \log T)$ and for the second step $O(T)$, overall therefore $O(T \log T)$ (where T is the number of tokens).

4.3 Single-Pass In-Memory Indexing

As with BSBI, the collection is sharded into different blocks. But the posting list for each block is built and then written back to disk (not the sorted pairs as with BSBI).

As with BSBI, the posting lists are merged in the end. To do this, the document IDs for a term are accumulated in memory and flushed to disk. Then, the document IDs for the next term are accumulated in memory and so on...

The complexity for the first step is $O(T \log M)$ (where M is the number of terms) because we only sort the terms (and the T comes from the parsing) and $O(T)$ for the merging, so overall $O(T)$.

4.4 MapReduce

MapReduce is a programming model where a program consists of a map phase, a shuffle phase and a reduce phase. The map function takes a series of key/value input pairs, processes each and generates zero or more output key/value pairs. The framework calls the applications reduce function once for each unique key in the sorted order.

MapReduce can be used to build an index. The mappers get documents as input and output postings. The reducers take all postings with the same term and generate the posting list for this term.

4.5 Online Index Construction

A simple solution is to reconstruct the index completely when new documents are added. But this can result in staleness of the results and a potentially unavailable or slower system during index construction.

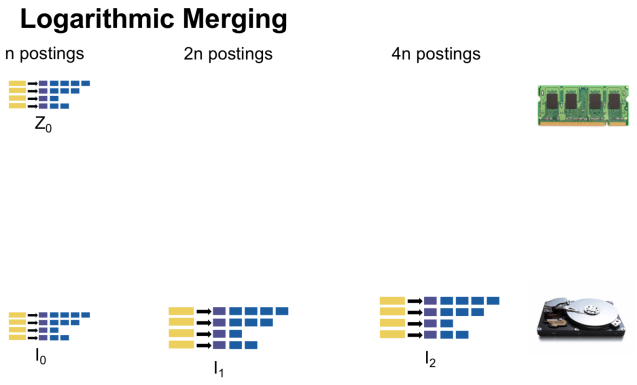
Another solution is to have the main index on disk and an auxiliary index (with the posting list of the added documents) in memory. Both indexes are then queried. When the auxiliary index gets too large, it is merged with the main index. For deletions, one can use an invalidation bit vector.

If we have T incoming postings and merge every n postings (i.e. always when the auxiliary index contains n postings, therefore at n total postings,

$2n$ total postings, etc...) we have a runtime of:

$$O(n + 2n + \ldots + \frac{T}{n}n) = O(\frac{T^2}{n})$$

The runtime can be improved with logarithmic merging. There, we have $\log_2(T/n)$ indexes of size $2^0 * n$, $2^1 * n$, $2^2 * n$, ... As before, up to n postings are accumulated in an in-memory auxiliary index. When n is reached, the postings are transferred to an index on disk. The next time the in-memory auxiliary index is full, it is merged with the index on disk to create an index of size $2n$. This is either stored on disk (if no index of size $2n$ exists on disk) or merged with an existing index of size $2n$ to create one of size $4n$. For instance, in the following example, we would create a new index of size $8n$, because there exists already one of size n, $2n$ and $4n$:



The construction time is $O(T \log(\frac{T}{n}))$ because each posting is processed once on each of the levels. But query time becomes $O(\log(\frac{T}{n}))$ because results of all levels need to get merged.

Chapter 5

Index Compression

5.1 Term Statistics

We use the following notation:

- N: Number of documents
- T: Number of tokens (positional postings)
- M: Number of terms (or types if stemming / lemmatization is used)

Heap's law states the relationship between the number of terms and the number of tokens. It is:

$$M = kT^b$$

In practice, b is around $\frac{1}{2}$, so we have $M = k\sqrt{T}$ with $30 \leq k \leq 100$.

Zipf's law states the relationship between term frequency and its rank. According to it, the relationship is:

$$\text{Frequency} = \frac{k}{\text{Rank}}$$

5.2 Compression Techniques

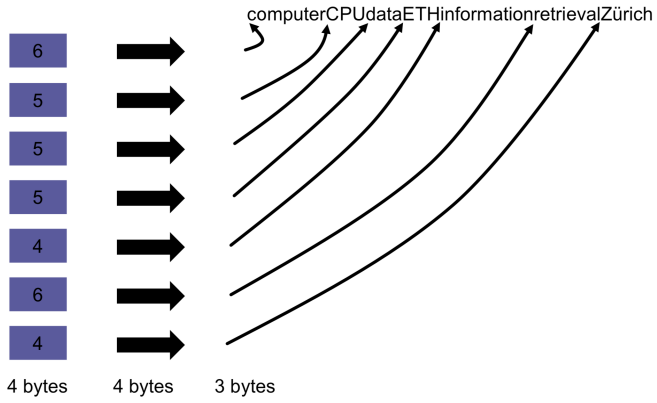
Some compression techniques that were already described are:

- Remove numbers
- Case folding
- Remove stopwords
- Stemming / Lemmatization

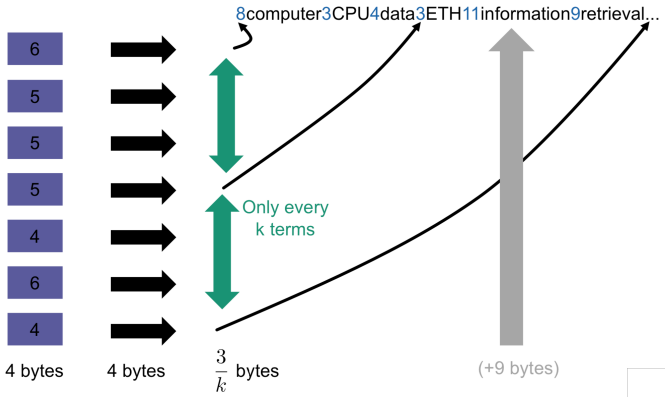
5.2.1 Dictionary Compression

A naive approach would be to store each dictionary entry as a fixed size array. Then, the maximum term length would be limited and a lot of space would be wasted.

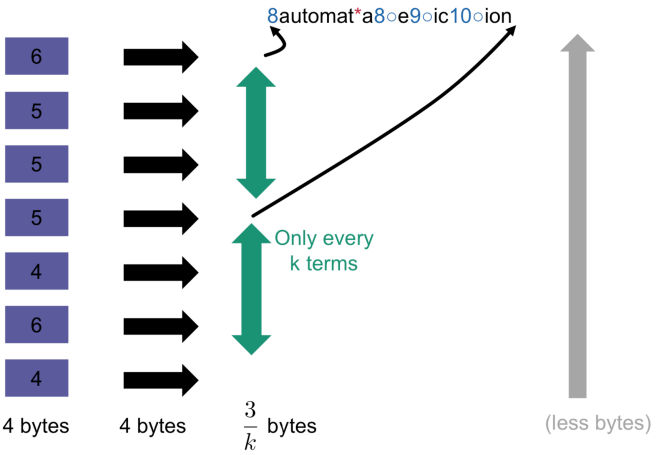
An alternative is to store one long string that contains all entries of the dictionary and store pointers to the word (in the image, black arrow indicates pointer to posting list):



Instead of storing pointers at every term, one can only store a pointer every k terms but store the word length in the string. This is called blocked storage:



Front coding further improves this by not storing common prefixes:



5.2.2 Postings File Compression

The standard storage for the postings file stores each document ID as a 4 byte integer. Instead of encoding the actual document ID, we can also encode the gaps between document IDs (but this only works for frequent terms). To encode those (hopefully small numbers), we can make use of variable length encodings.

5.3 Variable Length Encoding

Prefix codes allow us to deduce from the bits when a number stops / the next begins (in contrast to fixed length encoding where the boundaries are fixed).

5.3.1 Variable Byte Encoding

In a variable byte encoding, the first bit is a continuation bit (0 if the number doesn't end here, 1 otherwise) and the rest of a "packet" (where a packet can have different size, e.g. 8 bit) is used for the actual encoding.

Big packets lead to little overhead but also little compression whereas small packets lead to much compression (for small numbers) but there is a lot of bit manipulation involved.

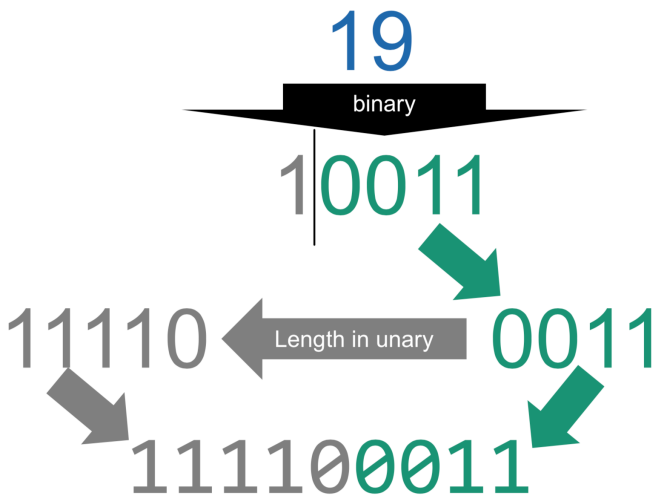
5.3.2 Gamma Encoding

Unary Code

Unary code, also called "thermometer code", is an encoding scheme that encodes a number n with n 1s, followed by a zero. 2 is therefore 110, 4 is 11110 and so on.

Elias Gamma Code

In the gamma encoding, a number is encoded in its binary representation, the leading 1 is removed and the length of the remaining bit-string is encoded in unary. This unary encoding and the remaining bit-string gives then the gamma encoding of the number:



0 has no gamma code and 1 is "0".

The gamma code is a universal encoding. This means that, for any probability distribution, the expected length of the gamma code is only a constant factor larger than optimal encoding ($H(X) = \mathbb{E}[I(X)]$).

5.4 Shannon Entropy

Shannon defined the entropy H as:

$$H(X) = \mathbb{E}[I(X)] = \mathbb{E}[-\log_2(p_X(X))] = -\sum_{x \in X(\Omega)} p_X(x) \log_2 p_X(x)$$

Where I is the information content of X (number of bits).

A deterministic distribution (with $p_X(0) = 1$) therefore has entropy 0 (because $-log_2 p_X(0) = 0$). The uniform distribution between 0 and 3 has entropy 2 ($4 * -p_X(i) * \log_2 p_X(i) = 4 * \frac{1}{4} * 2 = 2$) which is maximal.

Chapter 6

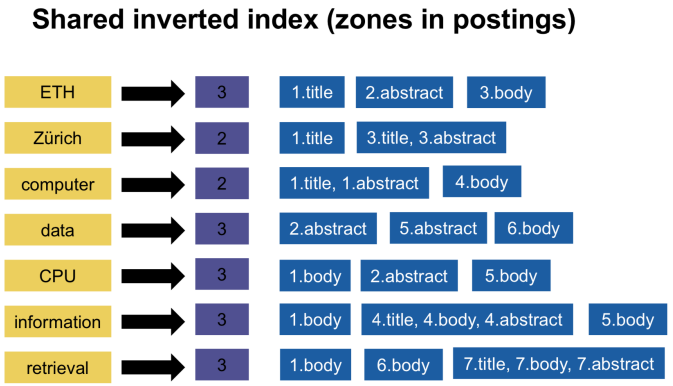
Ranked Retrieval

6.1 Parametric Indexes

We sometimes want to allow the user to search certain fields / metadata of a document. This is a classical database problem and we can build parametric indexes (using hash tables or trees such as B-trees / B+-trees) on the fields we expose to the user. The results can then be intersected (e.g. if a user wanted to retrieve all books with a given title and author).

6.1.1 Zone Search

On certain metadata fields, we want to allow full-text search (e.g. title). Therefore, we also build a standard inverted index on these fields. We can have individual inverted indexes or a shared inverted index. With a shared inverted index, we can have the zones in the terms by extending the terms with the field. We would then have terms such as "ETH.body", "ETH.title", etc... We can also have the zones in the postings by storing the field alongside each posting:



6.1.2 Scores

The score of a document is $\sum_{i=1}^{l} g_i s_i$ where the g_i's are the zone weights and s_i boolean results (1 if zone i contains the term, 0 otherwise). The results can then be sorted according to the score to get a ranked result.

To find the weights, we want to solve the following optimization problem using machine learning:

$$\text{argmin}_g \sum_j \text{error}_j(g) = \text{argmin}_g \sum_j (r_j - g_j.s_j)^2$$

Where r_j is a relevance judgment (is document with boolean vector s_j relevant?).

6.2 Ranked Retrieval

In ranked retrieval, we use the bag of words model. One possibility is to use the term-frequency (how often a term appears in a document) as document weights. If we simply use the term-frequency, a problem is that rarer terms have the same weight as very common terms.

The collection frequency counts how often a term appears (in the whole collection; i.e. the sum of the tf's over all documents) whereas the document frequency (df_t) counts in how many distinct documents a term appears. The inverse document frequency is defined as:

$$\text{idf}_t = \log \frac{N}{\text{df}_t}$$

Where N is the total number of documents. tf-idf multiplies the term frequency with the idf.

6.2.1 Vector-Space Model

In the bag of words model, we can represent a document as a vector of real numbers (that can be term frequency, tf-idf or some other weighting schemes). To get the similarity between two documents, we can calculate the inner product (which gives the cosine of the angle if the vectors are normalized). We can also represent queries as vectors and calculate the similarity between the query and the documents.

We can store the precomputed tf_{td} in the postings and the precomputed idf_t at the beginning of each terms posting list. To do the actual computation, we have accumulators for each document and iterate over all terms (term-at-a-time) or all documents (document-at-a-time). For optimization purposes, we can divide by the query length at the end.

We can then construct a priority queue ($O(N)$ for the construction) to extract the top K results in $O(\log(N))$ runtime.

6.2.2 Alternate Weights

Sometimes it's useful to scale the term frequency. In the natural term frequency scaling (n), we take the term frequency directly as weight.

In the sublinear term frequency scaling (l), we use:

$$\mathrm{wf}_{t,d} = \begin{cases} 1 + \log \mathrm{tf}_{t,d} & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Augmented term frequency scaling (a) is defined as:

$$\mathrm{wf}_{t,d} = a + (1-a)\frac{tf_{t,d}}{tf_{max(d)}}$$

Boolean term frequency scaling (b) simply assigns 1 if the term frequency is larger than 0, 0 otherwise.

No document frequency scaling (n) means multiplying by 1. Inverted document frequency scaling (t) is the idf_t as defined above. There's also prob idf document frequency scaling (p):

$$\text{prob-idf}_t = \max\left(0, \log\frac{N - \mathrm{df}_t}{\mathrm{df}_t}\right)$$

There are also different normalization schemes:

- No normalization (n)
- Cosine normalization (c): $\frac{\vec{x}}{||\vec{x}||} = \frac{\vec{x}}{\sqrt{\sum_{i=1}^{d} x_i^2}}$
- Byte-size normalization (b): $\frac{1}{\mathrm{CharLength}^{\alpha}}$
- Pivoted normalization (p): $a||d|| + (1-a)\mathrm{piv}$

The SMART notation is a way to summarize the used weighting schemes. The first three characters (before the dot) are about the document weights, the last three characters (after the dot) about the query weigths. The first character is about the term frequency, the second about the document frequency and the third about the normalization.

If we care about the relative ordering of the results, normalizing by the query length isn't needed because it's a constant factor for every document.

6.2.3 Inexact Top-K Document Retrieval

The main idea behind inexact top-K retrieval is to preselect some documents and only compute scores in this smaller set. A first approach would be to only consider the query terms with a high idf_t (i.e. rare terms) which results in much less documents to compute the scores for. Another approach would be to keep documents containing many (or all) query terms and drop those containing only a few terms.

One can also maintain so called champion lists. In these lists, the documents are sorted by decreasing term frequency. Then, only the top r documents are kept and at query time, the top r documents from each term are united (and weights are only computed for those documents). Instead of using only the term frequency, we can also assign documents a quality score and sort them by the term frequency plus the quality score.

With the champion lists, the documents are no longer sorted by ID, therefore the intersection algorithm won't work anymore. A solution to this is to keep the posting list in the common order and build the champion lists separately.

Impact ordering sorts the terms by idf and the documents by tf (or any other term-wise impact document score). We then start at the top left and go right and down until there are enough results.

We can use these approaches to build tiered indexes. If the Tier I indexes (that contain documents with the highest scores) don't return enough results, we query the Tier II indexes, and so on.

One thing that can also be considered is the query term proximity, i.e. the distance between two subsequent words of a query in a document.

Clustering

In this approach, we randomly pick leaders (e.g. $\sqrt{N}$). The document space is divided into clusters according to the nearest leader. We then pick the leader closest to the query and compute the scores only for the local cluster.

Chapter 7

Evaluation

In an evaluation setup, we pick a document and an information need and ask an expert if the document is relevant to the information need. There are multiple datasets like Cranfield or TREC where this has been done for many documents / information needs.

An information need is different from a query, a query is a way to represent a specific information need.

7.1 Metrics

7.1.1 Judgement

If we have such a table with the results of two judges:

	relevant	**not relevant**	**total**
relevant	18	8	26
not relevant	2	32	34
total	20	40	60

We can calculate the probability of agreement, which is the number of observed agreements divided by the total assessments, in this example therefore:

$$P(A) = \frac{50}{60}$$

The marginals for relevant / not relevant are the number of observed relevant / not relevant statements divided by the overall statements, in this example therefore:

$$P(R) = \frac{20+26}{60+60} = \frac{46}{120} \quad P(NR) = \frac{40+34}{60+60} = \frac{74}{120}$$

The probability of agreement by chance is:

$$P(E) = P(R)^2 + P(NR)^2$$

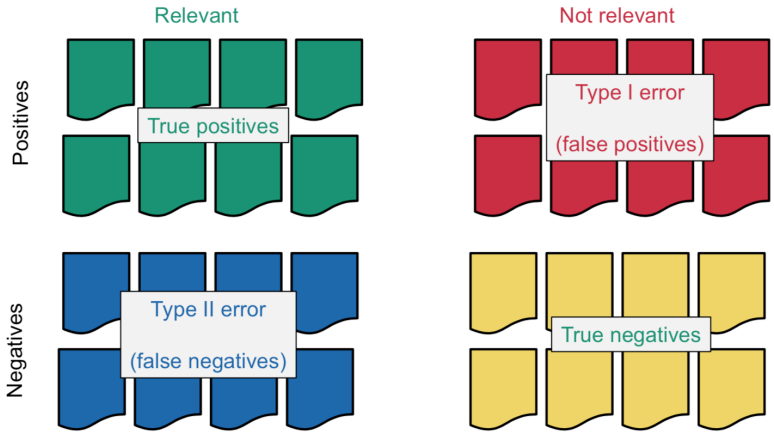
And the Kappa statistic is defined as:

$$\kappa = \frac{P(A) - P(E)}{1 - P(E)}$$

It is used to measure inter-rater reliability. If the raters are in perfect agreement, $\kappa = 1$. If there is no agreement among the raters other than what would be expected by chance, $\kappa = 0$. If the agreement is even worse than by chance, κ is negative.

7.1.2 Performance

We can divide the documents into four cases:



We then have the following metrics for assessing the performance:

- **Precision**: $\frac{\text{True positives}}{\text{Returned / Positives}}$
- **Recall**: $\frac{\text{True positives}}{\text{Relevant}}$
- **Specificity**: $\frac{\text{True negatives}}{\text{Not relevant}}$
- **Accuracy**: $\frac{\text{True positives + True negatives}}{N}$

Getting a high accuracy is quite easy (don't return anything because usually almost all documents are true negatives), getting 100% recall is also easy (return everything).

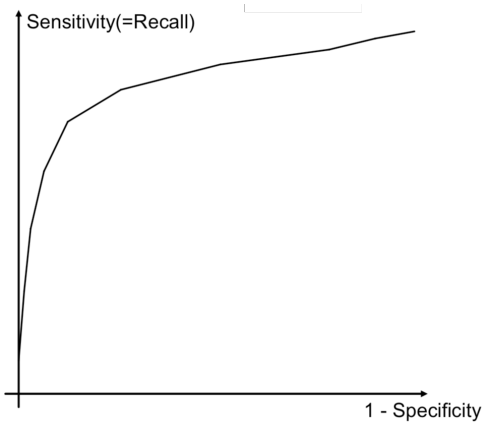
For this reason, the F measure is introduced:

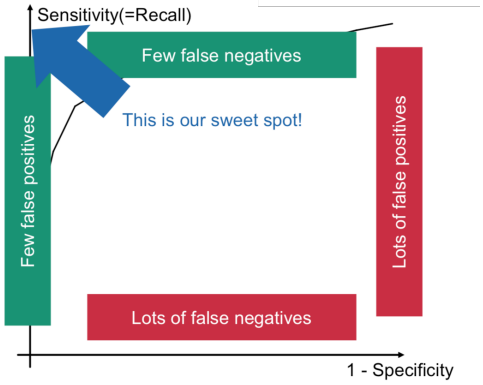
$$F_{0.5} = \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

The general F measure allows weighting of recall / precision:

$$F_\alpha = \frac{1}{\frac{\alpha}{P} + \frac{1-\alpha}{R}}$$

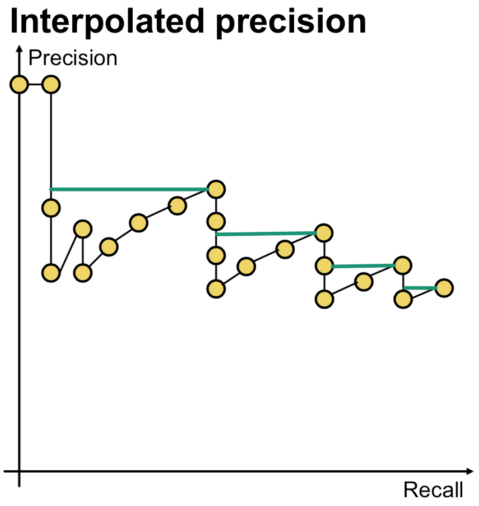
If we plot the sensitivity / recall against the false positive rate (1 - specificity), we get the ROC curve:





Ranked Retrieval

When we plot precision against recall in a ranked retrieval setting, we get a shape like this and can interpolate the precision:



The eleven-point interpolated average precision interpolates the precision against recall at eleven points.

Average precision is the area under the precision-recall curve. This curve is only for one query. If we average the curves for many / all queries and take the mean value, we get the mean average precision (MAP).

If we have k relevant documents and return the top k documents, precision is equal to recall. This point is called R-Precision and can also be visualized as the intersection of a line with slope 1 to the precision-recall curve.

Chapter 8

Probabilistic Information Retrieval

8.1 Probability Theory

We have a universe Ω with elementary events $\omega \in \Omega$. A probability distribution p assigns each element in Ω a probability between 0 and 1 and we have $\sum_{\omega \in \Omega} p(\omega) = 1$. An event E is a subset of Ω and $p(E) = \sum_{\omega \in E} p(\omega)$. The odds of an event are defined as $O_p(E) = \frac{p(E)}{p(\bar{E})}$. We have $P(E \cup F) = P(E) + P(F) - P(E \cap F)$ and therefore $P(E \cup F) = P(E) + P(F)$ for disjoint events.

The partition rule says: $p(E) = p(E \cap F) + p(E \cap \bar{F})$

Conditional probabilities are defined as: $p(E|F) = \frac{p(E \cap F)}{p(F)}$ From this, we can derive the chain rule: $p(E \cap F) = p(E|F) * p(F)$

Two events are independent if $p(E|F) = p(E)$ and therefore $p(E \cap F) = p(E) * p(F)$.

Bayes rule is:

$$p(E|F) = \frac{p(F|E) * p(E)}{p(F)}$$

Where we call $p(E|F)$ the posterior, $p(E)$ the prior.

A random variable assigns each event a value in some set S, i.e. $X : \omega \rightarrow X(\omega)$. We can then ask what the probability of a certain value is, which is defined as: $p_X : x \rightarrow \sum_{\omega | X(\omega) = x} p(\omega)$ An alternate notation is $P(X = x)$, $P(x)$ should never be used.

The joint probabilities are defined as: $p_{XY}(x, y) = \sum_{\omega | X(\omega) = x \wedge Y(\omega) = y} p(\omega)$

And the conditional probabilities: $p_{X|Y}(x, y) = \frac{p_{XY}(x,y)}{p_Y(y)}$

8.2 Probability Model

Like in the evaluation setup, we assume a collection of documents and a set of information needs. An elementary event is that we gave a domain expert one query and one document and the domain expert said whether the document is relevant to the query.

We can then regard the query, document and relevance as random variables. We want to find out $P(R = 1|D = d \wedge Q = Q)$ i.e. the probability that for a query q and a document d, d is relevant for query q. In ranked retrieval, we would sort by these probabilities. In boolean retrieval, return the document if the probability is greater than 0.5. Using this model, we can also have a system where false positives / negatives have different costs. If a false positive has cost $-C_0$ and a false negative $-C_1$, we would sort by increasing $C_0 * P(R = 0|D = d \wedge Q = Q) - C_1 * P(R = 1|D = d \wedge Q = Q)$

We denote by $P(D_k = d_k)$ the probability that the document contains term k (we therefore use the set-of-words model) and make the naive Bayes assumption that terms occur independently in documents, therefore:

$$P(D = d) = \prod_{k=1}^{M} P(D_k = d_k)$$

From Bayes formula, we can derive:

$$P(R = 1|D = d \wedge Q = q) = \frac{P(D = d|R = 1 \ \wedge Q = q) * P(R = 1|\ Q = q)}{P(D = d|Q = q)}$$
$$P(R = 0|D = d \wedge Q = q) = \frac{P(D = d|R = 0 \ \wedge Q = q) * P(R = 0|\ Q = q)}{P(D = d|Q = q)}$$

Hence:

$$\begin{aligned} O(R = 1|\ D = d \wedge Q = q) &= \frac{P(R = 1|D = d \wedge Q = q)}{P(R = 0|D = d \wedge Q = q)} \\ &= \frac{P(D = d|R = 1 \ \wedge Q = q) * P(R = 1|\ Q = q)}{P(D = d|R = 0 \ \wedge Q = q) * P(R = 0|\ Q = q)} \\ &= \frac{P(D = d|R = 1 \ \wedge Q = q)}{P(D = d|R = 0 \ \wedge Q = q)} * O(R = 1|Q = q) \\ &= \prod_{k=1}^{M} \frac{P(D_k = d_k|R = 1 \ \wedge Q = q)}{P(D_k = d_k|R = 0 \ \wedge Q = q)} * O(R = 1|Q = q) \end{aligned}$$

Introducing

$$p_k = P(D_k = 1|R = 1 \ \wedge Q = q) \quad u_k = P(D_k = 1|R = 0 \ \wedge Q = q)$$

we can rewrite this:

$$\prod_{k|d_k=1} \frac{p_k}{u_k} * \prod_{k|d_k=0} \frac{1-p_k}{1-u_k} * O(R=1|Q=q)$$

If we assume, that when a term doesn't appear in the query ($q_k = 0$), $p_k = u_k$ holds (probability that it appears in the document doesn't depend on the relevance), we can rewrite this to:

$$\prod_{k|d_k=1 \wedge q_k=1} \frac{p_k}{u_k} * \prod_{k|d_k=0 \wedge q_k=1} \frac{1-p_k}{1-u_k} * O(R=1|Q=q)$$

$$\prod_{k|d_k=1 \wedge q_k=1} \frac{p_k * (1-u_k)}{u_k * (1-p_k)} * \prod_{q_k=1} \frac{1-p_k}{1-u_k} * O(R=1|Q=q)$$

Only the first product depends on d, so it's the only thing we care about. Taking the logarithm gives the retrieval status value:

$$\mathrm{RSV}_d = \sum_{k|d_k=1 \wedge q_k=1} \log \frac{p_k}{1-p_k} - \log \frac{u_k}{1-u_k}$$

Where the first term is the log of the odds of containing term k in relevant documents and the second term the log of the odds of containing term k in non-relevant documents.

If we assume 1 for the first term and $\frac{\mathrm{df}_k}{N-\mathrm{df}_k}$ for the second, we get approximately the inverted document frequency:

$$\mathrm{RSV}_d = \sum_{k|d_k=1 \wedge q_k=1} \log \frac{N-\mathrm{df}_k}{\mathrm{df}_k} \approx \sum_{k|d_k=1 \wedge q_k=1} \log \frac{N}{\mathrm{df}_k}$$

I.e. we fall back to ranked information retrieval with cosine similarity and tf-idf.

Relevance feedback uses user feedback to subsequently calculate posteriors for the p_k's.

Chapter 9

Language Models

A document $D = (d_1, d_2, d_3)$ (where we assume the list of words model) has the following probability according to the chain rule (where we make the simplifying assumption that the probability of stopping doesn't depend on L_D):

$$1 - p_{stop}$$

$$P(D = (d_1, d_2, d_3)) = P(L_D >= 1) \times (D_1 = d_1 | L_D >= 1) \times P(L_D >= 2 | L_D >= 1 \wedge D_1 = d_1) \times P(D_2 = d_2 | L_D >= 2 \wedge D_1 = d_1) \times P(L_D >= 3 | L_D >= 2 \wedge D_1 = d_1 \wedge D_2 = d_2) \times P(D_3 = d_3 | L_D >= 3 \wedge D_1 = d_1 \wedge D_2 = d_2) \times P(L_D = 3 | L_D >= 3 \wedge D_1 = d_1 \wedge D_2 = d_2 \wedge D_3 = d_3)$$

$$p_{stop}$$

The markov rule with $(n+1)$-grams says we should only look at the previous n terms:

$$P(D_k = d_k) | L_D >= k \wedge D_1 = d_1 \wedge D_2 = d_2 \wedge ... \wedge D_{k-1} = d_{k-1})$$

=

$$P(D_k = d_k) | L_D >= k \wedge \underbrace{D_{k-n} = d_{k-n} \wedge ... \wedge D_{k-1} = d_{k-1}}_{\text{Only look at the previous n terms!}})$$

E.g. with unigrams, the probability of a term doesn't depend on the previous terms and we have:

$$P(D = (d_1, d_2, d_3)) = (1 - p_{stop}) p_{d_1} (1 - p_{stop}) p_{d_2} (1 - p_{stop}) p_{d_3} p_{stop}$$

Where the p_{d_k} are just the empirical frequencies of the terms.

To get rid of the order (and use the bag of words model), we have to sum over all documents d matching the bag d', which results in a multinomial distribution.

If we use a bigram model, the probability of a term depends on the previous one. We have:

$$P(D = (d_1, d_2, d_3)) = (1 - p_{stop})p_{d_1}(1 - p_{stop})p_{d_1,d_2}(1 - p_{stop})p_{d_2,d_3}p_{stop}$$

We can generate a model for every document and then use these models to generate text. We can then also ask which model (of which document) is the most likely to have generated a query q, i.e.:

$$P(D = d|Q = q) = \frac{P(Q = q|D = d) * P(D = d)}{P(Q = q)}$$

We can ignore $P(D = d)$ (assume to be uniform) and $P(Q = q)$ (constant for a given query). Therefore, we only need to estimate $P(Q = q|D = d)$ which is the probability of q under the model built from d. This probability is a weighted product of the term frequency (divided by the document length), assuming a unigram model.

If we introduce smoothing (with the collection frequency) to this model, it becomes somewhat similar to tf-idf.