Information Retrieval and Web Search

Academic Year 2022/2023

Nicola Aggio 880008

Index

1 Introduction

1.1 Course structure

The exam is composed of the following parts:

- a **written exam**, providing the **60%** of the overall grade;
- the **discussion of a project** (software and report) or the **presentation of a paper**, providing the remaining **40%** of the grade. For the discussion of a project, there may be 1 or 2 extra points for good projects.

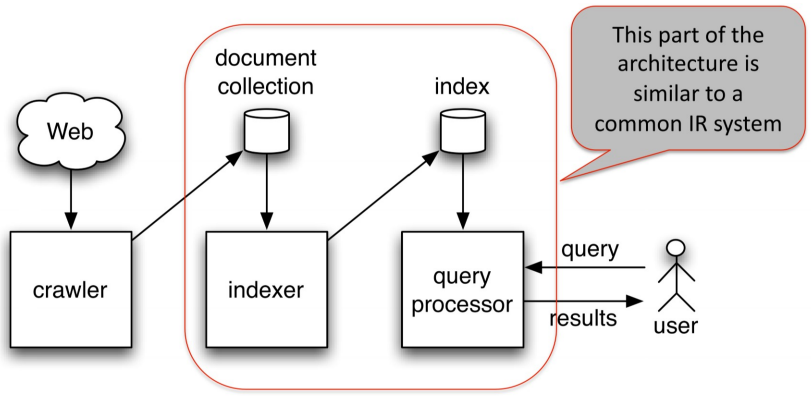
1.2 What Information Retrieval Is

Information retrieval (IR) is about finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers). Information retrieval is fast becoming the dominant form of information access, overtaking traditional database-style searching. The first example of information retrieval system can be considered the *Memex Machine* conceived by Vannevar Bush in the 30s, which actually cannot be classified as a real IR system, since it was more similar to a browsing hypertext system. In this sense, the principal driver of innovation for IR has been the World Wide Web, and in turn the continuous optimization of IR effectiveness, i.e. the quality of its results, and efficiency, i.e. the speed of the search, has driven web search engines to new quality levels.
Nowadays, the largest fraction of the data volume and the market cap is represented by **unstructured data**, i.e. data which has not a clear easy-for-a-computer structure, for example text documents etc.. An important factor for the growth of the volume of unstructured data has been the Web growth, along with the Web pages size growth and the search engine usage. In general, a *Search Engine Result Page (SERP)* is not represented exclusively by an algorithmic search results, but also with images, videos, query suggestions and advertisements. More specifically, the agents involved in the Web search field are the *users*, who try to access the information and request relevance and speed, the *search engine*, which tries to attract more users, to increase the ads revenues and to reduce the operational costs, and the advertisers, which wants to attract the users to their sites by paying little money. In general, the Web Search Engine is composed of three major components, as represented in Picture 1.2:

- the *web crawling* phase;
- the *indexing* phase;
- the *query processing* phase.

1.3 Basic assumptions

A **collection** is a set of documents, that we assume to be static at the moment, and the goal of IR is to retrieve documents with information that is relevant to the user's information need and helps the user to complete a task. In order to evaluate the quality of the retrieved documents, we introduce two basic metrics:



- the *precision*, which represents the fraction of retrieved documents that are relevant the the user's information need;
- the *recall*, which represents the fraction of all relevant documents in the collection that are retrieved.

2 Boolean Retrieval

In this chapter we begin with a very simple example of an information retrieval problem, and introduce the idea of a term-document matrix and the central inverted index data structure. We will then examine the Boolean retrieval model and how Boolean queries are processed.

2.1 An example of IR problem

Suppose that we want to determine which plays of Shakespeare contain the words *Brutus* AND *Caesar* AND NOT *Calpurnia*.
The simplest solution to this problem is grepping through the text, which can be a very effective process, especially given the speed of modern computers, and often allows useful possibilities for wildcard pattern matching through the use of regular expressions. However, this solution has some disadvantages:

- it is slow, because it requires a linear scan of the entire collection of documents. This problem can be avoided exploiting an index;
- it does not allow to implement other operations, for example "find the word *Romans* near *countrymen*";
- we would like to have a ranked retrieval, i.e. to return the best documents according to the user's information need.

Two important operations in IR are **tokenization** and **vectorization**. Tokenization's goal is to segment a text into words or characters, while tokenization is the process of transforming text into numeric vectors. Suppose we record for each document whether it contains each word out of all the words Shakespeare used (Shakespeare used about 32,000 different words): the result is a binary **term-document incidence matrix**, as represented in Picture 2.2.

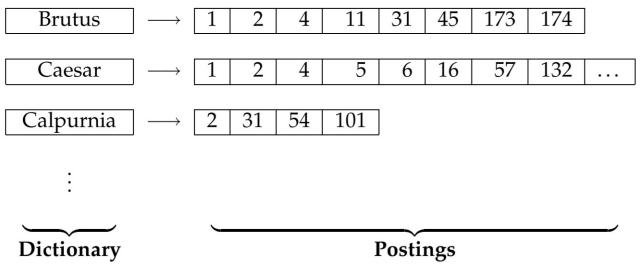
	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	…
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
…							

In this matrix, for each term we have a binary vector representing the set of documents including the term. Using this representation, we can answer the previous query by computing a bitwise AND between the vectors of *Brutus*, *Caesar* and *Calpurnia* (complemented). The result is: 110100 AND 110111 AND 101111 = 100100, i.e. the plays that satisfy the query are *Antony and Cleopatra* and *Hamlet*.
In this sense, a **Boolean retrieval model** is a model for information retrieval in which we can pose any query which is in the form of a Boolean expression of terms, that is, in

which terms are combined with the operators AND, OR, and NOT. In this model each document is represented by just using a set of words.
Let's consider a more realistic scenario, and suppose that we have a collection composed of $N = 1$ million documents, each of which is in turn composed of 1000 words (so 1G words in total). If we assume an average of 6 bytes per word, then the size of the collection is 6GB, and if there are $M = 500K$ distinct terms among the collection, then the matrix would have a dimension 500K x 1M. Moreover, since the collection only contains 1G of words, the matrix would be extremely sparse. For this reason, the matrix representation is not feasible for real world data dimension, so a much better representation is given by storing for each term only the documents in which that term appears. This approach can be implemented using the inverted index.

2.2 Inverted Index

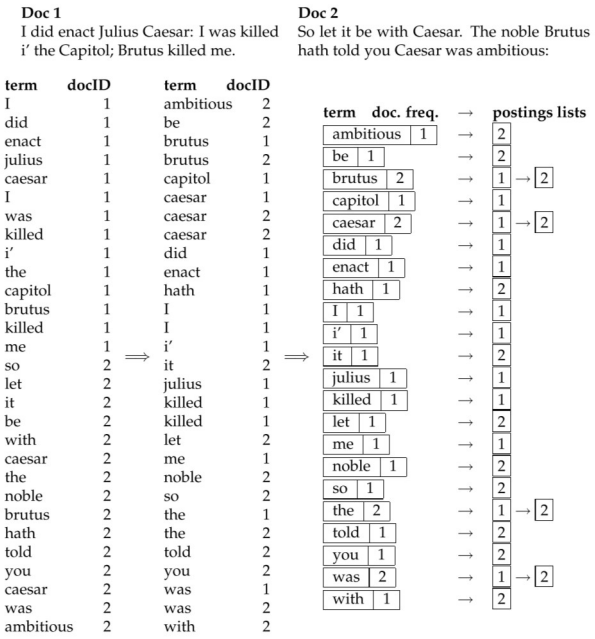
As introduced before, the idea of the **inverted index** structure is that for each term t, we only store a list of all the documents that contain t, the so called postings list.



Usually, the dictionary is stored in the memory, with pointers to each postings list, which is stored on disk. The inverted index construction follows the following steps:

1. Collect the documents to be indexed;
2. Tokenize the text, turning each document to a list of tokens;
3. Apply some linguistic preprocessing in order to obtain the indexing terms (ex: stemming, removing stop words, etc..);
4. Create the inverted index combining dictionary and postings.

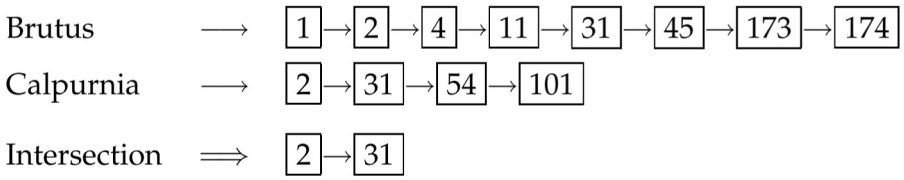
An example of this building process in represented in Picture 5.2
As we can see in the example, the terms of the dictionary are sorted in alphabetical order, and for arch term the document frequency information, i.e. the length of its postings list, is added for each term, and it can be helpful for improving the efficiency of the search engine at query time. Since both the dictionary and the postings list are stored, their size is important. Concerning the postings list, a fixed length array would be wasteful as some words occur in many documents, and others in very few, so two good alternatives are **singly linked lists** or **variable length arrays**. However, variable length arrays win in space requirements by avoiding the overhead for pointers and in time requirements because their use of contiguous memory increases speed on modern processors with memory caches.



2.3 Processing boolean queries

Now the question is: how do we process a query using the inverted index structure and the boolean retrieval model? In we consider the query "*Brutus* AND *Caesar*", then the result can be obtained with the following steps:

1. locate *Brutus* in the dictionary;
2. locate *Caesar* in the dictionary;
3. compute the intersection of the two postings lists;



As we can imagine, the intersection (or merging) operation is crucial: it must be able to retrieve in a fast way the documents that contain both terms. A possible implementation of the intersection operation is represented in Picture 2.3: in this case the complexity of the operation is $O(x + y)$, where x and y are the lengths of the two postings lists. Note that a crucial assumption of this implementation is that the two lists are sorted.
One possible way to speedup the intersection operation is represented by the usage of the *postings list with skip pointers* data structure, as shown in Picture 2.3. Skip pointers are effectively shortcuts that allow us to avoid processing parts of the postings list that will not figure in the search results: at each position of the postings list we can either process the following element or use the pointer to skip irrelevant elements.
Now the two questions are **where** to place skip pointers and **how** to do efficient merging using skip pointers. Suppose we're considering the two postings lists of Picture 2.3, and

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then p1 ← next(p1)
 9                else p2 ← next(p2)
10  return answer
```

Figure 1: Basic implementation of the intersection operation

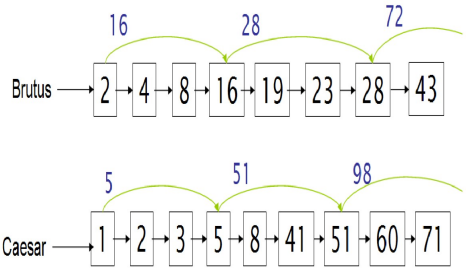


Figure 2: Postings list with skip pointers

that we have matched 8 on each list: we advance both pointers and we reach 16 in the first list, and 41 in the second. Now, since the skip list pointer points to 28, which is still less than 41, we can exploit it to jump directly to 28, without processing 19, and 23. Notice that this is possible thanks to the assumption that the lists are sorted. Ax example of **algorithm** that implements the merge operation using the postings lists with skip pointers is shown in Picture 2.3.

```
INTERSECTWITHSKIPS(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
 9                        then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
10                             do p1 ← skip(p1)
11                        else p1 ← next(p1)
12                else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
13                        then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
14                             do p2 ← skip(p2)
15                        else p2 ← next(p2)
16  return answer
```

Figure 3: Postings lists intersection with skip pointers

Regarding the **place** in which inserting the skip pointers, there's a **trade-off** between

the number of pointers and the space that they occupy. On the one hand, the more skip pointers, the shorter the skip spans will be, resulting in many comparisons to skip pointers and a lot of space used to store them. On the other hand, the fewer skip pointers, the longer the skip spans will be, resulting in less opportunities to skip and, consequently, in less successful skips. A simple **heuristic** indicates that for postings list of length P we should use $\sqrt{P}$ evenly-spaced skip pointers, ignoring the distribution of query terms.
In general, building effective skip pointers is easy if an index is relatively static, but it is harder if a postings list keeps changing because of updates.

2.4 Properties

Among the advantages, we underline:

- It is **precise**, if the right strategies for searching are known;
- It is **efficient** for computers.

On the other hand, the disadvantages are:

- The user must learn the **boolean logic**;
- The **boolean logic** is **insufficient** to capture the **richness of the language**;
- There's **no control** over the **size of the result set**: either too many or none documents are returned. Moreover, **all** the documents in the results are considered **equally good**, i.e. there's no rank for the documents in the solution;
- It **does not support partial matches**.

2.5 Query optimization

An important issue about *Boolean retrieval model* is represented by understanding which is the best order for query processing. Suppose we're dealing with the following query *Brutus* AND *Calpurnia* AND *Caesar*: for each term we have to get their postings and AND them together. In this case, the best choice would be to process the postings in order of increasing frequency, i.e. start with the smallest postings, by exploiting the document frequency information which is stored. By following this approach, the *intersect* operation is implemented as shown in Picture 2.5.

```
INTERSECT(⟨t_1, ..., t_n⟩)
1  terms ← SORTBYINCREASINGFREQUENCY(⟨t_1, ..., t_n⟩)
2  result ← postings(first(terms))
3  terms ← rest(terms)
4  while terms ≠ NIL and result ≠ NIL
5  do result ← INTERSECT(result, postings(first(terms)))
6     terms ← rest(terms)
7  return result
```

If we consider another query, for example (*Brutus* OR *Caesar*) AND NOT (*Anthony* OR *Cleopatra*) AND (*Brutus* OR *Cleopatra*), then another approach to speedup the query could be estimate the size of each OR operation by the sum of its document frequencies, and then process in increasing order of the OR sizes.

2.6 Positional postings and phrase queries

Most recent search engines support a double quotes syntax (*"stanford university"*) for **phrase queries**, which has proven to be very easily understood and successfully used by users, as 10% of web queries are phrase queries, and many more are implicit phrase queries (such as person names), entered without use of double quotes.
To be able to support such queries, it is no longer sufficient for postings lists to be simply lists of documents that contain individual terms. In this section we consider three approaches to supporting phrase queries and their combination in an efficient way.

2.6.1 Biword indexes

One approach could be considering each pair of consecutive terms (biword) in a document as a phrase: for example, the text *"Friends, Romans, Countrymen"* would generate two biwords: *"friends romans"* and *"romans countrymen"*. Each of the **biword** is now a **dictionary term**, so in this particular case we would be able to solve a two-word phrase query processing in an immediate way. In general, longer queries can be processed by breaking them down: for example the query *"stanford university palo alto"* can be broken into the following boolean query on biwords: *"stanford university"* AND *"university palo"* AND *"palo alto"*.
Two important issues about biwords are:

- There can be some **false positives**: without examining the documents, we cannot verify that the documents matching the Boolean query do actually contain the original word phrase;
- The problem of the **index blowup** due to bigger dictionaries.

For these reasons, the biword index is not the standard solution, but it can be part of a compound strategy.

2.6.2 Positional indexes

This approach is most commonly employed, and it is based on the idea of storing, for each term in the vocabulary, the postings of the form docID: <position1, position2, .., >, where each position is a token index in the document. Each posting will also usually record the term frequency. An example of positional index is provided in Picture 2.6.2: as we can see, for each term, the term frequency is indicated, along with the term frequencies and the positions for each document. In this case, *to* has a document frequency of 993,427, and it appears 6 times in document 1 at positions 7, 18, etc..
To process a phrase query, we still need to access the inverted index entries for each distinct term. As before, we would start with the least frequent term and then work to further restrict the list of possible candidates. In the **merge operation**, the same general technique is used as before, but rather than simply checking that both terms are in a document, we also need to **check** that **their positions** of appearance in the document are compatible with the phrase query being evaluated. This requires working out offsets between the words. For example, suppose that the positional index for the words *to* and *be* are the ones in Picture 2.6.2, and that the query is *to be or not to be* in

to, 993427:
⟨ 1, 6: ⟨7, 18, 33, 72, 86, 231⟩;
2, 5: ⟨1, 17, 74, 222, 255⟩;
4, 5: ⟨8, 16, 190, 429, 433⟩;
5, 2: ⟨363, 367⟩;
7, 3: ⟨13, 23, 191⟩; . . . ⟩

be, 178239:
⟨ 1, 2: ⟨17, 25⟩;
4, 5: ⟨17, 191, 291, 430, 434⟩;
5, 3: ⟨14, 19, 101⟩; . . . ⟩

Figure 4: Positional index

this case we first look for the documents that contain both terms, in this case doc1, doc4 and doc5. Then, we look for places in the lists where there is an occurrence of *be* with a token index one higher than a position of *to*, and then we look for another occurrence of each word with token index 4 higher than the first occurrence. In the above list, these conditions are guaranteed for doc4, since *to* appears in positions 429 and 433, while *be* appears in positions 430 and 434. Notice that doc4 represents a good candidate for solving the query, but an additional check must be performed for terms *or* and *not*.
In general, positional indexes allow to solve k word proximity searches, i.e. searches in which we fix a windows, while biword indexes cannot. Picture 2.6.2 shows an algorithm for implementing the k word proximity searches: the algorithm finds places where the two terms appear within k words of each other and returns a list of triples giving docID and the term position in $p1$ and $p2$.
A positional index **expands postings storage substantially**, even if the positions/offsets are compressed. Indeed, the asymptotic complexity of a postings intersection operation is no longer bounded by the number of documents N, but by the total number of tokens in the document collection T. However, positional indexes are now **standardly used**, because of the power and usefulness of phrase and proximity queries. From the point of view of space implication, since a posting needs an entry for each occurrence, the **index size** depends on the **average document size**: the average web page has less than 1000 terms, but financial documents, books etc.. may easily reach 100,000 terms. If we consider an average term frequency of 0.1%, then large documents cause an increase of two orders of magnitude in the space required to store the postings list, as represented in Picture 2.6.2.
In general, according to some rules of thumb, a positional index is 2 to 4 times as large as a non-positional index, and a compressed positional index is about one third to one half the size of the raw text (after removal of markup, etc.) of the original uncompressed documents.

2.6.3 Combination schemes

This strategy combines the biword and the positional indexes approach, since it uses a phrase index, or just a biword index, for certain queries and uses a positional index for other phrase queries. The idea behind this approach is that it is usually inefficient to

```
POSITIONALINTERSECT(p1, p2, k)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then l ← ⟨ ⟩
 5             pp1 ← positions(p1)
 6             pp2 ← positions(p2)
 7             while pp1 ≠ NIL
 8             do while pp2 ≠ NIL
 9                do if |pos(pp1) − pos(pp2)| ≤ k
10                      then ADD(l, pos(pp2))
11                      else if pos(pp2) > pos(pp1)
12                              then break
13                   pp2 ← next(pp2)
14                while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
15                do DELETE(l[0])
16                for each ps ∈ l
17                do ADD(answer, ⟨docID(p1), pos(pp1), ps⟩)
18                pp1 ← next(pp1)
19             p1 ← next(p1)
20             p2 ← next(p2)
21        else if docID(p1) < docID(p2)
22                then p1 ← next(p1)
23                else p2 ← next(p2)
24  return answer
```

Figure 5: Algorithm for proximity intersection of two postings lists

Document size	Expected postings	Expected entries in positional posting
1000	1	1
100,000	1	100

Figure 6: Average number of positional postings with different document sizes

keep on merging positional postings lists, so for this reason the most common queries are included in the phrase index. Williams et al. (2004) evaluated a more sophisticated mixed indexing scheme, composed of biwords and positional indexes, and a typical web query mixture was executed in one-fourth of the time of using just a positional index. However, it required 26% more space than having a positional index alone.

3 The term vocabulary

We recall that the main steps for constructing an inverted index are:

1. Collect the documents;
2. Tokenize the text;
3. Do some linguistic pre-processing of the tokens;
4. Create the inverted index.

In this chapter we first examine the possible formats and aspects of the input document, and then we will focus on the steps of tokenization and linguistic pre-processing of them.

3.1 Documents

Digital documents that are the input to an indexing process are typically bytes in a file or on a web server. The first step of processing is then to convert this byte sequence into a linear sequence of characters: however, many formats for the input file exist, along with many different encoding in which the convertion can take place. In this sense, the problem of choosing the best encoding could be addressed by a machine learning algorithm, but it is usually handled heuristically. Moreover, there exist some open source libraries that can handle the presence of multiple languages and formats in a document. Finally, the last issue regards the granularity of the documents: in particular, there's a trade-off between precision of the IR system and the recall. If the units get too small, we are likely to miss important passages because terms were distributed over several mini-documents, while if units are too large we tend to get spurious matches and the relevant information is hard for the user to find. In the following sections we assume that a suitable document size is chosen.

3.2 Tokenization

Given a character sequence and a defined document unit, **tokenization** is the task of chopping it up into pieces, called *tokens*, perhaps at the same time throwing away certain characters, such as punctuation. Despite being often called *terms* or *types*, there's a difference between these entities:

- 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 the class of all tokens containing the same character sequence;
- a *term* is a (perhaps normalized) type that is included in the IR system's dictionary.

For example, if the document to be indexed is *to sleep perchance to dream*, then there are 5 tokens, but only 4 types (since there are two instances of *to*).
Now the question is: what are the correct tokens to use? How do we extract them? In general, for each language there are some tricky cases (e.g. if we have *aren't*, we could tokenize it into *are* and *n't* or *aren* and *t* etc..), and for this reason the **issues** of tokenization are **language-specific**. Some examples of issues of tokenization are (more on Chapter 2 of the book):

- How to split ambiguous words (e.g. *aren't*, *O'Neill* etc..);
- How to deal with special types of characters, such as e-mail addresses, web URLs, IP addresses etc.. A possible solution could be represented by omitting them, resulting on the other hand in restricting a lot what people could search for;
- In English, *hyphenation* is a popular technique for splitting up vowels in words or for joining nouns as names; however, handling hyphens automatically could be hard, and it usually done in an heuristic way;
- Other language-specific issues (e.g. French, German, Chinese etc..).

3.3 Stop words

We define **stop words** as extremely common words (around 30%) that would appear to be of little value in helping selecting documents matching a user need. The general strategy for determining a *stop list* is to sort the terms by collection frequency, and then to take the most frequent terms and discard them. The usage of the stop list allows both to significantly **reduce the number of postings** that the system has to store and to save a lot of time in the indexing phase. However, in some cases the stopwords are very useful, for example in the *phrase queries*, such as *President of United States*, in *relational queries*, such as *Flights to Venice* or in some song titles, for example *Let it be*.
The general trend in IR systems over time has been from standard use of quite large stop lists (200–300 terms) to very small stop lists (7–12 terms) to **no stop list** whatsoever. Web search engines generally do not use stop lists.

3.4 Normalization

Having broken up our documents (and also our query) 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. For instance, if you search for *USA*, you might hope to also match documents containing *U.S.A.* In this sense, *token normalization* is the process of canonicalizing 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**, which are normally named after one member of the set. For instance, if the tokens *anti-discriminatory* and *antidiscriminatory* are both mapped onto the term *antidiscriminatory*, in both the document text and queries, then searches for one term will retrieve documents that contain either.
Another issue about tokenization is represented by **accents** and **diacritics**, which in some languages are a regular part of the writing system and distinguish different sounds and meanings. In these cases, the important question is how the users are likely to write their queries for these words: even in languages that standardly have accents, users often may not type them (lazyness, speed etc..). In these cases, it might be the best to equate all words to a form without diacritics. A common strategy is to do **case-folding** by reducing all letters to lower case: in this case, all the terms will be retrieved, regardless of the correct capitalization. However, such case folding could equate words that might be better be kept apart, for example *General Motors*, *Fed* etc..

3.5 Stemming and Lemmatization

The goal of both **stemming and lemmatization** is to **reduce inflectional forms** and sometimes of derivationally related forms of a word to a common base form. For instance, *am, are* and *is* become *be*: if we apply this to a sentence, *the boy's car are of different colors* becomes *the boy car be different color*. However, there'a little difference between stemming and lemmatization:

- *Stemming* usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often 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**, which have been shown to be empirically effective, is **Porter's algorithm**. This algorithm consists of 5 phases of word reductions applied sequentially, and each phase consists of a set commands: the usual convention is to select the command that applies the longest suffix. Picture 3.5 shows the set of commands of the first phase.

Rule			Example		
SSES	→	SS	caresses	→	caress
IES	→	I	ponies	→	poni
SS	→	SS	caress	→	caress
S	→		cats	→	cat

Figure 7: Set of commands of the first phase

Many of the later rules use the concept of the *measure* of a word, i.e. they check the number of syllables to see whether a word is long enough that is reasonable to regard the matching portion of a rule as a suffix rather than as part of the stem of a word. For example, the rule *(m > 1) EMENT* would map *replacement* to *replac*, but not *cement* to *c* (since the prefix *c* is not long enough).
In general, stemmers use language-specific rules, but they require less knowledge than a lemmatizer, which needs a complete vocabulary and morphological analysis to correctly lemmatize word. The **advantages** of performing stemming is that it helps increasing the **recall** of the IR system, i.e. the amount of its results, but on the other hand it could harm the precision, i.e. the quality of the results, of other IR system. In general, this operation helps on reducing the size of the vocabulary, and it was shown that is very useful for languages with much more morphology, such as Spanish, German and Finnish.

4 Dictionaries and Tolerant Retrieval

In Chapters 2 and 3 we developed the ideas underlying inverted indexes for handling Boolean and proximity queries, so now we develop techniques that are robust to typographical errors in the query, as well as alternative spellings (tolerant retrieval).
In section 4.1 we develop data structures that help the search of terms in the vocabulary in an inverted index, in Section 4.2 we study the idea of *wildcard queries*, that are used when the user is uncertain of the spelling of a query term, while in Section 4.3 we focus on spelling errors.

4.1 Search structures for dictionaries

Given an **inverted index** and a **query**, our first task is to determine whether each query term exists in the vocabulary and if so, identify the pointer to the corresponding postings. This operation is usually implemented by exploiting the **vocabulary** data structure: in our case, the dictionary stores the term vocabulary (representing the *keys* of the dictionary), its document frequency and the pointer to the corresponding posting list. Usually, a vocabulary is implemented by using **hash tables** or **search trees**, and the choice between the two solutions depend on many factors, for example the number of keys that we have, whether this number is likely to change or not, the relative frequencies of each key etc..

4.1.1 Hash Tables

In the **hashing** technique, each vocabulary term is hashed into an integer (we assume that this operation minimizes the collisions), which is then used to access the corresponding posting list. The main **advantage** of this solution is that the vocabulary lookup is very fast, $O(1)$. However, there are several **disadvantages**: the hash table does not provide an easy way to find minor variants of the term, since different terms results in two different integers, and does not allow to search for a term using the prefix (tolerant retrieval). Finally, one important issue is that if the vocabulary keeps growing, a re-hashing operation of the whole vocabulary must be done, and this is usually very expensive.

4.1.2 Search Trees

Search trees are data structures that overcome some of the issues of the hash tables. The best-known search tree is the *binary tree*, in which each internal node has two children, and it can be considered as a binary test that affects the search for a term, in the sense that the outcome of the test determines one of the two sub-trees along which the search proceeds. Picture 4.1.2 gives an example of binary tree used for a dictionary.
The main issue about binary trees is that in order to implement efficient search, i.e. with complexity $O(logM)$, the tree must be **balanced**, so a big disadvantage of this structure is given by the overhead of maintaining the tree balanced when a new document is inserted. To overcome this problem, usually the **B-tree** data structure is used: the *B-tree* is a generalization of the binary tree in which every internal nodes has a number of children in the interval $[a, b]$, where a and b are appropriate natural numbers. Picture 4.1.2 shows a B-tree where $a = 2$ and $b = 4$. We notice that, because a range of child nodes is permitted,

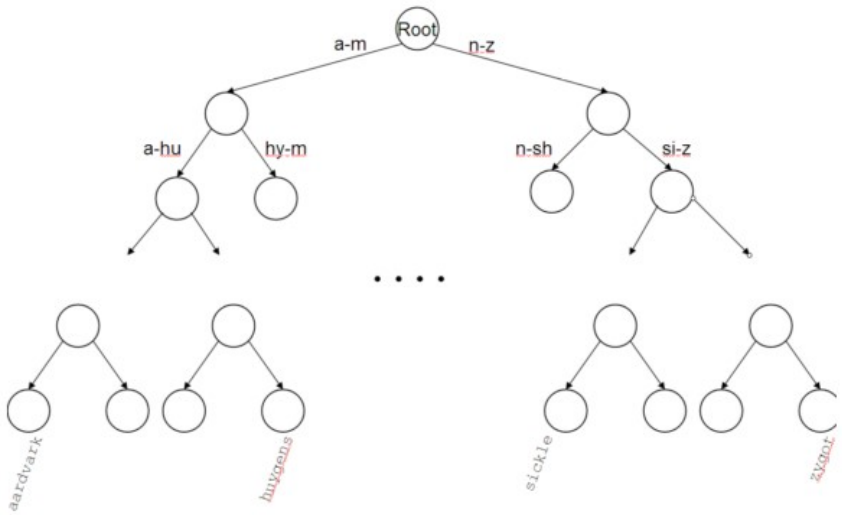


Figure 8: Binary tree for a dictionary

B-trees do not need re-balancing as frequently as binary trees, but on the other hand they may waste more space when the nodes are not entirely full.

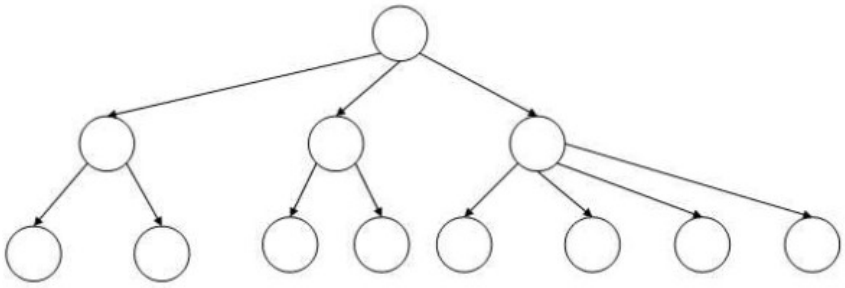


Figure 9: B-tree for a dictionary

In general, the simplest search tree to implement is the *binary tree*, but *B-trees* are the most used. We notice that trees, unlike hash tables, require a **standard ordering** of the characters, which is usually represented by the *lexicographic order*. The **advantage** of using the *search trees* is that they solve the prefix problem, i.e. they allow to search for a term by providing its prefix. The **disadvantages** are that they are usually slower than hash tables ($O(logM)$ vs $O(1)$), and they may require the re-balancing operation, which is quite expensive, although the B-trees mitigate this problem.

4.2 Wildcard queries

As we introduced before, the wildcard queries can be used when the user is uncertain of the spelling of a query, or when the user wants to retrieve documents containing all the variants of a term etc..
An example of wildcard query is *mon**, and it is called **trailing wildcard query**, since the symbol * occurs only once at the end of the string. This type of wildcard query is

quite simple to be answered by using a binary tree (or B-tree) lexicon, since we have to search for all the words s.t. mon $\leq$ word $\leq$ moo, by traversing the tree.
A slight generalization of this typology is the **leading wildcard query**, where the queries are in the form **mon*: these queries can be answered by maintaining an additional reverse B-tree that stores the terms backwards.
Finally, we can handle an even more general case: wildcard queries in which there is a single * in the middle of the query, such as *co*tion*. To solve these queries, we use the regular B-tree to retrieve the set of terms beginning with the prefix *co*, then we use the reverse B-tree to retrieve the set of terms ending with the suffix *tion*, and finally we intersect the two term sets using the standard inverted index. Obviously, this solution is quite expensive, and for this reason a possible improvement can be brought by transforming the wildcard queries so that the *'s occur always at the end of the terms. This approach gives rise to the **permuterm index**.

4.2.1 Permuterm index

The **permuterm index** is a form of inverted index in which all the possible permutations of a term all link to the original vocabulary term, as represented in Picture 4.2.1. Notice that the character $ is a special symbol that marks the end of a term: this character increases the vocabulary size, since we now store $n + 1$ terms, where n is the term size, rather than just 1. The set of rotated terms in the permuterm index is called *permuterm vocabulary*.

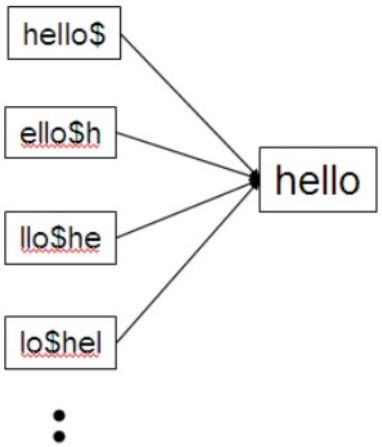


Figure 10: A portion of a permuterm index

As we introduced before, the goal of the *permuterm index* is to rotate the wildcard query so that the * symbol appears at the end of the string. For example, if the query is *h*e*, we insert the symbol $, so *h*e$*, and our goal is to consider the term *e$h**. Finally we can search the terms starting with *e$h** using a standard B-tree lookup to retrieve the matching documents.
Now the problem is how to handle wildcard queries containing multiple * symbols, such as *fi*mo*er*. In this case we first enumerate the terms in the dictionary that are in the permuterm index of *er$fi**. Not all such dictionary terms will have the string *mo* in the middle, so we filter these out by exhaustive enumeration, checking each candidate to see

if it contains *mo*. Finally, we run the surviving terms through the standard inverted index for document retrieval.
One **disadvantage** of the permuterm index is that its dictionary becomes like 4 to 10 times larger than the original one, so we now introduce another method to answer wildcard queries.

4.2.2 k-gram indexes for wildcard queries

A k-gram is a sequence of k characters, so *cas*, *ast* and *stl* are all 3-grams occurring in the term *castle*. We use a special character \$ to denote the beginning or end of a term, so the full set of 3-grams generated for castle is: *\$ca*, *cas*, *ast*, *stl*, *tle*, *le\$*.
In the *k-gram index*, the vocabulary contains all the k-grams that occur in any term in the vocabulary, and each postings list of a k-gram contains all the terms containing the corresponding k-gram. An example is provided in Picture 4.2.2.

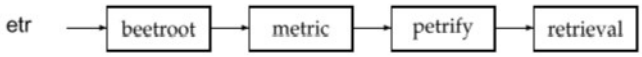


Figure 11: Postings list of a k-gram

Now, this particular index can be used for solving wildcard queries in the following way: suppose that the query is *re*ve*, we then run the Boolean query *\$re* AND *ve\$*, and each of the matching terms is then looked up in the standard inverted index to yield documents matching the query. However, this approach leads to the retrieval of some **false positives**. Consider for example the query *red**: following the approach described above, we would perform the Boolean query *\$re* AND *red*, but this would lead to match terms like *retired*, which is a false positives w.r.t. the original query. In this sense, a **post-processing** phase must be introduced, in order to filter the false positives against the original query: the surviving terms are then looked up in the standard inverted index.

4.2.3 Permuterm index vs k-gram index

As we said before, **permuterm index** is not space efficient: if n is the average length of a word, using the character \$ as a suffix, we add n rotations to the vocabulary; overall, we need $(n+1)*(n+1)$ chars for a term of n chars.
With 2-grams, considering the leading and trailing \$, we use $(n+1)*2$ chars, where $n+1$ is the number of bigrams for each term considering the symbol \$. Finally, with 3-grams we need $n*3$ chars.
In general, wildcard queries can result in very expensive query execution, and for this reason this capability is hidden behind an interface (say "Advanced Query"), which most users do not use.

4.3 Spelling correction

We now consider the problem of **spelling correction** in queries, and in particular we focus on two possible solutions: the first one is based on *edit distance*, while the second one on *k-gram overlap*.
In general, the two main uses of the spelling correction are:

- correcting the documents being indexed, and in particular this in important if the documents are retrieved using OCR (optical character recognition);
- correcting user queries to retrieve the correct answers (*Did you mean..?*).

4.3.1 Forms of spelling correction

We focus on two specific forms of spelling correction:

- **isolated-term correction**, in which we attempt to correct a single query term at a time. This type of correction does not catch typos resulting in correctly spelled words, e.g. *flew form Heathrow*, since each term n the query is correctly spelled in isolation;
- **context-sensitive correction**, in which the surrounding words are considered. In this case, the mispelling of the previous example is detected.

In general, given a lexicon and a char sequence Q, the goal of this technique is to return the words in the lexicon closest to Q, and, in particular, among alternative correct spellings for a mis-spelled query, choose the "nearest" one. If two correctly spelled queries are tied, then select the most common.

Isolated-term correction

We first analyze two techniques for addressing isolated-term correction: edit distance and k-gram overlap.

Given two strings S_1 and S_2, the *edit distance* (or *Levenshtein distance*) is the minimum number of edit operations to convert one to the other. The operations are typically character-level, and they're usually *insert*, *delete*, *replace* and *transposition*. For example, the *edit distance* between *cat* and *act* is 2, since two replacements are needed. A more accurate measure can be also obtained using the *weighted edit distance*, where different weights are used for different kind of edit operations, depending on the likelihood of letters that are replaced (e.g. m is more likely to be mis-typed as n than as q, so replacing m by n is a smaller edit disance than by q). In general, the (weighted) edit distance can be computed using dynamic programming with complexity $O(|S_1|\text{x}|S_2|)$, and in the case of the weighted version, a weight matrix must be provided as input to the algorithm.

The spelling correction problem however demands more than computing edit distance: given a set S of strings (corresponding to terms in the vocabulary) and a query string q, we seek the string(s) in V of least edit distance from q. The obvious way of doing this is to compute the edit distance from q to each string in V, before selecting the string(s) of minimum distance: however, this naive solution is quite expensive and extremely slow ($O(n^2)$). For this reason, some heuristics are adopted to cut the set of candidate dictionary terms:

- restrict the search to dictionary terms beginning with the same letter as the query string: in this case we hope that spelling errors do not occur in the first letter of the query;
- generate everything up to edit distance $k = 1, 2$ and then intersect these candidates with terms in the index lexicon.

Another way to limit the set of vocabulary terms for which we compute edit distances to the query term is by using the ***k*-gram index** that we introduced for wildcard queries. In particular, this index is used to retrieve vocabulary terms that have many k-grams in common with the query: for example, the bigram index in Picture 4.3.1 shows the postings for the three brigrams of the word *bord*. Suppose that we want to retrieve the vocabulary terms that contain at least two of these bigrams, then a single scan of the postings would result in retrieving *aboard*, *border*, and *boardroom*.

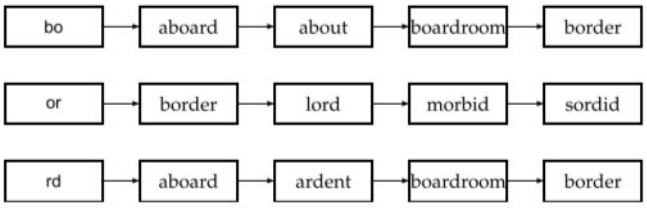


Figure 12: k-gram for spelling correction

However, we would like to consider a more precise measure of overlap, and one option could be the *Jaccard coefficient*, which is defined as $\frac{|A \cap B|}{|A \cup B|}$, where A and B are, respectively, the set of k-grams in the query and the set of k-grams in a vocabulary term. As the scan proceeds, we proceed from one vocabulary term t to the next, computing on the fly the *Jaccard coefficient* between q and t. If the coefficient exceeds a preset **threshold**, we add t to the output; if not, we move on to the next term in the postings.

Context-sensitive approach

As we underlined before, isolated-term correction would fail to correct typographical errors such as *flew form Heathrow*, where all three query terms are correctly spelled: our desired output would be that the search engine may offer the corrected query *few from Heathrow*. The simplest way to do this is to enumerate corrections of each of the three query terms (using the methods we described before) even though each query term is correctly spelled, then try substitutions of each correction in the phrase. For the example *flew form Heathrow*, we enumerate such phrases as *fled form Heathrow* and *flew fore Heathrow*. For each such substitute phrase, the search engine runs the query and determines the number of matching results: finally, the engine suggests the alternative that has lots of hits. Obviously, this enumeration can be expensive if we find many corrections of the individual terms, since we could encounter a large number of combinations of alternatives. A possible heuristic that can be used for reduce the search space is to retain only the most frequent combinations in the collection or in the query logs, which contain previous queries by the users. For example, we would retain *flew from* as an alternative to try and extend to a three-term corrected query, but perhaps not *fled fore* or *flea form*.

5 Index construction

In this chapter we focus on how to construct an inverted index, i.e. the process of *index construction*.

5.1 Hardware basics

When building an information retrieval (IR) system, many decisions are based on the characteristics of the computer hardware on which the system runs. The following list provides the hardware basics that are needed to motivate IR systems.

- Access to data in **memory** is much **faster** than access to data on **disk**: consequently, we want to keep as much data as possible in memory, especially those data that we need to access frequently. We call the technique of keeping frequently used disk data in main memory **caching**;
- When doing a disk read or write, it takes a while for the disk head to move to the part of the disk where the data are located, and **no data** are being **transferred during this seek time**. To maximize data transfer rates, chunks of data that will be read together should therefore be stored contiguously on disk;
- Operating systems generally **read** and **write** entire **blocks**. Thus, reading a single byte from disk can take as much time as reading the entire block;
- **Data transfers** from disk to memory are handled by the **system bus**, not by the processor. This means that the processor is available to process data during disk I/O. We can exploit this fact to speed up data transfers by storing compressed data on disk. Assuming an efficient decompression algorithm, the total time of reading and then decompressing compressed data is usually less than reading uncompressed data;
- Servers used in IR systems typically have several **gigabytes** (GB) of **main memory**, sometimes tens of GB. Available disk space is several orders of magnitude larger;
- **Fault tolerance** is very **expensive**, so it is much cheaper to use many regular machines rather than one fault-tolerant machine.

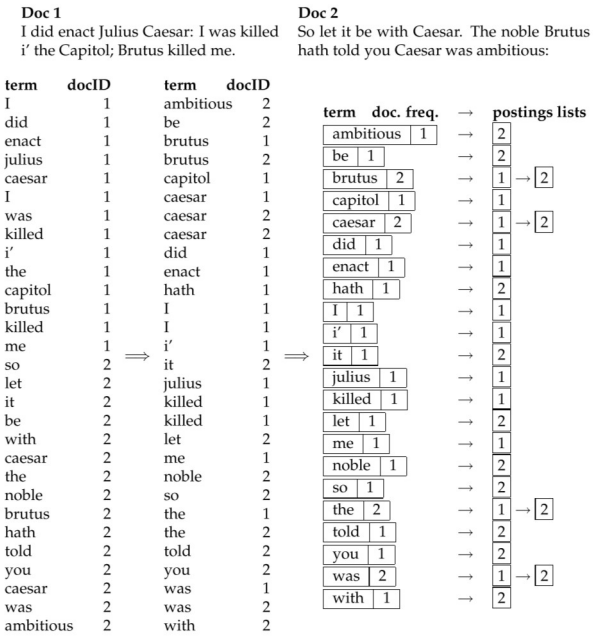
In particular, Picture 5.1 shows the hardware assumptions for this chapter.

Symbol	Statistic	Value
s	average seek time	$5\text{ ms} = 5 \times 10^{-3}$ s
b	transfer time per byte	$0.02\ \mu\text{s} = 2 \times 10^{-8}$ s
	processor's clock rate	$10^9\ \text{s}^{-1}$
p	lowlevel operation	
	(e.g., compare & swap a word)	$0.01\ \mu\text{s} = 10^{-8}$ s
	size of main memory	several GB
	size of disk space	1 TB or more

Figure 13: Hardware assumptions for this chapter

5.2 Blocked sort-based indexing (BSBI)

The basic steps in constructing an inverted index are represented in Picture 5.2: the first phase is passing through the collection to assemble all the term-docId pairs, then the pairs are sorted and the docIds are organized for each term into a postings list, computing statistics like term and document frequencies. For small collections, all of this can be done in memory, but in this Chapter we analyze some methods for large collections, that require secondary memory storage.



To make index construction more efficient, we represent terms as **termIDs**, where each termID is a unique serial number, and we work with the *Reuters-RCV1* collection as our model collection in this Chapter, whose statistics are showed in Picture 6.3.

Symbol	Statistic	Value
N	documents	800,000
L_{ave}	avg. # tokens per document	200
M	terms	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
T	tokens	100,000,000

Figure 14: Main statistics of *Reuters-RCV1* collection

As represented in Picture 5.2, after all the documents have been parsed, the inverted file is sorted by terms. *Reuters-RCV1* contains 100 million tokens, so collecting all termID-docID pairs of the collection using 4 bytes for each pair would require 0.8 GB of storage: clearly, it is not possible to sort them in memory, so we need an *external sorting algorithm.* The main requirement of this algorithm is to minimize the number of random disk seeks during sorting.
One solution is the *blocked sort-based indexing* algorithm or *BSBI* of Picture 5.2. This algorithm:

1. Segments the collection into parts of equal size;
2. Sorts the termID-docID pairs of each part in memory;
3. Stores the intermediate results on disk;
4. Merges all intermediate results into the final index.

```
BSBINDEXCONSTRUCTION()
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4     block ← PARSENEXTBLOCK()
5     BSBI-INVERT(block)
6     WRITEBLOCKTODISK(block, f_n)
7  MERGEBLOCKS(f_1, ..., f_n; f_merged)
```

Figure 15: *BSBI* algorithm

Some notes:

- The algorithm parses documents into termID–docID pairs and accumulates the pairs in memory until a block of a fixed size is full (function PARSENEXTBLOCK in the code). We choose the block size to fit comfortably into memory to permit a fast in-memory sort;
- Then, the termID-docID pairs are sorted;
- Next, we collect all termID–docID pairs with the same termID into a postings list, where a posting is simply a docID. The result, an inverted index for the block we have just read, is then written to disk;
- Finally, the algorithm simultaneously merges the blocks into one large merged index.

If we apply this approach to *Reuters-RCV1* collection, and if we assume that we can fit 10 million termID-docID pairs into memory, we end up with 10 blocks, each of which is an inverted index of one part of the collection. Finally, the blocks are merged into the final index. An example with two blocks is shown in Picture 5.2.

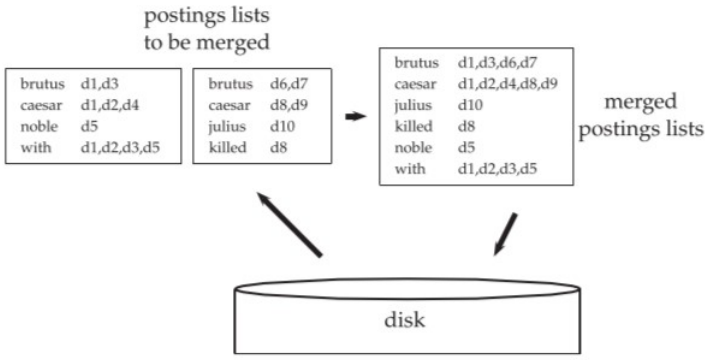


Figure 16: Example of the *BSBI* algorithm

One crucial operation of this algorithm is clearly the final **merge phase**: one possible way to implement it could be by performing a **binary merge**, but using this strategy we would

need to re-write increasingly larger blocks to disk, resulting in a very expensive operation. For this reason, a **multi-way merge** is preferred, and it consists of opening all the block files simultaneously and maintain small read buffers for the ten blocks we are reading and a write buffer for the final merged index we are writing. By using this technique, we do not perform too many disk seeks, so the algorithm has better performances.
The **complexity** of *BSBI* algorithm is $\theta(TlogT)$, and it derives by the sorting phase (assuming that we use QuickSort), where T is the number of termID-docID pairs. However, the actual indexing time is dominated by the parsing operation (PARSENEXTBLOCK) and the final merging operation (MERGEBLOCKS).
Some **issues** about this algorithm are:

- The assumption to use this algorithm is that we're able to keep all the dictionary in memory, and this is not always the case;
- Again, the dictionary is needed to produce the mapping from *term* to *termID*: we could also work with term-docID postings, but then the intermediate files would become too large, resulting in a scalable but very slow algorithm.

5.3 Single-pass in-memory indexing (SPIMI)

The *SPIMI* algorithm is a more scalable alternative to *BSBI* that is based on two key ideas:

- It uses terms instead of termID, so no mapping between term and termID is needed across the blocks;
- This algorithm does not sort the postings, so the docIDs are generated and assigned sequentially to the documents.

Using these two ideas, we can generate a complete inverted index for each block, and then merge them together into one big index. The *SPIMI* algorithm is shown in Picture 5.3: notice that the part of the algorithm that parses the documents and turns them into term-docID pairs (tokens) is omitted. The algorithm receives in input the token stream, and run until the entire collection has been processed.

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Figure 17: *SPIMI* algorithm

Some notes:

- *Line 4*: each token is processed one at a time;
- *Line 5 and 6*: when a term occurs for the first time, it is added to the dictionary and a new empty postings list is created (*Line 6*);
- *Line 7*: if the term already belong to the dictionary, the corresponding postings list is retrieved;
- *Line 8 and 9*: if the postings list is full, the size is doubled (some memory is wasted);
- *Line 10*: the docID of the token is directly inserted into the postings list. Contrary to *BSBI*, where the termID-docID pairs were sorted, here each postings list is dynamic, i.e. its size is adjusted as it grows (*Line 8 and 9*). This approach has two advantages: on the one hand, it is faster since no sorting is required, on the other it saves a lot of memory because the termID of postings need not to be stored;
- *Line 11*: when the memory is exhausted, the terms are sorted in order to write the postings lists in lexicographic order, which helps the final merging step;
- *Line 12*: the index of the block (dictionary and postings list) is written to disk;
- Finally, the blocks are merged (not shown in the picture)

Another important feature of this algorithm is **compression**: both the postings and the dictionary terms can be stored compactly on disk if we employ compression. Compression increases the efficiency of the algorithm further because we can process even larger blocks, and because the individual blocks require less space on disk. We will exploit the compression in the following Chapter.
The **time complexity** of *SPIMI* algorithm is $\theta(T)$, because all operations are at most linear in the size of the collection.

5.4 Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine: Web search engines use **distributed indexing** algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines, either according to term or according to document.
The distributed index construction method we describe in this section is an application of **MapReduce**, a general architecture for distributed computing, which is designed for large computer clusters (tightly coupled computers that work together closely). Each cluster is composed of cheap commodity machines, or *nodes*, that are able to solve large computing problems, so one the main requirement of *distributed indexing* is to divide the work up into chunks that can be easily assigned and re-assigned to idle nodes in the pool. The *master node* directs the process of assigning jobs to individual nodes, and it is assumed to be "safe", i.e. to be fault tolerant.
The steps of the MapReduce are shown in Picture 5.4.

1. The input data (a collection of Web pages), is split into *n splits*, where the size of each split is chosen so that the load is is distributed evenly and efficiently among the nodes. Usually, 16 or 64 MB are good choices;
2. The splits are assigned to the nodes by the *master node* on a ongoing basis: as a machine finishes processing one split, it is assigned to the next one;

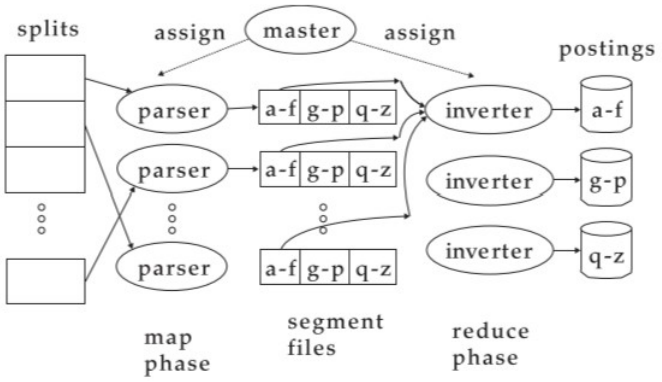


Figure 18: MapReduce

3. The *map phase* consists of mapping splits to term-docID pairs, and since it is the same parsing in *BSBI* and *SPIMI*, we call the machines that execute the map phase **parsers**. Each parser writes its output to local intermediate files, called *segment files*;
4. In the reduce phase, we would like to have all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by **partitioning the keys** into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Picture 5.4, the term partitions are according to first letter, and $j = 3$. Finally, each parser writes the corresponding segment file, one for each term partition, and each term partition thus corresponds to r segment files, where r is the number of parsers;
5. The *reduce phase* is performed by the **inverters**, which collect all values (docIDs) for a given key (termID) into one list. The master assigns each term partition to a different inverter and, as in the case of parsers, reassigns term partitions in case of failing or slow inverters. Each term partition (corresponding to r segment files, one on each parser) is processed by one inverter;
6. Finally, the list of values is sorted for each key and written to the final sorted postings list ("postings" in the Picture).

It is important to notice that parsers and inverters are not separate sets of machines, i.e. the same machine can be a parser in the map phase and an inverter in the reduce phase. In general, MapReduce offers a robust and conceptually simple framework for implementing index construction in a distributed environment. By providing a semiautomatic method for splitting index construction into smaller tasks, it can scale to almost arbitrarily large collections.
Picture 5.4 shows an example of usage of MapReduce

5.5 Dynamic indexing

Thus far, we have assumed that the document collection is static, but most collections are modified frequently with documents being added, deleted, and updated. This means that new terms need to be added to the dictionary, and postings lists need to be updated for existing terms.

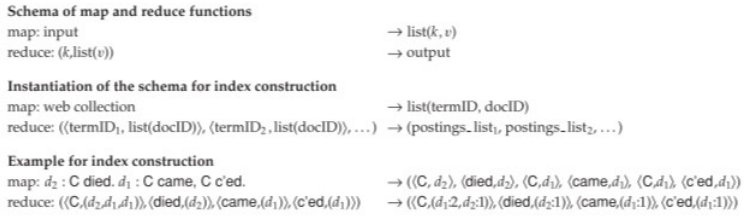


Figure 19: Example of usage of MapReduce

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 a delay in making new documents searchable is acceptable 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, one solution is to maintain two indexes: a *large main index* and a *small auxiliary index* that stores new documents. The auxiliary index is kept in memory, and the searches are run across both indexes and results merged. Deletions are stored in an invalidation bit vector, and we can then filter out deleted documents before returning the search result. Documents are updated by deleting and reinserting them. The main **issues** about this strategy may regard the merging phase: however, the merge of the auxiliary index into the main index could be efficient if we keep a separate file for each postings list. In this case, the merge operation would be the same as a simple append, but at the same time we would need to store a lot of files, which is inefficient for the OS. The simplest alternative, which is the one we assume for our purposes, is to store the index as one large file, i.e. a concatenation of postings lists; however, in reality a compromise between the two options is chosen.
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.

5.6 Other types of indexes

- positional indexes;
- character n-gram indexes: as the text is parsed, the n-grams are enumerate. Then, for each n-gram we consider the pointers to all dictionary terms that contain it, i.e. the postings.

6 Index Compression

In this chapter, we employ a number of compression techniques for dictionary and inverted index that are essential for efficient IR systems.
There are two more subtle **benefits** of compression:

1. Increased use of **caching**: with compression, we can fit a lot more information into main memory. Instead of having to expend a disk seek when processing a query, we instead access its postings list in memory and decompress it. As we will see below, there are simple and efficient decompression methods, so that the penalty of having to decompress the postings list is small. As a result, we are able to **decrease the response time** of the IR system substantially;
2. **Faster transfer of data from disk to memory**. Efficient decompression algorithms run so fast on modern hardware that the total time of transferring a compressed chunk of data from disk and then decompressing it is usually less than transferring the same chunk of data in uncompressed form.

However, if the goal of compression is to conserve disk space and to decrease the response time of the IR system, then the **speed of the compression algorithm** itself is crucial: the compression algorithms we discuss in this chapter are highly efficient and can therefore serve all the purposes of index compression.
NOTE: in this Chapter, we define a *posting* as a docID in a postings list. For example, (6; 20, 45, 100), where 6 is the termID of the list's term, contains three postings.

6.1 Statistical properties of terms in IR

As in Chapter 5, we use *Reuters-RCV1* as our model collection. We recall it main statistics in Picture 6.3, while in Picture 6.1 we provide some terms and postings statistics of the collection.

Symbol	Statistic	Value
N	documents	800,000
L_{ave}	avg. # tokens per document	200
M	terms	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
T	tokens	100,000,000

Figure 20: Main statistics of *Reuters-RCV1* collection

	(distinct) terms			nonpositional postings			tokens (= number of positi entries in postings)		
	number	Δ%	T%	number	Δ%	T%	number	Δ%	T%
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	−2	−2	100,680,242	−8	−8	179,158,204	−9	−9
case folding	391,523	−17	−19	96,969,056	−3	−12	179,158,204	−0	−9
30 stop words	391,493	−0	−19	83,390,443	−14	−24	121,857,825	−31	−38
150 stop words	391,373	−0	−19	67,001,847	−30	−39	94,516,599	−47	−52
stemming	322,383	−17	−33	63,812,300	−4	−42	94,516,599	−0	−52

Figure 21: Terms and postings statistics of *Reuters-RCV1* collection

Some notes about these statistics follow:

- the "Δ%" represents the reduction in size form the previous line;
- the "T%" is the cumulative reduction from unfiltered;
- in general, the statistics show that **pre-processing** affects the size of the **dictionary** and the number of **non-positional postings** greatly: as we can see, stemming and case folding (i.e. considering "Rome" and "rome" as the same term) reduce the number of distinct terms by 17%;
- the treatment of stopwords is also important: eliminating the 30 and 150 most common words from indexing cuts 31% and 47% of the total number of entries in postings, and 14% and 30% of the non-positonal postings;
- however, we notice that the usage of case folding and stemming does not change the number of entries in postings: this happens beacuse the positions of the terms are still kept in the postings;
- the same thing happens with the stopwords w.r.t. the number of distinct terms: we're only cutting off 30/150 words over a total of half million terms!

The compression techniques we describe in the remainder of this chapter are **lossless**, that is, **all information is preserved**. Better compression ratios can be achieved with **lossy** compression, which discards some information. Case folding, stemming, and stop word elimination are forms of lossy compression. In general, lossy compression makes sense when the "lost" information is unlikely ever to be used by the search system. For example, web search is characterized by a large number of documents, short queries, and users who only look at the first few pages of results. As a consequence, we can discard postings of documents that would only be used for hits far down the list. Thus, there are retrieval scenarios where lossy methods can be used for compression without any reduction in effectiveness.
Before introducing techniques for compressing the dictionary, we want to estimate the **number of distinct terms** (vocabulary) M in a collection. First of all, we notice that we cannot assume an upper bound; in practice, we will discover that the vocabulary size keep growing with the collection size.

6.1.1 Heap's Law: estimating the number of terms

The **Heap's Law** estimates the vocabulary size as a function of the collection size, in particular:

$$M = kT^b$$

, where:

- T is the number of tokens in the collection;
- $30 \leq k \leq 100$;
- $b \sim 0.5$.

The motivation of this law resides in the **linear relationship** between the collection size and the vocabulary size in a **log-log plot**, as represented in Picture 6.1.1.

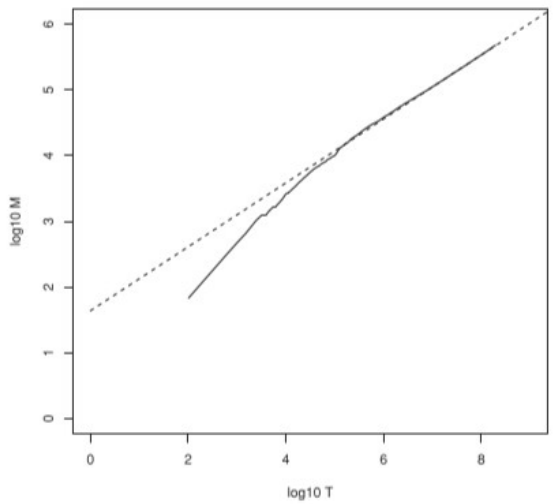


Figure 22: Heap's Law: linear relationship between M and T in a log-log scale

As we can see, for RCV1 collection, the dashed line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ represents the best least-square fit: in this case, $b = 0.49$ and $k = 44$. For example, for the first 1,000,020 tokens, Heap's Law predicts $44 * 1,000,020^{0.49} = 30,323$ terms, while the dashed line predicts 38,365 terms, so it is a good approximation.
Notice that the parameter k is quite variable because vocabulary growth depends a lot on the nature of the collection and how it is processed. Case-folding and stemming reduce the growth rate of the vocabulary, whereas including numbers and spelling errors increase it.
In general, regardless of the values of the parameters for a particular collection, Heaps' law suggests that:

1. The dictionary size continues to increase with more documents in the collection, rather than a maximum vocabulary size being reached;
2. The size of the dictionary is quite large for large collections

These two hypotheses have been empirically shown to be true of large text collections, so dictionary compression is a crucial operation for an effective IR system.
Exercise: Compute the vocabulary size for this scenario. There are 3,000 different terms (M_1) in the first 10,000 tokens (T_1), and 30,000 different terms (M_2) in the first 1,000,000 tokens(T_2). Assume a search engine indexes a total of 20,000,000,000 ($= 2^{10}$) pages, each containing 200 tokens on average. What is the size of the vocabulary of the indexed collection as predicted by the Heap's Law?

6.1.2 Zipf's Law

While Heap's Law gives the vocabulary size, **Zipf's Law** helps us understanding how the terms are distributed across documents, so it takes into account the relative frequencies of terms, as we know that in natural languages, there are a few very frequent terms and very many rare terms.
If we assume that the terms are ranked from the most frequent to the leas frequent, we can consider:

- t_i to be the term at rank i in the collection;
- cf_i to be the collection frequency, i.e. the number of occurrences of the term t_i, i.e. the term at rank i, in the collection;
- df_i to be the document frequency, i.e. the total number of documents that contain the term t_i in the corpus;
- $tf_{i,d}$ to be the term frequency, i.e. the total number of occurrences of the term in a document

Zipf's Law states that if $t1$ is the most common term in the collection, $t2$ is the next most common, and so on, then the collection frequency cf_i of the i-th most common term is proportional to $1/i$, i.e.:

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

In this sense, if the most frequent term occurs cf_i times, then the second most common will occur half of the occurrences, and the third one third of the occurrences and so on.. Another way of stating Zipf's Law is:

$$cf_i = ci^{-1}$$

or

$$\log cf_i = \log c - \log i$$

, c is a constant. As we can see, there is a linear relationship between $\log cf_i$ and $\log i$, with slope -1, as represented in Picture 7.2.

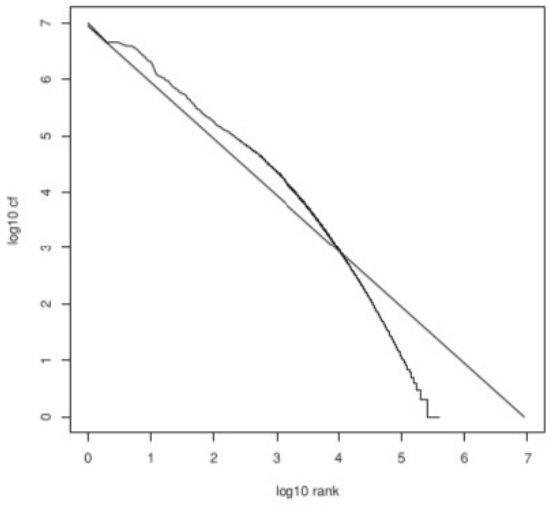


Figure 23: Zipf's Law: linear relationship between cf_i and i in a log-log scale

Moreover, if we consider the relationship $cf_i = ci^{-1}$, we can recognize a **Power Law relationship**: the Power Law states that $p(i) \approx ki^{-\alpha}$, i.e. that roughly 80% of the total popularity comes from 20% of the terms. Notice that the quantity α represents the slope in the log-log plot of p and i.

6.2 Dictionary compression

This section presents a series of dictionary data structures that achieve increasingly higher compression ratios. The main reason for compressing the dictionary is to fit it in memory, in order to reduce the number of disk accesses when processing a query. Indeed, some embedded/small devices may have very little memory.

6.2.1 Dictionary as a string

The simplest data structure for the dictionary is to sort the vocabulary lexicographically and store it in an **array of fixed-width entries** as shown in Picture 6.2.1.

	term	document frequency	pointer to postings list
	a	656,265	⟶
	aachen	65	⟶
	…	…	…
	zulu	221	⟶
space needed:	20 bytes	4 bytes	4 bytes

Figure 24: Dictionary as an array of fixed-width entries

If we allocate 20 bytes for the term itself, 4 bytes for its document frequency and 4 bytes for its pointer to its postings list, then for Reuters-RCV1 we need 400,000 x 28 = 11.2 MB for storing the dictionary. Notice that the lexicographical order allows us to perform a binary search for looking up terms. However, using fixed-width entries for terms is clearly wasteful, since the average length of a term in English is about eight characters, so on average we are wasting twelve characters. Also, we have no way of storing terms with more than twenty characters.
We can overcome these issues by storing the dictionary terms as one **long string of characters**, as shown in Picture 6.2.1.

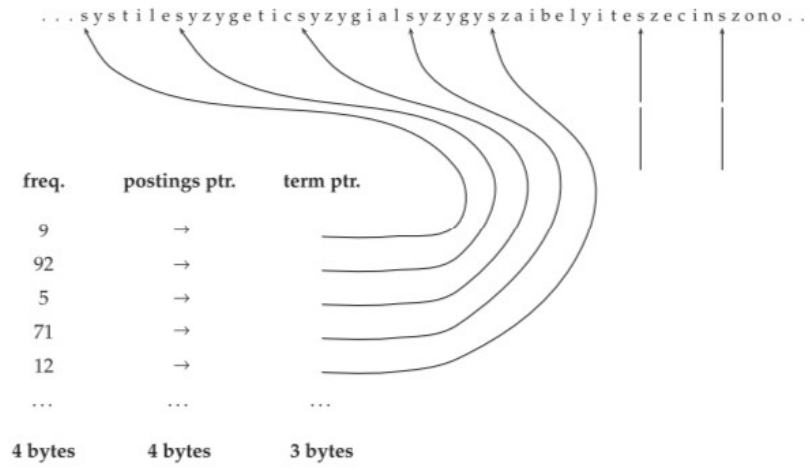


Figure 25: Dictionary as a string

In this case, for each term we use 4 bytes for storing the term frequency, 4 bytes for storing the pointer to the corresponding postings list, 3 bytes for storing the term pointer

and 8 an average of 8 bytes for storing the term in the string. In this way, we only use an average of 11 bytes per term, versus the 20 bytes we used before, resulting in a 60% storage save. In this case, we have a total of 19 bytes, so 400,000 x 19 = 7.6 MB, against the 11.2 MB used in the fixed-width entries. Notice that, as before, the look for a term can be done by performing a binary search operation, and that in this case the end of the string is not stored, since the term pointer of the next term is used as indicator of the end of the current term.

6.2.2 Blocked storage

We can further 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, as represented in Picture 6.2.2.

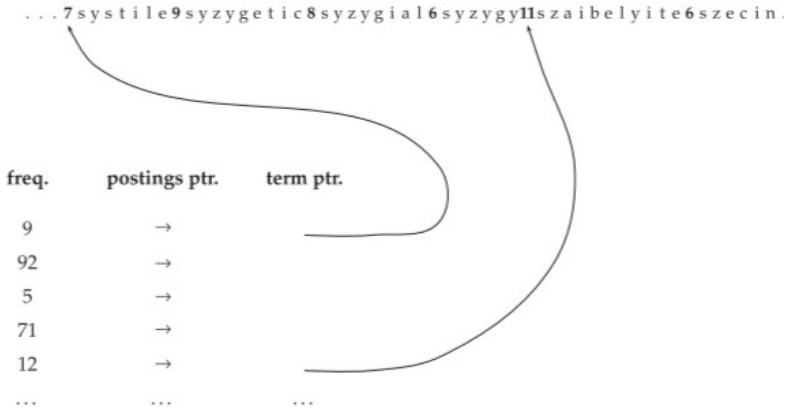
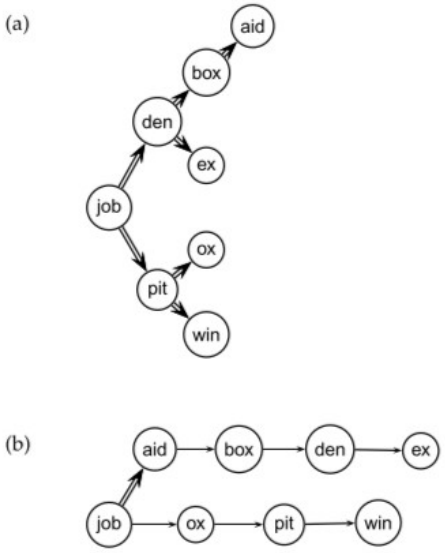


Figure 26: Blocked storage

In this case we store the length of the term in the string as an additional byte at the beginning of the term: we thus eliminate $k1$ term pointers, but need an additional k bytes for storing the length of each term. If $k = 4$ we save $(k - 1)$ x 3 = 9 bytes for term pointers, but we need an additional $k = 4$ bytes for term lengths. In the case of Reuters-RCV1, we reduce 5 bytes per four-term block, resulting in a total of 0.5 MB, bringing us down to 7.1 MB.
By increasing the block size k, we get better compression. However, there is a **tradeoff** between **compression** and the **speed of term lookup**. Suppose we have an eight-term dictionary, as represented in Picture 6.2.2.
In the Picture, steps in binary search are showed as double lines, and steps in list search as simple lines. We search for terms in the uncompressed dictionary by binary search (a). In the compressed dictionary, we first locate the term's block by binary search and then its position within the list by linear search through the block (b). Searching the uncompressed dictionary in (a) takes on average $(0 + 1 + 2 + 3 + 2 + 1 + 2 + 2)/8$, i.e. more or less 1.6 steps, assuming each term is equally likely to come up in a query. For example, finding the two terms, *aid* and *box*, takes three and two steps, respectively. With blocks of size k = 4 in (b), we need $(0+1+2+3+4+1+2+3)/8 = 2$ steps on average, i.e. 25% more. By increasing k, we can get the size of the compressed dictionary arbitrarily



close to the minimum of 400,000 × (4 + 4 + 1 + 8) = 6.8 MB, but term lookup becomes prohibitively slow for large values of k.
However, by now we did not exploit the redundancy in the dictionary, in particular the fact that consecutive entries in an alphabetically sorted list share common prefixes. The exploitation of this property leads to *front coding*.

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

⇓

...further compressed with front coding.
8automat∗a**1**⋄e**2**⋄ic**3**⋄ion

Figure 27: Front coding: first implementation

In this case, a **common prefix** is identified for a sub-sequence of the term list, and then referred to with a special character. As we can see, the compression with front coding includes the common prefix, and each of the following characters for each term in the uncompressed sub-sequence. Together with the characters, the extra length beyond the suffix is stored, leading to a much smaller representation. In the case of Reuters-RCV1, this strategy saves another 1.2 MB. However, this is not the only possible implementation of the idea of front coding: Picture 6.2.2 show another implementation in which each term exploits a prefix of the previous one.
However, even with the best compression scheme, it may not be feasible to store the entire dictionary in main memory for very large text collections and for hardware with limited memory. Picture 6.2.2 provides a summary of the dictionary compression achieved using the data structures we described by now.

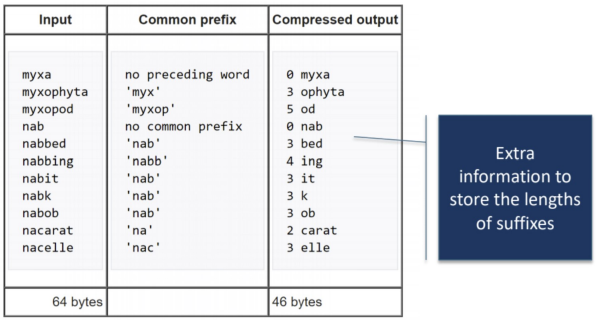


Figure 28: Front coding: second implementation

Dictionary compression for Reuters-RCV1.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
∼, with blocking, $k = 4$	7.1
∼, with blocking & front coding	5.9

Figure 29: Dictionary compression: summary

6.3 Postings compression

We recall the statistics of Reuters-RCV1 collection: as we can see, the number of postings, i.e. the number of docIDs in a postings list, is 100,000,000. DocIDs aere $\log_2 800,000 \approx 20$ bits long, so the size of the collection is 800,00 x 200 x 6 bytes = 960 MB, while the size of the uncompressed postings file is 100,000,000 x 20/8 = 150 MB.

Symbol	Statistic	Value
N	documents	800,000
L_{ave}	avg. # tokens per document	200
M	terms	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
T	tokens	100,000,000

Figure 30: Main statistics of *Reuters-RCV1* collection

To use a more efficient representation of the postings file, i.e. a representation that uses less than 20 bits per document, we can exploit the fact that postings for frequent terms are close together. As represented in Picture 6.3, a very **frequent term** will have postings whose **gaps** are very **close**, and for which much less than 20 bits can be used, so in this case the idea is to store the gaps instead of the docIDs; on the other hand, gaps for a **rare term** have the **same order of magnitude as the docIDs**, so they need 20 bits. In this sense, we need a *variable encoding* method that uses fewer bits for smaller gaps. To encode small numbers in less space than large numbers, we look at two types of methods: **bytewise compression** and **bitwise compression**. As the names suggest, these methods attempt to encode gaps with the minimum number of bytes and bits, respectively.

	encoding	postings list							
the	docIDs	...		283042		283043		283044		283045
	gaps				1		1		1	
computer	docIDs	...		283047		283154		283159		283202
	gaps				107		5		43	
arachnocentric	docIDs	252000		500100						
	gaps	252000	248100							

Figure 31: Encoding gaps

6.3.1 Variable byte encoding

Variable byte (VB) encoding uses an integral number of bytes to encode a gap G:

- the last 7 bits of a byte are *payload* and encode part of the gap;
- the first bit of the byte is a *continuation bit*: it is set to 1 for the last byte of the encoded gap and to 0 otherwise.

To decode a variable byte code, we read a sequence of bytes with continuation bit 0 terminated by a byte with continuation bit 1. We then extract and concatenate the 7-bit parts. An example of VB-encoded postings list is provided in Picture 6.3.1.

docIDs	824		829	215406
gaps			5	214577
VB code	00000110	10111000	10000101	00001101 00001100 10110001

Figure 32: Example of VB encoding

A key property of VB encoding is that VB-encoded postings are **uniquely prefix-decodable**, i.e. no whole code word in the system is a prefix of any other code word. This means that no special marker is needed to separate codes.
With VB compression, the size of the compressed index for Reuters-RCV1 is 116 MB as we verified in an experiment: this is a more than 50% reduction of the size of the uncompressed index.
The idea of VB encoding can also be applied to larger or smaller units than bytes: 32-bit words, 16-bit words, and 4-bit words or nibbles. **Larger words** further **decrease** the **amount of bit manipulation** necessary at the cost of less effective (or no) compression. In general, **bytes** offer a **good compromise** between **compression ratio** and **speed of decompression**, and for most IR systems **variable byte codes** offer an excellent **tradeoff** between **time** and **space**, and they are also simple to implement. However, if the disk space is a scarce resource, we can achieve better compression ratios by using bit-level encodings, in particular two closely related encodings: γ codes, which we will turn to next, and δ codes.

6.3.2 Elias-γ encoding

While VB codes use a variable number of bytes for encoding a gap, bit-level codes use a finer grained bit level. The simplest bit-level code is the **unary code**, which represents n as a string of n 1's followed by a 0, as showed in the first column of Picture 6.3.2.
Clearly, the unary code is not efficient, but it will become handy later.

number	unary code	length	offset	γ code
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	1111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110,0000000001

Figure 33: Examples of unary and γ codes

In general, how efficient can a code be in principle? 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. Our goal is to get as close to this lower bound as possible.
A method that is within a factor of optimal is γ encoding. γ codes implement **variable-length encoding** by **splitting** the representation of a gap G into a pair of **length** and **offset**:

- the *offset* is G in binary, but with the leading 1 removed. For example, for 13 (binary 1101), the *offset* is 101;
- the *length* encodes the length of *offset* in unary code. For 13, the length of *offset* is 3 bits, which is 1110 in unary.

The γ code is then the **concatenation** of *length* and *offset* (in the example, the γ code of 13 is therefore 1110101). Picture 6.3.2 provides some other examples. The **encoding** of a γ code works as follows:

1. Read the unary code up to the 0 that terminates it, for example the four bits 1110 when decoding 1110101. Now we know that the offset is long 3 bits;
2. The offset is read correctly, and the 1 is preprended, leading to $1101 = 13$.

In general:

- the length of the *offset* is $\lfloor \log_2 G \rfloor$ bits;
- the length of the *length* is $\lfloor \log_2 G \rfloor + 1$

so the length of the entire code is $2\lfloor \log_2 G \rfloor + 1$ bits.
All the γ codes have an odd number of bits, and they are within a factor of 2 of the optimal encoding $\log_2 G$ for any distribution, which makes the γ codes to be **universal**. In addition to universality, γ codes are **prefix free**, i.e. no γ code can be the prefix of another, meaning that there is always a unique decoding of a sequence of γ codes. The other property is that γ codes are **parameter free**, which simplifies the implementation and the decompression of this code. However, a **disadvantage** of this encoding is that

it is relatively inefficient for large numbers, since the unary code is used for encoding the offset.
The result of running γ compression on Reuters-RCV1 is that the size of the compressed index is 101 MB, which is less than VB encoding.
Picture 6.3.2 shows a summary of the results of the compression techniques we exploited by now.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9
collection (text, xml markup etc)	3600.0
collection (text)	960.0
term incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ encoded	101.0

Figure 34: Summary of compression techniques including postings

6.3.3 Elias-δ encoding

A slight variation of the γ codes are the δ, in which again the gap G is represented into a pair of *length* and *offset*:

- the *offset* is G in binary, but with the leading 1 removed. For example, for 13 (binary 1101), the *offset* is 101. It is the same as γ codes;
- the *length* encodes the length of *offset* in γ code.

In this case, δ encoding uses $2\lfloor \log_2 \lfloor \log_2 G \rfloor \rfloor + 1$ bits.

6.3.4 Golomb-Rice encoding

Golomb-Rice encoding works as follows: we choose a quantity $M = 2^j$, where j is a parameter of the method, and we encode an integer n by splitting n into two components, the quotient $q(n)$ and the remainder $r(n)$, where:

- $q(n) = \lfloor \frac{n-1}{M} \rfloor$;
- $r(n) = (n-1) mod M$, i.e. $0 \leq r(n) \leq M - 1$.

Then, n is encoded by writing $q(n) + 1$ in unary, followed by $r(n)$ as a binary number represented on $\log_2 M$ $(= j)$ bits. In general, j is chosen so that 2^j is centered around the mean of the elements to be represented.

6.3.5 Interpolative coding

A completely different approach is taken by the binary interpolative coding, which directly encodes strictly monotone sequences, and in particular this method recursively splits the sequence in two halves: the element in the middle is encoded, then the recursion is applied in the two halves.
We consider an example: suppose we have the following inverted list: <7; 3, 8, 9, 11, 12, 13, 17> in a collection of $N = 20$ documents. Suppose that the value of the second pointer is somehow known before the first must be coded. In the example, if it is known that the second pointer is to document 8, then the first pointer is restricted to some document number in the range 1 to 7 inclusive. A simple assignment of binary codewords then suffices to represent this first document pointer in three bits. Suppose now that the fourth as well as the second document number is known. The fourth document pointer is to document 11, so the third pointer is constrained to the range 9 to 10. Again, a simple code (in this case just one bit long) can be used to represent the third pointer. The brevity of this codeword is a direct consequence of the fact that there is a cluster and that both the upper and lower bounding pointers are also in the cluster. As an even more extreme example, if both the fourth and the sixth pointers are known (to documents 11 and 13, respectively), then the fifth document pointer can be represented using a codeword zero bits long, since it must be to document 12.
This representation is based upon the supposition that the second, fourth, and sixth pointers are known. To represent them, a list <3; 8,11,13> must have been previously coded. The same technique can be used for this list too. If the second pointer (to document 11) is known, then the first pointer (to document 8) takes at most four bits. Indeed, since there must be a pointer to the left and a pointer to the right of this document, the range can be further narrowed to 2 .. . 9 inclusive, and a three-bit code can be used. By a similar argument, the third pointer must lie between 13 = 12 + 1 and 19 = 20 — 1 inclusive, and 3 - $\lfloor \log 7 \rfloor$ bits suffice.
It is possible to show that at its worst the interpolative code is only slightly inefficient compared with a Golomb code, and, because contiguous sets of values can be coded in less than one bit each, at its best it can be dramatically better. Moreover, in practice the interpolative code usually results in very good compression. Indeed, the only real drawback of the method is its complexity of implementation—encoding and each decoding make use of a stack of pending values, and the encoding and decoding loops become rather more detailed than for the simpler Golomb and γ codes.

6.3.6 Information theory

Shannon showed that if we know that a random variable X takes values with a discrete probability distribution $P = p_1, .., p_n$, then the minimum expected message length we can achieve with any binary prefix-free code is between $H(P)$ and $H(P) + 1$, where $H(P)$ is the **entropy**, which is defined as:

$$H(P) = \sum_i p_i \log \frac{1}{p_i}$$

For example, Elias-γ encoding is optimal when the integers to be compressed conform the following probability distribution: $P(n) = \frac{1}{2n^2}$, i.e. the probability of small numbers is

very high, or, in other terms, the probability of the numbers is inversely proportional to the magnitudes of the gaps.
Huffman encoding is known to be **optimal** w.r.t. the Shannon theory: in particular, this encoding is based on the entropy, and its goal is to represent more common symbols using fewer bits than less common ones. This encoding works for every possible probability distribution of integers, but its drawback is that it requires traversing a tree or performing slow table lookups.

7 Scoring, term weighting and the vector space model

7.1 Ranked retrieval

Thus far, we've only considered Boolean queries, and in this sense, either a document satisfies a query or not. This methodology is good for expert users with precise understanding of their needs and the collection, but it is not good for most of the users, who are not capable of writing Boolean queries. Another problem of Boolean queries is the so-called *feast or famine* problem, i.e. Boolean queries often result in either too few or too many results.
For these reasons, rather then retrieving a set of documents satisfying a query expression, in **ranked retrieval** the system reorders the top documents of a collection w.r.t. a given query. In this case, rather than a query language consisting in operators and expressions, the system deals with **free text queries**, i.e. a collection of words in human language. Notice that in the case of *ranked retrieval*, the *feast or famine* issue is no longer a problem, since we just provide the top-k results.

7.2 Parametric and zone indexes

We have thus far viewed a document as a sequence of terms. In fact, most documents have **additional structure**. Digital documents generally encode, in machine-recognizable form, certain **metadata** associated with each document, such as its author(s), title and date of publication etc.. This metadata would generally include *fields* such as the date of creation and the format of the document, as well the author and possibly the title of the document. The possible values of a field should be thought of as finite (for instance, the set of all dates of authorship).
Consider queries of the form "find documents authored by William Shakespeare in 1601, containing the phrase *alas poor Yorick*". Query processing then consists as usual of postings intersections, except that we may merge **postings** from standard inverted as well as **parametric indexes**. There is **one** parametric index **for each field**, and it allows us to select only the documents matching a date specified in the query (in the example query above, the year 1601 is one such field value). The search engine may support querying ranges on such ordered values; to this end, a structure like a *B-tree* may be used for the field's dictionary.
Zones are similar to *fields*, except the **contents** of a zone can be **arbitrary free text**, while a field may take on a relatively small set of values. For instance, document titles and abstracts are generally treated as zones. We may build a separate **inverted index for each zone of a document**, to support queries such as "find documents with *merchant* in the title and *william* in the author list and the phrase *gentle rain in the body*". This has the effect of building an index that looks like Picture 7.2.
We can reduce the size of the dictionary by encoding the zone in which a term occurs in the postings. In Picture 7.2 for instance, we show how occurrences of *william* in the title and author zones of various documents are encoded. Such an encoding is useful when the size of the dictionary is a concern (because we require the dictionary to fit in main memory).
But there is another important reason why the encoding of Figure 7.2 is useful: the efficient computation of scores using a technique we will call weighted zone

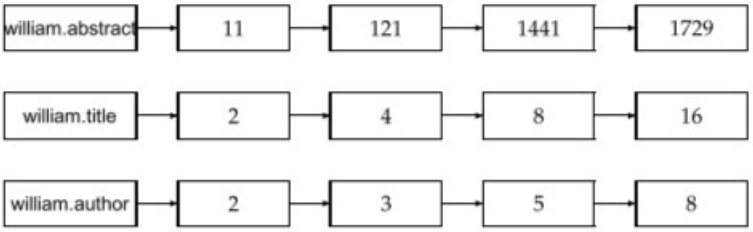


Figure 35: Basic zone index

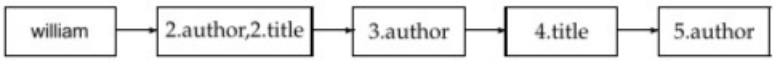


Figure 36: Zone index in which the zone is encoded in the postings rather than the dictionary

7.3 Term frequency and weighting

Obviously, the core of the *ranked retrieval* approach relies on the way we provide the **score** to each of the document, and in particular on the way we measure the importance of a query term w.r.t. the documents in the collection.
One approach could be represented by considering the **Jaccard** coefficient between the vectors representing the query and the document. For example, if the query is "*ides of march*", and the first document is "*caesar died in march*" and the second document is "*the long march*", then the Jaccard coefficients are:

$$J(q, d_1) = \frac{1}{3 + 4 - 1} = \frac{1}{6}$$

and

$$J(q, d_2) = \frac{1}{3 + 3 - 1} = \frac{1}{5}$$

In this way, the result is that the second document is more relevant than the first one, but this is clearly not the case: clearly the Jaccard coefficient does not consider the *term frequency*, i.e. the number of times a term occurs in a query
Another approach is to assign the weight to be equal to the **number of occurrences** of term t in document d. This weighting scheme is referred to as **term frequency** and is denoted $tf(t, d)$. Notice that for a document d, the set of weights determined by the *term frequency* does not depend on the order of the terms in the document, since we only focus on the information about the number of occurrences of each term. This model is often called as *bag of words* models.

7.3.1 Log-frequency and inverse document frequency

Clearly, this raw definition of term frequency is not what we want from the IR system to be implemented: a document d_1 with 10 occurrences of a term is more relevant than a document d_2 with only 1 occurrence of a term, but d_1 is not 10 times more relevant than

d_2. In other words, we do not want the relevance of a term to be directly proportional to its frequency.
One possible solution could be of considering the **log-frequency**, i.e.:

$$\log tf(t,d) = \begin{cases} 1 + \log_{10} tf(t,d) & \text{if } tf(t,d) > 0 \\ 0 & \text{otherwise} \end{cases}$$

In this way, the score of a query-document pair is defined as:

$$\text{score}_{\text{LOG}}(q,d) = \sum_{t \in q \cap d} (1 + \log tf(t,d))$$

Another possible solution exploits the **document frequency** concept: the *document frequency* of a term t represents the number of documents in the collection that contain the term t. The idea when considering *document frequency* is that rare terms are more informative than frequent terms, so the documents that contain these rare terms are more likely to be relevant than documents that do not. In this sense, the *document frequency* is an **inverse measure** of the **informativeness** of a term t. Notice that, for each term t, $df(t) \leq N$, where N is the number of documents in the collection. The way the *document frequency* is used for measuring the relevance of a term t is represented by the **inverse document frequency**, which is defined as:

$$idf(t) = \log_{10}(\frac{N}{df(t)})$$

Some notes:

- the $\log_{10}$ is used as a smoothing factor for the effect of idf;
- very frequent terms have small values of idf;
- rare terms have large values of idf.

In this case, the score of a query-document pair is defined as:

$$\text{score}_{\text{IDF}}(q,d) = \sum_{t \in q \cap d} idf(t) = \sum_{t \in q \cap d} \log_{10}(\frac{N}{df(t)})$$

Notice that idf alone can produce a ranking of the matching documents only for queries with at least two terms, if all the docs do not contain all query terms.

7.3.2 tf-idf

We now combine the definitions of term frequency and inverse document frequency, to produce a composite weight for each term in each document, the *tf-idf*:

$$tf\text{-}idf(t,d) = tf(t,d) * idf(t)$$

Notice that:

- the weight is higher for a term that appears many times within a small number of documents;

- the weight is lower for a term that occurs fewer times in a document, or that occurs in many documents. The lowest weight is obtained when the term occurs in all the documents.

At this point, we may view each document as a vector with one component corresponding to each term in the dictionary, together with a weight for each component that is given by the *tf-idf*; in this case, we can compute the score as:

$$\text{score}_{\text{TF-IDF}}(q, d) = \sum_{t \in q \cap d} \textit{tf-idf}(t, d)$$

Example: if we consider the table of term frequencies and idf values of Picture 7.3.2 ,the score of Doc 1 is given by:

	Doc1	Doc2	Doc3	df_t	idf_t
car	27	4	24	18,165	1.65
auto	3	33	0	6723	2.08
insurance	0	33	29	19,241	1.62
best	14	0	17	25,235	1.5

- $car = 27 * 1.65 = 44.55$;
- $insurance = 0$;
- $auto = 3 * 2.08 = 6.24$;
- $best = 14 * 1.5 = 21$

7.4 Vector space model for scoring

Thus far we developed the notion of a document vector that captures the relative importance of the terms in a document. The representation of a set of documents as vectors in a common vector space is known as the **vector space model** and is fundamental to a host of information retrieval operations ranging from scoring documents on a query, document classification and document clustering. The peculiarities of the documents as vectors are:

- They're very **high-dimensional vectors**, i.e. they belong to an high-dimension vector space;
- They are very **sparse vectors**, i.e. most of the entries are equal to 0.

We recall that the goal of *ranked retrieval* is to rank the documents of the collection according to their proximity to a given query. In this sense, we must provide both the vector representation for queries and the definition of proximity/similarity between vectors.

7.4.1 Vector similarity

We first deal with the concept of vector similarity, and we start our reasoning by considering the **euclidean distance** between two vectors. Given a document d and a query q, the euclidean distance is defined as:

$$dist_{\text{EUCLIDEAN}}(d, q) = \sqrt{\sum_{i=1}^{n}(d_i - q_i)^2}$$

, where n is the dimensionality of the vectors. In general, the Euclidean distance is not a good measure of similarity, since it assumes large values for vectors of different lengths, i.e. the Euclidean distance of two close vectors can be very high due to their length, as represented in Picture 7.4.1. As we can see, the value of the Euclidean distance of q and d_2 is very high, even if the distribution of their terms is quite similar.

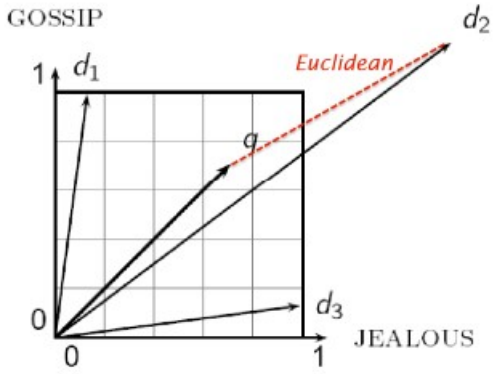


Figure 37: Example of Euclidean distance

Another possible measure of the similarity of two vectors could rely on the **angle** between the vectors. In this sense, we could rank documents in increasing order of **cosine similarity** w.r.t. the query:

$$sim_{\text{COSINE}}(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum_i q_i d_i}{\sqrt{\sum_i q_i^2}\sqrt{\sum_i d_i^2}} = \cos\theta$$

, where:

- $\vec{q} \cdot \vec{d}$ represents the *dot product* between q and d;
- $|\vec{q}|$ and $|\vec{d}|$ are the *Euclidean lengths* of $\vec{q}$ and $\vec{d}$, and they're used to normalize the vectors $\vec{q}$ and $\vec{d}$ to unit vectors $\vec{q}/|\vec{q}|$ and $\vec{d}/|\vec{d}|$. In this way, long and short documents have now comparable weights.

Notice that the definition of sim_{COSINE} is equal to the cosine of the angle θ between the two vectors, as showed in Picture 7.4.1.
What is the usage of cosine similarity? Given a document d (potentially one of the d_i in the collection), consider searching for the documents in the collection most similar to d: we reduce the problem of finding the document(s) most similar to d to that of finding the

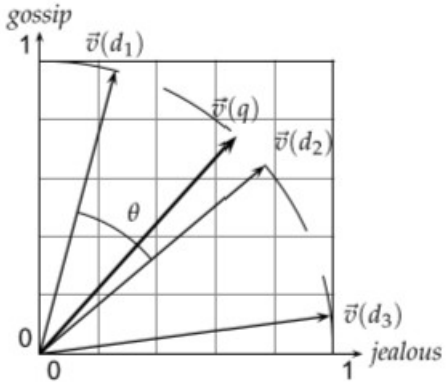


Figure 38: Example of cosine similarity

d_i with the highest dot products $\vec{d} \cdot \vec{d_i}$. We could do this by computing the dot products between $\vec{d}$ and each of $\vec{d_1}, .., \vec{d_n}$, then picking off the highest resulting values.
Example: Picture 7.4.1 represents the term frequencies for 4 terms in 3 different novels: in this sense, each of the novels is represented as a unit vector in three dimensions, resulting in the term frequencies weights of Picture 7.4.1.

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6

term	SaS	PaP	WH
affection	0.996	0.993	0.847
jealous	0.087	0.120	0.466
gossip	0.017	0	0.254

In this case, $sim_{\text{COSINE}}(\vec{SaS}, \vec{PaP}) = 0.996 * 0.993 + 0.087 * 0.120 + 0.017 * 0 = 0.999$
Viewing a collection of N documents as a collection of vectors leads to a natural view of a collection as a term-document matrix: this is an $M \times N$ matrix whose rows represent the M terms of the N columns, each of which corresponds to a document. As always, the terms being indexed could be stemmed before indexing;

7.4.2 Queries as vectors

After discussing the concept of vectors similarity, we can now notice that we can also view a query as a vector. In this way, we can assign to each document d a score equal to the dot product:

$$\text{score} = \vec{q} \cdot \vec{d}$$

In other words, we can consider a query as a very short document, so we can use the cosine similarity between the query vector and a documents vector as a measure of the score of the document for that query:

$$\text{score} = sim_{\text{COSINE}(\vec{q},\vec{d})} = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}||\vec{d}|}$$

Finally, the resulting scores can be used to select the top-scoring documents for a query. However, computing the cosine similarities between the query vector and each document

vector in the collection, sorting the resulting scores and selecting the top-K documents can be expensive, since a single similarity computation can entail a dot product in tens of thousands of dimensions, demanding many arithmetic operations.

7.4.3 Computing vector scores

In a typical setting we have a collection of documents each represented by a vector, a free 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. Picture 7.4.3 shows the algorithm for computing vector space scores.

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

Figure 39: Algorithm for computing vector space scores

Some notes:

- the array *Length* stores the lengths for each of the N documents;
- the array *Scores* stores the scores for each of the documents: when the scores are finally computed in Step 9, the top-k documents are return in Step 10;
- the loop in Step 3 iterates over all the query terms t, while in Step 5 we calculate the weight in the query vector for term t;
- Steps 6-8 update the score of each document by adding in the contribution from term t. This process of adding in contributions one query term at a time is sometimes known as *term-at-a-time* scoring or accumulation, and the N elements of the array *Scores* are therefore known as *accumulators*. For this purpose, it would appear necessary to store, with each postings entry, the weight $wf_{t,d}$ of term t in document d (we have thus far used either tf or $tf - idf$ for this weight, but leave open the possibility of other functions to be developed). In fact this is wasteful, since storing this weight may require a floating point number. Two ideas help alleviate this space problem. First, if we are using inverse document frequency, we need not precompute idf_t; it suffices to store N/df_t at the head of the postings for t. Second, we store the term frequency $tf_{t,d}$ for each postings entry.

7.5 Variant tf-idf functions

For assigning a weight for each term in each document, a number of alternatives to tf and $tf - idf$ have been considered. We discuss some of the principal ones here.

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

Figure 40: SMART Notation for tf-idf variants

Picture 7.5 lists some of the principal weighting schemes in use for each of $\vec{d}$ and $\vec{q}$, together with a mnemonic for representing a specific combination of weights; this system of mnemonics is sometimes called SMART notation. The mnemonic for representing a combination of weights takes the form *ddd.qqq* where the first triplet gives the *term weighting* of the *document* vector, while the second triplet gives the *weighting* in the *query* vector. The first letter in each triplet specifies the term frequency component of the weighting, the second the document frequency component, and the third the form of normalization used.
It is quite common to apply different normalization functions to $\vec{q}$ and $\vec{d}$. For example, a very standard weighting scheme is **lnc.ltc**, where the document vector has log-weighted term frequency, no idf (for both effectiveness and efficiency reasons), and cosine normalization, while the query vector uses log-weighted term frequency, idf weighting, and cosine normalization.

8 Computing scores in a complete search system

In the previous Chapter we developed the theory underlying term weighting in documents for the purposes of scoring, leading up to vector space models and the basic cosine scoring algorithm. In this Chapter we begin in Section 8.1 with heuristics for speeding up this computation; many of these heuristics achieve their speed at the risk of not finding quite the top K documents matching the query. Some of these heuristics generalize beyond cosine scoring. With Section 8.1 in place, we have essentially all the components needed for a complete search engine. We therefore take a step back from cosine scoring, to the more general problem of computing scores in a search engine. In Section 8.2 we outline a complete search engine, including indexes and structures to support not only cosine scoring but also more general ranking factors such as query term proximity.

8.1 Efficient scoring and ranking

Before analyzing an efficient version of the cosine ranking algorithm, we focus on the difference between two possible query evaluation techniques: **TAAT**, or **term at a time** and **DAAT**, or **document at a time**.
In the *TAAT* technique, the scores for all the documents are computed concurrently, one query term at a time, whereas in the *DAAT* technique we compute the total score for each doc, considering all the query terms, before proceeding to the following one. In general, *TAAT* requires a larger number of documents to be scored, and thus **more memory** to accumuate their scores before sorting. On the other hand, *DAAT* technique requires **less memory**, as less documents scores are stored overall. For example, the algorithm in Picture 7.4.3 implements a *TAAT* strategy.
In this case we are solving the following problem: find the k documents in the collection "nearest" to the query, i.e. retrieve the top-k largest query-doc cosines. A cosine ranking is defined **efficient** if the cosine values are computed efficiently and if the k largest cosines are chosen efficiently. Since we're dealing with sparse query/documents vector, the problem of finding the k-nearest neighbors to a query vector is a solvable problem (more difficult for high-dimensional spaces and dense vectors).

8.1.1 TAAT

In this case the idea for an efficient ranking is to pick the k-nearest neighbors without totally ordering all the scores. In other words, let J be the number of docs with non-zero cosines, our goal is to retrieve the k best of these J scores, rather that N.
An alternative is to use a MaxHeap to select the top-k. A MaxHeap is a priority queue with the following properties:

- it is a complete binary tree, i.e. if a node is stored at index k, then its left/right children are stored at index $2k+1$ and $2k+2$;
- node's values are $\geq$ than the values of the children.

Given a complete array of *accumulators*, $2J$ comparisons are required to build a MaxHeap; then, each of the k "winners" is retrieved in $\log J$ steps. Thus, the total cost is $2J+k\log J$.

Once we established a method for fast ranking, we now focus on some strategies for fast scoring. The main idea is to assume that each query term occurs only once, i.e. $tf = 1$: in this way, each query term has not any weight, so in the ranking phase the query vector is not needed to be normalized. In this sense, *idf* is only used for computing $wf_{t,d}$ and not for $wf_{q,d}$. The motivations for this choice relies on the following inequality, where $\vec{q}$ represents the normalized vector of the query q, while $\vec{Q}$ is the un-normalized one, i.e. all the nonzero components are equal to 1:

$$\vec{Q} \cdot \vec{d_1} > \vec{Q} \cdot \vec{d_2} \iff \vec{q} \cdot \vec{d_1} > \vec{q} \cdot \vec{d_2}$$

These simplifications lead to the algorithm in Picture 8.1.1.

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

Figure 41: A faster algorithm for computing vector space scores

Some notes:

- Line 7: since $wf_{t,q} = 1$, the multiply-add of algorithm 7.4.3 becomes an addition here;
- Line 11: the top-k documents are retrieved using the MaxHeap strategy described above.

8.1.2 DAAT

In this case the idea is to use a binary MinHeap of k elements. It takes $O(\log k)$ operatios to insert a new *accumulator*, then k winners are read off in order. In this sense, the goal is to keep the top-k documents seen so far, by using a binary MinHeap and performing the following operations when processing a new document d' with score s':

1. Get the current minimum h_m of heap (cost is $O(1)$);
2. If $s' < h_m$, skip to the next document;
3. if $s' \geq h_m$, perform an heap deletion of the root (cost if $O(\log k)$), and add $s' : d'$ (cost is $O(\log k)$)

Clearly, in the worst case scenario we have to add to the min heap all the new documents. An example of the functioning of this approach is given in Picture 8.1.2: as we can see, documents 6, 10 and 11 are not added to the MinHeap.

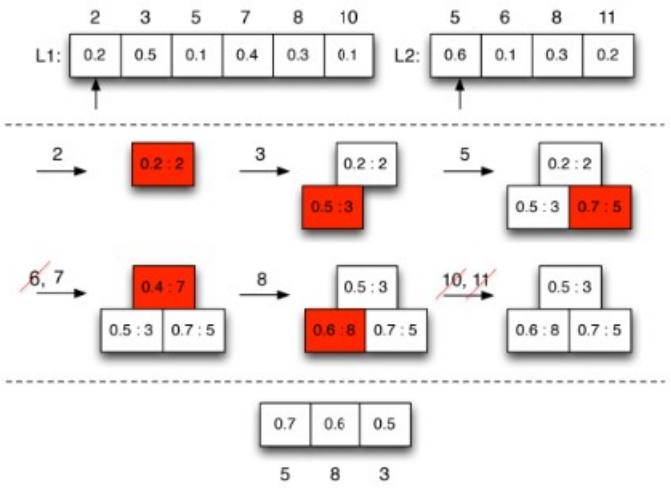


Figure 42: Example of MinHeap of 3 elements

8.1.3 WAND

Another possible method for query evaluation in the *DAAT* approach is provided by the **WAND scoring**. Notice that the normal DAAT still computes all the J scores (i.e. all the non-zero scores), while the WAND approach is the following:

- we maintain a running threshold score, e.g. the k-th highest score computed so far, which corresponds to the least score in the MinHeap;
- we prune away all the docs whose cosine scores are guaranteed to be below the current threshold;
- we compute the exact cosine scores for only the un-pruned documents, that are much less than the original J scores.

In this sense, the WAND scoring technique is based on a two-level evaluation:

1. The first level evaluation is used to quickly establish whether a document merits to be fully evaluated, i.e. whether it has a chance of being a top-k document. Notice that this phase cannot lead to *false negatives*, i.e. all the potential matches have to be kept, while on the other hand it could lead to few *false positives*;
2. The second level consists on the full evaluation of the candidates retrieved in the first phase to enter the MinHeap.

In this approach the **postings are ordered** by docID, and we assume there exists a **special iterator** on the postings with the following form: "go to the first docID $\geq$ X", i.e. an iterator that can skip blocks of documents. In a typical state, we have a **"finger"** at some docID in the postings of each query term, and each finger only moves to the right, i.e. to larger docIDs, by using the special iterator if needed. Clearly, the invariant is that all the docIDs lower than any finger have already been processed, meaning that either these docIDs have been pruned away or their cosine scores have been computed (in this case we can have either true positives or false positives).

Moreover, we assume that each query term t is associated with an **upper bound** UB_t on its maximal contribution to any document score in the postings list:

$$UB_t \geq \alpha_t \max(w(t, d_1), w(t, d_2), ..)$$

, where $w(t, d_i)$ is the weight of a term, given by *tf-idf*.
In addition, we set a **threshold** τ, which defines the min score of the MinHeap, and whenever a documents d succeeds in entering the MinHeap, the threshold grows.
A crucial operation in WAND is the **pivoting**, i.e. the searching of the **pivot** element: the pivot element is the first term for which $\sum_t UB_t > \tau$. If we have a situation like the one of Picture 8.1.3, the pivot is represented by the term *in*.

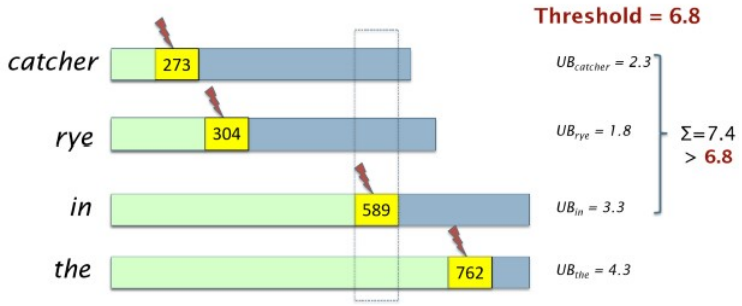


Figure 43: Pivot

Once the pivot is determined, all the fingers are moved to the position of the pivot, using the iterator we introduced before: in this way, we perform the pruning operation of the documents that have no hope of being in the top-k.

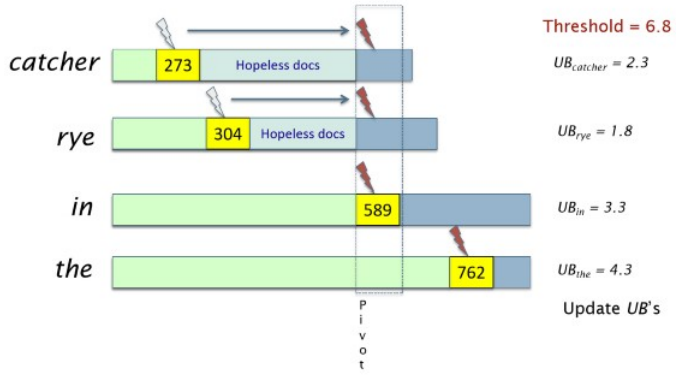


Figure 44: Pruning phase

Finally, if the docID of the pivot is present in all the three postings, we compute its full cosine score, otherwise we move the fingers and we consider another pivot.
The results of the WAND algorithm show that it leads to a 90+% reduction in the score computation, and that it provides the best gains on longer queries. Notice that the functioning of the algorithm is not specific for the cosine ranking, but we only need the scoring metric to be additive by term. Finally, we notice that TAAT, DAAT, WAND and other variants of this algorithm provide a **safe ranking**, i.e. we're guaranteed that the top-k documents are associated with the k highest scored documents.

8.1.4 Block-Max

Block-Max represents a variant of the WAND algorithm. While in WAND algorithm the maximum "impact score" for each term/postings list is stored, the Block-Max algorithm stores the maximum "impact score" for each **block** of a compressed inverted list in uncompressed form. This behaviour enables the skip of large parts of the lists, without decompressing, so it speedups WAND. In particular, the algorithm splits the inverted lists into blocks of 64 or 128 docIDs, s.t. each block can be uncompressed separately; moreover, for each compressed block an additional table is created, which stores for each block:

- the maximum (or minimum) docID;
- the maximum impact value for each block;
- the block size.

As in WAND, the **special iterator** is implemented to avoid decompressing blocks that are not relevant.
From Picture 8.1.4 we notice that for each block we maintain an upper bound approximation of the impact scores of each list, which corresponds to the maximum of the values, as we said before. Moreover, inside each block there may be many 0 values, that can be retrieved by decompressing the block. Finally, we notice that the blocks are not uniform in size.

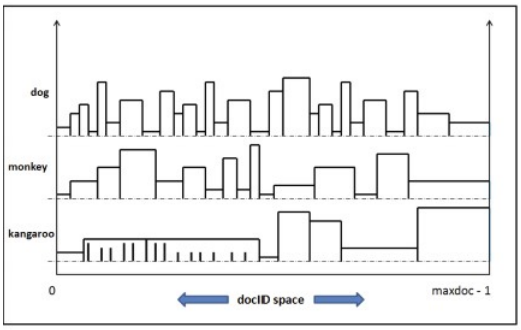


Figure 45: Blocks in Block Max algorithm

The *Block Max* algorithm is implemented as follows:

1. Sort the list from top to bottom, according the the docIDs of the current postings in the lists, as in WAND;
2. Find the pivot ID as in WAND;
3. Use the *global maximum scores* (UB_t) to determine a candidate pivot, as in WAND;
4. Use the *block maximum scores* (which is $< UB_t$) to check if the candidate pivot is a real pivot.
    - If it is a real pivot, move the current pointers of the previous query terms to the first postings in their lists, with docIDs $\geq$ the pivot ID, like in WAND;

- Otherwise, move them to ID = $\min(\text{ID}_i)$, where the various ID_i are the maximum docIDs in the current blocks.

Picture 8.1.4 shows the average query processing time of different algorithms w.r.t. different number of terms in the query, while Picture 8.1.4 show the average number of processed documents by the algorithms. In both cases we notice that Block Max algorithm obtains the best performances.

TREC 2006						
	avg	2	3	4	5	> 5
exhaustive OR	225.7	60	159.2	261.4	376	646.4
WAND	77.6	23.0	42.5	89.9	141.2	251.6
SC	64.3	12.2	36.7	75.6	117.2	226.3
BMW	27.9	4.07	11.52	33.6	54.5	114.2
exhaustive AND	11.4	10.3	10.8	14.0	15.4	15.2
TREC 2005						
	avg	2	3	4	5	> 5
exhaustive OR	369.3	62.1	238.9	515.2	778.3	1501.4
WAND	64.4	23.5	43.7	73.4	98.9	265.9
SC	63.5	14.2	37.5	119.7	172.9	316.9
BMW	21.2	3.5	12.7	25.2	39	104
exhaustive AND	6.86	6.4	7.3	9.2	4.7	5.9

Block Max WAND

Block Max WAND

Table 1: Average query processing time in ms for different numbers of query terms, using different algorithms on the TREC 2006 and 2005 query logs. Exhaustive OR, WAND, SC, and BMW are for disjunctive queries, while Exhaustive AND is for conjunctive queries.

Figure 46: Comparison of different retrieval algorithms

	evaluated docs	decoded ints	dpm	spm
exhaustive OR	3815676	9356032	15.9M	–
WAND	178391	6274432	1.18M	–
SC	–	965248	–	–
BMW	21921	2642752	0.42M	0.76M
exhaustive AND	20026	1939584	0.25M	–

Table 2: The average number of evaluated docIDs, decoded integers, deep pointer movements (dpm), and shallow pointer movements (spm) for different methods on the TREC 2006 query log. Exhaustive OR, WAND, SC, and BMW are for disjunctive queries, while Exhaustive AND is for conjunctive queries.

Figure 47: Comparison of different retrieval algorithms

8.1.5 Inexact top-k retrieval

Thus far, we have focused on retrieving precisely the k highest-scoring documents for a query. We now consider schemes by which we produce k **documents that are likely to be among the k highest scoring documents** for a query. In doing so, we hope to **dramatically lower** the **cost of computing** the k documents we output, without

materially altering the user's perceived relevance of the top k results. Consequently, in most applications it suffices to retrieve k documents whose scores are very close to those of the k best.
In general, the inexact top-k retrieval is not a bad thing from the user's point of view, since the ranking function is only a proxy for the user's happiness, so the corresponding result may not be exact w.r.t. the user's will.
As an example, the WAND algorithm can be made inexact on two possible ways:

- the first is a **safe** aggressive pruning, and it consists of considering smaller values for k: in this way, the threshold τ becomes higher, resulting in more pruning of low-scoring documents;
- the second is an **unsafe** aggressive pruning, and it consists of considering larger values of the threshold τ, for example by considering a new threshold $\tau' = F \cdot \tau$, where $F \geq 1$ being a tunable parameter: in this way, a document deserves to be fully evaluated only if the final score is likely much greater than the current threshold τ.

However, we now consider a series of ideas designed to **eliminate a large number of documents** without computing their cosine scores. The heuristics have the following two-step scheme:

1. Find a set A of documents that are contenders, where $k < |A| << N$. A does not necessarily contain the k top-scoring documents for the query, but is likely to have many documents with scores near those of the top k;
2. Return the k top-scoring documents in A.

From the descriptions of these ideas it will be clear that many of them require parameters to be tuned to the collection and application at hand, but it is important to notice that this approach can be used also for other (non-cosine) scoring functions.
Index elimination
For a multi-term query q, it is clear that we compute the cosine only for the documents containing at least one of the query terms: we can take this observation further considering the following heuristics:

1. We only consider documents containing terms whose *idf* exceeds a preset threshold, i.e. we only traverse the postings for terms whose *idf* exceeds the threshold. Using this technique, the set of documents for which we compute cosines is greatly reduced. One way of viewing this heuristic: low *idf* terms are treated as stop words and do not contribute to scoring. For instance, on the query *catcher in the rye*, we only traverse the postings for *catcher* and *rye*;
2. We only consider documents that contain many (and as a special case, all) 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. However, this strategy is quite easy to implement

Champion lists

The idea of **champion lists** is to **precompute**, for each term t in the dictionary, the set of the r **documents** with the **highest weights** for t; the value of r is chosen in advance. For $tfi - df$ weighting, these would be the r documents with the highest tf values for term t. We call this set of r documents the *champion list* for term t.

Now, 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 now restrict cosine computation to only the documents in A.

A critical parameter in this scheme is the **value** r, which is highly **application dependent**: intuitively, r should be large compared with k, but it is set at the time of index construction, whereas k is application dependent and may not be available until the query is received. In general, there is no reason to have the same value of r for all terms in the dictionary; it could for instance be set to be **higher for rarer terms**.

Static quality scores and ordering

We now further develop the idea of champion lists, in the somewhat more general setting of *static quality scores*. In many search engines, we have available a **measure of quality** $g(d)$ for each document d that is **query-independent** and thus **static**, where $0 \leq g(d) \leq 1$. For instance, this quantity may refer to the number of citations of a paper, or the PageRank score etc..

The **net score** for a document d is some combination of **quality/authority**, defined by $g(d)$, together with the query-dependent score representing the **relevance**, for example the cosine score:

$$\text{net-score}(q, d) = g(d) + \text{cosine}(q, d) = g(d) + \frac{\vec{q} \cdot \vec{d}}{|\vec{q}||\vec{d}|}$$

Notice that $g(d)$ and $\text{cosine}(q, d)$ could be weighted in some way.

The **first idea** is to **order** the **documents** in the postings list for each term by **decreasing value** of $g(d)$. This allows us to perform the postings intersection algorithm of Picture 2.3, so we can concurrently traverse the query terms' postings for both postings intersection and cosine score computation. This idea is a direct extension of champion lists: for a well-chosen value r, we maintain for each term t a global champion list of the r documents with the highest values for $g(d) + tf - idf_{t,d}$. Then, at query time we only compute the net scores for documents in the union of these global champion lists. Intuitively, this has the effect of focusing on documents likely to have large net scores.

The reason why we perform an ordering by $g(d)$ is that under $g(d)$ ordering top-scoring docs are likely to appear early in postings traversal, so in time-bound applications, for example, this strategy allows us to stop the postings traversal early with good results, since we avoid computing scores for non-relevant documents.

The **second idea** is to maintain for each term t **two postings lists** consisting of **disjoint sets of documents**, each sorted by $g(d)$ values. The first list, which we call *high*, contains the m documents with the highest tf values for t. The second list, which we call *low*, contains all other documents containing t. When processing a query, we first scan only the *high* lists of the query terms, computing net scores for any document on the *high* lists of all (or more than a certain number of) query terms. If we obtain scores for k documents in the process, we terminate. If not, we continue the scanning into the *low* lists, scoring documents in these postings lists.

More generally, this second idea belongs to a more general aproach that consists of exploiting **tiered indexes**: the idea of these indexes is to break the postings up into a hierarchy of lists of decreasing importance, where the importance may be defined by $g(d)$ or by $g(d) + tf - idf(t, d)$. The structure of tiered indexes is provided in Picture 8.1.5.

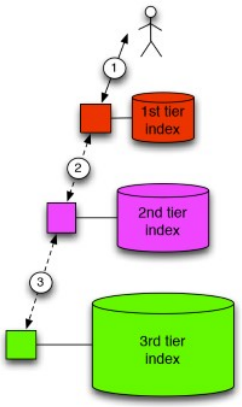


Figure 48: Structure of a tiered index

As we can see:

- The tiered index is composed of several sub-indexes;
- Former sub-indexes are small and keep more important documents, while later sub-indexes are larger and keep less important documents;
- At query time, only the top tier indexes are used to retrieve the top-k documents: if the retrieval fails, it proceeds to lower tiers.

This method is characterized by the following issues:

- The choice of the value of importance according to which a document is placed in each tier;
- How to place documents in different tiers (offline);
- At which tier do we stop the query processing (online).

Impact ordering

Thus far, we've only considered common ordering of documents, either by docID or by static quality score, e.g. net-score. However, this ordering imposes a DAAT scoring, i.e. a concurrent traversal of all of the query terms' postings computing the score for each document as we encounter it. We now consider a TAAT approach, that allows to preclude such a concurrent traversal.
The idea is to order the documents d in the postings list of term t by decreasing order of $tf(t, d)$. Thus, the ordering of documents will vary from one postings list to another, and we cannot compute scores by a concurrent traversal of the postings lists of all query terms. Given postings lists ordered by decreasing order of $tf(t, d)$, two ideas have been found to significantly lower the number of documents for which we accumulate scores:

1. When traversing the postings list for a query term t, we stop either after a fixed number of documents r have been seen, or after the value of $tf(t, d)$ has dropped below a threshold. Then, we take the union of the resulting sets of documents, and we compute the scores only for the documents in this union;
2. When accumulating cosine scores we only consider the query terms in decreasing order of *idf*, so that the query terms likely to contribute the most to the final scores are considered first.

Cluster pruning

In cluster pruning we have a preprocessing 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.
Specifically, the preprocessing step is as follows:

1. Pick $\sqrt{N}$ documents at random from the collection, the *leaders*. Alternatively, we can use k-means to choose them;
2. For each document that is not a *leader*, we pre-compute its nearest *leader*, that are called *followers*. Intuitively, the expected number of followers for each leader is $\approx N/\sqrt{N} = \sqrt{N}$;
3.

Next, query processing proceeds as follows:

1. Given a query q, find the *leader* L that is closest to q. This entails computing cosine similarities from q to each of the $\sqrt{N}$ *leaders*;
2. The candidate set A consists of L together with its *followers*. We compute the cosine scores for all documents in this candidate set;

A scheme of this approach is provided in Picture 8.1.5.

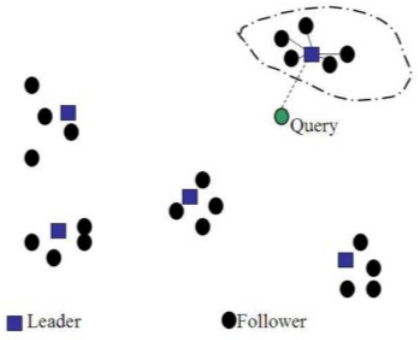


Figure 49: Cluster pruning

The use of randomly chosen *leaders* for clustering is fast and likely to reflect the distribution of the document vectors in the vector space: a region of the vector space that is dense in documents is likely to produce multiple *leaders* and thus a finer partition into sub-regions.

Variations of cluster pruning introduce additional parameters b_1 and b_2, both of which are positive integers. In the pre-processing step we attach each *follower* to its b_1 closest leaders, rather than a single closest leader. At query time we consider the b_2 leaders closest to the query q. Clearly, the basic scheme above corresponds to the case $b_1 = b_2 = 1$. Further, increasing b_1 or b_2 increases the likelihood of finding k documents that are more likely to be in the set of true top-scoring k documents, at the expense of more computation. In general, this variation implements the concept of *fuzzy/soft clustering*, and in general it produces larger clusters.

8.2 Components of an IR system

In this section we combine the ideas developed so far to describe a rudimentary search system that retrieves and scores documents. We first develop further ideas for scoring, beyond vector spaces. Following this, we will put together all of these elements to outline a complete system.

8.2.1 Tiered index

The first component is the *tiered index*, which we already presented in the previous section.

8.2.2 Query-term proximity

Especially for **free text queries** on the web, i.e. a set of terms typed into a query box, users prefer a **document** in which **most** or all of the **query terms appear close to each other**, because this is evidence that the document has text focused on their query intent.
Consider a query with two or more query terms, $t_1, t_2, .., t_k$. Let ω be the **width of the smallest window** in a document d that **contains all the query terms**, measured in the number of words in the window. For instance, if the document were to simply consist of the sentence "The quality of mercy is not strained", the smallest window for the query "strained mercy" would be 4. Intuitively, the **smaller** that ω is, the **better** that d matches the query. In cases where the document does not contain all of the query terms, we can set to be some enormous number. We could also consider variants in which only words that are not stop words are considered in computing ω. Such **proximity-weighted scoring functions** are a departure from pure cosine similarity and closer to the "soft conjunctive" semantics that Google and other web search engines evidently use.
How can we design such a proximity-weighted scoring function to depend on ω? We treat the integer ω as yet another **feature** in the **scoring function**, whose importance is assigned by **machine learning**, as will be developed in the following sections.

8.2.3 Query parsers

Common search interfaces tend to **mask query operators** from the end user, in order to hide the complexity of these operators from the largely non-technical audience for such applications, inviting **free text queries**. Given such interfaces, how should a search equipped with indexes for various retrieval operators treat a query such as "rising interest rates"?

The answer of course depends on the user **population**, the **query distribution** and the **collection of documents**. 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:

1. Run the user-generated query string as a *phrase query* "rising interest rates" and rank the matching docs using vector space scoring;
2. 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, as well;
3. If we still have fewer than ten results, run the vector space query consisting of the three individual query terms

Each of these steps (if invoked) may yield a list of scored documents, for each of which we compute a score. This score must combine contributions from vector space scoring, static quality, proximity weighting and potentially other factors. This demands an **aggregate scoring function** that accumulates *evidence* of a document's relevance from multiple sources. How do we devise a query parser and how do we devise the aggregate scoring function?
The answer depends on the setting. In many enterprise settings we have application builders who make use of a toolkit of available scoring operators, along with a query parsing layer, with which to manually configure the scoring function as well as the query parser. This setting works in collections whose characteristics change infrequently. **Web search** on the other hand is faced with a **constantly changing document collection** with new characteristics being introduced all the time. It is also a setting in which the number of scoring factors can run into the hundreds, making hand-tuned scoring a difficult exercise. To address this, it is becoming increasingly common to use **machine-learned scoring**, that we will explore later in the notes.

8.2.4 Putting it all together

The overall search system that supports free-text queries, as well as Boolean, zone and filed queries is represented in Picture 8.2.4: note that the paths are shown primarily for a free text query.
The steps are the following:

1. The documents stream in for **parsing** and **linguistic processing** (tokenization, stemming etc..). The resulting stream of tokens feeds into two modules:
   - First, we retain a copy of each parsed document in a **document cache**. This will enable us to generate **results snippets**, i.e. snippets of text accompanying each document in the results list for a query. This snippet tries to give a succinct explanation to the user of why the document matches the query;
   - A second copy of the tokens is fed to a bank of **indexers**.
2. The **indexers** create a bank of indexes including zone and field indexes that store the metadata for each document, tiered indexes, indexes for spelling correction and structures for accelerating inexact top-k retrieval;

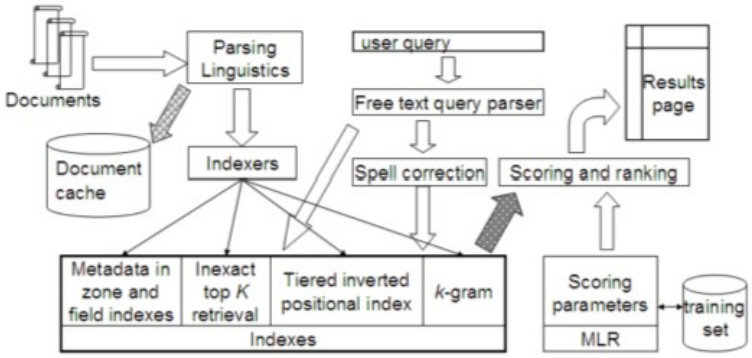


Figure 50: A complete search system

3. A **free text user query** is sent down to the indexes both directly and through a module for generating spelling-correction candidates;
4. **Retrieved documents** are passed to a **scoring module** that computes scores based on machine-learned ranking (MLR) for **scoring** and **ranking** documents;
5. Finally, these ranked documents are rendered as a **results page**.

9 Evaluation in IR

9.1 IRS quality evaluation

In general, the evaluation of a IR system depends on two important measures: the **efficiency**, i.e. how fast the system indexes a collection and how fast it searches, and the **effectiveness/quality** of the system, which can be measured by the number of clicks in the result pages (actually not a good measure since a user suffers from many biases) or by the number of repeated visitors/buyers or the dwell time.
The most common proxy for measuring the quality of an IR system is represented by the **relevance** of the search results. In order to measure the relevance of the results, we need a **test collection** consisting of:

1. A document collection;
2. A query collection;
3. A set of relevance judgments, standardly a binary assessment of either relevant (1) or non-relevant (0) for each query-document pair.

The standard approach to information retrieval system evaluation revolves around the notion of **relevant** and **non-relevant** documents. With respect to a user information need, a document in the test collection is given a binary classification as either relevant or non-relevant. This decision is referred to as the gold standard or **ground truth judgment of relevance**. Notice that 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 test document collection and suite of information needs have to be of a reasonable size: you need to average performance over fairly large test sets, as results are highly variable over different documents and information needs. In this sense, we may notice a crucial issue. Suppose that the query collection contains 50,000 queries, and that the document collection contains 5M documents: then, the matrix containing the relevance judgments would contain 0.25×10^{12} elements, and if we suppose that each judgement takes a human 2.5 seconds, then the matrix is created in 170M hours per person. However, **crowd sourcing** can be exploited to generate such test collection.
Finally, we also need the test queries, which on the one hand must be relevant to the available documents, and on the other they must be representative of actual user needs. In general, using random query terms from the documents is not a good idea, so usually a sample from the query logs is performed, or hand-crafted queries.

9.2 Standard test collections

Picture 9.2 shows the statistic of some of the most important test collections for IR evaluation.

9.3 Evaluation of unranked retrieval sets

The two most frequent and basic measures for information retrieval effectiveness are **precision** and **recall**. These are first defined for the simple case where an IR system returns

TABLE 4.3 Common Test Corpora

Collection	NDocs	NQrys	Size (MB)	Term/Doc	Q-D RelAss
ADI	82	35			
AIT	2109	14	2	400	>10,000
CACM	3204	64	2	24.5	
CISI	1460	112	2	46.5	
Cranfield	1400	225	2	53.1	
LISA	5872	35	3		
Medline	1033	30	1		
NPL	11,429	93	3		
OSHMED	34,8566	106	400	250	16,140
Reuters	21,578	672	28	131	
TREC	740,000	200	2000	89-3543	» 100,000

Figure 51: Test collections

a set of documents for a query. We will see later how to extend these notions to ranked retrieval situations.
Precision (P) is the fraction of retrieved documents that are relevant:

$$\text{Precision} = \frac{\text{retrieved relevant documents}}{\text{retrieved documents}} = P[\text{relevant}|\text{retrieved}]$$

, while the *Recall (R)* is the fraction of relevant documents that are retrieved:

$$\text{Recall} = \frac{\text{retrieved relevant documents}}{\text{relevant documents}} = P[\text{retrieved}|\text{relevant}]$$

If we consider the following contingency table

	Relevant	Nonrelevant
Retrieved	true positives (tp)	false positives (fp)
Not retrieved	false negatives (fn)	true negatives (tn)

then

$$P = \frac{tp}{tp + fp}$$

$$R = \frac{tp}{tp + fn}$$

Another measure is the *Accuracy*, i.e. the fraction of classification of the IR system that are correct:

$$\text{accuracy} = \frac{tp + tn}{tp + fp + tn + fn}$$

However, there is a good reason why **accuracy** is **not an appropriate measure** for information retrieval problems. In almost all circumstances, the **data** is **extremely**

skewed: normally over 99.9% of the documents are in the non-relevant category. A system tuned to maximize accuracy can appear to perform well by simply deeming all documents non-relevant to all queries, but labeling all documents as non-relevant is completely unsatisfying to an information retrieval system user.
The advantage of having the two numbers for precision and recall is that one is more important than the other in many circumstances. Typical web surfers would like every result on the first page to be relevant (**high precision**) but have not the slightest interest in knowing let alone looking at every document that is relevant. In contrast, various professional searchers such as paralegals and intelligence analysts are very concerned with trying to get as **high recall as possible**. Nevertheless, the two quantities clearly **trade off** against one another: you can always get a recall of 1 (but very low precision) by retrieving all documents for all queries! **Recall** is a **non-decreasing function of the number of documents retrieved**. On the other hand, in a good system, **precision** usually **decreases** as the **number of documents retrieved is increased**. In general we want to get some amount of recall while tolerating only a certain percentage of false positives.
A single measure that trades off precision and recall if *F measure*, which represents the weighted harmonic mean between the two measures:

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

, where $\beta^2 = \frac{1-\alpha}{\alpha}$ and $\alpha \in [0,1]$, thus $\beta^2 \in [0,\infty]$.
The default *balanced F measure* corresponds to $\alpha = 0.5$ or $\beta = 1$, and it is commonly written as F_1:

$$F_1 = \frac{1}{0.5\frac{1}{P} + 0.5\frac{1}{R}} = \frac{2PR}{P+R}$$

Notice that values of $\beta < 1$ emphasize precision, while values of $\beta > 1$ emphasize recall. The choice of the **harmonic mean** resides in the fact that it always **less than or equal** to the **arithmetic mean** and the **geometric mean**, so when the values of precision and recall differ greatly, the harmonic mean is closer to their minimum than to their arithmetic mean. This is useful, for example, if we have a system that return all the documents. In this case the recall is 100%, thus using the arithmetic mean we would get a 50% of evaluation, while using the harmonic mean and assuming that the precision is very low, the score is 0.02%, which is more explanatory.

9.4 Evaluation of ranked retrieval results

Precision, *recall*, and the *F measure* are set-based measures, i.e. they are computed using unordered sets of documents. We need to extend these measures (or to define new measures) if we are to evaluate the **ranked retrieval results** that are now standard with search engines. In a ranked retrieval context, appropriate sets of retrieved documents are naturally given by the top-k retrieved documents.

9.4.1 Precision-recall

For each such set, precision and recall values can be plotted to give a **precision-recall curve**, such as the one shown in Picture 9.4.1.

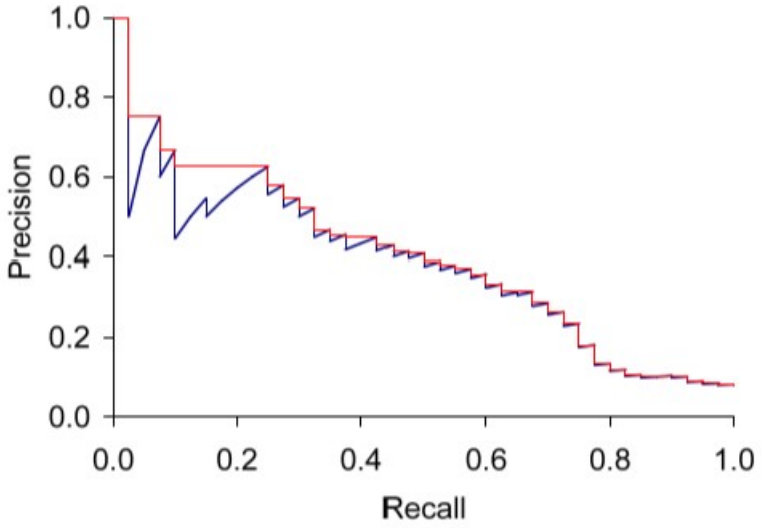


Figure 52: Precision-recall graph

Precision-recall curves have a distinctive **saw-tooth shape**: if the $(k+1)$-th retrieved document is non-relevant, then recall is the same as for the top-k documents, but precision has dropped. If it is relevant, then both precision and recall increase, and the curve jags up and to the right. In this sense, both precision and recall increases with each relevant document retrieved, while only precision drops when a non-relevant doc is retrieved.

9.4.2 Interpolated precision

Since the precision-recall is not a function we consider the **interpolated precision**: the *interpolated precision* p_{interp} at a certain recall level r is defined as the highest precision found for any recall level $r' \geq r$:

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

In Picture 9.4.1, the *interpolated precision* is depicted in red.
However, despite being very informative, there is the desire to boil the information of the *interpolated precision* down to a few numbers, or perhaps even a single number. The traditional way of doing this is the *11-point interpolated average precision*. For each information need, the interpolated precision is measured at the 11 recall levels of 0.0, 0.1, 0.2, . . . , 1.0. For the precision-recall curve in Picture 9.4.1, these 11 values are shown in the following table:
For each recall level, we then calculate the arithmetic mean of the interpolated precision at that recall level for each information need in the test collection. A composite precision recall curve showing 11 points is showed in Picture 9.4.2.

Recall	Interp. Precision
0.0	1.00
0.1	0.67
0.2	0.63
0.3	0.55
0.4	0.45
0.5	0.41
0.6	0.36
0.7	0.29
0.8	0.13
0.9	0.10
1.0	0.08

Figure 53: 11-point interpolated average precision

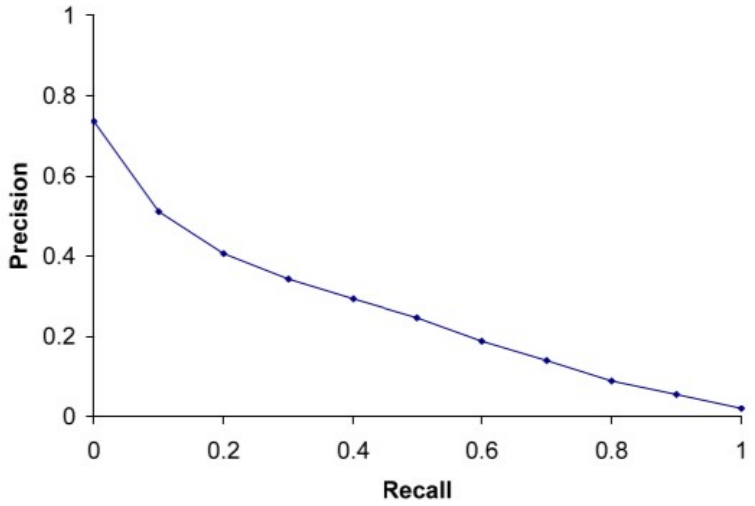


Figure 54: Averaged 11-point interpolated average precision across 50 queries

9.4.3 MAP

In recent year, other measures have become more common. One example **Mean Average Precision** or **MAP**, which has been shown to have especially good **discrimination** and **stability**. If the set of documents for a query $q_j \in Q$ is $\{d_1, .., d_m\}$ and R_{jk} is the set of ranked retrieval results from the top result until you get to document d_k, then:

$$MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} \text{Precision}(R_{jk})$$

, i.e. it is the arithmetic mean of average precision across multiple queries/ranking.
Notice that when a relevant document is not retrieved at all, the precision value in the above equation is taken to be 0. For a **single query**, the average precision approximates the area under the uninterpolated precision-recall curve, and so the MAP is roughly the **average area under the precision-recall curve for a set of queries**.
Example: Average Precision
Suppose we have two rankings, as in Picture 9.4.3, with the corresponding precision and recall.
Then:

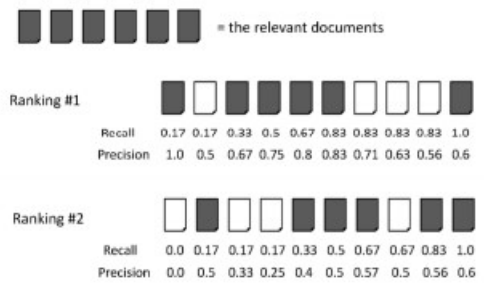


Figure 55: Example of AP

$$\text{AP}_1 = (1.0 + 0.67 + 0.75 + 0.8 + 0.83 + 0.6)/6 = 0.78$$

and

$$\text{AP}_2 = (0.5 + 0.4 + 0.5 + 0.57 + 0.56 + 0.6)/6 = 0.52$$

, so both rankings retrieve all the relevant documents, but the first one has a better AP.
Example: Mean Average Precision
Suppose we have two rankings, as in Picture 9.4.3, with the corresponding precision and recall.

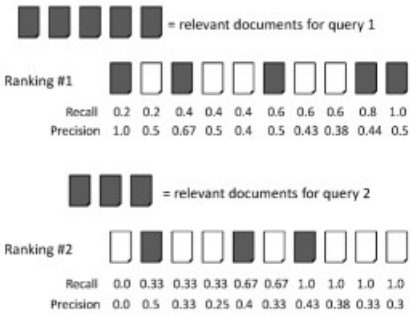


Figure 56: Example of MAP

Then:

$$\text{AP}_1 = (1.0 + 0.67 + 0.5 + 0.44 + 0.5)/5 = 0.62$$

and

$$\text{AP}_2 = (0.5 + 0.4 + 0.43)/3 = 0.53$$

Finally,

$$\text{MAP} = (0.62 + 0.44)/2 = 0.53$$

, so both rankings retrieve all the relevant documents, but the first one has a better AP.

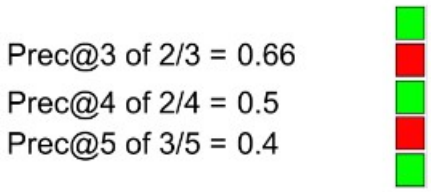
9.4.4 P@K, AP@K and MAP@K

The above measures factor in precision at all recall levels, but for many prominent applications, particularly web search, what matters is rather **how many good results there are on the first page or the first three pages**. This leads to measuring precision at fixed low levels of retrieved results, such as 10 or 30 documents. This is referred to as **Precision@K**, e.g. Precision@10. From a mathematical point of view, the Precision@K is defined as:

$$P@k(q_i) = \frac{1}{K} \sum_{k=1}^{K} \text{rel}(q_i, k)$$

, where

$$\text{rel}(q_i, k) = \begin{cases} 1 & \text{if document at rank } k \text{ is relevant} \\ 0 & \text{otherwise} \end{cases}$$



It has the **advantage** of **not requiring any estimate of the size of the set of relevant documents** but the **disadvantages** that it is the **least stable of the commonly used evaluation measures** and that it **does not average well**, since the total number of relevant documents for a query has a strong influence on Precision@k.
In a similar way, we can define AP@K as:

$$\text{AP@K}(q_i) = \frac{1}{K_i} \sum_{k=1}^{K} P@k(q_i) \cdot \text{rel}(q_i, k)$$

, where

$$K_i = \sum_{k=i}^{K} \text{rel}(q_i, k)$$

Finally, we can define MAP@k as:

$$MAP@k(Q) = \frac{1}{|Q|} \sum_{q_i \in Q} AP@K(q_i)$$

9.4.5 MRR

Another important measure is the **Mean Reciprocal Rank**, which is defined as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i}$$

, where rank_i represents the rank of the first relevant document returned for a query $q_i \in Q$.

9.4.6 DCG

Finally, we explore the **DCG** measure. We make two assumptions:

- Highly relevant documents are more useful than marginally relevant documents;
- The lower the ranked position of a relevant document, the less useful it is for the user, since it is less likely to be examined.

Now, this measure uses the notion of **gain** for measuring the usefulness of a document: the gain is accumulated, starting at the top of the ranking and may be reduced, or discounted, at lower ranks. If the judgements of the n documents are $r_1, r_2, .., r_n$ (in ranked order), then the **cumulative gain** is

$$CG = r_1 + r_2 + .. + r_n$$

In the **discounted cumulative gain**, the gain is discounted if the relevant documents appear at a rank greater than 1: a typical discount is $\frac{1}{\log(rank)}$. The DCG at a particular rank p is:

$$DCG_p = rel_1 + \sum_{i=2}^{p}\frac{rel_i}{\log_2 i}$$

An alternative formulation can be considered, where high relevance judgements become much more important:

$$DCG@p = rel_1 + \sum_{i=2}^{p}\frac{2^{rel_i}}{\log_2(1+i)}$$

In this sense, this measure emphasis on retrieving highly relevant documents.
Example: DCG
Suppose that the relevance judgements for 10 documents on a 0-3 scale are the following: [3, 2, 3, 0, 0, 1, 2, 2, 3, 0]. Then:

$$\frac{rel_i}{\log_2 i} = [3, 2, \frac{3}{1.59}, 0, 0, \frac{1}{2.59}, \frac{2}{2.81}, \frac{2}{3}, \frac{3}{3.17}, 0] = [3, 2, 1.89, 0, 0, 0.39, 0.71, 0.67, 0.95, 0]$$

Then, we can compute DCG@p as:

$$DCG@p = [3, 5, 6.89, 6.89, 6.89, ..]$$

9.4.7 NDCG

The idea of **NDCG** is to **normalize DCG** at rank n by the DCG value at rank n of the **ideal ranking**, where the ideal ranking would first return the documents with the highest relevance score, then the second highest etc..

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

, where:

- Q is a set of queries;
- Z_{kj} is a normalization factor calculated to make it so that a perfect ranking's NDCG@k for query j is 1;
- $R(j,m)$ is the relevance score given to document d for query j.

Notice that this normalization is particularly useful for contrasting queries with varying number of relevant documents, and for this reason it is quite popular in evaluating IR systems.
Example: NDCG
Suppose that the perfect ranking is given by the following order of relevance judgements [3,3,3,2,2,2,1,0,0,0]. Then, the ideal DCG values would be [3, 6, 7.89, 8.89, 9.75, ..].
Suppose that the actual rank is [3, 2, 3, 0, 0, 1, 2, 2, 3, 0], then the actual DCG is [3, 5, 6.89, 6.89, 6.89, 7.28, ..].
Thus, the NDCG values are [1, 0.83, 0.87, ..]. Notice that $\forall k, NDCG@k \leq 1$.

9.5 User utility

Since the **human judgements** that are used to estimate the relevance of documents are expensive, inconsistent (both between different users and over time), and are not always a good representative of real users, we may exploit **users' clicks**. From a study we can notice that the users suffer from a strong **position bias**, i.e. they're more likely to click on links in the first position more than others, even if the link is not correct.

9.5.1 Kendall's τ

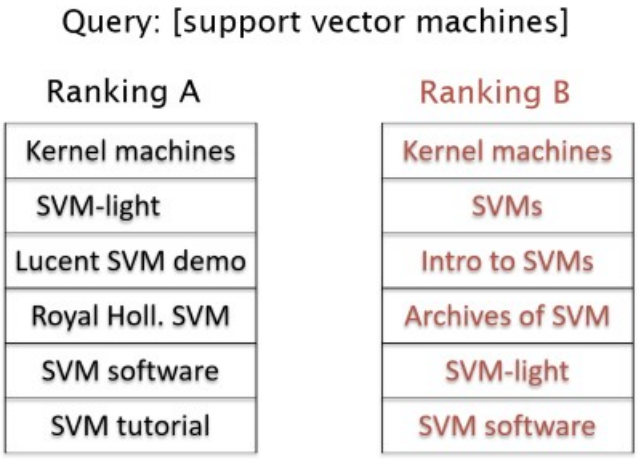
Given a set of pairwise preferences P, without relevance judgements assigned to documents, we can measure the quality of two ranking A and B by defining a proximity measure between A and P and B and P. The winning ranking is the one with better proximity with P. Clearly, the proximity measure should **reward** agreements with P, and **penalize** disagreements with P.
A possible proximity measure could be **Kendall's** τ: let X be the number of agreements between ranking A and P, and Y be the number of disagreements, then the Kendall's τ correlation between A and P is defined as:

$$\tau(A,B) = \frac{X-Y}{X+Y}$$

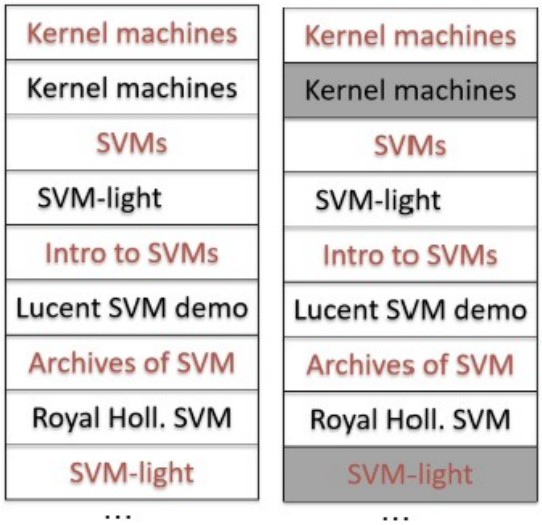
Notice that $0 \leq \tau(A, B) \leq 1$, and if $\tau(A, B) = 1$, then we have a perfect agreement, if $\tau(A, B) = -1$, then we have perfect disagreement, i.e. one ranking is the reverse of the other, and if $\tau(A, B) = 0$, then X and Y are independent.

9.5.2 A/B testing

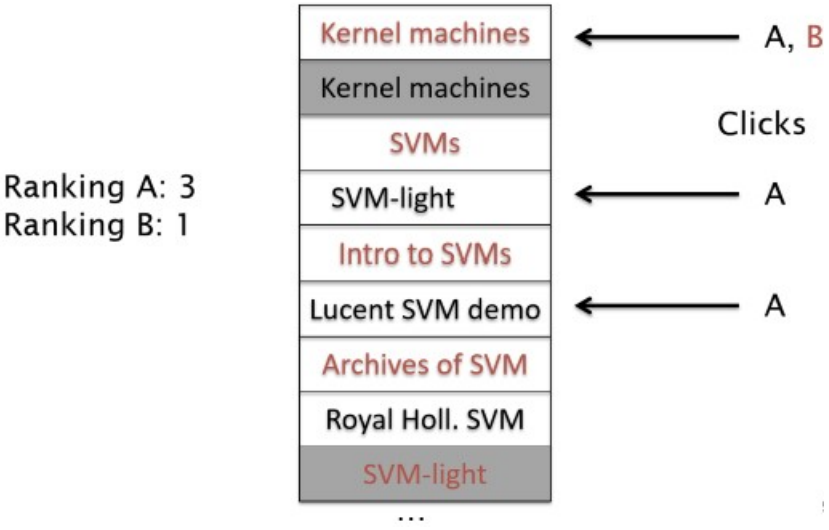
If an IR system has been built and is being used by a large number of users, the system's builders can evaluate possible changes by deploying variant versions of the system and recording measures that are indicative of user satisfaction with one variant vs. others as they are being used. This method is frequently used by web search engines.
The most common version of this is **A/B testing**. For such a test, precisely **one thing** is **changed** between the **current system** and a **proposed system**, and a **small proportion of traffic** (say, 1–10% of users) is randomly **directed to the variant system**, while most users use the current system.
In IR systems, A/B testing works as follows: suppose we have two rankings for the same query, as showed in Picture 9.5.2.



We can now interleave the results of the two rankings and remove the duplicates, as shown in Picture 9.5.2.



Finally, we can count the clicks on results from A versus the results from B, as showed in Picture 9.5.2: better rankings will (on average) get more clicks.



In practice, though, A/B testing is widely used, because A/B tests are easy to deploy, easy to understand, and easy to explain to management.

10 Relevance feedback and query expansion

In most collections, the same concept may be referred to using different words. This issue, known as *synonymy*, has an **impact** on the **recall** of most information retrieval systems. For example, you would want a search for *aircraft* to match *plane* (but only for references to an airplane, not a woodworking plane), and for a search on *thermodynamics* to match references to *heat* in appropriate discussions. Users often attempt to address this problem themselves by manually refining a query, but n this chapter we discuss ways in which a system can help with query **refinement**, either **fully automatically** or with the **user** in the loop.
The methods for tackling this problem split into two major classes: **global methods** and **local methods**. **Global** methods are **techniques** for **expanding** or reformulating query terms **independent of the query** and **results** returned from it, so that changes in the query wording will cause the new query to match other semantically similar terms. Global methods include:

- Query expansion with a thesaurus or WordNet;
- Query expansion via automatic thesaurus generation;
- Techniques like spelling correction.

Local methods adjust a query **relative** to the **documents** that initially appear to match the query, and the basic method is **relevance feedback**.

10.1 Relevance feedback

The idea of **relevance feedback** (**RF**) is to **involve** the **user** in the **retrieval process** so as to improve the final result set. In particular, the user gives feedback on the relevance of documents in an initial set of results. The basic procedure is:

1. The user issues a (short, simple) query;
2. The system returns an initial set of retrieval results;
3. The user marks some returned documents as relevant or non relevant;
4. The system computes a better representation of the information need based on the user feedback;
5. The system displays a revised set of retrieval results.

The process exploits the idea that it may be difficult to formulate a good query when you don't know the collection well, but it is easy to judge particular documents, and so it makes sense to engage in iterative query refinement of this sort. In such a scenario, relevance feedback can also be **effective** in **tracking a user's evolving information need**: seeing some documents may lead users to refine their understanding of the information they are seeking.
Picture 10.1 shows a textual IR example where the user wishes to find out about new applications of space satellites.
In the Picture:

(a) Query: New space satellite applications

(b)
+ 1. 0.539, 08/13/91, NASA Hasn't Scrapped Imaging Spectrometer
+ 2. 0.533, 07/09/91, NASA Scratches Environment Gear From Satellite Plan
3. 0.528, 04/04/90, Science Panel Backs NASA Satellite Plan, But Urges Launches of Smaller Probes
4. 0.526, 09/09/91, A NASA Satellite Project Accomplishes Incredible Feat: Staying Within Budget
5. 0.525, 07/24/90, Scientist Who Exposed Global Warming Proposes Satellites for Climate Research
6. 0.524, 08/22/90, Report Provides Support for the Critics Of Using Big Satellites to Study Climate
7. 0.516, 04/13/87, Arianespace Receives Satellite Launch Pact From Telesat Canada
+ 8. 0.509, 12/02/87, Telecommunications Tale of Two Companies

(c)
2.074 new 15.106 space
30.816 satellite 5.660 application
5.991 nasa 5.196 eos
4.196 launch 3.972 aster
3.516 instrument 3.446 arianespace
3.004 bundespost 2.806 ss
2.790 rocket 2.053 scientist
2.003 broadcast 1.172 earth
0.836 oil 0.646 measure

(d)
* 1. 0.513, 07/09/91, NASA Scratches Environment Gear From Satellite Plan
* 2. 0.500, 08/13/91, NASA Hasn't Scrapped Imaging Spectrometer
3. 0.493, 08/07/89, When the Pentagon Launches a Secret Satellite, Space Sleuths Do Some Spy Work of Their Own
4. 0.493, 07/31/89, NASA Uses 'Warm' Superconductors For Fast Circuit
* 5. 0.492, 12/02/87, Telecommunications Tale of Two Companies
6. 0.491, 07/09/91, Soviets May Adapt Parts of SS-20 Missile For Commercial Use
7. 0.490, 07/12/88, Gaping Gap: Pentagon Lags in Race To Match the Soviets In Rocket Launchers
8. 0.490, 06/14/90, Rescue of Satellite By Space Agency To Cost $90 Million

Figure 57: Example of relevance feedback process

- (a) shows the initial query;
- (b) shows how the user marked some documents as relevant, using the +;
- (c) shows the terms that are used to expand the initial query, along with the weights for each term;
- (d) shows the revised top results. Notice that the first two documents are the same of the first top-8, while many others were not included initially.

10.1.1 The Rocchio algorithm for relevance feedback

The Rocchio algorithm is the classic algorithm for implementing relevance feedback in the vector space model.
The goal of the algorithm is to find a query vector $\vec{q}$ that maximizes the similarity with relevant documents while minimizing similarity with non-relevant documents. If C_r is the

set of relevant documents, and C_{nr} the set of non-relevant documents, then we wish to find:

$$\vec{q}_{opt} = \arg\max_{\vec{q}}[\text{sim}(\vec{q}, \mu(C_r)) - \text{sim}(\vec{q}, \mu(C_{nr}))]$$

, where μ is the centroid of the two sets, and it is define as $\mu(D) = \frac{1}{|D|}\sum_{d \in D}\vec{d}$.
Notice that if we use the cosine similarity, then the equation becomes:

$$\vec{q}_{opt} = \frac{1}{|C_r|}\sum_{\vec{d}_j \in C_r}\vec{d}_j - \frac{1}{|C_{nr}|}\sum_{\vec{d}_j \in C_{nr}}\vec{d}_j$$

, i.e. the **optimal query** is the vector **difference** between the **centroids** of the relevant and non-relevant documents, as showed in Picture 10.1.1.

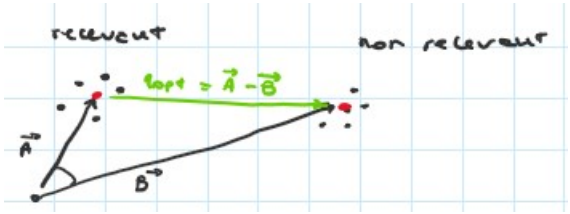


Figure 58: Optimal query in Rocchio's algorithm

Notice that a strong **assumption** of this algorithm is that the full **sets of relevant and non-relevant documents are known**, which is not necessarily true in real systems. For this reason, the algorithm that was introduced and popularized in SMART system proposed to use the modified query $\vec{q}_m$:

$$\vec{q}_m = \alpha\vec{q}_0 + \beta\mu(D_r) - \gamma\mu(D_{nr})$$

, where:

- D_r is the set of **known relevant document** vectors;
- D_{nr} is the set of **known non-relevant document** vectors;
- $\vec{q}_m$ is the resulting modified query vector;
- $\vec{q}_0$ is the original query vector;
- α, β and γ are weights that can be hand-chosen or set empirically;
- $\mu(D_r)$ and $\mu(D_{nr})$ represent, respectively, the centroid of the known relevant documents and the centroid of the known non-relevant documents.

Some notes:

- Since $|C_r| << |D_r|$ and $|C_{nr}| << |D_{nr}|$, $\vec{q}_m$ represents an **approximation** of $\vec{q}_{opt}$;
- The weights control the balance between trusting the judged document set versus the query: if we have a lot of judged documents, we would like a higher β and γ;

- Starting from q_0, the **new query moves** you some distance **toward** the **centroid** of the **relevant documents** and some distance **away** from the **centroid** of the **non-relevant documents**;
- Negative term weights are ignored, i.e. they are set to 0.

Picture 10.1.1 shows an application of Rocchio's algorithm: after some documents are labeled as relevant and non-relevant, the initial query vector is moved in response to this feedback

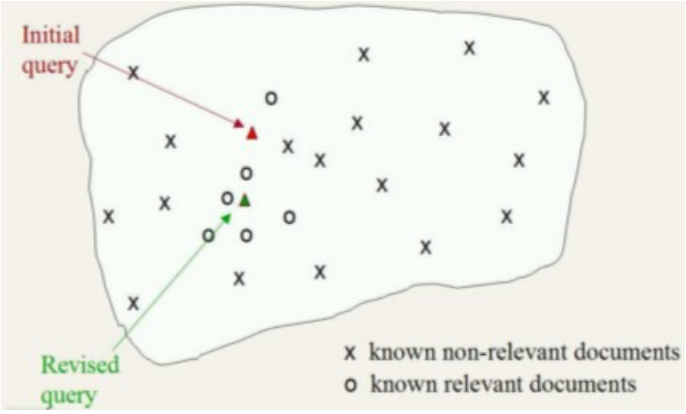


Figure 59: Example of Rocchio's algorithm

Relevance feedback can **improve** both **recall** and **precision**. But, in practice, it has been shown to be most useful for increasing recall in situations where recall is important. This is partly because the technique expands the query, but it is also partly an effect of the use case: when they want high recall, users can be expected to take time to review results and to iterate on the search.
Positive feedback also turns out to be **much more valuable** than **negative feedback**, and so most IR systems set $\gamma < \beta$. Reasonable values might be $\alpha = 1$, $\beta = 0.75$, and $\gamma = 0.15$. In fact, many systems allow only positive feedback, which is equivalent to setting $\gamma = 0$.
In general, the **disadvantages** of relevance feedback are the following:

- It is **expensive**: it creates long modified queries,as the resulting vectors are less sparse, which are expensive to process;
- Users are reluctant to provide explicit feedback;
- It's often hard to understand why a particular document was retrieved after applying relevance feedback.

10.1.2 Pseudo-relevance feedback

Pseudo relevance feedback provides a **method** for **automatic local analysis**. It **automates** the **manual part of relevance feedback**, so that the user gets improved retrieval performance without an extended interaction.
The method is to do normal retrieval to find an **initial set of most relevant documents**, to then **assume** that the **top-k** ranked documents are **relevant**, and finally to do

relevance feedback as before under this assumption. This automatic **technique** mostly **works**, but can go horribly wrong for some queries, due to the *query drift* problem.

10.2 Query expansion

In relevance feedback, users give additional input on documents (by marking documents in the results set as relevant or not), and this input is used to re-weight the terms in the query for documents. In **query expansion** on the other hand, users give additional input on query words or phrases, possibly **suggesting additional query terms**. Some search engines (especially on the web) suggest related queries in response to a query (*Also try..*); the users then opt to use one of these alternative query suggestions.
The central question in this form of query expansion is **how to generate alternative or expanded queries for the user**. The most common form of query expansion is **global analysis** (i.e. **not query dependent**), using some form of **thesaurus**. For each term t in a query, the **query** can be automatically **expanded** with **synonyms** and **related words** of t from the thesaurus. Use of a thesaurus can be combined with ideas of **term weighting**: for instance, one might weight added terms less than original query terms.
Methods for building a thesaurus for query expansion include:

- Use of a controlled **vocabulary** maintained by human editors;
- A **manual thesaurus**, i.e. a set of synonyms;
- An **automatically derived thesaurus**, formed by exploiting co-occurrence statistics over collections of documents and word embeddings;
- Using **query log mining**, i.e. exploiting the manual query reformulations, the session analysis and the co-clicks of other users to make suggestions to a new user. This requires a huge query volume, and is thus particularly appropriate to web search.

Thesaurus-based query expansion has the **advantage** of **not requiring any user input**. Use of query expansion generally **increases recall**, but it may significantly **reduce** the **precision**, particularly when the query contains ambiguous term. In this sense, a domain specific thesaurus is required: general thesauri and dictionaries give far too little coverage of the rich domain-particular vocabularies of most scientific fields. It is widely used in many science and engineering fields. Moreover, it is very expensive to create a manual thesaurus and to maintain it over time.

10.2.1 Automatic thesaurus generation

As we introduced before, an automatic thesaurus can be exploited for query expansion.
Before analyzing how an automatic thesaurus can be generated, we now focus on how we can represent the **words** as **vectors**. Notice that this task is particularly useful, since we can exploit the vector representations in order to compute similarity between words. One possibility is to consider a **one-hot representation** of terms, i.e. each term is represented by an extremely sparse vector $v \in \mathbb{R}^N$, where N is the lexicon size, and $v[i] = 1$ if the term is equal to the one in the lexicon, 0 otherwise. Clearly, this representation is not very suitable, since:

- It results in very high dimensional vectors;
- It does not provide a notion of similarity between vectors: e.g. *car* and *automobile*, despite being two similar words, are orthogonal vectors.

A possible solution is to **learn to encode similarity in the vectors** themselves, and in particular by representing **words** by their **context**. In this case, a word's meaning is given by the words that frequently appear close-by, and in order to represent a word, we use many contexts of it.
This notion is particularly useful for **automatic thesaurus generation**, whose attempt is to generate a thesaurus by analyzing a collection of documents. There are two main approaches:

1. Exploit **word co-occurrence**. We say that words co-occurring in a document or paragraph are likely to be in some sense similar or related in meaning, and simply count text statistics to find the most similar words;
2. Use a shallow **grammatical analysis** of the text and to exploit grammatical relations or grammatical dependencies. For example, we say that entities that are grown, cooked, eaten, and digested, are more likely to be food items.

Word	Nearest neighbors
absolutely	absurd, whatsoever, totally, exactly, nothing
bottomed	dip, copper, drops, topped, slide, trimmed
captivating	shimmer, stunningly, superbly, plucky, witty
doghouse	dog, porch, crawling, beside, downstairs
makeup	repellent, lotion, glossy, sunscreen, skin, gel
mediating	reconciliation, negotiate, case, conciliation
keeping	hoping, bring, wiping, could, some, would
lithographs	drawings, Picasso, Dali, sculptures, Gauguin
pathogens	toxins, bacteria, organisms, bacterial, parasite
senses	grasp, psyche, truly, clumsy, naive, innate

Figure 60: Example of an automatically generated thesaurus

Simply using **word co-occurrence** is **more robust** (it cannot be misled by parser errors), but using **grammatical relations is more accurate**.
The simplest way to compute a co-occurrence thesaurus is based on **term-term similarities**. We begin with a **term-document matrix** A, where each cell $A_{t,d}$ is a weighted count $w_{t,d}$ for term t and document d, with weighting so A has length-normalized rows. Notice that the dimensionality of A is $M \times N$, where M is the size of the lexicon, and N is the size of the collection, i.e. the number of documents. If we then calculate $C = AA^T$, then $C_{u,v}$ is a **similarity score** between terms u and v, with a **larger** number being **better**.
While some of the thesaurus terms are good or at least suggestive, others are marginal or bad. Term ambiguity easily introduces irrelevant statistically correlated terms. For example, a query for *Apple computer* may expand to *Apple red fruit computer*. In general these thesauri suffer from both **false positives** and **false negatives**. Moreover, since the terms in the automatic thesaurus are highly correlated in documents anyway (and

often the collection used to derive the thesaurus is the same as the one being indexed), this form of **query expansion may not retrieve many additional documents**. Another very important way to compute a co-occurrence thesaurus is to use **word embeddings**: we build a **dense vector for each word**, with the property that it is similar to vectors of words that appear in similar context. In particular, we can measure **similarity** between documents by using the **dot product**. A famous framework that is used for learning word vectors is **Word2vec**, which works as follows:

- A large corpus of text is provided as input to the model;
- Each word is the represented by a vector;
- The model goes through each position t in the text, which has a center word c and a list of context words o (usually not so large): the similarity between the vectors c and each context word o is used to calculate the probability of o given c (or vice-versa), as showed in Picture 10.2.1.

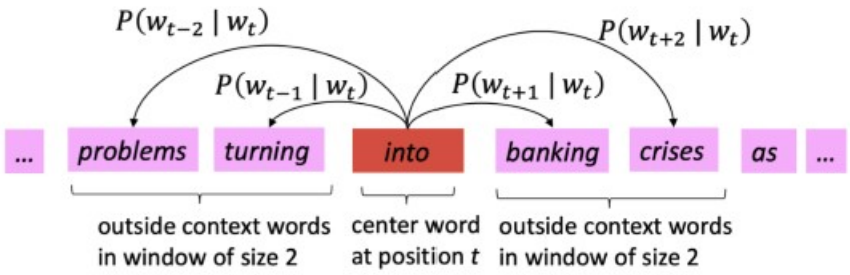


Figure 61: Example of windows and process for computing the probabilities

Then, the model keeps adjusting the word vectors to maximize this probability.

In other words, the goal of Word2vec is to train a **Neural Network** on a large corpus of pairs (c, o), i.e. center and context terms pairs. An example of training samples is provided in Picture 10.2.1, while Picture 10.2.1 shows the Neural Network that is used for implementing Word2vec.

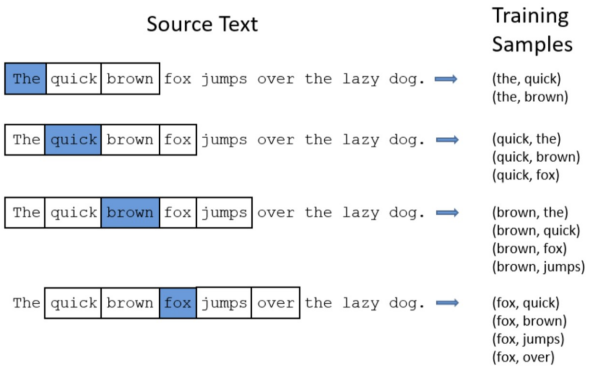


Figure 62: Example of training samples

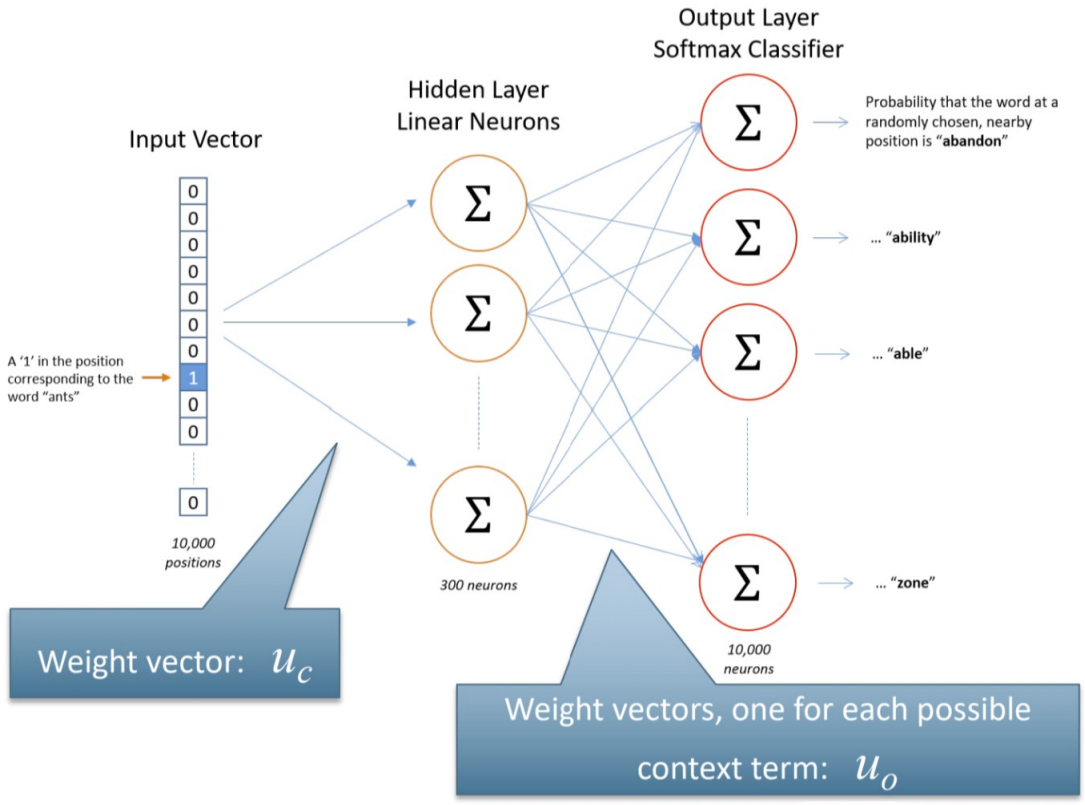


Figure 63: Neural Network of Word2vec

As we can see, the Network receives in input a vector in which there's a 1 in the position corresponding to the center word c, and for each term in the lexicon, it produces a probability that the word at a randomly chosen, nearby position is the term itself. Notice that the output layer is represented by a **Softmax classifier**, whose goal is to map a real number (in this case the similarity between vectors) to a probability distribution. In particular, for each word w, we use two different vectors:

- v_w when w is a center word;
- u_W when w is a context word.

Then, for a center word c and a context word o, we compute:

$$P(o|c) = \frac{\exp u_o^T v_c}{\sum_{w \in V} \exp u_w^T v_c}$$

Then, network minimizes the **cross-entropy** using Stochastic Gradient Descent.
In the case of query expansion, Word2vec can be used by expanding a query retrieving the k-nearest neighbors w.r.t. the term-vector similarity. The method works as follows:

- Each query is represented by a set of terms: $q = \{t_1, t_2, .., t_m\}$;
- We build a set of $m \times k$ candidate terms: $C = \bigcup_{t \in q} \text{k-NN}(t)$;

- Terms in C are sorted on the basis of the following score: $\text{AvgSim}(t, q) = \frac{1}{|q|} \sum_{t_i \in q} \text{cosine}(t, t_i)$, and the top-$k$ candidates are finally selected as expansion terms.

11 Web crawling

Web crawling is the process by which we **gather pages from the Web**, in order to index them and support a search engine. The objective of crawling is to **quickly** and **efficiently** gather as **many useful web pages as possible**, together with the link structure that interconnects them.
The goal of this chapter is not to describe how to build the crawler for a full-scale commercial web search engine. We focus instead on a range of **issues** that are generic to **crawling** from the student project scale to substantial research projects.

11.1 Features

We now list some features that a web crawler must provide:

- **Robustness**: the Web contains servers that create spider traps, which are generators of web pages that mislead crawlers into getting stuck fetching an infinite number of pages in a particular domain. Crawlers must be designed to be **resilient** to such **traps**;
- **Politeness**: a crawler must **respect** both the implicit and the explicit (i.e. specifications on which portions os a site can be crawled) **policies** of Web servers regulating the rate at which a crawler can visit them.

Other features that a Web crawler should provide are:

- **Distributed**, i.e. it should be designed to run on multiple distributed machines;
- **Scalable**, i.e. it should be designed to increase the crawl rate by adding more machines;
- **Performance and efficiency**, i.e. it should permit full use of available processing and network resources;
- **Quality**, i.e. the crawler should be biased towards fetching "useful" pages first;
- **Extensible**, i.e. it should be adaptable to new data formats and protocols;
- **Freshness**, i.e. it should continue fetching fresh copies of a previously fetched page.

11.2 Crawling

The basic operation of any hypertext crawler is as follows.

1. The crawler begins with **one or more URLs** that constitute a **seed set**;
2. It picks a **URL** from this seed set, then **fetches** the web page at that URL;
3. The fetched page is then parsed, to extract both the **text** and the **links** from the page (each of which points to another URL).
    - The extracted text is fed to a **text indexer**;

- The extracted links (URLs) are then added to a **URL frontier**, which at all times consists of URLs whose corresponding **pages** have **yet to be fetched** by the crawler. Initially, the URL frontier contains the seed set; as pages are fetched, the corresponding URLs are deleted from the URL frontier.

A visual representation of the seed set and the frontier is provided in Picture 11.2: as we can see, there are many crawlers that get the URL from the frontier, implemented as a queue.

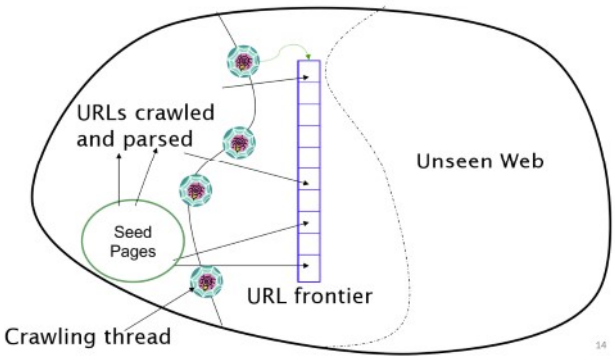


Figure 64: Visual representation of the crawling process

This seemingly **simple recursive traversal** of the web graph is **complicated** by the many **demands** on a practical web crawling system: the crawler has to be distributed, scalable, efficient, polite, robust and extensible while fetching pages of high quality. We examine the effects of each of these issues.

11.2.1 Crawler architecture

The simple scheme outlined above for crawling demands several modules that fit together as shown in Picture 11.2.1.

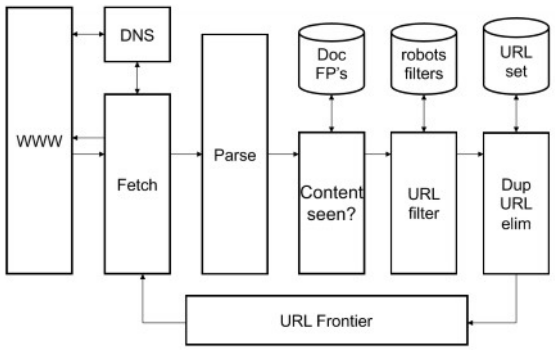


Figure 65: Architecture of a crawler

1. The **URL frontier**, containing URLs yet to be fetched in the current crawl (in the case of continuous crawling, a URL may have been fetched previously but is back in the frontier for re-fetching);

2. A **DNS resolution module** that determines the web server from which to fetch the page specified by a URL;
3. A **fetch module** that uses the http protocol to retrieve the web page at a URL;
4. A **parsing module** that extracts the text and set of links from a fetched web page;
5. A **duplicate elimination module** that determines whether an extracted link is already in the URL frontier or has recently been fetched.

Crawling is performed by anywhere from **one** to potentially **hundreds of threads**, each of which loops through the logical cycle in Picture 11.2.1. These threads may be run in a **single process**, or be **partitioned** amongst **multiple processes** running at different nodes of a distributed system. We begin by assuming that the **URL frontier** is **in place** and **non-empty**. Then, we follow the progress of a single URL through the cycle of being fetched, passing through various checks and filters, then finally (for continuous crawling) being returned to the URL frontier.
A crawler thread begins by **taking a URL** from the **frontier** and fetching the web page at that URL, generally using the *http* protocol. The **fetched page** is then **written** into a **temporary store**, where a number of operations are performed on it. Next, the page is **parsed** and the text as well as the links in it are extracted: the text is passed on to the indexer, while the link goes through a series of tests to determine whether the link should be added to the URL frontier.
First, the thread **tests** whether a **web page** with the **same content** has **already been seen** at another URL: the simplest implementation for this would use a simple fingerprint, while more sophisticated tests would use shingles. Next, a **URL filter** is used to determine whether the **extracted URL** should be **excluded** from the **frontier** based on one of **several tests**. For instance, the crawl may seek to exclude certain domains. Notice that a similar test could be **inclusive** rather than **exclusive**: many hosts on the Web place certain **portions** of their **websites off-limits to crawling**, under a standard known as the **Robots Exclusion Protocol**. This is done by placing a file with the name *robots.txt* at the root of the URL hierarchy at the site. Here is an example *robots.txt* file that specifies that no robot should visit any URL whose position in the file hierarchy starts with */yoursite/temp/*, except for the robot called *"searchengine"*.

```
User-agent: *
Disallow: /yoursite/temp/

User-agent: searchengine
Disallow:
```

Figure 66: Example of *robots.txt* file

Next, a URL should be **normalized**, and, finally, the URL is **checked** for **duplicate elimination**: if the URL is already in the frontier or (in the case of a non-continuous crawl) already crawled, we do not add it to the frontier. When the URL is added to the frontier, it is assigned a **priority** based on which it is eventually removed from the frontier for fetching.

11.2.2 DNS resolution

Given a URL in textual form, the process of translating it to an IP address is called **DNS resolution**: during this phase, the component of the web crawler contacts a *DNS server* that returns the translated IP address.
DNS resolution is a well-known **bottleneck** in web crawling. Due to the **distributed nature** of the Domain Name Service, DNS resolution may entail **multiple requests** and round-trips across the internet, requiring seconds and sometimes even longer. Right away, this puts in **jeopardy** our goal of **fetching several hundred documents a second**. A standard remedy is to introduce **caching**: URLs for which we have recently performed DNS lookups are likely to be found in the DNS cache, avoiding the need to go to the DNS servers on the internet. However, obeying politeness constraints limits the of cache hit rate.
There is another important difficulty in DNS resolution; the **lookup implementations** in standard libraries (likely to be used by anyone developing a crawler) are generally **synchronous**. This means that **once a request is made** to the Domain Name Service, **other** crawler **threads** at that node **are blocked** until the first request is completed. To circumvent this, most web crawlers implement their **own DNS resolver** as a component of the crawler.

11.2.3 URL frontier

Two important considerations govern the **order** in which **URLs** are **returned by the frontier**:

1. **Freshness**, i.e. **high-quality pages** that change frequently should be **prioritized** for **frequent crawling**. Thus, the **priority** of a page should be a **function** of both its **change rate** and its **quality** (using some reasonable quality estimate). The combination is necessary because a large number of spam pages change completely on every fetch;
2. **Politeness**, i.e. we must **avoid repeated fetch requests to a host within a short time span**. A common heuristic is to insert a time gap between successive requests to an host.

Clearly, these goals may conflict with each other.
A state-of-the-art (continuous) web crawler maintains two separate queues for prioritizing the download of URLs:

- The **discovery queue**, which downloads pages that are not previously downloaded, pointed by already discovered links;
- The **refreshing queue**, which re-downloads already downloaded pages, and thus tries to increase the freshness of the repository.

Since the Web is dynamic, the general problem is that the client must periodically poll the remote source to detect changes with local copies and refresh them. In general, the *freshness* F and the *age* A of a page e_i are defined as:

$$F(e_i;t) = \begin{cases} 1 & \text{if} e_i \text{is up-to-date at time} t \\ 0 & \text{otherwise} \end{cases}$$

$$A(e_i;t) = \begin{cases} 0 & \text{if} e_i \text{is up-to-date at time} t \\ (t - \text{modification time of } e_i) & \text{otherwise} \end{cases}$$

The evolution in time of *freshness* and *age* are represented in Picture 11.2.3.

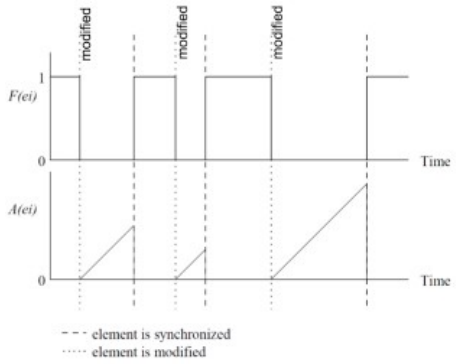


Figure 67: Evolution of freshness and age over time

In general, the goal of the refreshing operation is to maintain the average number of fresh pages as high as possible, and the average age of the pages as low as possible. In this sense, the update can be either uniform, i.e. regardless of the age of the page, or proportional to the change rate of a page: in a simulated environment, the uniform strategy is surprisingly better, since the crawler wastes a lot of time in trying to keep updated pages changing often, so in order to improve freshness, we must penalize pages changing too often.

11.2.4 Distributed crawler

We have mentioned that the threads in a crawler could run under different processes, each at a different node of a distributed crawling system. Such distribution is essential for scaling; it can also be of use in a **geographically distributed crawler system** where each node crawls hosts "near" it. Partitioning the hosts being crawled amongst the crawler nodes can be done by a hash function, or by some more specifically tailored policy. For instance, we may locate a crawler node in Europe to focus on European domains, although this is not dependable for several reasons.
How do the **various nodes** of a distributed crawler **communicate** and **share URLs**? The idea is to replicate the flow of Picture 11.2.1 at each node, with one essential difference: following the URL filter, we use a **host splitter** to dispatch each surviving URL to the crawler node responsible for the URL; thus the set of hosts being crawled is partitioned among the nodes. This modified flow is shown in Picture 11.2.4. The output of the host splitter goes into the Duplicate URL Eliminator block of each other node in the distributed system.
In this sense, the idea is to assign a partition of the Web to each of the crawlers, and this **Web partitioning** can be, for example Hash-based, i.e. based on the MD5 hashes of either canonical URLs or host names. Moreover, regarding **fault tolerance**, when a crawling node dies, its URLs are redistributed over the remaining crawler nodes.

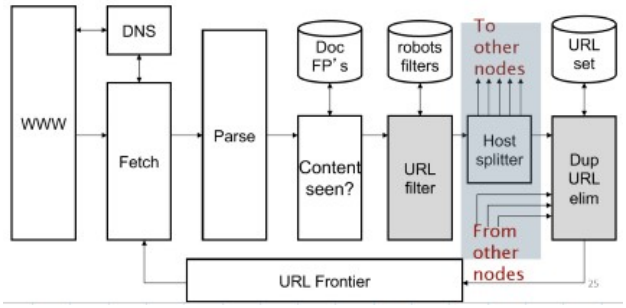


Figure 68: Distributed version of the crawl architecture

12 Link analysis

12.1 Introduction

Early search engines retrieved **relevant pages** for the user based primarily on the content **similarity** of the user **query** and the **indexed pages** of the search engines. Starting from **1996**, it became clear that **content similarity** alone was **no longer sufficient** for search due to two reasons:

1. The **number of Web pages grew rapidly** during the middle to late 1990s. Given any query, the number of relevant pages can be huge, and this abundance of information causes a major problem for ranking, i.e., how to choose only 10–30 pages and rank them suitably to present to the user;
2. Content **similarity methods** are easily **spammed**. A page owner can **repeat some important words** and add many remotely related words in his/her pages to **boost the rankings of the pages** and/or to make the pages relevant to a large number of possible queries.

Starting from around 1996, researchers in academia and search engine companies began to work on the problem, and they resorted to **hyperlinks**. Unlike **text documents** used in traditional information retrieval, which are often considered **independent** of one another (i.e., with no explicit relationships or links among them except in citation analysis), **Web pages** are **connected** through **hyperlinks**, which carry important information. **Some** hyperlinks are used to **organize** a large amount of information at the same Web site, and thus **only point to pages in the same site**. **Other** hyperlinks point to **pages in other Web sites**. Such out-going hyperlinks often indicate an implicit conveyance of authority to the pages being pointed to. Therefore, those **pages** that are **pointed to** by **many other pages** are likely to **contain authoritative or quality information**. Such linkages should obviously be used in page evaluation and ranking in search engines. During the period of 1997-1998, two most influential hyperlink based search algorithms **PageRank** and **HITS** were designed. PageRank is the algorithm that powers the successful search engine Google. Both PageRank and HITS were originated from **social network analysis**,and they both exploit the hyperlink structure of the Web to rank pages according to their levels of "prestige" or "authority".
Notice that apart from search ranking, hyperlinks are also useful for **finding Web communities**. A Web community is a cluster of densely linked pages representing a group of people with a common interest. Beyond explicit hyperlinks on the Web, explicit or implicit links in other contexts are useful too, e.g., for discovering communities of named entities (e.g., people and organizations) in free text documents and for analyzing social phenomena in emails and friendship networks on social networking sites.

12.2 Network properties

Before analyzing the two algorithms defined above, we now focus on some important properties of **networks**, in particular the one of the Web graph.
A network is defined as **scale-free** if its degree distribution, i.e. the probability $P(k_i)$ that a node selected uniformly at random has K_i degree, follows a **power law** distribution:

$$P(k_i) = c \cdot k_i^{-\alpha}$$

Picture 12.2 shows the difference between a random network and a scale-free network.

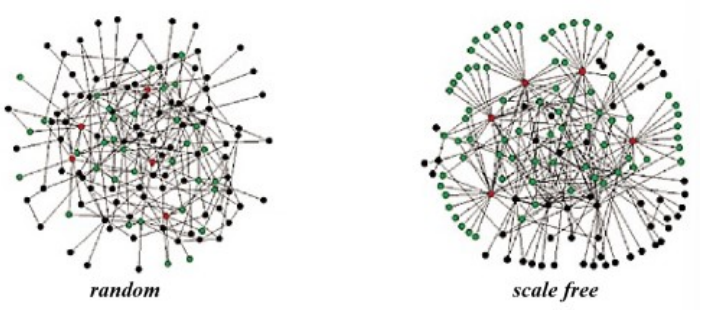


Figure 69: Random vs scale-free network

In other word, the property of scale-free networks ensures that:

- The **ratio** of very **connected nodes** and the number of nodes in the rest of the network remains **constant** as the network changes in size;
- Scale-free networks may show almost **no degradation** as "random" nodes fail: if failures occur randomly, since most nodes have a small degree, the likelihood that a hub (with many links) would be affected is almost negligible. But such networks highly **suffer from attacks toward high degree nodes**, which may cause disasters in the global inter-connectivity;
- They have a high value of the so-called **clustering coefficient** C. Given a vertex v, let k_v be the number of neighbors of v, then $C_v = \frac{\text{existing links}}{\text{all possible existing links}} = \frac{\text{existing links}}{k_v(k_v-1)}$. Then C is computed as the average across all the values of C_v. This coefficient in scale-free networks is about five times higher than the coefficient of a random graph. Notice that, in general, the coefficient decreases as the node degrees increase.

Notice that the name of scale-free networks comes from the fact that the power-law distribution looks the same, no matter the scale of the network. In this sense, the **shape** of the power-law **distribution** remains **unchanged**, except for a multiplicative constant. From a mathematical point of view, a probability distribution $p(x)$ is scale-free if exists $g(b)$ s.t. $p(bx) = g(b)p(x)$, for each b and x. If we consider the power-law distribution $p(x) = c \cdot x^{-\alpha}$:

$$p(bx) = c \cdot (bx)^{-\alpha} = b^{-\alpha} \cdot c \cdot x^{-\alpha}$$

from which we derive that $g(b) = b^{-\alpha}$, so that the property $p(bx) = g(b)p(x)$ is ensured. Scale-free networks arise from the **preferential attachment process**, a pattern of network growth in which new nodes prefer to link to highly connected nodes. In other words, given a new node, the probability that it connects to node$_i$ is proportional to the number of existing links k_i to node$_i$:

$$P(\text{linking to node}_i) = \frac{k_i}{\sum_j k_j}$$

If we think of a citation network, we are more likely to cite pages that already have a lot of links, and hence there's a positive feedback loop.
Finally, one last property on the Web network is that is is a **small world**, i.e. it has a short average path length d: in particular, $d = 0.35+2.06 \log N$. If we consider $N = 8*10^8$, then $d = 18.59$, i.e. two randomly chosen documents on the Web are on average 19 clicks away from each other.

12.3 Social network analysis

Social network is the study of **social entities** and their interactions and **relationships**. The interactions and **relationships** can be **represented** with a network or **graph**, where each vertex (or node) represents an actor and each link represents a relationship. From the network we can study the **properties of its structure**, and the role, position and prestige of each social actor. We can also find various kinds of sub-graphs, e.g., **communities** formed by groups of actors.
Social network analysis is useful for the Web because the Web is essentially a virtual society, and thus a virtual social network, where each page can be regarded as a social actor and each hyperlink as a relationship.
In this section, we introduce two types of social network analysis, **centrality** and **prestige**, which are closely related to hyperlink analysis and search on the Web. Both centrality and prestige are **measures of degree of prominence of an actor in a social network**.

12.3.1 Centrality

Important or prominent actors are those that are **linked** or involved with **other actors extensively**. In the context of an organization, a person with extensive contacts (links) or communications with many other people in the organization is considered more important than a person with relatively. A **central node** is a node involved in **many relationships**. There are different types of links or involvements between actors. Thus, several types of centrality are defined on undirected and directed graphs. We discuss three popular types below.
Degree centrality
Let the total number of actors in the network be n. In an **undirected graph**, the degree centrality of an actor i (denoted by $C_D(i)$) is simply the **node degree** (the number of edges) of the actor node, denoted by $d(i)$, **normalized** with the **maximum degree**, $n-1$:

$$C_D(i) = \frac{d(i)}{n-1}$$

Notice that $0 \leq C_D(i) \leq 1$
In a **directed graph**, we need to distinguish **in-links** of actor i (links pointing to i), and **out-links** (links pointing out from i). The degree centrality is defined based on only the out-degree (the number of out-links or edges), $d_o(i)$:

$$C'_D(i) = \frac{d_o(i)}{n-1}$$

Closeness centrality
This view of centrality is based on the closeness or **distance**. The basic idea is that an **actor** xi is **central** if it can **easily interact** with **all other actors**. That is, its **distance to all other actors is short**. Thus, we can use the shortest distance to compute this measure.
Let the shortest distance from actor i to actor j be $d(i, j)$ (measured as the number of links in a shortest path): then, for an undirected graph the closeness centrality $C_C(i)$ of actor i is defined as

$$C_C(i) = \frac{n-1}{\sum_{j=1}^{n} d(i,j)}$$

Notice that in this case we're assuming that the graph is connected, i.e. there exists a path from any point to any other point s.t. $d(i, j) \geq 0$. Also in this case $0 \leq C_C(i) \leq 1$.
The **same equation** can be used for a **directed graph**.
Betweenness centrality If two non-adjacent actors j and k want to interact and actor i is on the path between j and k, then i may have some control over their interactions. Betweenness measures this **control of** i **over other pairs of actors**. Thus, if i is on the **paths** of **many** such **interactions**, then i is an **important actor.**
For an **undirected graph**, let p_{jk} be the number of shortest paths between actors j and k. The betweenness of an actor i is defined as the number of shortest paths that pass i (denoted by $p_{jk}(i)$, $j \neq i$ and $k \neq i$) normalized by the total number of shortest paths of all pairs of actors not including i:

$$C_B(i) = \sum_{j<k} \frac{p_{jk}(i)}{p_{jk}}$$

Note that there may be multiple shortest paths between actor j and actor k. Some pass i and some do not. We assume that all paths are equally likely to be used. $C_B(i)$ has a **minimum** of 0, attained when i falls on no shortest path. Its **maximum** is $(n-1)(n-2)/2$, which is the number of pairs of actors not including i. In the network of Picture 12.3.1, actor 1 is the most central actor. It lies on all 15 shortest paths linking the other 6 actors. $C_B(1)$ has the maximum value of 15, and $C_B(2) = C_B(3) = C_B(4) = C_B(5) = C_B(6) = C_B(7) = 0$.

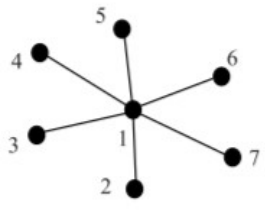


Figure 70: Betweenness centrality

If we are to ensure that the value range is between 0 and 1, we can normalize it with $(n-1)(n-2)/2$, which is the maximum value of $C_B(i)$. The standardized betweenness of actor i is defined as:

$$C'_B(i) = \frac{2\sum_{j<k} \frac{p_{jk}(i)}{p_{jk}}}{(n-1)(n-2)}$$

Unlike the closeness measure, the betweenness can be computed even if the graph is not connected.
For a **directed graph**, the same equation can be used but must be multiplied by 2 because there are now $(n-1)(n-2)$ pairs considering a path from j to k is different from a path from k to j.

12.3.2 Prestige

Prestige is a **more refined measure of prominence** of an actor than centrality as we will see below. We need to distinguish between ties sent (**out-links**) and ties received (**in-links**), and to compute the **prestige** of an actor, we only look at the **in-links**. Hence, the prestige cannot be computed unless the relation is directional or the graph is directed. The main difference between the concepts of centrality and prestige is that **centrality** focuses on **out-links** while **prestige** focuses on **in-links**.
We define three prestige measures. The third prestige measure (i.e., rank prestige) forms the basis of most Web page link analysis algorithms, including PageRank and HITS.
Degree prestige Based on the definition of the prestige, it is clear that an actor is **prestigious** if it **receives many in-links** or nominations. Thus, the simplest measure of prestige of an actor i (denoted by $P_D(i)$) is its in-degree,

$$P_D(i) = \frac{d_I(i)}{n-1}$$

where $d_I(i)$ is the in-degree of i (the number of in-links of i) and $n-1$ is the maximum in-degree of a node. As in the degree centrality, dividing by $n\breve{}1$ standardizes the prestige value to the range from 0 and 1. The **maximum** prestige value is 1 when **every other actor links to or chooses actor** i.
Proximity prestige The degree prestige of an actor i only considers the actors that are adjacent to i. The proximity prestige **generalizes** it by **considering** both the **actors directly and indirectly linked to actor** i. That is, we consider every actor j that can reach i, i.e., there is a directed path from j to i.
Let I_i be the set of actors that can reach actor i (notice that $|I_i| \leq 1$). The **proximity** is defined as **closeness** or distance of **other actors to** i. Let $d(j,i)$ denote the shortest path distance from actor j to actor i: to compute the proximity prestige, we use the average distance, which is

$$\sum_{j \in I_i} \frac{d(j,i)}{|I_i|}$$

If we look at the **ratio** or proportion of **actors** who **can reach** i to the **average distance that these actors are from** i, we obtain the **proximity prestige**, which has the value range of [0, 1]:

$$P_P(i) = \frac{\frac{|I_i|}{n-1}}{\sum_{j \in I_i} \frac{d(j,i)}{|I_i|}}$$

where $|I_i|/(n-1)$ is the **proportion** of **actors** that **can reach actor** i. In one extreme, every actor can reach actor i, which gives $|I_i|/(n-1) = 1$. The denominator is 1 if every actor is adjacent to i. Then, $P_P(i) = 1$. On the other extreme, no actor can reach actor i. Then $|I_i| = 0$, and $P_P(i) = 0$.

Rank prestige The above two prestige measures are based on in-degrees and distances. However, an important factor that has not been considered is the prominence of individual actors who do the "voting" or "choosing." In the **real world**, a **person** i **chosen** by an **important person** is **more prestigious** than chosen by a less important person. If one's circle of influence is full of prestigious actors, then one's own prestige is also high. Thus one's **prestige** is **affected** by the **ranks or statuses of the involved actors**. Based on this intuition, the rank prestige $P_R(i)$ is defined as a **linear combination of links that point to** i:

$$P_R(i) = \sum_{j=1}^{n} A_{ji} \cdot P_R(i)$$

where $A_{ji} = 1$ if j points to i, and 0 otherwise. This equation says that an **actor's rank prestige** is a **function** of the **ranks of the actors who vote or choose the actor**, which makes perfect sense.

Since we have n equations for n actors, we can write them in the **matrix notation**. We use P to represent the **column vector** that contains all the rank prestige values, i.e., $P = (P_R(1), P_R(2), \ldots, P_R(n))^T$. We use matrix A (where $A_{ij} = 1$ if i points to j, and 0 otherwise) to represent the **adjacency matrix** of the network or graph. Then, we have

$$P = A^T P$$

This equation is precisely the **characteristic equation** used for finding the **eigensystem** of the matrix A^T, so P is an **eigenvector** of A^T.

This equation and the idea behind it turn out to be very useful in Web search. Indeed, the most well known ranking algorithms for Web search, PageRank and HITS, are directly related to this equation.

12.4 Co-Citation and Bibliographic Coupling

Another area of research concerned with links is the **citation analysis** of scholarly publications. A scholarly publication usually cites related prior work to acknowledge the origins of some ideas in the publication and to compare the new proposal with existing work. Citation analysis is an area of bibliometric research, which **studies citations** to **establish** the **relationships** between **authors and their work**.

When a publication (also called a paper) cites another publication, a relationship is established between the publications. Citation analysis uses these relationships (links) to perform various types of analysis. A **citation** can represent many types of **links**, such as links between **authors**, **publications**, **journals** and conferences, and fields, or even between countries. We will discuss two specific types of citation analysis, co-citation and bibliographic coupling.

12.4.1 Co-citation

Co-citation is used to **measure** the **similarity** of **two papers**. If papers i and j are both cited by paper k, then they may be said to be related in some sense to each other, even though they do not directly cite each other. Picture 12.4.1 shows that papers i and j are co-cited by paper k. If papers i and j are cited together by many papers, it means that i and j have a strong relationship or similarity. The **more papers they are cited by**, the **stronger** their **relationship** is.

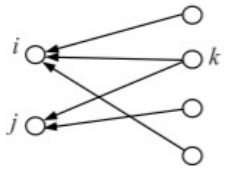


Figure 71: Co-citation

Let L be the **citation matrix**, where $L_{ij} = 1$ if paper i cites paper j, and 0 otherwise. **Co-citation** (denoted by C_{ij}) is a **similarity measure** defined as the **number of papers that co-cite** i **and** j, and is computed with

$$C_{ij} = \sum_{k=1}^{n} L_{ki} L_{kj} = (L^T L)_{ij}$$

where n is the total number of papers. Notice that C_{ii} is naturally the number of papers that cite i. A square matrix C can be formed with Cij, and it is called the **co-citation matrix**. Co-citation is symmetric, $C_{ij} = C_{ji}$, and is commonly **used** as a **similarity measure** of two **papers** in **clustering** to group papers of similar topics together.

12.4.2 Bibliographic coupling

Bibliographic coupling operates on a similar principle, but in a way it is the **mirror image of co-citation**. Bibliographic coupling **links papers** that **cite the same articles** so that if papers i and j both cite paper k, they may be said to be related, even though they do not directly cite each other. The **more papers they both cite**, the **stronger** their **similarity** is. Picture 12.4.2 shows both papers i and j citing (referencing) paper k.

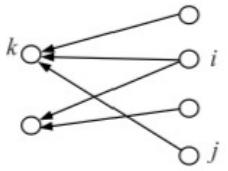


Figure 72: Bibliographic coupling

We use B_{ij} to represent the number of papers that are cited by both papers i and j:

$$B_{ij} = \sum_{k=1}^{n} L_{ik} L_{jk} = (LL^T)_{ij}$$

B_{ii} is naturally the **number of references** (in the reference list) of paper i. A **square matrix** B can be formed with B_{ij}, and it is called the **bibliographic coupling matrix**. Bibliographic coupling is also **symmetric** and is regarded as a similarity measure of two papers in clustering. We will see later that two important types of pages on the Web, hubs and authorities, found by the HITS algorithm are directly related to co-citation and bibliographic coupling matrices.

12.5 PageRank

The year 1998 was an important year for Web link analysis and Web search, since both the PageRank and the HITS algorithms were reported in that year. The main ideas of PageRank and HITS are really quite similar. However, it is their dissimilarity that made a huge difference as we will see later. Since that year, **PageRank** has emerged as the **dominant link analysis model** for Web search, partly due to its **query-independent evaluation of Web pages** and its **ability** to **combat spamming**, and partly due to Google's business success.
PageRank relies on the **democratic nature of the Web** by using its vast link structure as an indicator of an individual page's quality. In essence, PageRank interprets a **hyperlink** from page x to page y as a **vote**, by page x, for page y. However, PageRank looks at **more than just the sheer number of votes** or links that a page receives. It also analyzes the **page** that **casts** the vote. **Votes** casted by pages that are themselves **"important"** weigh more heavily and help to make other pages more "important." This is exactly the idea of rank prestige in social networks.

12.5.1 PageRank algorithm

PageRank is a **static ranking of Web pages** in the sense that a **PageRank** value is **computed for each page off-line** and it does **not depend on search queries**. Since PageRank is based on the measure of prestige in social networks, the **PageRank value** of each page can be regarded as its **prestige**.
From the perspective of prestige, we use the following to derive the PageRank algorithm:

1. A **hyperlink** from a **page** pointing **to** another **page** is an implicit conveyance of authority to the target page. Thus, the **more in-links** that a page i receives, the **more prestige** the page i has;
2. Pages that point to page i also have their own prestige scores. A page with a **higher prestige** score **pointing** to i is **more important** than a page with a **lower prestige** score pointing to i. In other words, a **page** is **important** if it is **pointed** to by other **important pages**.

According to rank prestige in social networks, the importance of page i (i's **PageRank score**) is **determined** by **summing up the PageRank scores of all pages that point to** i. Since a page may point to many other pages, its **prestige score should be**

shared among all the pages that it points to. Notice the difference from rank prestige, where the prestige score is not shared.
To formulate the above ideas, we treat the Web as a directed graph$G = (V, E)$, where V is the set of vertices or nodes, i.e., the set of all pages, and E is the set of directed edges in the graph, i.e., hyperlinks. Let the total number of pages on the Web be n (i.e., $n = |V|$). The PageRank score of the page i (denoted by $P(i)$) is defined by:

$$P(i) = \sum_{(j,i)\in E} \frac{P(j)}{O_j} \tag{1}$$

, where O_j is the number of out-links of page j, and it represents the "quantity" of prestige which is spread among the neighbors of i.
Mathematically, we have a system of n linear equations with n unknowns, so we can use a **matrix** to represent **all the equations**. Let P be a n-dimensional column vector of PageRank values, i.e.,

$$P = (P(1), P(2), \ldots, P(n))^T$$

Let A be the adjacency (usually sparse) matrix of our graph with

$$A_{ij} = \begin{cases} \frac{1}{O_i} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

We can write the system of n equations with as

$$P = A^T P \tag{2}$$

This is the **characteristic equation of the eigensystem**, where the **solution to P** is an **eigenvector** with the corresponding **eigenvalue of 1** ($\lambda = 1$). Since this is a **circular definition**, an **iterative algorithm is used to solve it**. It turns out that if **some conditions** are satisfied (which will be described shortly), **1 is the largest eigenvalue** and the **PageRank vector** P **is the principal eigenvector**. A well known mathematical technique called **power iteration** can be used to find P:

$$P_k = A^T P_{k-1}$$

Notice that the power iteration method stops when $P_k \approx P_{k-1}$.
However, the problem is that Equation 2 does not quite suffice because the Web graph does not meet the conditions. To introduce these conditions and the enhanced equation, let us derive the same Equation based on the **Markov chain**. In the Markov chain model, **each Web page** or node in the Web graph is **regarded as a state**. A **hyperlink** is a **transition**, which leads **from one state to another state with a probability**. Thus, this framework models **Web surfing as a stochastic process**. It models a Web surfer randomly surfing the Web as a state transition in the Markov chain. Recall that we used O_i to denote the number of out-links of a node i. Each **transition probability is** $1/O_i$ if we assume the Web surfer will click the hyperlinks in the page i **uniformly** at random, the "back" button on the browser is not used and the surfer does not type in an URL. Let A be the state transition probability matrix, a square matrix of the following format:

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix}$$

, where A_{ij} represents the **transition probability** that the **surfer** in **state** i (page i) will move to **state** j (page j).
Given an **initial probability distribution vector** that a surfer is at each state (or page) $p_0 = (p_0(1), p_0(2), \dots, p_0(n))^T$ (a column vector) and an $n \times n$ transition probability matrix A, we have

$$\sum_{i=1}^{n} p_0(i) = 1 \tag{3}$$

$$\sum_{j=1}^{n} A_{ij} = 1 \tag{4}$$

Equation 4 is not quite true for some Web pages because they have no out-links. If the matrix A satisfies Equation 4, we say that A is the **stochastic matrix of a Markov chain**. Let us assume A is a stochastic matrix for the time being and deal with it not being that later.
In a Markov chain, a question of common interest is: **given the initial probability distribution** p_0 at the beginning, what is the **probability** that m **steps**/transitions later that the **Markov chain will be at each state** j? We can determine the probability that the system (or the random surfer) is in state j after 1 step (1 state transition) by using the following reasoning:

$$p_1(j) = \sum_{i=1}^{n} A_{ij}(1) p_0(i)$$

where $A_{ij}(1)$ is the probability of going from i to j in 1 transition, and $A_{ij}(1) = A_{ij}$. We can write it with a matrix:

$$p_1 = A^T p_0$$

In general, the probability distribution after k steps is:

$$p_k = A^T p_{k-1} \tag{5}$$

By the **Ergodic Theorem of Markov chains**, a **finite Markov chain** defined by the stochastic transition matrix A has a **unique stationary probability distribution** if A **is irreducible and aperiodic**.
The **stationary probability distribution** means that **after a series of transitions** p_k **will converge to a steady-state probability vector** π **regardless of the choice of the initial probability vector** p_0, i.e.,

$$\lim_{k \to \infty} p_k = \pi$$

When we reach the steady-state, we have $p_k = p_{k+1} = \pi$, and thus $\pi = A^T\pi$. π is the **principal eigenvector** of A^T with **eigenvalue of 1**. In PageRank, π is used as the PageRank vector P. Thus, we obtain:

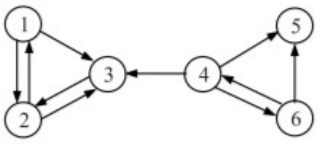
$$P = A^T P \tag{6}$$

Using the **stationary probability distribution** π as the **PageRank vector** is **reasonable** and quite **intuitive** because it reflects the long-run probabilities that a random surfer will visit the pages. A page has a high prestige if the probability of visiting it is high.

Now let us come back to the real Web context and **see whether the above conditions are satisfied**, i.e., whether A is a stochastic matrix and whether it is irreducible and aperiodic. In fact, none of them is satisfied. Hence, we need to **extend** the ideal-case Equation 6 to produce the **"actual PageRank model"**. Let us look at each condition below.

First of all, A is **not a stochastic** (transition) **matrix**. A stochastic matrix is the transition matrix for a finite Markov chain whose entries in each row are non-negative real numbers and sum up to 1. This requires that **every Web page must have at least one out-link**. This is **not true on the Web because many pages have no out-links**, which are reflected in transition matrix A by **some rows of complete 0's**. Such pages are called the **dangling pages (nodes)**.

Example: Dangling node in a graph: as we can see, Picture 12.5.1 shows that node 5 is a dangling node, and the corresponding transition matrix is not a stochastic matrix.



$$A = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix}.$$

Figure 73: Example of dangling node

We can fix this problem in several ways in order to convert A to a stochastic transition matrix. We describe only two ways here:

1. **Remove** those **pages** with **no out-links** from the system **during the PageRank computation** as these pages do not affect the ranking of any other page directly. Out-links from other pages pointing to these pages are also removed. After PageRanks are computed, these pages and hyperlinks pointing to them can be added in. Note that the **transition probabilities** of those pages with removed links will be **slightly affected** but not significantly;
2. **Add** a **complete set of outgoing links from each such page** i **to all the pages on the Web**. Thus the **transition probability** of going from i to every page is $1/n$ assuming uniform probability distribution. That is, **we replace each row containing all 0's** with e/n, where e is n-dimensional vector of all 1's.

$$\bar{A} = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 1/3 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix}.$$

Figure 74: Transformation of matrix A into $\bar{A}$

If we use the second method to make A a stochastic matrix by adding a link from page 5 to every page, we obtain a stochastic matrix:
Second, A is **not irreducible**. Irreducible means that the Web graph G is **strongly connected**, i.e. if and only if, for each pair of nodes $u, v \in V$, there is a path from u to v. A **general Web graph** represented by A is **not irreducible** because **for some pair of nodes** u and v, **there is no path** from u to v. For example, in Picture 12.5.1, there is no directed path from node 3 to node 4. The adjustment of adding a complete set of out-links is not enough to ensure irreducibility.
Finally, A is **not aperiodic**: a state i in a Markov chain being **periodic** means that **there exists a directed cycle that the chain has to traverse**. More formally, a state i is periodic with period $k > 1$ if k is the smallest number such that all paths leading from state i back to state i have a length that is a multiple of k. If a state is not periodic (i.e., k = 1), it is aperiodic. A **Markov chain** is **aperiodic** if **all states** are **aperiodic**.
Example: Periodic chain: Picture 12.5.1 shows a periodic Markov chain with $k = 3$. The transition matrix is given on the left. Each state in this chain has a period of 3. For example, if we start from state 1, to come back to state 1 the only path is 1-2-3-1 for some number of times, say h. Thus any return to state 1 will take $3h$ transitions.

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

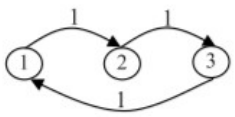


Figure 75: Example of periodic chain

It is easy to deal with the above **two problems** with a single strategy, called **teleportation**: we **add** a **link from each page to every page** and give **each link** a **small transition probability** controlled by a **parameter** d, with $0 < d < 1$.
In this way, the augmented transition matrix becomes **irreducible** because it is clearly **strongly connected**. It is also **aperiodic** because the situation in Picture 12.5.1 no longer exists as we now have **paths of all possible lengths from state** i **back to state** i. Finally, since the augmented transition matrix is both irreducible and aperiodic, we're ensured that it has a single stationary distribution.
After this augmentation, we obtain an **improved PageRank model**. In this model, at a page, the random surfer has two options:

1. With probability d, he randomly chooses an out-link to follow, i.e. it **clicks on a hyperlink**;

2. With probability $1 - d$, he jumps to a random page without a link.

Thus, the improved model is

$$P = ((1 - d)\frac{E}{n} + dA^T)P \tag{7}$$

where E is ee^T (e is a column vector of all 1's) and thus E is a $n \times n$ square matrix of all 1's. In this sense, E represents the adjacency matrix of a completely connected graph, and we multiply it by $(1 - d)/n$, which is the probability of clicking on a teleportation link.

Example: Improved PageRank model: Picture 12.5.1 shows the resulting matrix after applying the improved model described above. On the left we have the original matrix made stochastic, while on the left the resulting matrix. Notice that the original matrix is quite sparse, while the resulting one is dense. In this case the parameter $d = 0.9$.

$$\overline{A} = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 1/3 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix}. \quad (1-d)\frac{E}{n} + dA^T = \begin{pmatrix} 1/60 & 7/15 & 1/60 & 1/60 & 1/6 & 1/60 \\ 7/15 & 1/60 & 11/12 & 1/60 & 1/6 & 1/60 \\ 7/15 & 7/15 & 1/60 & 19/60 & 1/6 & 1/60 \\ 1/60 & 1/60 & 1/60 & 1/60 & 1/6 & 7/15 \\ 1/60 & 1/60 & 1/60 & 19/60 & 1/6 & 7/15 \\ 1/60 & 1/60 & 1/60 & 19/60 & 1/6 & 1/60 \end{pmatrix}$$

Figure 76: Resulting matrix

Notice that $(1-d)\frac{E}{n} + dA^T$ is a **stochastic matrix**, **irreducible and aperiodic**. Moreover:

$$P = ((1 - d)\frac{E}{n} + dA^T)P = (1 - d)\frac{1}{n}ee^T P + dA^T P$$

, since $E = ee^T$ and $e^T P = 1$, since P is a probability vector (i.e. it sums up to 1). For this reason, we obtain:

$$P = (1 - d)\frac{1}{n}e + dA^T P$$

, and if we scale this equation by multiplying both sides by n, and considering that $e^T P = n$, then:

$$P = (1 - d)e + dA^T P$$

This gives us the PageRank formula for each page i as follows:

$$P(i) = (1 - d) + d\sum_{j=1}^{n} A_{ji}P(j) = (1 - d) + d\sum_{(j,i)\in E}^{n} \frac{P(j)}{O_j}$$

The parameter d is called **damping factor**, $0 < d < 1$, and $d = 0.85$ was used in the paper.

The computation of PageRank values of the Web pages can be done using the well known **power iteration method**, which produces the principal eigenvector with the eigenvalue of 1. The algorithm is simple, and is given in Picture 12.5.1.

PageRank-Iterate(G)
$P_0 \leftarrow e/n$
$k \leftarrow 1$
repeat
$P_k \leftarrow (1-d)e + dA^T P_{k-1}$;
$k \leftarrow k+1$;
until $||P_k - P_{k-1}||_1 < \varepsilon$
return P_k

Figure 77: Power iteration method for PageRank

The iteration ends when the PageRank values do not change much or converge: in the Picture above, the iteration ends after the 1-norm of the residual vector is less than a pre-specified threshold ϵ. Note that the 1-norm for a vector is simply the sum of all the components.
Example: PageRank computation: Picture 12.5.1 shows an example of PageRank computation: in this case the PageRank equation is not scaled, so we have $e^T P = 1$ and thus $P(i) = \frac{1-d}{n} + d \sum_{(j,i) \in E} \frac{P(j)}{O_j}$.

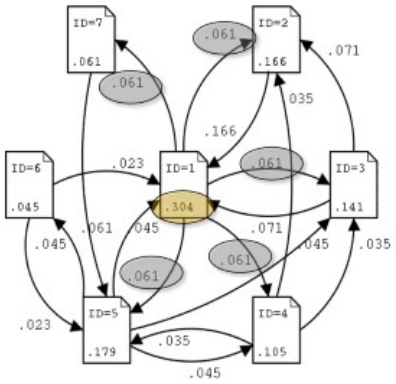


Figure 78: Example of PageRank

As we can see, the score of PageRank is written at the bottom of the page, e.g. $P(1) = 0.304$, and its score is distributed among all its 5 out-links $0.304 = 0.061 * 5$. In particular, the PageRank is obtained as $P(1) = \frac{1-d}{n} + d(0.023 + 0.166 + 0.071 + 0.045) = 0.304$, considering $d = 0.85$

12.5.2 Advantages and disadvantages of PageRank

The main **advantage** of PageRank is its **ability to fight spam**. A page is important if the pages pointing to it are important. Since it is **not easy** for Web page owner to **add in-links** into his/her page from other important pages, it is thus **not easy** to **influence PageRank**: nevertheless, there are reported ways to influence PageRank. Another major **advantage** of PageRank is that it is a **global measure** and is **query independent**. That is, the **PageRank values** of all the pages on the Web are **computed** and **saved off-line rather** than at the **query time**. At the **query time**, only a **lookup** is **needed** to find the value to be integrated with other strategies to rank the pages. It is thus **very efficient at the query time**. Both these two advantages contributed greatly to Google's success.

The main **criticism** is also the **query-independence** nature of PageRank. It **could not distinguish between pages that are authoritative in general** and pages that are **authoritative on the query topic**. Another **criticism** is that PageRank **does not consider time**, and we note again that the **link-based ranking** is **not** the **only strategy** used in a search engine. Many other information retrieval methods, heuristics, and empirical parameters are also employed.

12.6 HITS

HITS stands for **Hypertext Induced Topic Search**. Unlike **PageRank** which is a **static** ranking algorithm, **HITS** is **search query dependent**. When the user issues a search query, **HITS first expands the list of relevant pages returned by a search engine** and then **produces two rankings** of the expanded set of pages, **authority ranking** and **hub ranking**.
An **authority** is a **page** with **many in-links**. The idea is that the page may have **authoritative content** on some topic and thus many people trust it and thus link to it. A **hub** is a **page** with **manyout-links**. The page serves as an **organizer of the information** on a particular topic and points to many good authority pages on the topic. When a user comes to this hub page, he/she will find many useful links which take him/her to good content pages on the topic. Picture 12.6 shows an authority page and a hub page.

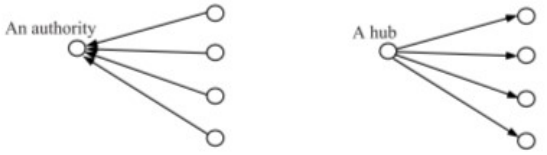


Figure 79: An authority and a hub page

The **key idea** of HITS is that a **good hub points** to **many good authorities** and a **good authority** is **pointed** to by **many good hubs**. Thus, authorities and hubs have a mutual reinforcement relationship. Picture 12.6 shows a set of densely linked authorities and hubs (a bipartite sub-graph).

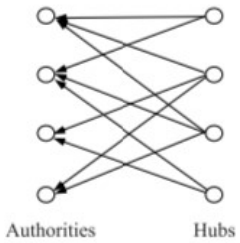


Figure 80: A densely linked set of authorities and hubs

Below, we first present the HITS algorithm, and also make a connection between HITS and co-citation and bibliographic coupling in bibliometric research. We then discuss the strengths and weaknesses of HITS, and describe some possible ways to deal with its weaknesses.

12.6.1 HITS algorithm

Before describing the HITS algorithm, let us first **describe** how **HITS collects pages** to be ranked. Given a broad search query, q, HITS collects a set of pages as follows:

1. It sends the **query** q to a **search engine system**. It then **collects** t ($t = 200$ is used in the HITS paper) **highest ranked pages**, which assume to be highly relevant to the search query. This set is called the **root set** W;
2. It then **grows** W by **including** any **page pointed to by a page** in W and **any page that points to a page** in W. This gives a larger set called S. However, this set can be very large. The algorithm **restricts its size** by allowing each page in W to bring at most k pages ($k = 50$ is used in the HITS paper) pointing to it into S. The set S is called the **base set**.

HITS then works on the pages in S, and **assigns every page** in S an **authority score** and a **hub score**. Let the number of pages to be studied be n. We again use $G = (V, E)$ to denote the (directed) link graph of S. V is the set of pages (or nodes) and E is the set of directed edges (or links). We use L to denote the adjacency matrix of the graph:

$$L_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Let the **authority score** of the page i be $a(i)$, and the **hub score** of page i be $h(i)$. The mutual reinforcing relationship of the two scores is represented as follows:

$$a(i) = \sum_{(j,i) \in E} h(j)$$

, i.e. the authority score is given by the sum of the hub scores of the incoming links, and

$$h(i) = \sum_{(i,j) \in E} a(j)$$

, i.e. the hub score is given by the sum of the authority scores of the outgoing links.
Writing them in the **matrix form**, we use a to denote the column vector with all the authority scores, $a = (a(1), a(2), \ldots, a(n))^T$, and use h to denote the column vector with all the hub scores, $h = (h(1), h(2), \ldots, h(n))^T$:

$$a = L^T h \tag{1}$$

$$h = La \tag{2}$$

The computation of authority scores and hub scores is basically the same as the computation of the PageRank scores using the **power iteration method**. If we use a_k and h_k to denote authority and hub scores at the k-th iteration, the iterative processes for generating the final solutions are:

$$a_k = L^T L a_{k-1} \tag{3}$$

and

$$h_k = LL^T h_{k-1} \tag{4}$$

starting with $a_0 = h_0 = (1, 1, \ldots, 1)$. After each iteration, the **values** are also **normalized** (to keep them small) so that

$$\sum_{i=1}^{n} a(i) = 1$$

and

$$\sum_{i=1}^{n} h(i) = 1$$

The algorithm is shown in Picture 12.6.1.

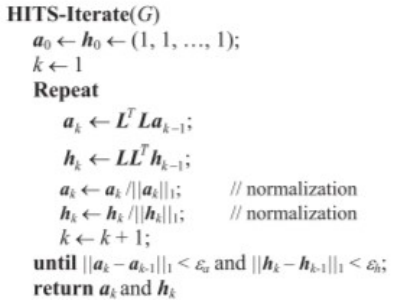


Figure 81: HITS algorithm

The iteration ends after the 1-norms of the residual vectors are less than some thresholds ϵ_a and ϵ_h. The pages with large authority and hub scores are better authorities and hubs respectively. HITS will select a few top ranked pages as authorities and hubs, and return them to the user.
Although **HITS** will **always converge**, there is a problem with **uniqueness of limiting (converged) authority and hub vectors**. It is shown that for certain types of graphs, **different initializations** to the power method **produce different final authority and hub vectors**.

12.6.2 Relationships with co-citation and bibliographic coupling

Authority pages and hub pages have their matches in the bibliometric citation context. An **authority page** is like an **influential research paper** (publication) which is cited by many subsequent papers. A **hub page** is like a **survey paper** which cites many other papers (including those influential papers). It is no surprise that there is a connection between authority and hub, and co-citation and bibliographic coupling.
Recall that co-citation of pages i and j, denoted by C_{ij}, is computed as

$$C_{ij} = \sum_{k=1}^{n} L_{ki} L_{kj} = (L^T L)_{ij}$$

This shows that the **authority matrix** $(L^T L)$ of **HITS** is in fact the **co-citation matrix** C in the **Web context**. Likewise, recall that bibliographic coupling of two pages i and j, denoted by B_{ij}, is computed as

$$B_{ij} = \sum_{k=1}^{n} L_{ik} L_{jk} = (LL^T)_{ij}$$

, which shows that the **hub matrix** (LL^T) of **HITS** is the **bibliographic coupling matrix** B in the **Web context**.

12.6.3 Advantages and disadvantages of HITS

The main **strength** of **HITS** is its **ability to rank pages according to the query topic**, which may be able to provide more relevant authority and hub pages. The **ranking** may also be **combined** with **information retrieval based rankings**.
However, HITS has several **disadvantages**.

1. It **does not have the anti-spam capability of PageRank**. It is quite **easy to influence** HITS by **adding out-links from one's own page to point to many good authorities**. This **boosts** the **hub score** of the page. Because hub and authority scores are interdependent, it in turn **also increases the authority score** of the page;
2. Another problem of HITS is **topic drift**. In **expanding the root set**, it can easily **collect** many **pages** (including authority pages and hub pages) which have **nothing to do the search topic** because out-links of a page may not point to pages that are relevant to the topic and in-links to pages in the root set may be irrelevant as well because people put hyperlinks for all kinds of reasons, including **spamming**;
3. The **query time evaluation** is also a major drawback. Getting the root set, expanding it and then performing eigenvector computation are all **time consuming operations**.

13 Learning to rank

13.1 Introduction

Thus far, we've looked at methods for ranking documents (e.g. cosine similarity, IDF, proximity, TF, PageRank etc..), but we've also looked at methods for classifying documents using supervised ML classifiers (e.g. Naive Bayes, Rocchio's algorithm, kNN). Now the question is: can we also **use ML** to **rank** the **documents** displayed in search results? This is the problem addressed by **learning to rank**, or **LTR**.
Some issues some years ago:

- **Limited training data**: especially for real world use, it was very hard to gather test collection queries and relevance judgements that are representative of real user needs and judgements on documents returned;
- **Poor ML techniques**;
- **Insufficient customization** to IR problem;
- **Not enough features** for ML to show value;

In this sense, traditional ranking functions in IR used a very small number of features, e.g. TF, IDF and document length.. In modern systems, especially on the Web, many features are used, e.g. log frequency of the query word in anchor text, number of images on a page, number of (out)links on a page, URL length (because if the URL is long, than probably the page is not important), BM25, URL click count etc..

13.2 LTR

The usual **LTR framework** is composed of:

- A large training set, composed of queries and ideal document ranking. In particular, for each query, a list of (document,relevant score) is provided;
- A set of features that are used for training the ML model. Many types of features exist:
  - Query-only features, i.e. query features with the same value for each document, e.g. query type, query length;
  - Query-dependent features, i.e. document features that depend on the query, e.g. query-URL click count, BM25 etc..;
  - Query-independent features, i.e. document features with the same value for each query, e.g. PageRank, URL length etc..
- A ML model, usually a regressor, that produces a final ranked list of documents. Notice that even if the ML models are regressors that score each candidate document, the goal is to guess the overall ranking, not the specific relevance judgement labels.

, as shown in Picture 13.2.
Picture 13.2, on the other hand, shows the pipeline of the LTR.

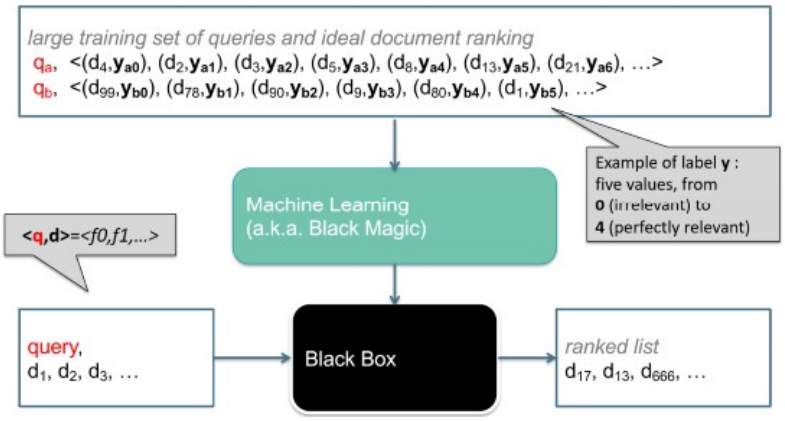


Figure 82: LTR framework

- In the first step, the base ranker produces a first ranking of the documents, exploiting cosine similarity, BM25 etc..;
- Then, this ranking is provided as feature to the LTR algorithm, which in turn is used to build a top ranker (which may be composed as a cascade of rankers). Finally, this top ranker provides the user a top-k documents for the given query.

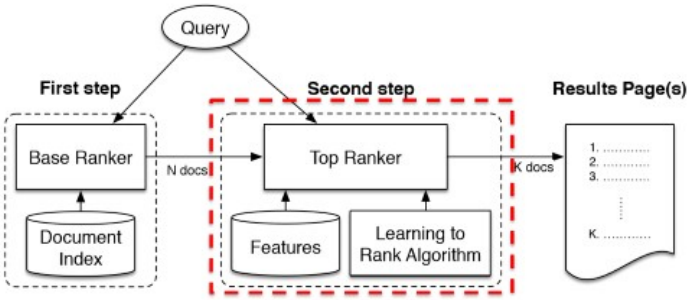


Figure 83: LTR pipeline

Notice that budget considerations are very important for commercial Web Search Engines, so the computational cost of LTR model must be strictly accounted in the time budget available for processing queries in the incoming stream.In other words, the step of the ranking pipeline involving the LTR model must be as efficient as possible.

13.2.1 LTR approaches

There exist many approaches for LTR framework:

- **Pointwise**: in this case **each** query-document **pair** is associated with a **score**, and the **objective** of the ML model is to **predict** such **score**. In this sense, it can be considered ad a pure **regression problem** (if the score is "continuous"), or a **multi-class classification problem** (if the score is "discrete"). Notice that this approach does not consider the position of a document into the final ranking;

- **Pairwise**: in this case we're given **pairwise preferences**, e.g. d_1 is better than d_2 for a query q, so the **objective** is to **predict** a **score** that **preserves such preferences**. In this sense, it can be considered as a **binary classification problem**. Notice that this approach does not consider the relevance of a document into the final ranking;
- **Listwise**: finally, in this case we're given the **ideal ranking** of results for each query, so the **objective** is to **maximize the quality of the whole resulting ranked list** by exploiting the whole list at training time. This approach is also used in recommendation systems.

In general, a modern ranking architecture should be:

- **Effective**, i.e. the users should be happy of the results they receive;
- **Efficient**, i.e. it should have a low response time (<0.1 sec);
- **Easy to adapt**, i.e. it should perform a continuous crawling of the Web, exploiting users' feedback.

In this sense, the state-of-the-art is represented by **Additive Regression Trees** and **LambdaMART** models, which will be presented in the following Sections.
We now focus on the *pointwise* and *pairwise* approaches.

13.2.2 Pointwise approach

We now discuss a first example of classification for IR.
In this case the **training set** is composed of (q, d, r) triples, where r represents the relevance of each document w.r.t. a query, and it is a binary score, i.e. *relevant* or *not relevant*. Moreover, each document is represented by a feature vector $x = (\alpha, \omega)$, where:

- α represents the cosine similarity;
- ω represents the minimum query window size, i.e. the shortest text span that includes all the query words, regardless of the order.

In this sense, the goal of the classifier is to predict the relevance r of a new document-query pair. A linear score function is :

$$\text{score}(q, d) = \text{score}(\langle \alpha, \omega \rangle) = a\alpha + b\omega$$

, so the linear classifier has to determine the parameters a, b and θ to decide that a document is *relevant* if $\text{score}(q, d) > \theta$, and *not relevant* if $\text{score}(q, d) \leq \theta$. This methodology is usually referred as **pointwise ranking**. A visual representation of such classifier is provided in Picture 13.2.2.

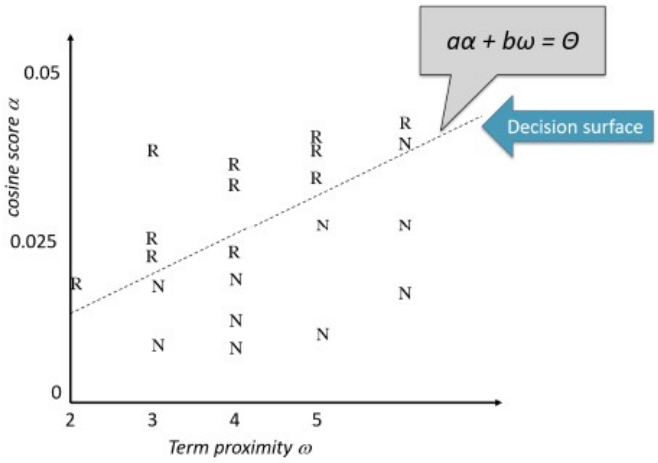


Figure 84: Linear classifier

13.3 BM25 and BM25F

BM25 is a *probabilistic* ranking function, *state-of-the-art* method that exploits term independence assumption to **approximate** the **probability** of a **document** of being **relevant**. The BM25 score is define as:

$$\text{score}_{\text{BM25}}(q,d) = \sum_{q_t \in q} TF_{\text{BM25}}(q_t,d) \cdot IDF(q_t)$$

, where:

- $IDF(q_t) = \log(\frac{|D|}{DF(q_t)})$. Notice that since $DF(q_t)$ is in the denominator, the more documents contain a term, i.e. the more frequent a term is, the less contribution the term given to the score;
- $TF_{\text{BM25}}(q_t,d) = \frac{TF(q_t,d)}{k+1-b+b\frac{l_d}{L}}$, where l represents the length of document d, and L the average document length in the collection. Notice that the parameter b determines the importance of l_d w.r.t. L.

  In general, if $l_d = L$, then $TF_{\text{BM25}}(q_t,d) = TF(q_t,d)$; if $l_d >> L$, for example $l_d = 2L$, then $TF_{\text{BM25}}(q_t,d) = \frac{TF(q_t,d)}{1+b}$, so we penalize more the contribution of the term frequency. In other words, $\frac{l_d}{L}$ penalizes or awards the term frequency following the schema of Picture 13.3.

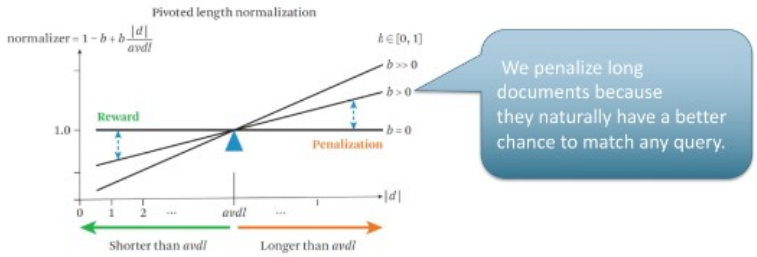


Figure 85: Parameter b

BM25 can be extended to handle *structure documents*, i.e. multi-field documents, where a field could be the title, the abstract, the body etc.. The extended function is called **BM25F**, and it is define as:

$$\text{score}_{\text{BM25F}}(q,d) \sum_{q_t \in q} TF_{\text{BM25F}}(q_t,d) \cdot IDF(q_t)$$

, where

$$TF_{\text{BM25F}} = \sum_s \frac{w_s \cdot TF(q_t,s)}{1 - b_s + b_s \frac{l_s}{L_s}}$$

, where:

- s represents a field;
- w_s is the weight of field s (of document d);
- l_s is the length of field s (of document d);
- b_s determines the importance of l_s;
- L_s is the average length of field s in the collection.

We notice that while BM25 has two parameters, b and k, BM25F has $2S+1$ free parameters, where S is the total number of fields: indeed, we have b_s and w_s for each of the S fields, plus the parameter k. Thus, in order to find the best parameters of BM25(F), we can fit it into a LTR framework.

13.4 LTR for BM25(F)

In general, the three main ingredients for exploiting ML in our system are:

- An **evaluation function**, i.e. a method for evaluating our results;
- The **ML algorithm**, and in particular the hypothesis space (linear combination, mixture models, etc..) and the optimization strategy (greedy, bounded approximation etc..);
- The **data**.

13.4.1 Using NDCG

In the case of BM25(F), we consider:

- The *NDCG@k* as evaluation function;
- As ML model, we must search over all the possible BM25(F) functions, using a gradient descent strategy over NDCG@k;
- The data are composed of a set of query/results, where each query has a set of candidate results, each of which was manually annotated with a relevance label (e.g. from 0 to 4).

In particular, given a query q and a set of documents $D = \{d_1, d_2, ..\}$, we want to learn a model h that allows to score and then rank the documents in D according to their relevance. In this sense, the results will be sort$\{h(d_1), h(d_2), ..\}$, where the model h is BM25F with a proper parameter set θ.
However, if we want to apply the gradient descent strategy, we would need to compute the gradient of the sort function w.r.t. θ, but sort is not a continuous and differentiable function, so we cannot apply this strategy.
The first solution to this problem is to optimize by **line search**, i.e. from a point in the parameter space, we perform a line search along each individual parameter co-ordinate axis, i.e. we try many values for each parameters, while the other are kept fixed. In our case, this means that NDCG is computed for each of the N sample points along the line, and the location of the best NDCG is recorded.

13.4.2 Pairwise approach

However, since NDCG cannot be directly optimized, we can change completely the optimization strategy by considering **pairwise** ordering. In this case we're given a collection R of document pairs (d_i, d_j), and for each pair we know that d_i is more relevant that d_j: our goal is to find the **best ranking function** r^* s.t. for each pair $(d_i, d_j) \in R$, $r^*(d_i) < r^*(d_j)$, where smaller ranks are better. In other words, the goal is to find the ranking function s.t. the smallest number of such constraints is violated. We now consider the **RankNet** approach.
Let the **training set** be **result pairs** (d_1, d_2) where d_1 is better than d_2 (we also say that the *true probability* that d_1 is better than d_2 is $T_{12} = 1$). Let $h(d)$ be the **score** of document d computed by the learned model: obviously, the higher the score, the better, and the document ranking is obtained by sorting the documents in decreasing order of $h(d)$. We then define the **score difference** $Y = h(d_1) - h(d_2)$: if $Y > 0$, then the documents are ranked correctly. We now **map** Y to the probability P_{12} that d_1 is better than d_2 with a **sigmoid function**:

$$P_{12} = \frac{1}{1 + e^{-Y}}$$

We now measure the **error of the model** using the **cross entropy** between P_{12} and T_{12}:

$$C_{12} = -T_{12} \log P_{12} - (1 - T_{12}) \log(1 - P_{12})$$

Notice that:

- The **cross entropy** is used to define a **loss function** in ML and optimization;
- Thus, in this case, it aims to **minimize** the number of inversions in ranking.

However, since we take the pair ordering of the training set as certain , then $T_{12} = 1$, so the loss function reduces to:

$$C_{12} = \log(1 + e^{-Y})$$

In this sense, the overall RankNet loss is defined as:

$$C = \sum_{d_i, d_j} C_{ij} = \sum_{d_i, d_j} \log(1 + e^{-Y})$$

We have that C is **minimum** if all pairs are ranked in the proper order, therefore my minimizing C we aim at **improving NDCG**. Notice that now C is a continuous function, so we can compute its **gradient** and directly apply **steepest descent**.
Thus, in this case the LTR framework is composed of:

- A proxy of the *NDCG@k* evaluation function, the *RankNet cost*;
- As a ML model, we still must search over all the possible BM25(F) functions, using a gradient descent strategy over RankNet cost;
- The data the same as before.

The results showed that this alternative formulation using the RankNet cost is a good proxy for NDCG optimization. However, the **pairwise approach** has some **disadvantages**. We know that its goal is to maximize the number of correctly classified pairs, but in general the number of document pairs violations might not be a good indicator, since errors that occurs at the bottom of the ranking are not the same as errors at the top of the ranking.

13.5 LambdaMART

13.5.1 Decision Tree and Regression Tree

We start our discussion by introducing the concept of **decision tree**. A decision tree is a tree-like structure in which each **internal node** denotes a **test** over an attribute/feature, and each **leaf node** is associated with a **class label**, in case of a classification task, or a **predicted value**, in case of a regression task. A visual representation of a decision tree is provided in Picture 13.5.1.
Then, the decision tree is used to label a new data instance on the basis of its structure: it runs several tests on the data instance attributes and it traverses the tree according to the tests results. The label associated to the leaf represents the prediction of the decision tree.
In case of a regression task, a **regression tree** is used: a regression tree is a tree in which each **internal node** is a predicate on some feature, and a **leaf** is a prediction. Given a set of features $X = \{X_1, X_2, ..\}$, the goal is to find the tree that best predicts Y on the training data. An example is provided in Picture 13.5.1.
Notice that in both the decision tree and the regression tree, each node induces a partitioning/splitting of the data.
Now the question is: how do we build a decision/regression tree? We must recursively decide the best feature for which performing a split of the dataset. The **best split** can be decided using a **greedy strategy**: for each possible candidate, i.e. for each **splitting criterion**, compute the prediction for the left and right child, where the prediction is given by the average of the target variable on the corresponding instances. Then, the **goodness of the split** is computed as an error reduction (usually MSE), and the split with best **error reduction** is chosen. Then, the data are splitted according to the chosen split criterion, and the operations are repeated recursively on the sub-trees.

age	income	student	credit_rating	buys_computer
<=30	high	no	fair	no
<=30	high	no	excellent	no
31…40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31…40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31…40	medium	no	excellent	yes
31…40	high	yes	fair	yes
>40	medium	no	excellent	no

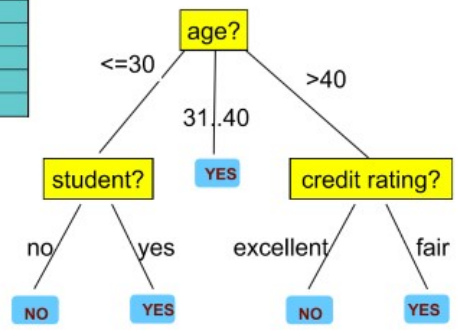


Figure 86: Example of decision tree

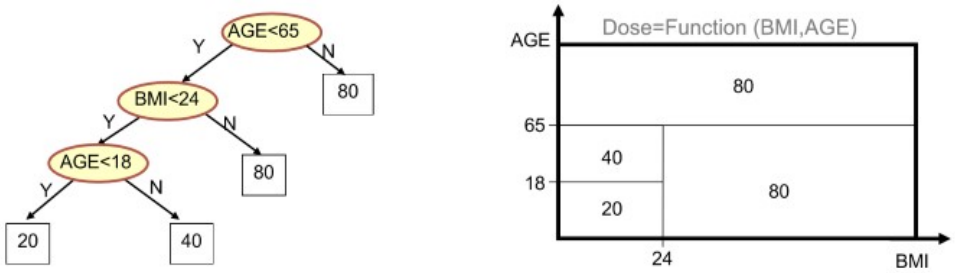


Figure 87: Example of regression tree

The **advantage** of a decision/regression tree is that it is very **explainable**, in the sense that a single prediction/classification can be easily analyzed by traversing the tree, but a **disadvantage** is that it is not very powerful for some tasks, and it is also prone to **overfitting**.

13.5.2 Ensemble of trees

In order to overcome the problems of a single decision/regression tree, an **ensemble method** can be exploited. An ensemble is a combination of trees that is able to combine their predictive power by reducing the overfitting problem. A famous technique for building an ensemble is **bagging**, which essentially performs a bootstrap operation and an aggregation.
An example of ensemble of trees is a **random forest**, which is a collection of randomly created decision trees, in which each test node in a decision tree works on a random subset of the features, and the final forest then combines the output of individual decision trees to generate the final output, e.g. by averaging the trees' leave values. A representation is provided in Picture 13.5.2.

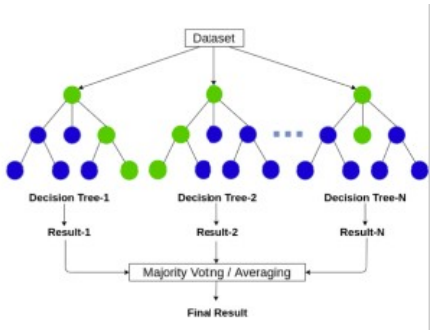


Figure 88: Example of Random Forest

13.5.3 MART

MART (Multiple Additive Regression Trees) is an **ensemble of trees** adopting **gradient boosting**. A gradient-boosted tree model is built on a stage-wise fashion as in other boosting methods:

- The model iteratively learns a weak learner, e.g. a shallow decision tree;
- Then, it adds this weak learner to the final strong classifier, i.e. to the ensemble;
- The next weak learner will focus more on mis-classified data items.

In this sense, MART is also known as **Gradient Boosting Regression Tree** (**GBRT**). From a mathematical point of view, given a ground truth $D = \{(x_1, y_1), .., (x_{|D|}, y_{|D|})\}$, where $x \in \mathbb{R}^{|F|}$ (in our case $x = (\text{query}, \text{doc})$ and $y_i = \text{score}$), GBRT learns incrementally an ensemble of m trees predicting the various y_i:

$$F_m(x) = F_{m-1}(x) + v \cdot h_m(x)$$

, where:

- $F_{m-1}(x)$ represents the summation of all the outputs of the trees in the ensemble so far;
- $h_m(x)$ represents the output of the current tree;
- v is a **shrinkage factor** or learning rate, which act as a regularization factor by shrinking the minimization step along the steepest descent direction.

Notice that $h_m(x)$ is sufficiently easy to be learnt, since it represents the output of a relatively shallow tree, and its goal is to reduce the error/cost function of $F_{m-1}(x)$ learned so far. When the method starts, h_0 represents the initial guess, e.g. the average of the various y_i.
The function $F(x)$ learned by GBRT minimizes a loss function that is averaged over all x_i, usually the mean squared error:

$$L(y_i, F(x_i)) = \frac{1}{2}(y_i - F(x_i))^2$$

Then, since we exploit the steepest descent method, we must compute the negative gradient:

$$-g_m(x_i) = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}|_{F(x)=F_{m-1}(x)} = y_i - F_{m-1}(x_i)$$

Intuitively, to improve our prediction for x_i, we have to fit a new regression tree h_m whose output is $y_i - F_{m-1}(x_i)$, which are called **residuals**.
Now, the question is: can we optimize NDCG rather than MSE? No, since we cannot compute the derivative function of NDCG. However, we showed that RankNet cost represents a good proxy for that measure: for this reason, we now introduce Lambda-Rank.

13.5.4 Lambda-Rank

Lambda-Rank is a GBRT that optimizes directly the derivable **RankNet pairwise loss**, i.e. it provides a new way for computing the residuals. In this case, the **gradients** λ of the cost w.r.t. the model score can be seen as **little arrows** attached to each document in the ranked list, indicating the direction we would like those documents to move to reduce the number of **pairwise violations**. In particular, we have that:

$$\lambda_{ij} = \frac{\partial C(s_i - s_j)}{\partial s_i} = \frac{1}{1 + e^{s_i - s_j}}$$

, where s_i and s_j are the scores produced by the model that we're training, i.e. $s_i = F_{m-1}(x_i)$ and $s_j = F_{m-1}(x_j)$.
Then, we assign a gradient λ_i to each document d_i for a given query q:

$$\lambda_i^q = \sum_{j:(i,j)\in I^q} \lambda_{ij}^q - \sum_{j:(j,i)\in I^q} \lambda_{ji}^q$$

, where $I^q = \{(i,j) : y_i^q > y_j^q\}$, i.e. the set of all the document pairs (i,j) ranked according to the relevance judgements y_i and y_j for a given query q.
In this sense, if we think of each λ as a force applied to documents, the first sum accounts for all the forces coming from less relevant documents, which therefore pushes d_i up in the ranking; on the other hand, the second sum, finally subtracted, accounts for all the forces coming from more relevant documents, which therefore pushes d_i down in the ranking. A visual representation of this behaviour is provided in Picture 13.5.4.

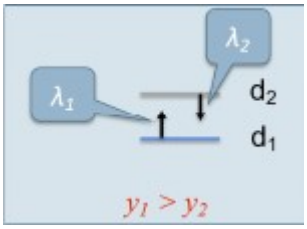


Figure 89: λ_i as a force

As we can see, in this case $y_1 > y_2$, i.e. the relevance judgement of y_1 is greater than the relevance judgement of y_2, but the ranking is different. For this reason, λ_1 forces d_1 up in the ranking, while λ_2 forces d_2 down in the ranking.
In general, Lambda-Rank is a good model, since it tries to minimize the pairwise violations, but still is not able to directly optimize NDCG. For this reason, we now discuss about Lambda-MART.

13.5.5 Lambda-MART

Lambda-MART uses an approximation of NDCG gradient. In particular, for each training instance x_i, Lambda-MART estimates the **benefit of increasing or decreasing** the currently predicted score $F_{m-1}(x_i)$ in terms of NDCG.
The new λ-gradient λ_i is computed as a before, but we multiply each λ_{ij} by $|\Delta NDCG_{ij}|$: after sorting by score the documents for each query q, for each pair (i, j) where the i-th document is more relevant the the j-th, i.e. $(i, j) \in I^q$, we compute $|\Delta NDCG_{ij}|$. Indeed, $|\Delta NDCG_{ij}|$ is the change in terms of NDCG given by swapping the rank positions of d_i and d_j, while leaving the other documents untouched.
In this sense, the new residuals are:

$$\lambda_{ij} = \frac{1}{1 + e^{s_i - s_j}} \cdot |\Delta NDCG|$$

Picture 13.5.5 provides a comparison between plain RankNet λ's (black arrows) and Lambda-MART one's (red arrows); notice that in the Picture, the blue indicates a relevant document.

Figure 90: Comparison between plain RankNet λ's and Lambda-MART one's

In the left we see a ranking that is very good in terms of NDCG, despite having a higher number of pairwise violations (13), while in the right we notice how RankNet λ's try to increase the score of the second relevant document more than Lambda-MART, since its goal is to minimize the number of pairwise violations. Moreover, we notice how the first relevant document is heavily pushed up by Lambda-MART, since optimizing NDCG, its goal is to have relevant documents in the first positions.
As a resume, in Lambda-MART framework is composed of:

- The *NDCG@k* as evaluation function;
- As ML model, mu must search over a sum of regression trees, where each tree is fitted w.r.t. new residuals, and injecting the evaluation function into the gradient of RankNet's cost;
- The data are the same as before.