Information Retrieval

With IR we want to find material, usually documents, but it can be also images or web pages or whatever, of an *unstructured nature* (no databases).

Information need: what we want to satisfy (usually expressed in natural language, not a formal query on a db). Usually we are not interested in the whole document, but only in an information (not data!) contained in a document.

Examples:

- web search: nowadays we have to scrape the web to get information (cannot use an authoritative source, moreover the links between the pages are a source of additional information);
- email search (kind of semistructured data: front, subject, text etc);
- searching documents;
- search from a knowledge base (e.g. book in a library search).

Terminology

Document: unit that we use in IR. It doesn't need to be a text document, whatever we want to find with the system!

Collection/corpus: group of documents on which we perform the search. What is our collection depends on what we are building. Some collections are static (build a system that does not need to be modified once finished), others are dynamic (need to find a way to update our system without having to build it from scratch every time).

Differences between a database record and a document:

- a document contains a significant amount of text, the db record has a predefined structure;
- documents can have some structure (title, author etc) but it is usually not well defined (i.e. the semantic is not well defined), while in db record the semantic is well define;
- in a db we perform a search comparing specific fields.

Information need: topic about which the user wants to know more (differentiated from a query).

Query: the way the user formulates his information need to the IR system (we want to satisfy what's behind the query: the information need). The same information need can be formulated in different way by different users.

The goal of the IR system is to interpret the information needs of the user and estimate the relevance of the documents wrt it.

We cannot just use *grep* (UNIX, `grep term_to_search corpus`) since this technique does not scale to large collections. We want a more flexible query language. Moreover we usually want a **ranked** retrieval (not just a set of relevant documents). In other words, we want to avoid to scan the entire collection for each query, so the corpus is scanned only once (for the static collection case) and an **index** is built.

An index will have for each term (e.g. word for text documents) all document containing that term: in this way we avoid to scan the entire collection of documents (only way to avoid linearly scanning the texts for each query).

A document is *relevant* if it is one that the user perceives as containing information of value with respect to their personal information need.

Structure of IR system

The query is interpreted in a *query representation* (e.g. point in a vector space, boolean formula etc…) by the system, which is matched with the index in a *matching mechanism* in order to give the answer to the query.

We have three main components:

- way to formally represent the query: boolean formulae, free form queries, images;
- way of formally represent documents: set of terms, point in a vector space, binary vector;
- way of matching documents (documents representation) to queries and a way to measure relevance (e.g. *exact matching*: document is either relevant or not relevant, *partial matching*: things that are sufficiently similar).

There are **two aspects of an IR system**: technical aspect (**efficiency**) and semantic aspect (**effectiveness**) which are usually in contrast, a trade off has to be made.

We can measure efficiency in a easy way, for effectiveness we use **precision** and **recall**:

$precision = \frac{relevant \cap retrieved}{retrieved} = P(relevant|retrieved)$: which fraction of the retrieved documents is relevant.

$recall = \frac{relevant \cap retrieved}{relevant} = P(retrieved|relevant)$: which fraction of the relevant documents has been retrieved.

# Boolean Retrieval

Only exact matching (not find documents that contain all words but one for example, either a document is relevant or not), the query is expressed like a Boolean formula: we can ask for the inclusion (or exclusion) of certain terms. Even if it can be extended, it is not flexible (very old IR system).

A document for the boolean model is the set of terms that it contains (the order and multiplicity of the terms is ignored in this representation, most representation are a trade off between being efficient and be faithful wrt the original document). **A term is identified by the collection of all the documents that contain a term** (this is what will be our index). Query is a formula in which we use terms instead of variables.

We can build and incidence matrix of terms and documents: terms are row (the row: indeed it is the collection of all documents containing it) and documents are columns (collection of terms).

The size requirements ($\#terms \times \#documents$) makes this data structure impractical (think about the english dictionary, or wikipedia). Moreover often this matrix is extremely sparse, a much better representation is to record only the things that do occur, that is, the 1 positions.

A more compact data structure is the **inverted index** (inverted since we go from terms to documents, but being an index it maps back from terms to the parts of a document where they occur): we want to avoid storing rows that will be mostly empty. So for each term, we store the ordered (any order is fine, to improve performances for union and intersection) list of documents that contain the term (similar to the difference between adjacency matrix and adjacency list for graphs), instead of the whole binary vector.

Usually we store the **DocID** (Document IDentifier) in the list, i.e. a unique serial number associated to each document.

The **dictionary** is the collection of all terms that we have in the inverted index, the **posting list** is the list of DocID associated to a term, the **posting** is the element of the list (not DocID, since it is associated to a term).

In order to build an inverted index (after collecting the documents to be indexed) for each document we extract the sequence of terms, we tag each term with the corresponding DocID and we **sort the list of terms** extracted from all the documents. We group together equal terms to create the posting list.

Recall that a term in this model is a set of documents, so we don't have duplicates!

However, in the worst case we are not gaining anything: we have the list containing all the documents $O(\#terms \times \#documents)$: this never happens in practice (it means that each term should be contained in each document), usually most documents contain a small subset of the terms.

The same reasoning applies to the time complexity of the operations performed on the set of documents.

Inverted index: union and intersection

How to implement an inverted index

If our query is composed by a single term, it is trivial to answer: all the document containing that term are in the posting list. We search the term in the dictionary and then we check for its posting list, i.e. we find the term in the list of terms and we return the associated list of term.

If we have the *conjunction* or the *disjunction* of two terms it is more complex.

Answering conjunction queries: we have to perform the intersection of the posting list. If the posting list are ordered we can perform the intersection linearly (complexity is linear in the lengths of the lists: we must advance once for every element of the two lists). Complexity of search intersection (+ index search): $O(n+m) + O(log(\#terms))$.

The size of the answer is at most as big as the smallest list, i.e. it is $\leq$ than the minimum of the lengths of the lists.

Answering disjunction queries (without duplicates): we still have to compare the two lists, we continuously are adding the missing elements (one copy of each DocID). Also in this case the complexity is linearly in the length of the lists: we advance once for each posting.

The size of the answer is at most, i.e. $\leq$ the sum of the lengths of the lists.

We want also to deal with the cases: $term1NEARterm2$ (near usually means not more than a few terms distant), "$term1term2$", $term1 \cdot wildcards$, etc.

Some terms are not useful: e.g. if a term is in all documents. Also our queries at the moment need to be extremely precise: for each possible word we have a different term, even if some terms are very similar semantically.

Recall that the query is only a way for the user to express its information need, we may want to put together all the posting lists of terms that are similar in meaning.

Improving the quality of retrieval

Tokenization

We have to go deeper in the process of extracting terms from the documents, i.e. **tokenization** (and perhaps at the same time throwing away certain characters ì, such as punctuation).

A **token** is an instance of a sequence of characters (in some particular document that are grouped together as a useful semantic unit for processing).

A **type** is a collection of al tokens with the same character sequence.

A **term** is a type (perhaps normalized) that is inserted into the IR dictionary.

In other words, tokenization is the process of (given a character sequence and a defined document unit) chopping character streams into tokens (perhaps at the same time throwing away certain characters, such as punctuation). Linguistic preprocessing deals with building equivalence classes of tokens, which are the set of terms that are indexed.

The first step of tokenization is deciding how to split our collection, i.e. decide what is the granularity of the indexing. Then we split a text sequence into tokens. For very long documents, the issue of *indexing granularity* arises.

The thing is that it is not that simple deciding where to split the text sequence (called *problematic tokenization*), in some languages (e.g. Japanese) it is not simple to see where a word ends and when the other starts. the issue of tokenization are language specific. It thus requires the language of the document to be known.

We can either use the same tokenizer for the documents and queries (no discrepancy btw tokens of the index and tokens of the queries) or use a collection of heuristics to decide where to split.

Even if we are able to solve the problem of tokenizing with a certain criteria, there is still the problem that some words are present in all the documents: we have to *drop common terms*.

Stop words removal

Sometimes, some extremely common words that would appear to be of little value in helping select documents matching a user need are excluded from the vocabulary entirely.

We want to have a collection of **stop words**: a collection of common words that are discarded from the indexing (posting lists that occupy space but are not useful) and querying process, since they are so common that do not help in selecting a document matching a user need.

The collection of stop words, the **stop list**, is specific for a language and for a corpus (if we have a corpus about a topic, maybe the name of the topic is present in all documents, i.e. stop lists can be specific by topic). Usually consists of the most frequent words, curated for their semantic. Using a stop list significantly reduces the number of postings that a system has to store.

We can check for the frequency of words in a corpus (*collection frequency*, i.e. the total number of times each term appears in the document collection), usually some of the words are extremely frequent: that words are good candidates for being stop words. Using a stop list significantly reduces the number of postings that a system has to store, and a lot of the time not indexing stop word does little harm.

The problem is that sometimes stop words are useful, we can have queries that are not answered if we remove stop words (e.g. "to be or not to be" is composed only of stop words, in general in phrase queries this can happen). So stop word removal allows us to be fast and to occupy less space, but we can have more than a few problems in answering some of the queries. Removing stop words can reduce the *recall*.

In general the trend is to have small lists of stop words (7-12 terms), or to have no stop list at all (e.g. web search engines generally do not use stop lists). Use compression techniques to reduce the storage requirements, use weighting to limit the impact of stop words, or we can use specific algorithms to limit the runtime impact of stop words.

Normalization

Having broken up our documents (and queries) into tokens, the easy case is if tokens in the query just match tokens in the token list of the document. However, there are many cases when two character sequences are not quite the same but you would like a match to occur.

Token normalization is the process of canonicalising tokens so that matches occur despite superficial differences in the character sequences of the tokens.

The problem is that the same word can be written in different ways, we may want to define some equivalence classes of terms (so that searching for any word in the equivalence class brings to the same result, this is the most standard way to normalize), e.g. ignoring capitalisation (case-folding), removing accents and diacritics and other normalization steps specific to the language, e.g. ignoring spelling difference (the person that wrote the document and the person that searches in the document use different spelling to mean the same thing).

As before, sometimes this is not the thing that we want (i.e. sometimes capitalization and other features are important).

We may decide to save a table of relationships between the unnormalised terms (query term | equivalent terms).The usual way is to index unnormalised tokens and to maintain a query expansion list of multiple vocabulary entries to consider for a certain query term. A query term is then effectively a disjunction of several posting lists (this requires more processing at query time). The alternative is to perform expansion during index construction (this requires more space for storing postings).

These approaches are more flexible than equivalence classes because the expansion list can overlap while not being identical, i.e. there can be an asymmetry in expansion.

Stemming and Lemmatization

The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related form of a word to a common base form. Howver, the two words differ in their flavour.

Stemming is a language specific technique to reduce all variants of a word to a "common root", a common base form. It is based on heuristics (completely syntactical, it essentially chops off the ends of the words). **Lemmatization** has the same goal, but uses vocabulary and morphological analysis of words (the *lemma* is the dictionary form of a word).

Example: the *Porter stemmer* is the most used stemmer for the English language (it has been shown to be empirically very effective). It consists of five stages applied sequentially, where each stage consists of a series of rewriting rules for words.

Rather than using a stemmer, we can use a *lemmatizer*, a tool from NLP that does full morphological analysis to accurately identify the lemma for each word.

Preprocessing pipeline

These steps are performed before indexing and before the queries, it is composed by (the starting point are the documents):

- normalization;
- stop words removal (eventually);
- stemming (or lemmatization).

After that we perform indexing, so the actual index will not contain words that are in the documents, but a normalized version of them. Recall that additional steps might be present and not all steps are mandatory.

Note that the same steps applied to the queries! This guarantees that a sequence of characters in a text will always match the same sequence typed in a query.

So now we have a system that given a corpus performs a set of preprocessing steps and returns an inverted index. Now we look for a way to answer *phrase queries*, i.e. queries consisting of multiple **consecutive words** (our goal is to extend the query language).

Biword indexes

One approach to handling phrases is to consider every pair of consecutive terms in a document as a phrase.

The **biword index** works on pair of words. Our terms are now pairs of words (each of the biwords is a vocabulary term). Also queries need to be rewritten, e.g. "inside the box" becomes "inside the" AND "the box". The problem here is that we can retrieve also documents that do not contain "inside the box" (false positive), but match both "inside the" and "the box". Hence rewriting the query may generate false positives, but in general works well.

Another problem of biword indexes is that we loose the single words, but some queries may ask for a single word. Hence we also need an index for single terms (another index to manage).

We can improve on some of those things: we can extend the idea of using pair of words as terms to sequences of any length (reduce the risk of false positives, at the price of increasing the amount of space needed). If the number of words in a term is variable (i.e. the index includes variable length word sequences), it is called *phrase index*.

We can even go as far as deciding where to split and what to index based on the part of speech (sort of tag).

Positional indexes

The idea is not to augment the original inverted index, for each posting we say the set of positions in the document where the term is (postings are of the form DocID:(position1, position2,…)). In this way we can check if two terms are consecutive (i.e. in adjacent positions). This is called **positional index**.

The positional index can be used to support the operators of the form $term_1/k$ $term_2$ with k integer indicating the maximum number of words that can be between $term_1$ and $term_2$ (biword indexes cannot).

To process a phrase query, in the merge operation we need to check both that terms are inside the document and that their positions of appearance in the document are compatible with the phrase query being evaluated.

The complexity of performing a query is not bounded anymore by the number of documents, but now we have to scan all documents and all terms in a document (additional list to scan) and so it is bounded by the number of terms.

The size of the index now depends on the average document size (we are saving all the positions). This means that it can be slower to answer a query when we have a positional index. Moving to a positional index also changes the asymptotic complexity of a postings intersection operation, because the number of items to check is now bounded not by number of documents but by the total number of tokens in the document collection.

We can combine biword and positional index: biword index for frequently asked queries and positional index for all other queries. In this way we can have the advantages of both, with few disadvantages (no advantage in terms of space).

Improving the inverted index

As data structure to represent the posting list we can use a *singly linked list*, which has cheap insertion and updates, but we have some pointer overhead, poor memory locality (pointer chasing, i.e. a common sequence of instructions that involves a repeated series of irregular memory access patterns that require the accessed data to determine the subsequent pointer address to be accessed, forming a serially-dependent chain of loads).

We could also use *variable length arrays* that do not have pointers overhead and are allocated contiguous in memory, drawbacks are that if we need to update the array we may need to re-allocate it.

Another possibility are **skip lists**. They have an additional pointer every k postings inside a list (augmenting posting lists with skip pointers at indexing time). So the forward pointer skips a certain number of postings (so to avoid processing parts of the posting list that will not figure in the search results). A rule of thumb is, for posting list of P posting, we use $\sqrt{P}$ evenly spaced pointers. At least the intersection is now faster. In some situations we only need $O(\sqrt{P})$ steps to traverse a list. There is a tradeoff in where to place skips: more skips means shorter skip spans but it also means a lot of comparisons to skip pointers, and lots of space storing skip pointers. Fewer skips means few pointer comparison, but then long skip spans, which means that there will be fewer opportunities to skip.