Web Information Retrieval Notes

Giuliano Abruzzo

May 29, 2020

Contents

1 Cap 1: Inverted Index & Boolean Retrieval

Information retrieval means finding *material* (usually *docs*) of an **unstructured** (a *data* which doesn't have an easy-for-a-computer *structure*) nature (usually *text*) that satisfies an information need from within large collections. The field of *information retrieval* also covers supporting *users* in *browsing* or *filtering* document collections. Given a set of documents, **clustering** is the task of coming up with a *good grouping* of the *documents* based on their contents. Given a set of topics, **classification** is the task of deciding which *class*, if any, each of a set of *documents* belongs to. *Information retrieval* systems can also be distinguished by the **scale** at which they operate, in *web search*, the system has to provide search over billions of *documents* stored on millions of computers instead at the other extreme there is *personal information retrieval*.

Information retrieval in the Web is different because:

- *Infos* can be **linked**;
- We have huge *sources* with different kinds of files;
- *Sources* can be not *homogeneous*;

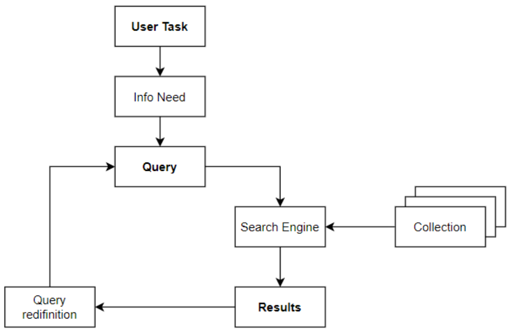
Links can be used for suggestion, from a *page* to another one, every *web page* has a **rank** that depends on how may *links* point to it, and this *rank* is used by Google for **relevance criteria**. We can have also a **quality** difference between *infos* on the same *source*. *Web* in past was different, so now we can distinguish:

- **Web 1.0**:
  - In 90s, *infos* on the *web* were almost always generated from a *source* instead *users* were quite passive, in fact web was used for reading, owning, consuming, so very few interactions of the *user*. The *web* was structured in a *hierarchical* way, called **Taxonomy**.
- **Web 2.0**:
  - Now, a lot of *web content* is **user-generated** and there are a lot of interactions between *users* and *information* that we can retrieve. The structure is not anymore *hierarchical*, we have *tags* and every *tag* is a *category*, this structure is called **Folksonomy**.

If we consider the *web* as a **graph**, where the *nodes* are the *pages* and *edges* are the *hyperlinks* between them:

- **Crawling the web** means that by visiting this *graph* we will collect a **web corpus** (*multiple documents*), starting from a *page* and through this *links* we iterate on another *pages*. Often we don't want to *crawl* some *pages*, cause we don't need them, or we want to add some kind of *restrictions*, and we need to be sure to visit all *relevant pages* in order to avoid *information missing*, so this problem requires an huge amount of *computing power* and *competitiveness*;
- We also need **Indexing the web**, so we need to give a *structure* to the *web corpus* that we crawled;
- Last, we need some **Search Algorithms**, in order to solve *user's information need* using *queries*, they need to be based on **relevance** (results are coherent with the query) and **authority** (pages must give user real information);

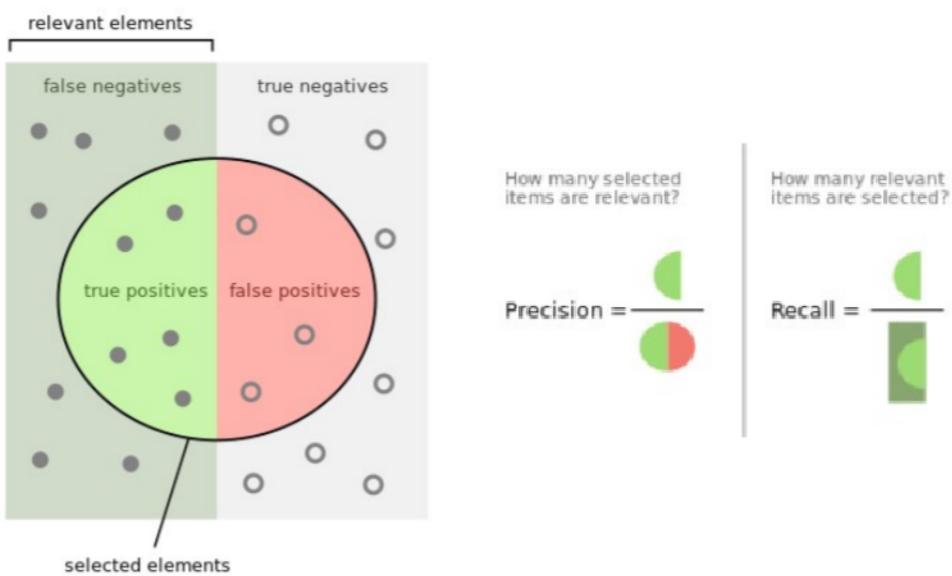
The **Meta-Search** is when we research using combined results from multiple *search engines*. A **collection** is a set of *documents*, not always *static*, and the **goal** is to collect *documents* that are relevant to *user's information* need. The **classic Search Model** is:



For example, the **user task** is: '*how to get rid of a mice in a politically correct way*', so the **info need** is '*remove mouse without killing it*', the **query** will be '*how trap mouse alive*'

Between **user task** and **query** there could be some *misconception* or *misformulation* by the *search engine*. In order to measure a **search engine performance**, we have to compare the *ground truth* (what we expect from a search), with the *result* we actually obtain, and this is done through **Precision** and **Recall**:

- **Precision**: $\frac{relevant\ docs}{returned\ docs}$
- **Recall**: how many documents over the relevant ones are shown;



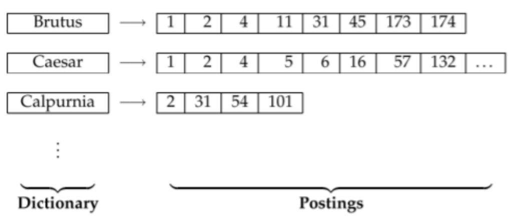
Usually we have a **trade-off** between them, so we need to obtain either a good *precision* or *recall* depending on the problem's type.

The first type of *query* we are going to see is the **Boolean IR query**, a *query* in which we research *keywords* in *documents*. For example we want to know which plays of Shakespeare contains '*brutus*' **and** '*caesar*' **and not** '*calpurnia*'. A way could be **grepping** (search a *string* in a *document*) all the *plays*, but this is **strongly inefficient** for large *docs* and it's not flexible for other *matching operations*. Another option is using a **binary term-document incidence matrix**, where the *words* are the *rows*, and the *plays* are the *column*, and we will have a 1 in the cell if that *word* appear in that *play*:

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
...							

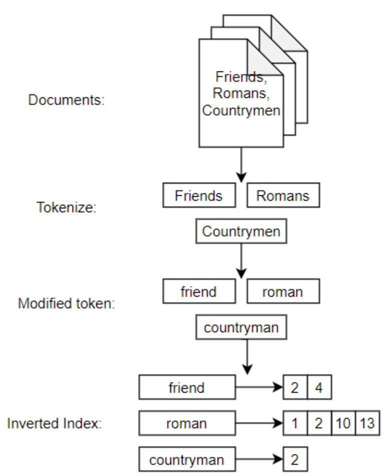
In order to *answer* the **query** we will take the vectors of *brutus*, *caesar* and *calpurnia* and we do a *bitwise and*: 101000 AND 110111 AND (NOT 010000) = 100100, so the answer will be *Antony and Cleopatra* and *Hamlet*.

But if we have **big collection**, we will have a *matrix* too big to fit in a *computer memory*, and also it will be full of *zeroes*, with few *non-zero entries*. A much better representation is to record only the *1's occurrences*, by using an **Inverted Index**:



This means that for each *term* we will have a *linked list* that records which *documents* the *term* occurs in. Every *document* must have a **docID** that identify it and these are *sorted*, so we avoid to use *fixed length array*.

In order to build an **inverted index** we need to:



- Collect the *documents* to be *indexed*;
- *Tokenize* the *text*, turning each *document* into a list of *tokens*;
- Do *linguistic preprocessing*, producing a list of *normalized tokens*, which are the *indexing terms*;
- Index the *documents* that each term occurs in by creating an *inverted index*, consisting of a *dictionary* and *postings*.

In *linguistic modules* we **stem** the text and also remove *stopwords*, not only *lowercase*. The core **indexing step** is *ordering* both *words* and *docID*, and often also the *frequency* of every *word* is added. So the *cost* for a *query* $word1\ AND\ word2$is $O(X+Y)$ where X and Y are the *postings length* of the two *words*. The *intersect* of the two *posting lists* is the crucial operation, and it's also called **merging**. The **query optimization** is the process of selecting how to organize the work of answering a *query*:

- **(... AND ...) AND (... AND ...):** we will start from *intersect* the two smallest *posting lists*, so the next *intersection* will be more efficient;
- **(... OR ...) AND (... OR ...):** we use the *frequency* to estimate which *OR* we will use, usually the one which *term's frequency sum* is lower;

Sometimes we want a *query* to match a *phrase*, for example we want to answer queries with '*stanford university*', we call these **Phrase Queries**. These *words* have to appear in these *precise order*, so a *posting list* is not good anymore, we could index every consecutive pair of *terms* in the *text* as a *phrase*, also called **Biword Indexes**, for example '*Friends, Romans, Countryman*' generate two biwords '*friends romans*' and '*romans countryman*', where each of these is a *dictionary term*, but we will have too many *dictionaries* and *false positive*. So we need to use a **Positional Index**, in which we consider the **position** of the *term* in the *docs*:

- -**Stanford**: 2: 1,12,24; 4: 8,16,190; ...
- -**University**: 1: 17,19; 4: 17,191; ...

Where at the left we have *docID*, at the right the *positions* of the *word*. Using position we can even do **proximity search** and we can also combine *Biword* and *Positional*.

2 Cap 2: Document Ingestion

2.1 Tokenization

We said that in order to construct an *inverted index*, we need to **tokenize** the *text*. **Tokenization** is the process of chopping *character streams* into *tokens*, a **token** is a sequence of chars grouped together as a *semantic unit* for *processing*. *Tokenization* depends on how we want to process the *index* (by *words*, by *biwords*, ...), the most simple strategy is to split on *whitespaces*, but there could be the need of grouping special sets of *characters*. The problem of *tokenization* is finding the correct *token* to use, for '*Finland's*' which is the best *tokenization*? '*Finland AND s*', '*Finlands*' or '*Finland's*', these issues are **language-specific**, so we require to know the *language* of the document, we can have also problems with *dates* and *numbers*.

2.2 Stopwords

Stopwords are *words* that are extremely common and their *semantic content* is almost useless, they are generated by *collection frequency* (number of times each *term* appears in *documents*). 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 words* does little harm: keyword searches with terms like *the* and *by* don't seem very useful. However, this is not true every time, for example, the meaning of *flights to London* is likely to be lost if the word *to* is stopped out. For most modern *IR systems*, the additional cost of including *stop words* is not that big, neither in terms of *index size* nor in terms of *query processing time*.

2.3 Normalization

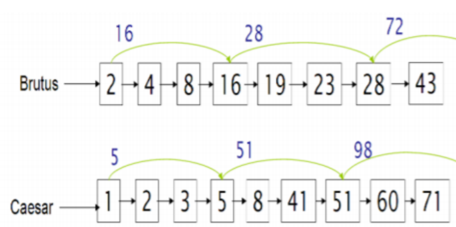
Normalization is the process of canonize *tokens* so that matches occur despite superficial differences in the character sequences of the *tokens*. The most standard way to *normalize* is to implicitly create **equivalence classes**, for example we delete dots ($U.S.A \rightarrow USA$) or hyphens (*anti-discriminatory* → *antidiscriminatory*). It can also be applied on accents or variants and there could misunderstanding too, but even on *synonyms*. During *normalization* all *token* characters are reduced to *lower case*.

2.4 Stemming and Lemmatization

For grammatical reasons, *documents* are going to use different forms of a *word*, such as *organize*, *organizes*, and *organizing*. The goal of both **stemming** and **lemmatization** is to reduce *inflectional forms*, for instance: *am*, *are*, *is* → *be*, or *car*, *cars*, *car's*, *cars'* → *car*. **Stemming** usually refers to a heuristic process that *chops off* the ends of *words* in the hope of achieving this goal correctly most of the time, includes the removal of *derivational affixes*. **Lemmatization** usually refers to doing things properly with the use of a *vocabulary* and *morphological analysis* of *words*, normally aiming to remove inflectional endings only and to return the base or *dictionary* form of a *word*, which is known as the **lemma**. The most common algorithm for *stemming* English is **Porter's Algorithm**, which consists of 5 phases of *word reductions*, applied sequentially, some typical rules are: substitution like $SSES \rightarrow SS$ or $IES \rightarrow I$, weight of word sensitive rules, $(m > 1)$ Ement: $replacement \rightarrow replac$ or $cement \rightarrow cement$.

2.5 Faster Postings List

In the basic **postings list intersection**, if the list lengths are m and n, the *intersection* takes $O(m + n)$ operations. One way to do better than this is to use a **skip list** by augmenting *postings lists* with **skip pointers** (at indexing time), which allow us to avoid *processing parts* of the *postings list* that will not figure in the *search results*. We have a *trade-off*, more *skips* means shorter *skip spans* (space of a skip), but also means lot of *comparisons* and lots of space storing *skip pointers*. Building effective *skip pointers* is easy if an *index* is relatively static, it is harder if a *postings list* keeps changing because of updates.

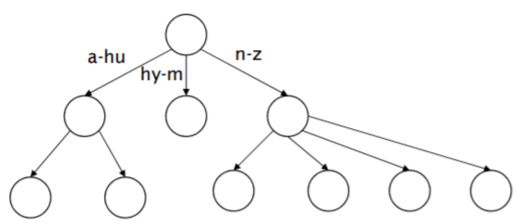


3 Cap 3: Dictionaries and Tolerant Retrieval

3.1 Dictionaries

Data Structure for *posting lists* are usually *dictionaries*, but we need to store in memory efficiently, and also we need to quickly look up elements at *query time*. There are two main choices: **Hash Tables** and **Trees**:

- **Hash Tables**:
  - Each *vocabulary term* (the **key**) is **hashed** into an *integer* over a large enough *space* (in order to avoid *hash collisions*), at *query time* we *hash* each *query* term separately and following a *pointer* to the corresponding *postings*. So the *lookup* is really fast $O(1)$, the problem is that it's not easy to find *similarity* (minor variants of a *query term*), and there is no *prefix search*. If the size of the *vocabulary* keeps growing we may need to *rehash* everything to avoid *collisions*.
- **Trees**:
  - The best-known *search tree* is the **binary tree**: the search for a *term* begins at the *root* of the *tree*, each *internal node* (including the *root*) represents a *binary test*. A **lookup** costs $O(log\ N)$ if the *tree* is **balanced** (number of *terms* under the two *sub-trees* of any *node* is equal or differ by one), else is $O(N)$. It solves the *prefix search* problem, but we can have problems of *re-balancing*, in fact when we insert new *terms* or we delete old ones the *tree* needs to be re-balanced. One approach to avoid it is to allow the number of *sub-trees* under an *internal node* to vary in a fixed interval, these are called **B-Trees**, in which every *internal node* has a number of children in the interval $[a, b]$. Unlike *hashing*, **search trees** demand that the characters used in the document collection have a **prescribed ordering**.



3.2 Wild-Card Queries

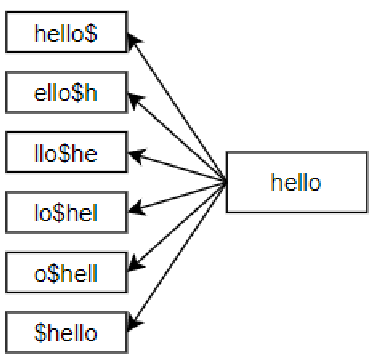
Wildcard Queries are used in any of the following situations:

- The user is uncertain of the spelling of a *query term*, like *Sydeny* vs *Sidney*, so the wildcard query is *S*dney*;
- The user is aware of multiple variants of spelling a term, and seeks documents containing any of the variants, like *color* vs *colour*;

- The user seeks documents containing variants of a term that would be caught by *stemming*, but is unsure whether the search engine performs stemming like *judicial* vs. *judiciary*, the wildcard query *judicia**;
- The user is uncertain of the correct rendition of a foreign word or phrase, like *Universit* Stuttgart*;

A *query* such as *mon** is **trailing wildcard query**, because the ** symbol* occurs only once, at the end of the search *string*. This query is easy to find in a B-tree, but the query **mon* is very harder, called **leading wildcard queries**, we can use a reverse *B-Tree* in which each *root-to-leaf* path corresponds to a term written backwards, for example *lemon* will be represented by the path *root-n-o-m-e-l*. Using a regular *B-Tree* together with a *reverse B-Tree* we can handle a general case *se*mon* where the regular B-tree is used to find the prefix *se*, and the reverse B-tree is used to find the suffix *mon*, and then we will **intersect** the two *sets* obtained, but this solution is pretty expensive.

A more efficient way is using **Permutation Index**, a form of *inverted index*, we will use a special symbol $ used to mark the end of a term.



If we have to search *hel*o*, the key is to rotate in such a way the ** symbol* is at the end of the string, we set $X = hel$ and $Y = o$, we search for $Y\$X*$ so $o\$hel*$. *we search in the regular b-tree, all the words that start with hel and ends with o (? probably?).*

If we have a query like *fi*mo*er* first we ignore the *mo*, and we search for *fi*er* that permuted is *er$fi**, then we will filter the words that doesn't contain *mo*. The problem with permutation index is that its dictionary becomes quite large.

In order to solve this problem, we can use the **Bigram Index** (or **K-gram Index** where *k-gram* is a sequence of *k* characters) for a single word, for example in '*April is the cruelest month*', we get the bigrams: 'a,ap,pr,ri,il,l,i,is,s,t,th,he,e,c,cr,ru,ue,el,le,es,st,t, m,mo,on,nt,h'. We maintain a *second inverted index* from *bigrams* to *dictionary terms* that match each *bigram*. For example $m will point to *mace*, *madden*, *mo* will point to *among*, *amortize*, *on* to *along, among*. The query *mon** can now be run as $m AND *mo* AND *on*, but these can return also the word *moon*, so we need another filter at the end. This method is *fast* and *space efficient* (compared to *permutation*). The processing of a *wildcard query* can be quite expensive because of the added *lookup* in the *special index*, *filtering* and finally the *standard inverted index*. A **search engine** may support such rich functionality, but most commonly, the capability is hidden behind an interface that most users never use.

3.2.1 Spelling Correction

There are two methods to solve the problem of **correcting spelling** errors in *queries*: the first based on **edit distance** and the second based on **k-gram overlap**. There are two basic principles underlying most spelling correction algorithms:

- Correct spellings for a *misspelled query*, so we choose the **nearest one**;
- When two correctly spelled *queries* are tied, select the one that is **more common**;

We will focus on two types of *spelling correction*: **isolated-term correction** and **context-sensitive correction**:

- **Isolated-Term correction**:
  - In **isolated-term correction**, we attempt to correct a single *query term* at a time, but such *isolated-term correction* would fail to detect, for instance, that the query *flew form Heathrow* contains a misspelling of the term *from*, because each term in the *query* is correctly spelled in isolation.
  - **Edit Distance**:
    * Given two strings s_1 and s_2, the **edit distance** between them is the minimum number of *edit operation* to convert one string to the other. *Edit distance* is also called **Levenshtein distance**, for example the edit distance of *cat* and *dog* is 3. There is also the **weighted edit distance** in which we have a *weight* based on the *character* involved (char m mis-typed as n more than z, so replacing m with n is a smaller edit then z), but these requires a *weight matrix* as input. In order to compute the *weighted edit distance* we need $O(|s_1| \times |s_2|)$ where $|s_1|$ denotes the length of the string. In order to correct the *spelling*, given a *query*, first we enumerate all *char sequences* within a preset *weighted edit distance*, then we intersect this set with list of *correct words*, then we show these *words* to the user as a suggestion.
  - **N-Gram Overlap**:
    * We can use the **n-gram index** to retrieve *vocabulary terms* that have many *n-grams* in common with the *query*, and the *retrieval process* is essentially that of a single scan through the *postings* for the *n-grams* in the *query string*. For example if the text is *november* (trigrams are *nov, ove, vem, emb, mbe, ber*) and the query is *december* (trigrams are *dec, ece, cem, emb, mbe, ber*) so we have 3 *trigrams* that overlap. The measure of *overlapping* is given by the **Jaccard Coefficent** by two sets X and Y and it is: $|X \cap Y| \setminus |X \cup Y|$, this will be a value between 0 and 1, so we will choose the *terms* over a certain *threshold*.
- **Context-Sensitive Correction**:
  - *Isolated-term correction* would fail to correct typographical errors where all *query terms* are correctly spelled, for example '*I flew form Milan*', '*form*' is an error (*from*). The simplest way to correct these errors is to enumerate *corrections* of each of the *query terms* even though each *query term* is correctly spelled, then try substitutions of each *correction* in the phrase. For each such *substitute phrase*, the *search engine* runs the *query* and determines the number of *matching results*. This enumeration can be expensive, several

heuristics are used to reduce the *search space*. We wanna rank alternatives probabilistically: $argmax_{corr}\ P(corr|query)$, that with bayes rules is: $argmax_{corr}\ P(query|corr) * P(corr)$, where *query* is the noisy channel, and *corr* the language model.

3.2.2 Phonetic Correction

Misspellings that arise because the user types a *query* that sounds like the *term target*. The main idea here is to generate, for each term, a **phonetic hash** so that *similar-sounding terms hash* to the same value. Algorithms for such *phonetic hashing* are commonly collectively known as **Soundex algorithms**, common steps are:

- We turn every *term* to be indexed into a *4-character reduced form*, we build an **inverted index** from these reduced forms to the original *terms* called *soundex index.*;
- We do the same with the *query terms*;
- When the *query* calls for a *soundex match*, search this *soundex index*;

4 Cap 4: Index Construction Algorithm

In order to answer a *query*, we need to build an *inverted index* for a set of terms, this construction process is called **Index Construction** or **Indexing**, that takes advantage of *secondary storage*. It's important to note that: *main memory >>> secondary storage*, it's way faster. There are some consideration:

- To optimize **transfer time** we have a big *chunk* of data and not several small *chunks*;
- **Disk I/O** is *block-based* (of fixed lights);
- **Fault tolerance** is handled with *replication* (several instead of a single fault-tolerant machine);
- The **main memory** is the better;

We need larger average *word token size* to handle *words*, especially if we strip out *stepwords*. The goal is construct the *inverted index*, but we can't do the whole sorting in *main memory*, we need intermediate steps.

4.1 BSBI Algorithm

BSBI Algorithm or **Blocked Sort-Based Indexing Algorithm**, is an **external sorting algorithm** which try to minimize the number of *random disk seeks* during *sorting*, for example to sort *100M postings* (made of pairs *term-docID*) we define *10 blocks* of *10M postings* each, in such a way that each *block* can fit in the *memory* and:

- For each *block*:
  - Accumulate *postings*;
  - Sort in *memory*;
  - Write to *disk*;
- Then merge all the *blocks sorted* in order to obtain the *final index*;

Algorithm 2 Blocked sort-based indexing. The algorithm stores inverted blocks in files $f_1, \dots, f_n$ and the merged index in f_{merged}.

function BSBI_CONSTRUCTION
    $n \leftarrow 0$
    while all documents have not been processed **do**
        $n \leftarrow n + 1$
        $block \leftarrow PARSE\_NEXT\_BLOCK()$
        $BSBI\_INVERT(block)$
        $WRITE\_BLOCK_TO\_DISK(block, f_n)$
    end while
    $MERGE\_BLOCKS(f_1, \dots, f_n, f_{merged})$
end function

The *algorithm* parses *documents* into *termID–docID* (*termID is a unique mapped in a dictionary from the term*) pairs and accumulates the pairs in *memory* until a *block* of a fixed size is full, the *block* is then inverted and written to *disk*. **Inversion** involves two steps, first we sort the *termID–docID pairs*, next, we collect all *termID–docID* pairs with the same *termID* into a *postings list*, where a *posting* is simply a *docID*, and the result is then written to *disk*. In the final step, the algorithm simultaneously merges the *blocks* into one large *merged index*. It's time complexity is $O(T\ log\ T)$, cause the step with the *highest time complexity* is *sorting*, and T is an *upper-bound* for the number of *items* we must sort. The actual *indexing time* is usually dominated by the time it takes to parse the document (*parse block function*), and the final merge (*merge blocks function*). The key decision is the **block size** that need to be *optimized*.

4.2 SPIMI Algorithm

Blocked sort-based indexing has excellent *scaling properties*, but it needs a *data structure* for mapping *terms* to *termIDs*. For very large collections, this *data structure* does not fit into *memory*. An alternative is **Single-Pass In-Memory Indexing or SPIMI**, which uses separate *dictionaries* for each *block* (so we don't need to maintain *term-termID* mapping across *blocks* like in *BSBI*), and the other idea is **don't sort**, so we accumulate *postings* in *postings lists* as they occur, so we can generate a complete *inverted index* for each *block* that we will merge into one *big index*. Each *postings list* is dynamic so it's immediately available to collect *postings*, and this has two advantages, it's *faster* (no sorting) and saves *memory* cause we keep track of the term a *posting lists* belongs to, so the *termID* of postings doesn't need to be stored. *SPIMI* has also an important component, the **compression**, in fact both *postings* and the *dictionary terms* can be store compactly on *disk*. The *time complexity* is $O(T)$ since no sorting is required and all operations are linear in the size of the collection. *SPIMI_INVERT* is called repeatedly on the *token stream* until the entire *collection* has been processed. *Tokens* are processed one by one during each successive call of *SPIMI_INVERT*. When a *term* occurs for the first time, it is added to the *dictionary*, and a new *postings list* is created, at the end it returns this *postings list* for subsequent occurrences of the term. (*NOT SURE we don't sort postings cause they are already sorted by the docsID that is incremental when received*)

Algorithm 3 Inversion of a block in single-pass in-memory indexing.

```
function SPIMI_INVERT(token_stream)
    output_file = NEW_FILE()
    dictionary = NEW_HASH()
    while free memory available do
        token ← next(token_stream)
        if term(token) ∉ dictionary then
            postings_list = ADD_TO_DICTIONARY(dictionary, term(token))
        else
            postings_list = GET_POSTINGS_LIST(dictionary, term(token))
            p2 ← next(p1)
        end if
        if full(postings_list) then
            postings_list = DOUBLE_POSTINGS_LIST(dictionary, term(token))
        end if
        ADD_TOPOSTINGS_LIST(postings_list, docID(token))
    end while
    sorted_terms ← SORT_TERMS(dictionary)
    WRITE_BLOCK_TO_DISK(sorted_terms, dictionary, output_file)
    return output_file
end function
```

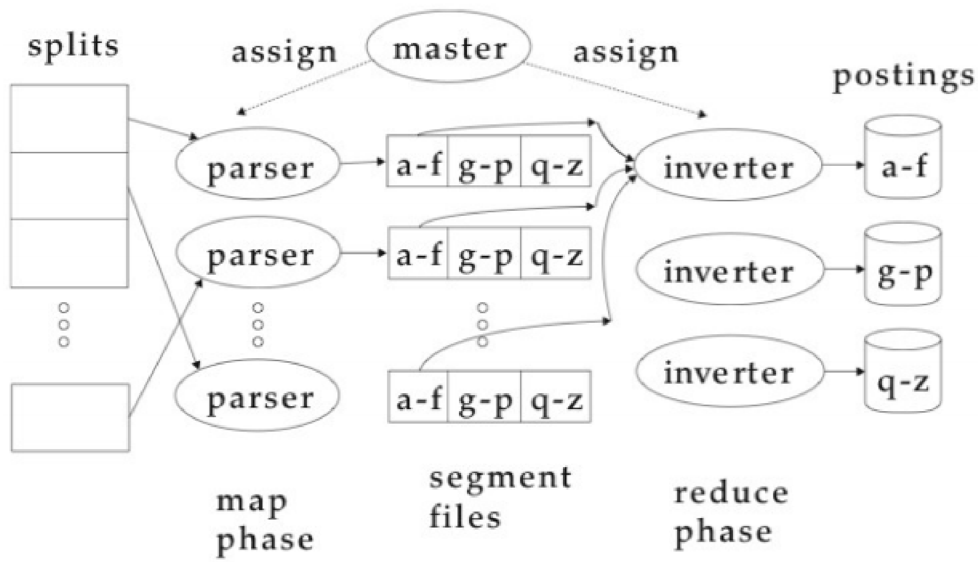
4.3 Distributed Indexing

Collections are often so large that we cannot perform **index construction** efficiently on a *single machine*, in this case we will use **distributed indexing algorithms** for *index construction.* The result of the construction process is a *distributed index* that is partitioned across several machines. We have a **Master Machine** which is a *fault-tolerant machine*, and **Idle Machines**, that are part of a *pool* and they have *indexing tasks* in parallel assigned by the *master.*

The distributed index construction method is an application of **MapReduce** a general architecture for *distributed computing.* A robust *distributed indexing* needs the work to be divided in re-assignable chunks, a **master node** directs the process of assigning and reassigning *tasks* to individual **worker nodes** (*idle machine*).

First, the input data, are split into n *splits* where the size of the *split* is chosen to ensure that the work can be distributed evenly and efficiently, these *splits* are not preassigned to *machines*, but are instead assigned by the *master node*: as a *machine* finishes processing one *split*, it is assigned the next one, if a machine dies, the *split* it is working on is simply reassigned to another machine. In general, *MapReduce* breaks a large computing problem into smaller parts by *recasting* it in terms of manipulation of *key-value pairs*, in the form of $\langle termID, docID \rangle$.

The **map phase** consists of *mapping splits* of the input data to *key-value pairs*, the machines that execute the map phase are called **parser**. Each *parser* writes its output to local intermediate files, the *segment files*. For the **reduce phase**, we want all values for a *given key* to be stored close together, this is achieved by *partitioning* the *keys* into *j term partitions* (*range of letters like a-f*) and having the *parsers* write *key-value pairs* for each term *partition* into a separate *segment file*. Collecting all values for a *given key* into one list is the task of the **inverters**, the *master* assigns each *term partition* to a different *inverter*. Finally, the list of *values* is sorted for each *key* and written to the final *sorted postings list*. *Parsers* and *inverters* are not separate sets of machines, the *master* identifies *idle machines* and assigns *tasks* to them, the same machine can be a *parser* in the *map phase* and an *inverter* in the *reduce phase*.



4.4 Dynamic Indexing

Most collections are *modified frequently* with *documents* being *added, deleted*, and *updated*, new *terms* need to be added to the *dictionary*, and *postings lists* need to be updated. The simplest way to achieve this is to *periodically reconstruct* the **index** from scratch, this is a good solution if the number of changes over time is small and if enough resources are available to construct a new *index* while the old one is still available for *querying*.

If there is a requirement that new *documents* be included quickly, the simplest approach is to use *two indexes*, a large **main index** on *disk*, and a small **auxiliary index** that store new documents in *memory*. *Searches* are run across both *indexes*, *deletions* are stored in an **invalidation bit vector**, then we filter out *deleted documents* before returning the *search* result. When the *auxiliary index* becomes too large, we merge it with the *main index*. Unfortunately, this scheme is infeasible because most *file systems* cannot efficiently handle very large numbers of files and merges are computationally expensive. The simplest alternative is to store the *index* as one large file as a concatenation of all *postings list*, but this would generate a lot of files.

An alternative is the **Logarithmic Merge**, that reduces the cost of *merging indexes* over time. We maintain *several indexes*, **each twice larger than the previous one**, and the smallest Z_0 is maintained in memory , and the others $(I_0, I_1, ...)$ are on disk, when Z_0 becomes too large, we will write it to disk as I_0, or we merge it with I_0 if it already exist and write the merger to I_1 and so on. It use a *binary number* to save which index is full for example 1011 (positions 3 2 1 0), means that I_2 has space, instead the others are full. For example if we have a as size of a *memory index*, then on the *disk* we will have *indexes* of size $2a$, $4a$ and so on, so *logarithmic scale*. The **time complexity** of *index construction* in the worst case is $O(T\ logT)$, because $log\ T$ is the number of indexes with T number of postings, so for a query we need to merge $O(log\ T)$ indexes and the worst case is $O(T\ log\ T)$ because each of T posting is merged $O(log\ T)$ times. This is more efficient than the $O(n^2)$ of the previous alternative. (*supponiamo che una parola ha T postings nella sua posting list, se devo fare il merge su tutte le I_i allora avrò $T\ logT$ dato che i vari I_i sono ognuno il doppio del precedente*)

```
LMergeAddToken(indexes, Z0, token)
  Z0 ← Merge(Z0, {token})
  if |Z0| = n
    then for i ← 0 to ∞
           do if Ii ∈ indexes
                then Zi+1 ← Merge(Ii, Zi)
                      (Zi+1 is a temporary index on disk.)
                     indexes ← indexes − {Ii}
                else Ii ← Zi    (Zi becomes the permanent index Ii.)
                     indexes ← indexes ∪ {Ii}
                     Break
           Z0 ← ∅

LogarithmicMerge()
  Z0 ← ∅    (Z0 is the in-memory index.)
  indexes ← ∅
  while true
  do LMergeAddToken(indexes, Z0, getNextToken())
```

Because of this *complexity* of *dynamic indexing*, some large search engines adopt a **reconstruction from-scratch strategy**. They do not construct *indexes dynamically*, instead, a new *index* is built from scratch periodically. *Query processing* is then switched from the *new index* and the *old index* is deleted.

5 Index Compression Algorithms

Compression techniques for *dictionary* and *inverted index* are essential for efficient *IR systems*, one benefit of *compression* is straightforward: we need less *disk space*. Compression can be **lossy** (discard some *infos*, like *stopwords* or *lowering*), or **lossless** (all *infos* are preserved). We will use some variables in statistics:

- **N**: documents;
- **L**: average number of tokens per document;
- **T**: average number of bytes per term, non-positional postings;
- **M**: word types, average number of bytes per token;

5.1 Heaps' Law

In order to get the number of *distinct terms* M in a collection is to use the **Heaps's Law** which estimates *vocabulary size* as a function of collection size:

$$M = kT^b$$

Where T is the number of *tokens* in the collection, k and b are two parameters usually: $30 \leq k \leq 100$ and $b \cong 0.5$, this law's suggests that the *dictionary size* continues to increase with more *documents* in the *collection*, and the size of the *dictionary* is quite large for large collections. If a *term* is *frequent*, it will not characterize a document so much, and the contrary, so we can *rank* terms by their frequency (their *relevance*)

5.2 Zipf's Law

We also want to understand how *terms* are *distributed* across documents, a commonly used model of the distribution of *terms* in a collection is **Zipf 's Law**. It states that, if t_1 is the most common *term* in the collection, t_2 is the next most common, and so on, then the **collection frequency** cf_i of the i-th most common *term* is proportional to $1/i$:

$$cf_i \propto \frac{1}{i}$$

So if the most *frequent term* occurs cf_i times, then the second most *frequent term* has half as many occurrences, the third most frequent *term* a third as many occurrences, and so on. The *frequency* decreases very rapidly with *rank*. The goal is to encode most *frequent terms* to smaller size *encoding*. So the most frequent *word* will appear cf_1 the second one will appear $\frac{1}{2}cf_2$ times and so on.

5.3 Dictionary Compression

One of the *primary factors* in determining the *response time* of an *IR system* is the number of *disk seeks* necessary to process a *query*. If parts of the **dictionary** are on *disk*, then many more *disk seeks* are necessary in *query evaluation*, so, the main goal of compressing the *dictionary* is to fit it in *main memory*.

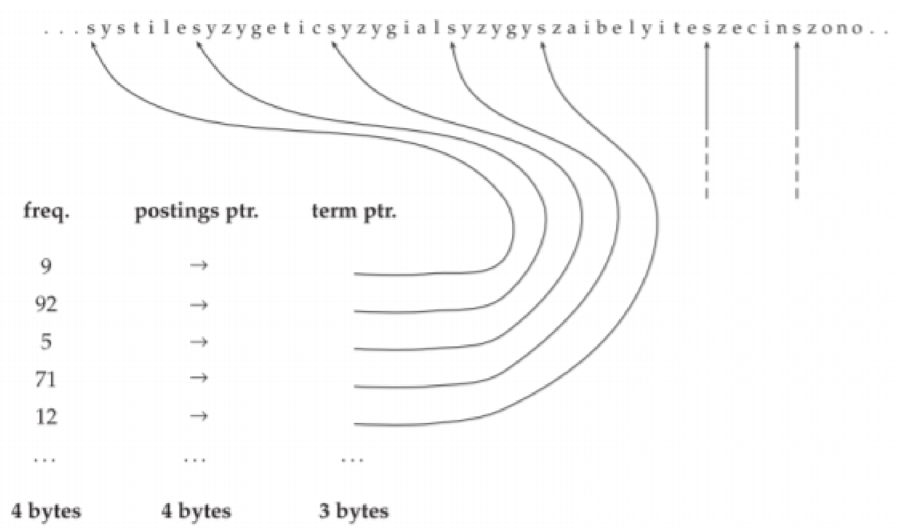
5.3.1 Dictionary as an array of fixed-width entry

The simplest *data structure* for the *dictionary* is to sort the *vocabulary lexicographically* and store it in an *array* of *fixed-width entries*. We allocate *20 bytes* for the *term* itself, *4 bytes* for its *document frequency*, and *4 bytes* for the *pointer* to its *postings list*. For example for *Reuters-RCV1* (a *dataset* with 400.000 elements) we need $M \times (20+4+4) = 400.000 \times 28 = 11.2MB$ for storing the *dictionary* in this scheme.

space needed:	20 bytes	4 bytes	4 bytes
	term	document frequency	pointer to postings list
	a	656,265	$\longrightarrow$
	aachen	65	$\longrightarrow$
	...	...	...
	zulu	221	$\longrightarrow$

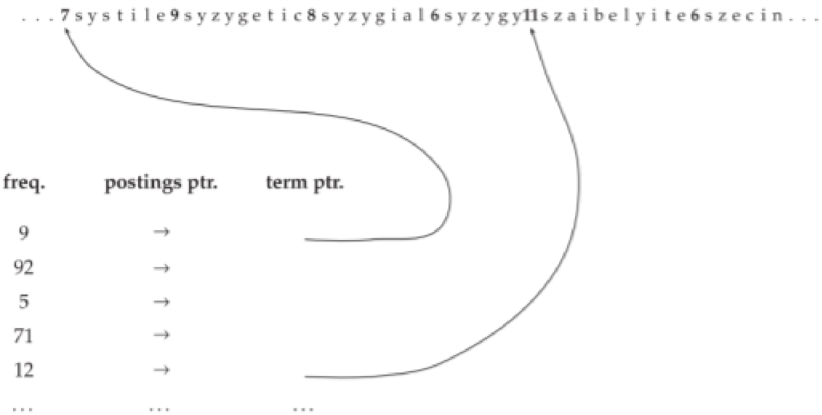
5.3.2 Dictionary as string

Using *fixed-width entries* for *terms* is wasteful, in fact average length of a *term* in English is about eight characters and we also we need a way of storing *terms* with more than twenty characters. We can overcome these problems by storing the *dictionary terms* as one **long string of characters**, so we use a pointer to demarcate the end of the current term. For Reuters-RCV1 we will use $(4+4+3+8) \times 400.000 = 7.6\ MB$, where 8 bytes are the average of the *term*.



5.3.3 Dictionary as string with blocking

We can also compress the *dictionary* by grouping *terms* in the *string* into **blocks** of size k and keeping a *term pointer* only for the first *term* of each *block*. We store the *length* of the *term* in the string as an *additional byte* at the beginning of the *term*, we thus eliminate $k-1$ *term pointers*, but we need an additional k bytes for storing the *length*. Dictionary of *Reuters-RCV1* is reduced by 0.5 MB, to 7.1 MB. (*se prima avevamo un puntatore da tre byte per ogni termine ora abbiamo un puntatore ogni blocco e un byte per ogni per la lunghezza e risparmiamo circa 5 byte per ogni blocco*)



There is a **tradeoff** between *compression* and *lookup time*, if we increase *block size k* we have better *compression*, but we have more *lookup time* for a *word* in that *block*. Consecutive *entries* in an *alphabetically sorted* list share common prefixes, this observation leads to **front coding**. A *common prefix* is identified for a *sub-sequence* of the *term list* and then referred to with a *special character*. In the case of Reuters, front coding saves another 1.2 MB.

One block in blocked compression ($k = 4$) . . .
8automata**8**automate**9**automatic**10**automation

⇓

. . . further compressed with front coding.
8automat$*$a**1**$\diamond$e**2**$\diamond$ic**3**$\diamond$ion

A sequence of terms with identical prefix ("automat") is encoded by marking the end of the prefix with $*$ and replacing it with $\diamond$ in subsequent terms. As before, the first byte of each entry encodes the number of characters.

5.4 Postings Compression

Posting lists total size is about 10 times larger than the *total dictionary size*, so we need to compress their *size*. One idea is to store **gaps** (*difference between the two document id index*) instead of *docsID*, for example the *word computer* $\rightarrow 28154, 28159, 28160, ...$ we can store is like *computer* $\rightarrow 28154, 5, 43, ...$ With the *original one* we use 20 bits for each *docID*, instead now with *gaps* that are usually shorter we use way less than *20 bits*. For an economical representation of this distribution of *gaps*, we need a *variable encoding method* that uses fewer bits for *short gaps*. To encode small numbers in less space than large numbers, we look at two types of methods: **bytewise compression** and **bitwise compression**.

5.4.1 Variable Bytecode

Variable Bytecode or **VB encoding** uses an *integral number* of bytes to encode a *gap*, and last *7 bits* of a byte are "*payload*" and encode part of the *gap*. It dedicate 1 bit to be a **continuation bit** c, if the gap G fits within 7 bit, binary encode it in the 7 available bits and we set $c = 1$ else we use more than a block (*at the beginning of each byte we can have 0 or 1, if 0 then is a number which requires more than a block, if it's 1 it fit in the 7 bit*). For example we have **docID** 824 and 829, so the gap is 5, 824 in binary is 1100111000 so it doesn't fit in 7 bit, we will write it as: **0**000110 **1**0111000 where the black numbers are *continuation bits* (so we use two blocks), instead *gap* 5 in binary is **1**0000101 cause it fit in 7 bits. The length of *7+1 bits* can be changed depending on how the gaps are, for small *gaps 4 bits block* are usually better.

5.4.2 Gamma Codes

VB codes use an adaptive number of *bytes* depending on the size of the *gap*, **Bit-level codes** (also called **Gamma Codes**) adapt the length of the *code* on the *finer grained bit level.* The simplest bit-level code is **unary code**, the *unary code* of n is a string of n 1s followed by a 0 (for example 3 = 1110, 4 = 11110), but this is not a very efficient code. **Gamma code** uses *length* and *offset* of a *gap* G. **The offset is the gap in binary without the leading 1**. For example 13: binary is 1101 and offset is 101, length encodes the length of offset in unary code, for 13 the length of offset is 3 bit that is 1110 in unary (*ao prendi l'offset, 101, vedi quant'è lungo, 3, lo schiaffi in unario, 1110*), so we have that the **gamma code** of 13 is: 1110 concat 101 = 1110101.

The *offset length* is: $\lfloor log_2\, G \rfloor$ bits, instead the length of *length part* is: $\lfloor log_2\, G + 1 \rfloor$ bits, so the length of the *entire code* is $2 \times \lfloor log_2\, G + 1 \rfloor$, this means that *gamma codes* are always of *odd length* and they are within a factor of 2 of what we claimed to be the optimal *encoding length* $log_2\, G$. In fact assuming the 2^n *gaps* G with $1 \leq G \leq 2^n$ are all equally likely, the *optimal encoding* uses n bits for each G, so some *gaps* cannot be encoded with fewer than $log_2\, G$ bits and our goal is to get as close to this **lower bound** as possible.

Gamma codes have also some important *properties*:

- **Prefix-Free**: no *gamma codes* is the *prefix* of the other, this means that there is always a unique *decoding* of a sequence of gamma codes, and we don't need *delimiters* between them.
- **Universal**: we can use it far any data distribution;
- **Parameter-Free**: there are no parameters in this procedure;

However machines have *word boundaries* so compressing at a *bit level* can be *expensive* (*slow*), for this reason the **VB code** can be a better solution.

6 Cap 6: Ranked Retrieval

Thus far, our queries have all been **Boolean**, *documents* either match or don't, this is good for *expert users* with precise understanding of their needs and of the *collection*, but it's not good for the majority of *users*. In fact for these *queries* we have:

- *Docs* that either match or don't (*too strict*);
- Thousands of results (*inefficient to present all of them in a web page*);
- Too few or too many results;

Users need a **ranked series** of results and, let's say, the first 10 ones, so not a complete result of thousands of elements. **Ranking** is done with respect to specific criteria of **relevance**, and is measured through a **score** in $[0, 1]$ assigned to each *query-document pair*. For example in the **1-Word Query**: the most the *query term* appears in a *doc*, the higher the *ranking* will be, if it doesn't appear *score* is 0. There are also some alternatives:

- **Jaccard Coefficient**:
  - A commonly used measure of overlap of two sets A and B, $JACCARD(A, B) = \frac{|A \cap B|}{|A \cup B|}$, A and B don't have to be the same size. The problem is that this method doesn't take *frequency* into consideration, and doesn't use *rare terms* (that are more informative than *frequent terms*).
- **Bags of words**:
  - A method in which we represent each *document* as a **count vector**, with *frequency* for every *term*, the problem is that we don't consider *order* so we will not use it;

The **term frequency** $tf_{t,d}$ of term t in document d, is defined as the number of times that t occurs in d. We need to use it when we compute *query-document match scores*, but we don't want that *relevance* is increased proportionally with the *term frequency*. We use a **log frequency weight** for t in d (can also be called $wf_{t,d}$):

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

So the **score** for a *document-query* pair is: $matching - score(q, d) = \sum_{k \in (q \cap d)} (1 + log_{10}\ tf_{t,d})$, which means the sum of the **weighted frequencies** of that *term* in both *query* and *document*.

We also want to use the **frequency of the term in the collection** for *weighting* and *ranking*, in fact let's consider a *term* that is really *rare* in a *collection*, it appears in only one *document*, this probably means that such *document* has a great relevance for the *collection*. So we want *high weights* for **rare terms** and *low weights* for *frequent words*. We will use **document frequency** to factor this into computing the *matching score*, that is the number of *documents* in the *collection* that the *term* occurs in.

We call df_t the number of *docs* in which *term t* occurs, it is an *inverse measure* of the **informativeness** of term t, so we define $idf_t = log_{10} \frac{N}{df_t}$ the measure of the *informativeness* of a term (where N is the number of *docs* in the *collection*). The best known weighting scheme is called **tf-idf weight** that is the product of *tf weight* (*term frequency* in a document) and *idf weight* (rarity measure of a *word* in the collection):

$$tf - idf_{t,d} = tf_{t,d} \times idf_t = (1 + log_{10}\, tf_{t,d}) \times log_{10} \frac{N}{df_t}$$

This scheme assign to term t a weight that is:

- Highest when t occurs many times within a small number of *documents*;
- Lower when t occurs fewer times in a document, or occurs in many *documents*;
- Lowest when the *term* occurs in all *documents*;

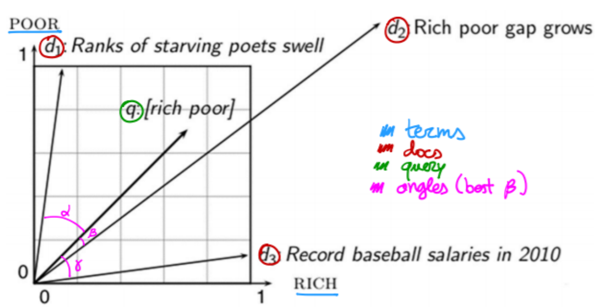
It's important to note that this formula uses the $wf_{t,d}$ (so the *log weighted frequency*), not just the *term frequency* itself.

6.1 Vector space model for scoring

The representation of a set of *documents* as *vectors* in a common *vector space* is known as the **vector space model**.

6.1.1 Dot Products

We denote by $\vec{V}(d)$ the **vector** derived from *document d*, with one component in the *vector* for each *dictionary* term. 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*. This *representation* loses the relative *ordering* of the terms in each document. We need to quantify the *similarity* between two documents in the *vector space*, we can consider the **magnitude** (*modulo*) of the *vector difference* between two *document vectors*, but two *documents* with very *similarity* content can have a significance vector difference if one document is much longer than the other.



Another way to quantify the similarity is to compute the **cosine similarity** of the *vector representations* of two documents:

$$sim(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)| \cdot |\vec{V}(d_1)|}$$

Where the numerator represents the dot product of the two vectors, while the denominator is the product of the **Euclidean Length** (that will length normalize the two vectors). This measure is the *cosine* of the angle θ between the two vectors. We can also rewrite as: $\vec{v}(d_1) = \frac{\vec{V}(d_1)}{|\vec{V}(d_1)|}$ and $\vec{v}(d_2) = \frac{\vec{V}(d_2)}{|\vec{V}(d_2)|}$ so the equation will be:

$$sim(d_1, d_2) = \vec{v}(d_1) \cdot \vec{v}(d_2)$$

We can also view the *query* as a *vector*, we consider a *query* q, this *query* into the *unit vector* $\vec{v}(q)$, so the idea is to assign to each document d a score equal to:

$$cos(\vec{q}, \vec{d}) = sim(\vec{q}, \vec{d}) = \frac{\vec{q}}{|\vec{q}|} \cdot \frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i \cdot d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \cdot \sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

Where:

- q_i is the **tf-idf weight** of the term i in the **query**;
- d_i is the **tf-idf weight** of the term i in the **document**;
- $|\vec{q}|$ and $|\vec{d}|$ **length** of $\vec{q}$ and $\vec{d}$;

Algorithm 4 The basic algorithm for computing vector space scores.

function COSINE_SCORE(q)
    float $Scores[N] = 0$
    Initialize $Length[N]$
    for all query term t **do** calculate $w_{t,q}$ and fetch postings list for t
        for all pair(d, tf$_{t,d}$) in postings list **do** Scores[d]$+= wf_{t,d} \times w_{t,q}$
        end for
    end for
    Read the array $Length[d]$
    for all d **do** $Scores[d] = Scores[d]/Length[d]$
    end for
    return Top K components of $Scores[]$
end function

In a typical setting we have a collection of *documents* each represented by a *vector*, a *text query* represented by a *vector*, and a positive integer K. We seek the K *documents* of the collection with the **highest vector space scores** on the given *query*. Typically, we seek these ordered by *decreasing score* (usually K = 10). The algorithm computes the *vector space scores*, and for each term t of the *query* it will update the *score* of the *document* by adding in the contribution from term t. This process is also known as **term-at-a-time scoring** or *accumulation*, and the N documents of the *array scores* are therefor known as *accumulators*. It is wasteful to store the *weight* of term t in document d since storing this *weight* may require a floating point number. The most complex and expensive operation is the *extraction* of the top K scores, this requires a *priority queue data structure*, often implemented using a **heap**. Each of the K top scores can be extracted from the *heap* at a cost of $O(log\ N)$ comparisons.

7 Cap 7: Computing Scores

We saw in the *Cosine Score Algorithm*, that we return the first K elements of *scores array*, so the most relevant ones, but now we want to do this without calculating all the *cosines*. So we are doing the **K-nearest neighbors problem** for a *query vector*. In general we don't know how to do this for *high dimensional spaces*, but it is solvable for *short queries*. We assume that each *query term* will occur only once, this means that for *ranking* we don't need to *normalize* query vector.

7.1 Efficient Scoring and Ranking schemes

7.1.1 Heap Tree

Let J be the total number of *non-zero cosine documents*, we need to find the K best of these J documents. The solution is to use a **heap**: in fact it takes $2J$ for constructing and $2\ log\ J$ for reading each *winner*. The *bottleneck* is that the *cosine computation* in this case has to be done for all the *tree elements* in order to build the *tree*.

7.1.2 Inexact top-K Retrieval

Another method is called **inexact top-K retrieval**, that is not from the user's perspective a bad thing, in fact we avoid all these calculations, even if we lose some *accuracy*: we find a set of **contenders** A with $K < |A| << N$, so A doesn't contain the top K but it has many *docs* from among the top K, and in such a way we return the top K docs in A, so we can think A as a **pruning non-contenders**, and this scheme is also used for *non-cosine based functions* of *scoring*.

7.1.3 Index Elimination

With **Index Elimination** scheme we use *two pruning*, we select only *query terms* with high *idf weight* and we only select *documents* which contains several *query terms*, for a **multi-term query** q:

- We only consider *documents* containing *terms* whose *idf weight* exceeds a preset threshold. In the *postings traversal* (*scorrere le varie posting list tipo un ciclo for*), we only traverse the *postings* for *terms* with high *idf*. The *postings lists* of *low-idf terms* are generally long, thus the set of documents for which we compute *cosines* is greatly reduced. *Low-idf terms* are treated as *stop words* and do not contribute to *scoring*.
- We only consider *documents* that contain many of the *query terms*. A danger of this *scheme* is that by requiring all (or even many) *query terms* to be present in a *document* before

considering it for *cosine computation*, we may end up with fewer than K candidate *documents* in the *output*.

7.1.4 Champion Lists

The idea of **champion lists** is to pre-compute, for each term t in the *dictionary*, the set r of documents with the *highest weights* for t. For *tf-idf weighting*, these would be the r documents with the highest tf values for term t. We call this set the **champion list** for term t.

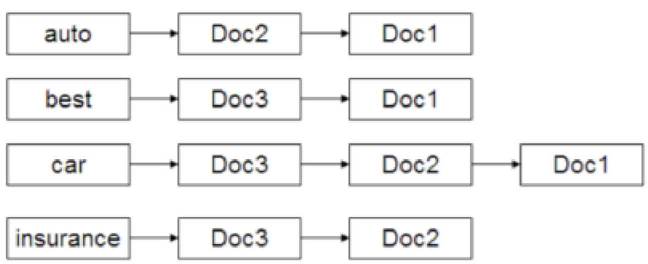
Given a query q we create a set A as follows: we take the union of the *champion lists* for each of the *terms* comprising q. We restrict *cosine computation* to only the *documents* in A. One issue here is that the value r is set at the time of **index construction**, whereas K is *application dependent* and may not be available until the *query* is received, so as a result we may find ourselves with a set A that has fewer than K documents.

7.1.5 Static Quality Score

In order to **top-ranking documents** we need to guarantee two important *properties*:

- **Relevance**: modeled by *cosine scores*;
- **Authority**: is *query-independent property* of a document that indicates its *validity*;

In many *search engines*, we have available a measure of **quality** $g(d) \in [0, 1]$ for each document d that is *query-independent* and thus **static**:



A static quality-ordered index. In this example we assume that Doc1, Doc2 and Doc3 respectively have static quality scores $g(1) = 0.25$, $g(2) = 0.5$, $g(3) = 1$.

The **Net Score** for a document d is a combination of $g(d)$ with the *query-dependent score*:

$$NetScore(q, d) = g(d) + cosine(q, d) = g(d) + \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)| \cdot |\vec{V}(d)|}$$

In this form, the static quality and the query score have equal contribution. Now we seek the top k docs by the net-score by using two parameters: α and β in order to give more importance to a term instead of the other:

$$NetScore(q, d) = \alpha \cdot g(d) + \beta \cdot cosine(q, d)$$

In order to get the first K documents, we need to **order** the documents with respect to $g(d)$, because this raises the probability to find the most relevant *docs* early in *posting traversal*, so we have better *performance*. There are three ways of acting:

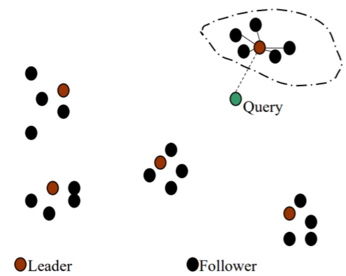
- **Global Champion List**:
  - We maintain for each term t a **global champion list** of the r documents with the highest values for $g(d) + tf - idf_{t,d}$ score, that will be sorted by a common order, so at *query time* we only compute the *net scores* for documents in the union of these *global champion lists*. We seek the *top-K results* from only the *docs* in these *champions lists*;
- **High and Low List**:
  - We maintain for each term t an **high list**, that contains the *documents* with the highest values for t, and a **low list** which contains the other *documents* containing t, we first scan only the *high list* of the *query terms*, if we obtain scores for K documents in the process we terminate, if not we continue in the *low list*;
- **Impact Ordering**:
  - In all the *postings lists* we order the documents by some *common ordering*. With **Impact order** we compute only *documents* with $wf_{t,d}$ high enough (should be *weighted log frequency* $1 + log_{10}\ tf_{t,d}$, *cosi de botto senza senso, sulle slide ogni tanto con tf intende wf e i due termini sembrano interscambiabili*), so we will sort each *postings list* by $wf_{t,d}$, this means that all the *postings lists* are in a different *order* (a *document* can be very relevant for a term, and useless for another). In order to compute the *score* for picking off top K there are two ways:
    * **Early termination**: when traversing t's *postings*, we stop after a fixed r docs or when $wf_{t,d}$ is too low, so we take the **union** of the resulting sets of *docs*, one for each *query term*, and we compute the *score* only for *documents* in this union;
    * **Idf-Ordered terms**: when considering *postings* of *query terms*, we look at them in order of decreasing **idf** (since high *idf* terms contribute most to the score). As we update *score contribution* for each *query term*, we stop if *document* scores relatively unchanged (?).

7.1.6 Cluster Pruning

Cluster Pruning is a technique in which we have a *pre processing step* during which we **cluster** the *document vectors*, then at *query time*, we consider only *documents* in a small number of *clusters* as *candidates* for which we compute *cosine scores*. In the *pre processing step* we take $\sqrt{N}$ *documents* randomly, and we call them **leaders**, for all the other, called **followers**, we compute the *nearest leader*.

The expected number of *followers* for each *leader* is $\approx \frac{N}{\sqrt{N}} = \sqrt{N}$. The *query processing* proceeds as follows:

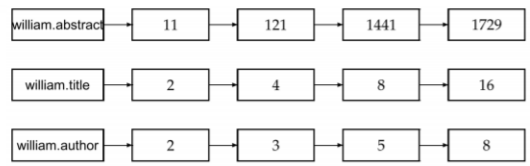
- Given a **query** q, find the **leader** L closest to q, this means computing *cosine similarity* between q and the $\sqrt{N}$ *leaders*;
- We seek for K nearest documents from the *leader's* $\sqrt{N}$ followers;



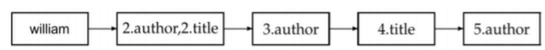
7.2 Combining components

7.2.1 Parametric and Zone Indexes

Some *docs* have specific *term* with *special semantics* called **metadata** which includes **fields** such as *format, author, year, ...* , and sometimes we want to search by these *fields* so we can build some **parametric indexes**, one for each *field*, in such a way we can select only the *documents* matching a specified *field* (like match a *date* in a *query*). There are also **zones**, similar to *fields*, but the contents of a *zone* can be arbitrary *free text*, like the *document title*. Whereas the *dictionary* for a *parametric index* comes from a *fixed vocabulary*, the *dictionary* for a *zone index* must structure whatever *vocabulary stems* from the text of that *zone*.



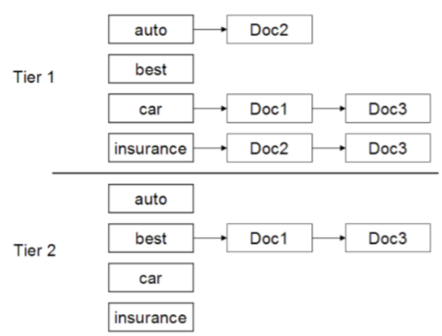
Basic zone index ; zones are encoded as extensions of dictionary entries.



Zone index in which the zone is encoded in the postings rather than the dictionary.

7.2.2 Tiered Indexes

We may occasionally find ourselves with a set of *contenders* that has fewer than K *documents*. A common solution in this case is the use of **Tiered Indexes**, which break *postings* up into a *hierarchy* of lists, from the most important to the last, this order can be done by $g(d)$ (*measure of quality of a document*) or another measure. At *query time* we seek *docs* in *top tier*, if it's not enough we go below.



Tiered indexes. If we fail to get K results from tier 1, query processing "falls back" to tier 2, and so on. Within each tier, postings are ordered by document ID.

7.2.3 Query-term Proximity

Especially for *free text queries*, users prefer a *document* in which most or all of the *query* terms appear **close** to each other, w is the smallest *window* in a *doc* which contains all the *query terms* (the *distance* between the *query terms* in the doc). The smallest w is, the better the *document* matches the query. Such **proximity-weighted scoring** functions are a departure from pure *cosine similarity* and closer to the *soft conjunctive semantics*.

7.2.4 Query Parser

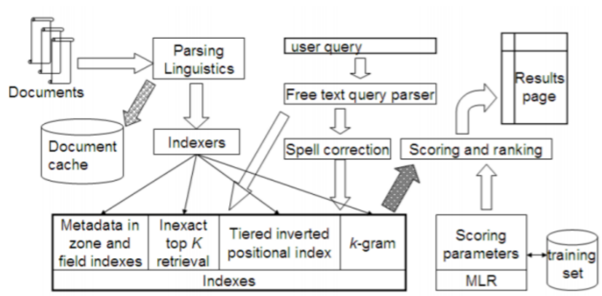
Common *search interfaces* tend to mask *query operators* from the end user, inviting **free text queries**. Typically, a **query parser** is used to translate the *user-specified keywords* into a *query* with various *operators* that is executed against the underlying *indexes*. This execution can entail *multiple queries* against the *underlying indexes*, for example, the *query parser* may issue a stream of *queries*:

- Run the *user-generated query string* as a **phrase query**, rank them by *vector space scoring* using as *query* the *vector* consisting of , for example, the 3 terms *rising interest rates*;
- If fewer than K *documents* contain the phrase *rising interest rates*, run the two **2-term phrase queries** *rising interest* and *interest rates*, rank these using *vector space scoring*;
- If we still have fewer than K results, run the *vector space query* consisting of the three individual *query terms*;

Each of these steps will compute a *score*, so we must combine these using an **aggregate scoring function** that accumulates evidence of a *document's relevance* from multiple sources.

7.3 Putting all together

Document will be **parsed** and will be applied a **language processing** (*tokenization*, *stemming*, ...). The resulting **tokens** will feeds into two modules: a copy of each parsed *document* will go in a **document cache**, instead a second copy is fed to the **indexers**, that will create *indexes* like **zone and field indexes**, **tiered positional index**, **indexes for spelling correction** and other *tolerant retrieval*, and structures for accelerating **inexact top-K retrieval**. A **free text user query** is sent down to the *indexes* both directly and through a module for generating *spelling-correction candidates*. **Retrieved documents** (dark arrow) are passed to a **scoring module** that computes scores based on **machine-learned ranking** (*MLR*). Finally, these **ranked documents** are rendered as a **results page**.



8 Evaluation of search results

8.1 Unranked Sets Measures

A *search engine* can be easily **evaluated** by measurable factors like:

- How fast does it **index** (*docs/hour*);
- How fast does it **search** (*latency as a function of index size*);
- Expressiveness of **query language**;

But the *key measure* is **User Happiness**, this cannot be quantified as the above factor, but it's something crucial for the *validity* of a *search engine*, it includes *speed of response*, *user-friendly UI*, and of course **relevance**. In order to measure ad hoc IR (*information retrieval*) we need a **test collection** composed by three things:

- A benchmark **document collection**;
- A benchmark **suite of queries**;
- A set of relevance *judgments*, usually binary assessment of either **relevant** or **non-revelant** for each *query-document pair*;

In fact for each *document*, with respect to a *user information need*, will be assigned a *binary classification* as *relevant* or *non-relevant*, and this decision is referred as the **ground truth** judgment of *relevance*. A *document* is **relevant** if it addresses the *stated information need*, not because it just happens to contain all the *words* in the *query*.

The two most *frequent* and basic measures for *information retrieval* effectiveness are **precision** and **recall**:

- **Precision, P**, is the fraction of *retrieved documents* that are *relevant*:

$$Precision = \frac{\#(relevant\ items\ retrieved)}{\#(retrieved\ items)} = P(relevant \mid retrieved)$$

- **Recall, R**, is the fraction of *relevant documents* that are *retrieved*:

$$Recall = \frac{\#(relevant\ items\ retrieved)}{\#(relevant\ items)} = P(retrieved \mid retrieved)$$

These notions can be summarized in this table:

	Relevant	Nonrelevant
Retrieved	true positives (tp)	false positives (fp)
Non retrieved	false negatives (fn)	true negatives (tn)

We can also write *precision* and *recall* as:

- $P = tp/(tp + fp) \quad F = tp/(tp + fn)$

An alternative is using **accuracy** $= (tp + tn)/(tp + fn + gn + tn)$, but this is useless for *IR*, because the *true negative documents* set is huge, and a system tuned to maximize *accuracy* will give a lot of *false positive documents*. *Precision* and *recall* will trade off against one another, for example you can always get a *recall* of 1 but with very low *precision* by retrieving all *documents* for all *queries*, in general we want to get some amount of *recall* while tolerating only a certain percentage of *false positive*.

Another measure of **relevance** that trades off *precision* versus *recall* is called **F-Measure**, which is the weighted harmonic mean of *precision* and *recall*:

$$F = \frac{1}{\alpha \cdot \frac{1}{P} + (1-\alpha) \cdot \frac{1}{R}} = \frac{(\beta^2 + 1) \cdot P \cdot R}{\beta^2 \cdot P + R} \quad where \;\; \beta^2 = \frac{1-\alpha}{\alpha}$$

Where $\alpha \in [0, 1]$ and $\beta \in [0, \infty]$, the most used is the so called **Balanced F-Measure** which uses $\alpha = 1/2$ and $\beta = 1$ commonly written as F_1 so the formula is simplified as:

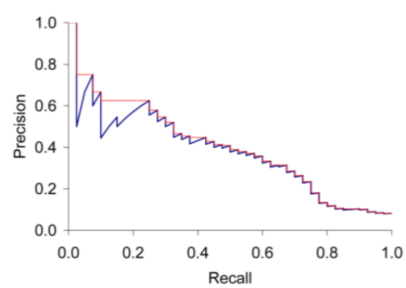
$$F_{\beta=1} = \frac{2 \cdot P \cdot R}{P + R}$$

Using values $\beta < 1$ will emphasize *precision*, instead values $\beta > 1$ will emphasize *recall.* We use **harmonic mean** because we want to punish *bad performance* either on *precision* or *recall*, and when we have number that differ greatly the *harmonic mean* in closer to their **minimum** than *arithmetic* or *geometric mean*:

> To explain, consider for example, what the average of 30mph and 40mph is? if you drive for 1 hour at each speed, the average speed over the 2 hours is indeed the arithmetic average, 35mph.
>
> However if you drive for the same distance at each speed -- say 10 miles -- then the average speed over 20 miles is the harmonic mean of 30 and 40, about 34.3mph.
>
> The reason is that for the average to be valid, you really need the values to be in the same scaled units. Miles per hour need to be compared over the same number of hours; to compare over the same number of miles you need to average hours per mile instead, which is exactly what the harmonic mean does.
>
> Precision and recall both have true positives in the numerator, and different denominators. To average them it really only makes sense to average their reciprocals, thus the harmonic mean.

8.2 Evaluation of ranked retrieval results

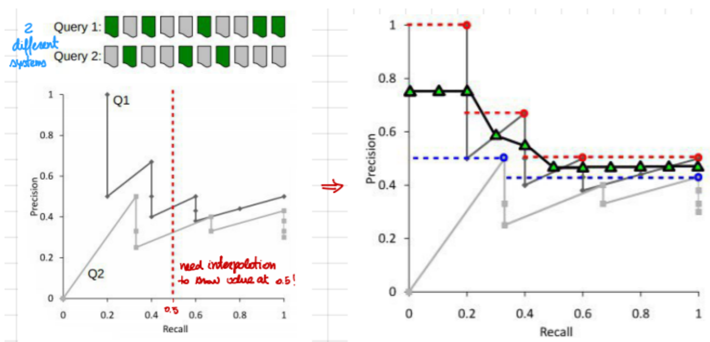
Precision, recall, and F, are measures made for **unranked sets**, in fact they are computed using *unordered sets of documents.* In a **ranked retrieval context**, the set of *retrieved documents* are naturally given by the top K *retrieved documents*, so *precision* and *recall* values can be plotted to give a **precision-recall curve**: if the $(k + 1)$-th document retrieved is *non-relevant*, then *recall* is the same of the top K documents, but *precision* has dropped, instead if it *relevant* then both *precision* and *recall* increase, and the curve jags to the right.



Precision/recall graph

In order to remove the *jiggles* the standard way is using the **interpolation precision** p_{interp}. The idea is, if locally *precision* increases with increasing *recall*, then you take the max of *precision* to right of value. At certain *recall level* r, the *interpolation* is defined as the **highest precision** found for any recall level $r' \geq r$:

$$p_{interp}(r) = \max_{r' \geq r} p(r')$$



Praticamente trovi il miglior bilancio tra precision e recall trovando i punti ottimali (i triangoli) dove ottieni il massimo tra le due

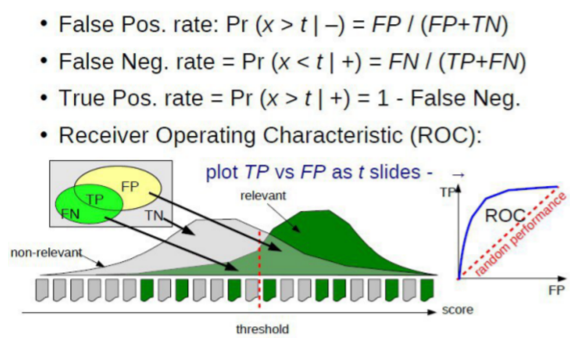
Examining the entire *precision-recall curve* is very informative, but there is often a desire to reduce this *information* down to a few numbers. The traditional way of doing this is the **11-point interpolated average precision**, which for each *information need*, the *interpolated precision* is measured at the **11 recall levels** of 0.0, 0.1, 0.2,..., 1.0, then for each *recall level* we will calculate the arithmetic *mean* of the *interpolated precision* for each *information need* in the test collection.

There are also other measures that have become more common, like:

- **MAP**:
  - The **Mean Average Precision**, for a *single information need*, is the average of the *precision* obtained for the set of top K documents, each time a *relevant document* is retrieved. It avoid to use *interpolation*, since it use **fixed recall levels**.
- **DCG**:
  - The **Discounted Cumulative Gain**, is a measure of *ranking quality*, given n documents, each has a rating $r_1, r_2, ..., r_n$:

$$DCG = r_1 + \frac{r_2}{log\ 2} + \frac{r_3}{log\ 3} + ... + \frac{r_n}{log\ n}$$

8.3 Relevance Assessment



In order to properly evaluate a *system*, the *test information* need to be sure that the *documents* are **relevant** for the usage of the *system*, this is a *time-consuming* and *expensive process* involving human beings. For large modern collections, it is usual for *relevance* to be assessed only for a subset of the *documents* for each *query*. The most standard approach is **pooling**, where *relevance* is assessed over a subset of the *collection* that is formed from the top K *documents* returned by a number of different *IR systems*, but the problem is a human is not a device, all have different *judgments system*. In the social sciences, a common measure for **agreement** between judges is the **Kappa Measure**, designed for *categorical judgments* and a simple *agreement rate for* the rate of *chance agreement*:

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

Where $P(A)$ is the proportion of times the judges agreed, $P(E)$ is the the proportion of times they would be expected to agree by change. The *Kappa value* will be 1 if two *judges* always agree, 0 if the agree only at the rate given by chance, and negative if they are worse than random, and if there are more than two judges, it is normal to calculate an average *pairwise kappa value*. Usually a Kappa vale $\in \left[\frac{2}{3}, 1\right]$ is acceptable. An example:

$P(A) = 370/400 = 0.925$
$P(non{-}relevant) = (10{+}20{+}70{+}70)/800 = 0.2125$
$P(relevant) = (10 + 20 + 300 + 300)/800 = 0.7878$
$P(E) = P(non - relevant)^2 + P(relevant)^2 =$
$0.2125^2 + 0.7878^2 = 0.665$
$kappa = (0.925 - 0.665)/(1 - 0.665) = 0.776$

Number of docs	Judge 1	Judge 2
300	Relevant	Relevant
70	Nonrelevant	Nonrelevant
20	Relevant	Nonrelevant
10	Nonrelevant	Relevant

Another problem with the *relevance-based assessment* is the distinction between *relevance* and **marginal relevance**: whether a *document* still has distinctive *usefulness* after the user has looked at certain other *documents*, even if a *document* is highly relevant, its information can be completely **redundant** with other *documents* which have already been examined. Maximizing *marginal relevance* requires returning *documents* that exhibit diversity and novelty.

8.3.1 Interactive Relevance Feedback

We make a query, we take the first results set and ask user to select relevant and non-relevant documents, so the user will see the result summaries