

CHAPTER ONE: INTRODUCTION TO INFORMATION RETRIEVAL

The main contents of this chapter are the following. Therefore, the Student must understand the concept of the following key terms and must have the ability to answer the listed questions after the end of the chapter.

- 1.1. What are the main difference between data and Information?
- 1.2. What is storage?
- 1.3. What is retrieval?
- 1.4. What is information retrieval (IR)?
- 1.5. What is information storage and retrieval (ISR)?
- 1.6. What are IR systems?
- 1.7. What are the basic structures of IR systems?

1.1. Data and information

Data can be defined as a representation of facts, concepts, or instructions in a formalized manner, which should be suitable for communication, interpretation, or processing, by human or electronic machines. It is raw fact, unprocessed information, it simply exists and has no full meaning (does not have meaning of itself). It can exist in any form, usable or not. It can be described as unprocessed facts and figures. It is represented with the help of characters such as alphabets (A-Z, a-z), digits (0-9) or special characters (+, -, /, *, , =, etc.). Whereas information is the processed data on which decisions and actions are based. It is data that has been processed into a form that is meaningful to the recipient and is of real or perceived value in the current or the prospective action or decision of the recipient. Furtherer, **information** is interpreted data; created from organized, structured, and processed data in a particular context. Or it is data that have been processed and has meaning of itself and the meaning is useful but does not have to be.

Example: - The temperature dropped 15 degrees and then it started raining.

2009 is the year that I never forget because of its event, especially in Ethiopia.

1.2. Information Storage and Retrieval (ISR)

Storage: The action of or method of storing something. The place where data is held in an electromagnetic or optical for access by a computer processor.

Retrieval: The process of getting something back from somewhere. The action of obtaining or consulting material stored in a computer system.

Example: find 'BRUTUS AND CAESAR AND NOT CALPURNIA' in the big book of shakespeare.

Information storage: The computers can store different types of information in different ways, depending on what the information is, how much storage it requires and how quickly it needs to be accessed.

Information retrieval

Information Retrieval (IR) is 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 technology has been central to the success of the Web.

Information Retrieval is the process of obtaining relevant information from a collection of informational resources. It does not return information that is restricted to a single object collection but matches several objects which vary in the degree of relevancy to the query. So, we have to think about what concepts IR systems use to model this data so that they can return all the documents that are relevant to the query term and ranked based on certain importance measures. These concepts include dimensionality reduction, data modeling, ranking measures, clustering etc.

These tools that IR systems provide would help you get your results faster. So, while computing the results and their relevance, programmers use these concepts to design their system, think of what data structures and procedures are to be used which would increase speed of the searches and better handling of data.

Generally, Information retrieval, as the name implies, concerns the retrieving of relevant information from the collection of information. It is basically concerned with facilitating the user's access to large amounts of (predominantly textual) information. The process of information retrieval involves the following stages:

- ✓ Representing Collections of Documents - how to represent, identify and process the collection of documents.
- ✓ User-initiated querying - understanding and processing of the queries.

✓ Retrieval of the appropriate documents - the searching mechanism used to obtain and retrieve the relevant document

Basic assumptions of Information Retrieval

- **Collection:** A set of documents

— Assume it is a static collection for the moment

- **Goal:** Retrieve documents with information that is relevant to the user's information need and helps the user complete a task

What is information storage and retrieval?

Since the 1940s the problem of information storage and retrieval has attracted increasing attention. It is simply stated: we have vast amounts of information to which accurate and speedy access is becoming ever more difficult. In libraries, many of which certainly have an information storage and retrieval problem, some of the more mundane tasks, such as cataloguing and general administration, have successfully been taken over by computers. However, the problem of effective retrieval remains largely unsolved. Suppose there is a store of documents and a person (user of the store) formulates a question (request or query) to which the answer is a set of documents satisfying the information need expressed by his question. He can obtain the set by reading all the documents in the store, retaining the relevant documents and discarding all the others. In a sense, this constitutes 'perfect' retrieval. This solution is obviously impracticable. A user either does not have the time or does not wish to spend the time reading the entire document collection, apart from the fact that it may be physically impossible for him to do so.

Information storage & Retrieval (IS&R) deals with the representation, storage, organization, and access to information items to satisfy user information needs. In other terms, Information storage & retrieval is the process of searching for relevant documents from an unstructured large corpus that satisfy the user's information need. And it is a tool that finds and selects from a collection of items a subset that serves the user's purpose.

Information storage and retrieval, is the systematic process of collecting and cataloging data so that they can be located and displayed on request. There are several basic types of information-storage-and-retrieval systems. **Document-retrieval** systems store entire documents, which are

usually retrieved by title or by key words associated with the document. In some systems, the text of documents is stored as data. This permits full text searching, enabling retrieval on the basis of any words in the document. In others, a digitized image of the document is stored, usually on a write-once optical disc.

Database systems store the information as a series of discrete records that are, in turn, divided into discrete fields (e.g., name, address, and phone number); records can be searched and retrieved on the basis of the content of the fields (e.g., all people who have a particular telephone area code). The data are stored within the computer, either in main storage or auxiliary storage, for ready access. **Reference-retrieval** systems store references to documents rather than the documents themselves. Such systems, in response to a search request, provide the titles of relevant documents and frequently their physical locations. Such systems are efficient when large amounts of different types of printed data must be stored. They have proven extremely effective in libraries, where material is constantly changing.

Generally, Information storage & retrieval is the science of searching for documents, for information within documents, as well as that of searching relational databases and the World Wide Web.

INFORMATION RETRIEVAL (IR) SYSTEM ARCHITECTURE

Before an information retrieval system can actually operate to retrieve some information the information must have already been stored inside the system this is true both for manual and computerized systems.

An IR system accepts a query from a user and responds with a set of documents. The system returns both relevant and non-relevant material and a document organization approach are applied to assist the user finding the relevant information in the retrieved set.

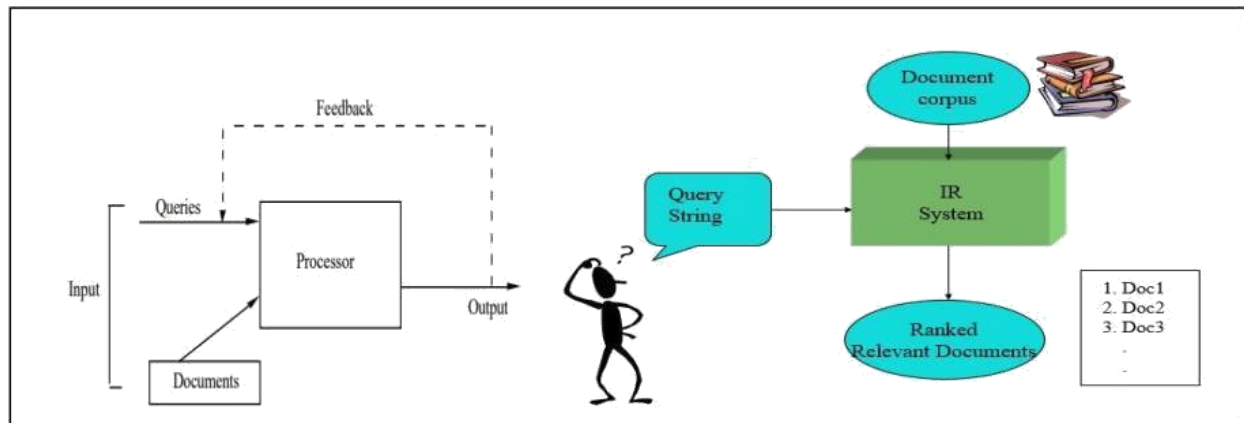


figure 1 IR SYSTEM ARCHITECTURE

Figure 1: is the diagram of a typical Information Retrieval System. It shows three main components: input, processor, and output.

From the **input side**, the need is to obtain a representation of each data and query suitable for a computer to use. Let me emphasize that Most computer-based retrieval systems store only a representation of the data (or query) which means that the text of a document is misplaced once it has been processed for the purpose of making or generating its representation.

A document representative might, for instance, be a list of extracted texts considered to be significant. Rather than have the computer process the natural language, an alternative approach is to have an artificial language within which all queries and documents can be formulated. **The processor** is that part of the retrieval system concerned with the retrieval process. The process may involve structuring the information in some appropriate way, such as classifying it. It will also involve performing the actual retrieval function that is, executing the search strategy in response to a query. In the diagram, the documents have been placed in a separate box to emphasize the fact that they are not just input but can be used during the retrieval process in such a way that their structure is more correctly seen as part of the retrieval process. The **output** is usually a set of citations or document numbers. In an operational system, the story ends here. However, in an experimental system, it leaves the evaluation to be done. Generally, in a typical Information Retrieval System.

The given is

- A corpus of textual natural-language documents.
- A user query in the form of a textual string

The **output** is the finding of a ranked set of documents that are relevant to the query. And there is a process in the middle.

Generally speaking, IR systems:

- Are systems which are build to retrieve documents highly likely relevant to the user
- Are systems built to reduce user's workload in searching, through the store of documents to find relevant one's
- Are systems that give information about the presence or absence of documents in accordance with the query

— Automated abstracts or summaries of documents were developed to further simplify access to search results

The main importance of IR systems are:

a. **Regulatory Compliance**

A well-organized information storage and retrieval system that follows compliance regulations and tax record-keeping guidelines significantly increases a business owner's confidence the business is fully complying.

b. **Efficiency and Productivity**

Any time a business owner or employees spend searching through stacks of loose files or spend trying locate missing or misfiled records is inefficient, unproductive and can prove costly to a small business. A good information storage and retrieval system, including an effective indexing system, not only decreases the chances information will be misfiled but also speeds up the storing and retrieval of information. The resulting time-saving benefit increases office efficiency and productivity while decreasing stress and anxiety.

c. **Improve Working Environment**

It can be disheartening to anyone walking through an office area to see vital business documents and other information stacked on top of file cabinets or in boxes next to office workstations. Not only does this create a stressful and poor working environment, but if customers see this, can cause customers to form a negative perception of the business. Contrast this with an office area in which file cabinets, aisles and workstations are clear and neatly organized to see how important it is for even a small business to have a well-organized information storage and retrieval system.

Electronic vs. Manual for IR

Although a very small business may choose to institute a manual system, the importance of electronic information storage and retrieval systems lie in the fact that electronic systems reduce storage space requirements and decrease equipment and labor costs. In contrast, a manual system requires budgetary allotments for storage space, filing equipment and administrative expenses to maintain an organized filing system. Additionally, it can be significantly easier to provide and monitor internal controls designed to deter fraud, waste and abuse as well as ensure the business is complying with information privacy requirements with an electronic system.

Major functions of IR systems

- Analyze contents of information items
- Represent the contents of the analyzed sources in a way suitable for matching with users' queries
- Analyze users information need and represent them in a form that will be suitable or matching with the database
- Match the search statement with the stored database
- Retrieve or generate information that are relevant in a ranking which reflects relevance
- Make necessary adjustments in the system based on feedback from users

INFORMATION RETRIEVAL PROCESS (Basic Structure of an IR System)

As we discussed above before any of the retrieval processes are initiated it is necessary to define the text database and this is usually done by the manager of the database and includes specifying the documents to be used, The operations to be performed on the text, The text model to be used (the text structure and what elements can be retrieved), The text operations transform the original

documents and the information needs and generate a logical view of them Once the logical view of the documents is defined.

The process of information retrieval involves the following stages:

- Representing Collections of Documents - how to represent, identify and process the collection of documents.
- User-initiated querving - understanding and processing of the queries.
- Retrieval of the appropriate documents - the searching mechanism used to obtain and retrieve the relevant document

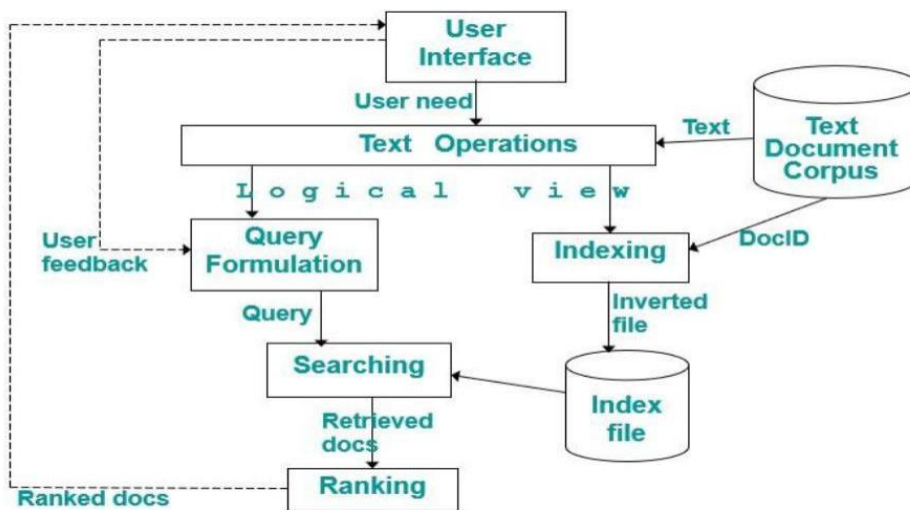


figure 2:Information Retrieval Process

There are three main ingredients to the IR process

- Texts or documents
- Queries
- The process of evaluation

For texts. the main problem is to obtain a representation of the text in a form which is amenable to automatic indexing. This is achieved (i.e., the representation) by creating an abbreviated form of the text, known as a text surrogate. Atypical surrogate would consist of a set of index terms or keywords or descriptors.

For queries. the query has arisen as a result of an information need on the part of the user. The query is then a representation of the information need and must be expressed in a language understood by the system. Due to the inherent difficulty of accurately representing the information need, the query in IR system is always regarded as approximate and imperfect. **For the evaluation,** The evaluation process involves a comparison of the text actually retrieved with those the user expected to retrieve. This often leads to some modification, typically of the query through possibly of the information need or even of the surrogates. The extent to which modification is required is closely linked with the process of measuring the effectiveness of the retrieval operation (recall and precision)

It is necessary to define the text database before any of the retrieval processes are initiated.

This is usually done by the manager of the database and includes specifying the following

— The documents to be used

—The operations to be performed on the text

— The text model to be used (the text structure and what elements can be retrieved)

The text operations transform the original documents and the information needs and generate a logical view of them

Once the logical view of the documents is defined, the database module builds an index of the text

— An index is a critical data structure

— It allows fast searching over large volumes of data

Different index structures might be used, but the most popular one is the inverted file (more on this later). Given the document database is indexed, the retrieval process can be initiated. The user first specifies a user need which is then parsed and transformed by the same text operation applied to the text. Then the query operations might be applied before the actual query, which provides the system representation for the user need, is generated. Matching- The query is then processed to obtain the retrieved documents. Before the retrieved documents are sent to the user, the retrieved documents are ranked according to the likelihood of relevance.

The user then examines the set of ranked documents in the search for useful information



Two choices for the user

— Reformulate query, run on entire collection

— Reformulate query, run on result set

At this point, he might pinpoint a subset of the documents seen as definitely of interest and initiate a user feedback cycle. In such a cycle, the system uses the documents selected by the user to change the query formulation. Hopefully, this modified query is a better representation of the real user need

Applications area

- Graphical interfaces to support information search
- Information Retrieval & Extraction
- XML retrieval
- Geographic Information Retrieval
- Multimedia information retrieval
- Cross-Language & Multilingual Information Retrieval
- Agent-based (like information filtering, tracking, routing) Information Retrieval

- Adversarial Information Retrieval
- Question answering
- Document Summarization
- Text classification
- Multi-database searching
- Document provenance
- Recommender systems
- Information Retrieval & Machine Learning
- Text Mining & Web Mining
- N-Grams in Information Retrieval

Chapter two

Text/document operations and automatic indexing

2.1. Text and Documents

2.1.1. Statistical Properties of Text

The chapter, therefore, starts with the original ideas of Luhn on which much of automatic text analysis has been built and then goes on to describe a concrete way of generating document representatives. Furthermore, ways of exploiting and improving document representatives through weighting or classifying keywords are discussed. In passing, some of the evidence for automatic indexing is presented.

Zipf's Law

Zipf's Law which states that the product of the frequency of use of words and the rank order is approximately constant. Zipf verified his law on American Newspaper English. Luhn used it as a null hypothesis to enable him to specify two cut-offs, an upper and a lower, thus excluding non-significant words. The words exceeding the upper cut-off were considered to be common and those below the lower cut-off are rare, and therefore not contribute significantly to the content of the article. He thus devised a counting technique for finding significant words. Consistent with this he assumed that the resolving power of significant words, by which he meant the ability of words to discriminate content, reached a peak at a rank order position halfway between the two cut-offs and from the peak fell off in either direction reducing to almost zero at the cut-off points. A certain arbitrariness is involved in determining the cut-offs. There is no oracle that gives their values. They have to be established by trial and error.

It is interesting that these ideas are really basic to much of the later work in IR. Luhn himself used them to devise a method of automatic abstracting. He went on to develop a numerical measure of significance for sentences based on the number of significant and non-significant words in each portion of the sentence. Sentences were ranked according to their numerical score and the highest-ranking was included in the abstract (extract really).

Edmundson and Wyllys⁸ have gone on to generalize some of Luhn's work by normalizing his measurements with respect to the frequency of occurrence of words in general text. There is no reason why such an analysis should be restricted to just words. It could equally well be applied to stems of words (or phrases) and in fact, this has often been done.

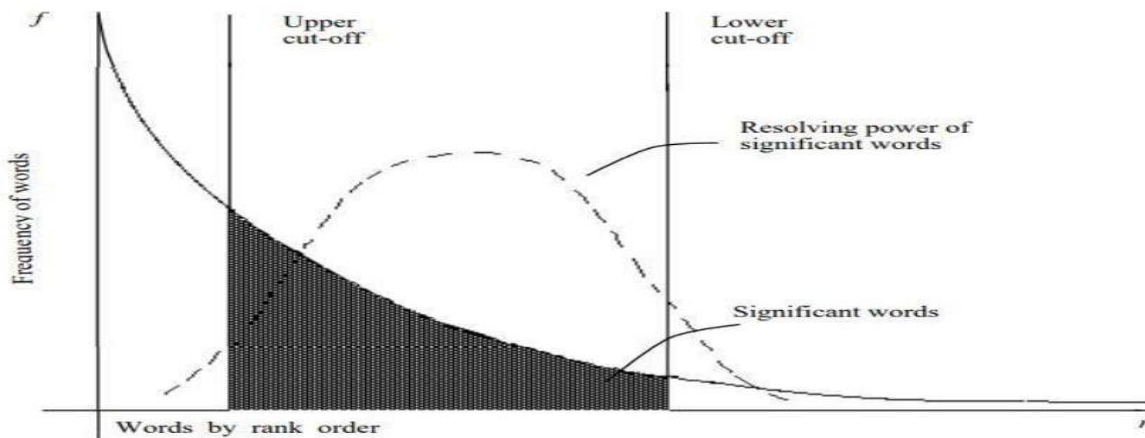


fig 2.1. A plot of the hyperbolic curve relating f , the frequency of occurrence, and r , the rank

order (Adapted from Schultz)

Luhn's ideas

In one of Luhn's⁶ early papers, he states: 'It is here proposed that the frequency of word occurrence in an article furnishes a useful measurement of word significance. It is further proposed that the relative position within a sentence of words having given values of significance furnish a useful measurement for determining the significance of sentences. The significant factor of a sentence will therefore be based on a combination of these two measurements.' This quote fairly summarizes Luhn's contribution to automatic text analysis. His assumption is that frequency data can be used to extract words and sentences to represent a document.

Luhn's model predicts the relative "resolving power of significant words" (Luhn, 1958) and Zipf's Law (Zipf, 1949) relates term frequencies and term ranks

Luhn's selection

Luhn's (1957), who was one of the earliest researcher into IR and the one who first suggested that certain words could be automatically extracted from texts to represent their content. He discovered that the distribution patterns of words could give significant information about the property of being content bearing. Much of text analysis has been built on the original idea of Luhn

Luhn's proposal is "The frequency of word occurrences in an article furnishes a useful measure of word significance..."

The quote fairly summarizes Luhn's contribution to automatic analysis. However, a high frequency term will be acceptable for indexing purposes only if its occurrence frequency is not equally high in all documents of the collection. According to his assumption, frequency data can be used to extract words (and sentences) to represent documents. Still today, the search engines that operate on the Internet index the documents based on this principle.

Luhn's observation:

- He noted that high frequency words tend to be common, non content bearing words
- He also recognized that one or two occurrences of a word in a relatively long text could not be taken significant in defining the subject matter

Luhn's model:

✓ Words which occur very infrequently in a collection are of little importance for indexing since they are unlikely to be specified in queries

Such rare terms are likely to be specific to the documents and they may not occur in users queries

- Words which occur very frequently in a collection are of little importance for indexing since they do not discriminate sufficiently between documents
- It is less likely to use these terms to discriminate the documents from others so not important for indexing

Therefore, the most important words for indexing are those which occur with intermediate frequencies. Thus, according to Luhn, medium frequency terms are better candidates for indexing. Therefore, mechanisms should be devised to get rid of rare and frequent terms.

Luhn used the work of Zipf to enable him specify the two cut-offs, an upper and lower, thus exclude non-significant words. The words exceeding the upper cut-off were considered to be common and those below the cut-off rare, and therefore not contributing significantly to the content of the document.

Luhn's Assumption

"...resolving power of significant words reaches a peak at a rank order position half way between the two cut-offs and from the peak fell off in either direction reducing to almost zero at the cut-off points"

The resolving power of words is the ability of words to discriminate content (i.e., document content). Words with low significance are at both tails of the distribution. Therefore, Luhn suggested using the words in the middle of the frequency range. These findings are the bases of a number of classical weighting schemes.

2.2. Text Operations

Text operations are the process of text transformations into logical representations. Not all words in a document are equally significant to represent the contents/meanings of a document some words carry more meaning than others for instance Noun words are the most representative of document content. Therefore, need to preprocess the text of a document in a collection to be used as index terms. Because using the set of all words in a collection to index documents creates too much noise for the retrieval task. So, reducing noise means reducing words that can be used to refer to the document.

Preprocessing is the process of controlling the size of the vocabulary or the number of distinct words used as index terms. Preprocessing will lead to an improvement in information retrieval.

performance. However, some search engines on the Web omit to preprocess every word in the the document is an index term.

The logical view of the document is provided by representative keywords or index terms, which are frequently used historically to represent documents in a collection. In modern computers, retrieval systems adopt a full-text logical view of the document. However, with very large collections, the set of representative keywords may have to be reduced. This process of reduction or compression of the set of representative keywords is called text operations (or transformation).

Text Processing System

Input text – full text, abstract, or title

Output – a document representative adequate for use in an automatic retrieval system

Generating Document Representatives

Ultimately one would like to develop a text processing system that by means of computable methods with the minimum of human intervention will generate from the input text (full text, abstract, or title) a document representative adequate for use in an automatic retrieval system. This is a tall order and can only be partially met. The document representative we aiming for is one consisting simply of a list of class names, each name representing a class of words occurring in the total input text. A document will be indexed by a name if one of its significant words occurs as a member of that class.

Such a system will usually consist of three parts:

- \emdash Removal of high-frequency words,
- \emdash Suffix stripping,
- \emdash Detecting equivalent stems.

The removal of high-frequency words, ‘stop’ words, or ‘fluff’ words is one way of implementing Luhn’s upper cut-off. This is normally done by comparing the input text with a ‘stop list’ of words that are to be removed.

Figure 2.1 gives a portion of such a list, and demonstrates the kind of words that are involved. The advantages of the process are not only that non-significant words are removed and will therefore not interfere during retrieval, but also that the size of the total document file can be reduced by between 30 and 50 percent.

The second stage, suffix stripping, is more complicated. A standard approach is to have a complete list of suffixes and to remove the longest possible one.

2.2.1. Lexical Analysis/Tokenization of Text

Lexical analysis or Tokenization is a fundamental operation in both query processing and automatic indexing. It is the process of converting an input stream of characters into a stream of words or tokens. Tokens are groups of characters with collective significance. In other words, it is one of the steps used to convert the text of the documents into the sequence of words, $w_1, w_2,$

- w_n to be adopted as index terms. It is the process of demarcating and possibly classifying sections of a string of input characters into words.

Generally, Lexical analysis is the first stage of automatic indexing, and of query processing. Automatic indexing is the process of algorithmically examining information items to generate lists of index terms. The lexical analysis phase produces candidate index terms that may be further

processed and eventually added to indexes. Query processing is the activity of analyzing a query and comparing it to indexes to find relevant items. Lexical analysis of a query produces tokens that are parsed and turned into an internal representation suitable for comparison with indexes.

- **Issues in Tokenization**

The main Objective of Tokenization is – the identification of words in the text document.

Tokenization is greatly dependent on how the concept of the word is defined.

The first decision that must be made in designing a lexical analyzer for an automatic indexing system is: What counts as a word or token in the indexing scheme?

Is that a sequence of characters, numbers, and alpha-numeric once? A word is a sequence of letters terminated by a separator (period, comma, space, etc).

The definition of letter and separator is flexible; e.g., a hyphen could be defined as a letter or as a separator. Usually, common words (such as “a”, “the”, “of”, ...) are ignored.

The standard tokenization approach is single-word tokenization where input is split into words using white space characters as delimiters and it ignores other characters rather than words. This approach introduces errors at an early stage because it ignores multi-word units, numbers, hyphens, punctuation marks, and apostrophes.

How to handle special cases involving hyphens, apostrophes, punctuation marks, etc? C++, C#, URLs, e-mail, ...

Sometimes punctuations (e-mail), numbers (1999), & cases (Republican vs. republican) can be a meaningful part of a token.

There are many Issues in Tokenization and some of these are listed below. **Frequently they are not.**

- Two words may be connected by hyphens.

Can two words connected by hyphens be taken as one word or two words? Break up hyphenated sequence as two tokens? In most cases hyphens – break up the words (e.g. state-of-the-art, state of the art), but some words, e.g. MS-DOS, B49 – are unique words that require hyphens

\emdash Two words may be connected by punctuation marks.

Punctuation marks: remove totally unless significant, e.g. program code: x.exe and xexe. What about Kebede's, www.command.com?

\emdash Two words (phrases) may be separated

by space E.g. Addis Ababa, San Francisco, Los

Angeles

Two words may be written in different ways lowercase, lower case, lower case? database, database, database?

Numbers: are numbers/digits words used as index terms?

- dates (3/12/91 vs. Mar. 12, 1991);
- phone numbers (+251923415005)
- IP addresses (100.2.86.144)
- Numbers are not good index terms (like 1910, 1999); but 510 B.C. is unique. Generally, don't index numbers as text, though very useful.

\emdash What about the case of letters (e.g. Data or data or DATA):

\emdash Cases are not important and there is a need to convert all to upper or lower.
Which one is mostly followed by human beings?

The simplest approach is to ignore all numbers and punctuation marks (period, colon, comma, brackets, semi-colon, apostrophe, ...) & use only case-insensitive unbroken strings of alphabetic characters as words. Will often index "meta-data", including creation date, format, etc. separately Issues of tokenization are language-specific and require the language to be known. The following is an example of how standard tokenization performed

Analyze text into a sequence of discrete tokens (words)?

\emdash Input: "Friends, Romans, and Countrymen"

\emdash Output: Tokens (an instance of a sequence of characters that are grouped together as a useful semantic unit for processing)

Friends

Romans and

Countrymen

\emdash Each such token is now a candidate for an index entry, after further processing, But what are valid tokens to emit?

2.2.2. Elimination of Stopwords

Stopwords are extremely common words across document collections that have no discriminatory power. They may occur in 80% of the documents in a collection. They would appear to be of little value in helping select documents matching a user's need and needs to be filtered out as potential index terms. Examples of stopwords are articles, prepositions, conjunctions, etc.:

articles (a, an, the); pronouns (I, he, she, it, their, his), Some prepositions (on, of, in, about, besides, against), conjunctions/ connectors (and, but, for, nor, or, so, yet), verbs (is, are, was, were), adverbs (here, there, out, because, soon, after) and adjectives (all, any, each, every, few, many, some) can also be treated as stopwords. Stopwords are language-dependent.

Why Stopword Removal?

Intuition:

- Stopwords have little semantic content; It is typical to remove such high-frequency words
- Stopwords take up 50% of the text. Hence, document size reduces by 30-50% Smaller indices for information retrieval
- Good compression techniques for indices: The 30 most common words account for 30% of the tokens in written text

With the removal of stopwords, we can measure a better approximation of the importance of text classification, text categorization, text summarization, etc.

How to detect a stopwords?

One method: Sort terms (in decreasing order) by document frequency (DF) and take the most frequent ones based on the cutoff point.

— **Another method:** Build a stop word list that contains a set of articles, pronouns, etc.

— Why do we need stop lists: With a stop list, we can compare and exclude from the index terms entirely the commonest words?

Stop word elimination used to be standard in older IR systems. But the trend is away from doing

nowadays most web search engines index stop words: Good query optimization techniques mean you pay little at query time for including stop words. You need stopwords for:

- Phrase queries: “King of Denmark”
- Various song titles, etc.: “Let it be”, “To be or not to be”
- “Relational” queries: “flights to London”

elimination of stopwords might reduce recall (e.g. “To be or not to be” – all eliminated except “be” – no or irrelevant retrieval)

2.2.3. Normalization

It is canceling tokens so that matches occur despite **superficial** differences in the character sequences of the tokens.

Need to “normalize” terms in the indexed text as well as query terms into the same form

- **Example:** We want to match **U.S.A.** and **USA**, by deleting periods in a term.

Case Folding: Often best to lower case everything, since users will use lowercase regardless of ‘correct’ capitalization...

- **Republican** vs. **republican**
- **Fasil** vs. **fasil** vs. **FASIL**
- **Anti-discriminatory** vs. **antidiscriminatory**
- **Car** vs. **Automobile?**

Normalization issues

- Good for:
 - Allow instances of Automobile at the beginning of a sentence to match with a query of automobile
 - Helps a search engine when most users type Ferrari while they are interested in a Ferrari car
- Not advisable for:
 - Proper names vs. common nouns
- E.g. General Motors, Associated Press, Kebede...
- Solution:
 - lowercase only words at the beginning of the sentence

In IR, lowercasing is most practical because of the way users issue their queries

2.2.4. Stemming/Morphological analysis

Morphology is the study of the structure of a word and how it is built. Martin and Jurafsky [20] describe it as small building blocks of a word called morphemes and they can be divided into two groups, stems, and affixes. A stem is the main part of a word and affixes are morphemes that are added to a stem to give different meanings to it. In the word dogs, for example, the dog is the stem and -s is the affix. Using affixes allow a word to occur in different forms, it can give it different inflections and derivations. How common do these variations differ between different languages. According to Hedlund et al. [17], a language can be considered to be simple or complex with regard to morphology. English, is considered to be simple while Swedish is a language that is considered to be morphologically complex. Stemming reduces tokens to their root form of words to recognize morphological variation.

The process involves the removal of affixes (i.e. prefixes & suffixes) with the aim of reducing variants to the same stem. There are two types of morphology and those are inflectional & derivational morphology Inflectional morphology: varies the form of words in order to express grammatical features, such as singular/plural or past/present tense. E.g. Boy → boys, cut → cutting.

Derivational morphology: makes new words from old ones. E.g. creation is formed from creating, but they are two separate words. And also, destruction → destroy. Correct stemming is language-specific and can be complex.

Stemming is one technique to provide ways of finding morphological variants of search terms. Used to improve retrieval effectiveness and to reduce the size of indexing files. a. Taxonomy for stemming algorithms

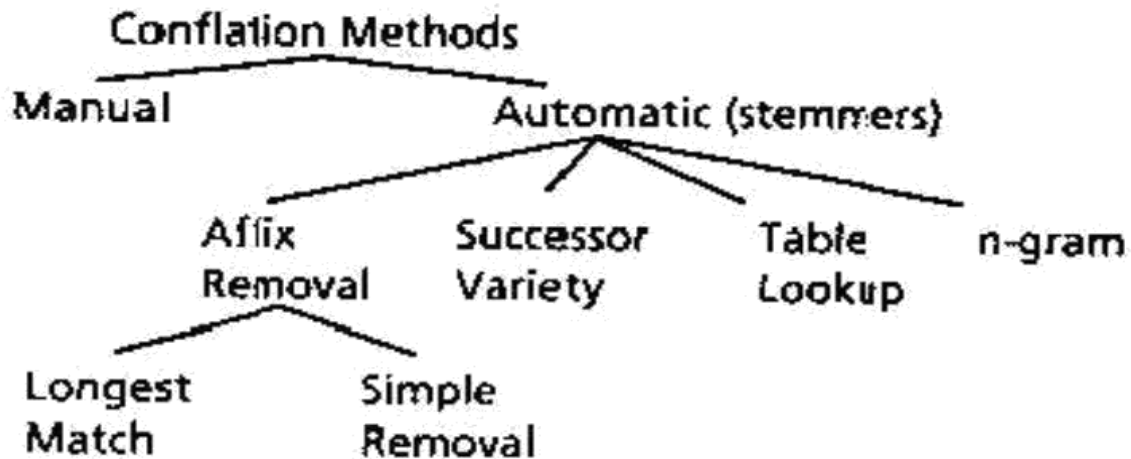


fig 2.3

Criteria for judging stemmers

a. Correctness

- o Over stemming: too much of a term is removed.
- o Under stemming: too little of a term is removed.

b. Retrieval effectiveness :- measured with recall and precision and on their speed, size, and so on

c. Compression performance :- greatly reduce the cost of storing common words

Way to implement stemming

There are basically two ways to implement stemming.

The first approach is to create a big dictionary that maps words to their stems.

* The advantage of this approach is that it works perfectly (insofar as the stem of a word can be defined perfectly): the disadvantages are the space required by the dictionary and the investment required to maintain the dictionary as new words appear.

The second approach is to use a set of rules that extract stems from words.

* The advantages of this approach are that the code is typically small. and it can gracefully handle new words; the disadvantage is that it occasionally makes mistakes.

* But, since stemming is imperfectly defined. anyway, occasional mistakes are tolerable, and the rule-based approach is the one that is generally chosen.

Type of stemming algorithms There are 4 basic types of stemming algorithms those are:

Table lookup approach

Successor Variety

n-gram stemmers

Affix Removal Stemmers

1. Table lookup approach

Stemming is done via lookups in the table. Store a table of all index terms and their stems, so terms from queries and indexes could be stemmed very fast. Problems

- There is no such data for English. Or some terms are domain dependent.
- The storage overhead for such a table, though trading size for time is sometimes warranted.

2. Successor Variety approach

Determine word and morpheme boundaries based on the distribution of phonemes in a large body of utterances. And then, The successor variety of a string is the number of different characters that follow it in words in some body of text. The successor variety of substrings of a term will decrease as more characters are added until a segment boundary is reached. See the following example

Table 2 examples of successor variety approach

Test Word: *READABLE*

Corpus: ABLE, APE, BEATABLE, FIXABLE, READ, READABLE,
READING, READS, RED, ROPE, RIPE

Prefix	Successor Variety	Letters
R	3	E,I,O
RE	2	A,D
REA	1	D
READ	<u>3</u>	A,I,S
READA	1	B
READAB	1	L
READABL	1	E
READABLE	1	(Blank)

cutoff method

some cutoff value is selected and a boundary is identified whenever the cutoff value is peak and plateau method

- segment break is made after a character whose successor variety exceeds that of the characters immediately preceding and following it

A criteria used to evaluate various segmentation methods. the number of correct segment cuts divided by the total number of cuts. After segmenting, if the first segment occurs in more than 12 words in the corpus, it is probably a prefix. The successor variety stemming process has three parts.

1. determine the successor varieties for a word
2. segment the word using one of the methods
3. select one of the segments as the stem
3. n-gram stemmers

Association measures are calculated between pairs of terms based on shared unique digrams. statistics => st ta at ti is st ti ic cs

unique digrams = at cs ic is st ta ti statistical => st ta at ti is st ti ic ca al unique digrams = al at
ca ic is st ta ti

Dice's coefficient (similarity)

$$S = \frac{2C}{A+B}$$

A and B are the numbers of unique diagrams in the first and the second words. C is the number of unique diagrams shared by A and B. Similarity measures are determined for all pairs of terms in the database, forming a similarity matrix. Once such a similarity matrix is available, terms are clustered using a single link clustering method.

4. Affix Removal Stemmers

Affix removal algorithms remove suffixes and/or prefixes from terms leaving a stem – If a word ends in “ies” but not “eies” or “aies ” (**Harman 1991**) Then “ies” -> “y”

– If a word ends in “es” but not “aes” , or “ees ” or “oes” Then “es” -> “e”

– If a word ends in “s” but not “us” or “ss ”

Then “s” -> “NULL”

The Porter algorithm

The most common stemmer is the Porter stemmer (Porter, 1980). It is designed to fit the characteristics of English language

Idea: Suffixes in the English language are mostly made up of a combination of smaller and simpler suffixes

- How does it work?

- The algorithm runs through five steps, one by one

- In each step, several rules are applied that change the word's suffix

Some (simplified!) examples of rules used in the Porter stemmer:

Table 3 examples of Porter algorithm Some transformations made by the Porter stemmer:

Input word	Stemmed word
gen	gen
gender	gender
genders	gender
general	gener
generally	gener
generals	gener
generation	gener
generations	gener
generative	gener
generosity	generos
generous	gener
genitive	genit
genitivo	genitivo

Stemming Studies: Conclusion

The majority of stemming's affection on retrieval performance have been positive Stemming is as effective as manual conflation

The effect of stemming is dependent on the nature of vocabulary used

There appears to be little difference between the retrieval effectiveness of different full stemmers

2.2.5. Index Term Selection

Index language is the language used to describe documents and requests. Elements of the index language are index terms which may be derived from the text of the document to be described, or may be arrived at independently. If a full text representation of the text is adopted, then all words in the text are used as index terms = full text indexing. Otherwise, need to select the words to be used as index terms for reducing the size of the index file which is basic to design an efficient searching IR system.

Some words are not good for representing documents, use of all words have computational cost, increase searching time and storage requirements and using the set of all words in a collection to index documents generates too much noise for the retrieval task, therefore, term selection is very important. The main objectives of term selections are:

- Represent textual documents by a set of keywords called index terms or simply terms

- Increase efficiency by extracting from the resulting document a selected set of terms to be used for indexing the document
- If full text representation is adopted then all words are used for indexing (not as such efficient as it will have an overhead, time and space)

Index term is also called keyword or is a word (a single word) or phrase (multiword) in a document whose semantics gives an indication of the document's theme (main idea)

- Capture subject discussed
- Help in remembering the documents main theme Index term is mainly noun (because nouns have meanings by themselves)

Indexing- User's ability to find documents on a particular subject is limited by the indexing process used to create index terms for the subject. Some definitions of indexing are the following. Indexing:

- Is the art of organizing information
- Is an association of descriptors (keywords, concepts) to documents in view of future retrieval
- Is a process of constructing document surrogates by assigning identifiers to text items
- Is the process of storing data in a particular way in order to locate and retrieve the data — Is the process of analyzing the information content in the language of the indexing system

The main important and well known Purpose/objective of indexing are

- To give access point to a collection that are expected to be most useful to the users of information
- To allow easy identification of documents (e.g., find documents by topic)
- To relate documents to each other
- To allow prediction of document relevance to a particular information need

There are 2 ways of indexing:

1. Manual indexing

Indexers decide which keywords to assign to documents based on controlled vocabulary (Human indexers assign index terms to documents). The indexers try to summarize the contents or about of the whole document in a few keywords, that is, indexers analyze and represent the content of a document through keywords which is based on intellectual judgment and semantic interpretation of (concepts, themes) of indexers. In manual indexing indexers prior knowledge of the following is important to come up with good keywords or index terms.

- Terms that will be used by the user
- Indexing vocabulary
- Collection characteristics

Indexers are normally provided with guidelines (input sheets, manuals and instructions, printed thesaurus) to determine the contents of a given document and are usually done in the library environment.

Advantage of Manual indexing

- Ability to perform abstraction (conclude what the subject is) and determine additional related terms
- Ability to judge the value of concepts .

disadvantage of Manual indexing

- Slow and expensive (significant cost)
 - Cost of professional indexers is very expensive
 - High probability of inconsistency or low consistency among indexers (maintaining consistency is difficult),
 - Labor intensive

Table 1: advantage and disadvantage of manual indexing

2. Automatic indexing

Automatic indexing is the assignment of content identifiers, with the help of modern computing technology. A computer system is used to record the descriptors generated by the human and the system extracts "typical"! "significant" terms. Then the human may contribute by setting the parameters or thresholds, or by choosing components or algorithms. The original texts of information items are used as a basis of indexing.

An automatic indexing is necessary because of the following reason

- Information overload
 - Enormous amount of information is being generated from day to day activities
- Explosion of machine-readable text
- Massive information available in electronic format and on Internet.
- Cost effectiveness
- Human indexing is expensive and labor intensive.

Procedures for automatic indexing

Generating document representatives through automatic indexing involves

- Lexical analysis
- Use of stop list
- Noun identification (optional)
- Phrase formation (optional)
- Use of conflation procedures (stemming, optional)
- Selection of index terms
- Weighting the resulting terms (optional)

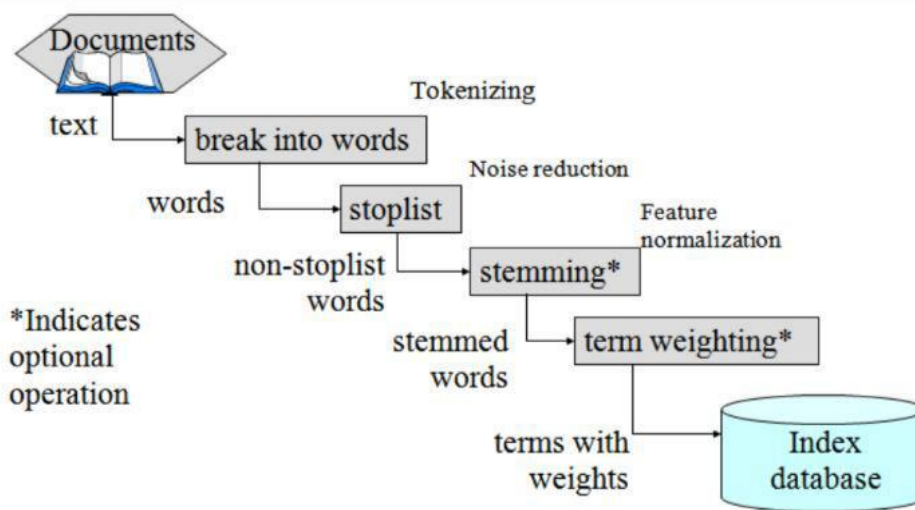


Figure 2.1: procedures of automatic indexing

Term weighting and similarity measures

Terms are usually stems. Terms can be also phrases, such as “Computer Science”, “World Wide Web”, etc. Documents and queries are represented as vectors or “bags of words” (BOW).

- Each vector holds a place for every term in the collection.
- Position 1 corresponds to term 1, position 2 to term 2, position n to term n. W=0 if a term is absent

Documents are represented by **binary weights** or **Non-binary weighted** vectors of terms.

$$D_i = w_{d1}, w_{d2}, \dots, w_{dn}$$

$$Q = w_{q1}, w_{q2}, \dots, w_{qn}$$

Document collection and term vector space

A collection of n documents can be represented in the vector space model by a term-document matrix. An entry in the matrix corresponds to the “weight” of a term in the document; zero means the term has no significance in the document or it simply doesn’t exist in the document. The Vector-Space Model (VSM) for Information Retrieval **represents documents and queries as vectors of weights**. Each weight is a measure of the importance of an index term in a document or a query, respectively.

n -dimensional space, where n is the number of different terms/tokens used to index a set of documents.

Document i , d_i , represented by a vector. Its magnitude in dimension j is w_{ij} ,

where: $w_{ij} > 0$ if term j occurs in document i

$w_{ij} = 0$ otherwise

w_{ij} is the weight of term j in document i .

$$\begin{pmatrix} D_1 & T_1 & T_2 & \dots & T_t \\ D_2 & w_{11} & w_{21} & \dots & w_{t1} \\ \vdots & w_{12} & w_{22} & \dots & w_{t2} \\ \vdots & \vdots & \vdots & & \vdots \\ D_n & w_{1n} & w_{2n} & \dots & w_{tn} \end{pmatrix}$$

Term weighting is a process to compute and assign a numeric value to each term in order to weight its contribution in distinguishing a particular document from others. By assigning a numerical value to a term representing its importance in the document, retrieval effectiveness can be improved [8]. Term weighting indicates how important each individual word is to the document and within the document collection. Thus the doc – John is quicker than Mary. is indistinguishable from the doc – Mary is quicker than John. The most popular approach is tfidf weighting scheme, i.e. term frequency (tf) times inverse document frequency (idf).

Binary Weights

- Only the presence (1) or absence (0) of a term is included in the vector
- Binary formula gives every word that appears in a document equal relevance.
- It can be useful when frequency is not important.
- **Binary Weights Formula:**

<i>docs</i>	<i>t1</i>	<i>t2</i>	<i>t3</i>
D1	1	0	1
D2	1	0	0
D3	0	1	1
D4	1	0	0
D5	1	1	1
D6	1	1	0
D7	0	1	0
D8	0	1	0
D9	0	0	1
D10	0	1	1
D11	1	0	1

$$freq_{ij} = \begin{cases} 1 & \text{if } freq_{ij} > 0 \end{cases}$$

Why use term weighting?

- Binary weights are too limiting.
 - terms are either present or absent.
 - Not allow to order documents according to their level of relevance for a given query
- Non-binary weights allow to model partial matching .
 - Partial matching allows retrieval of docs that approximate the query.
 - Term-weighting improves quality of answer set.
 - Term weighting enables ranking of retrieved documents; such that best matching documents are ordered at the top as they are more relevant than others.
- **Term Frequency:** In document d, the frequency represents the number of instances of a given word t. Therefore, we can see that it becomes more relevant when a word appears in the text, which is rational. Since the ordering of terms is not significant, we can use a vector to describe the text in the bag of term models. For each specific term in the paper, there is an entry with the value being the term frequency.

The weight of a term that occurs in a document is simply proportional to the term frequency.

$$tf(t,d) = \text{count of } t \text{ in } d / \text{number of words in } d$$

- **Document Frequency:** This tests the meaning of the text or measures the importance of document in whole set of corpus, which is very similar to TF, in the whole corpus collection. The only difference is that TF is frequency counter for a term t in document d , whereas DF is the count of **occurrences** of term t in the document set N . In other words, DF is the number of documents in which the word is present. We consider one occurrence if the term consists in the document at least once, we do not need to know the number of times the term is present. In other words, the number of papers in which the word is present is DF.

$df(t)$ = occurrence of t in documents

- **Inverse Document Frequency:** Mainly, it tests how relevant the word is. The key aim of the search is to locate the appropriate records that fit the demand. Since tf considers all terms equally significant, it is therefore not only possible to use the term frequencies to measure the weight of the term in the paper. First, find the document frequency of a term t by counting the number of documents containing the term:

$df(t) = N(t)$

where

$df(t)$ = Document frequency of a term t

$N(t)$ = Number of documents containing the term t

Term frequency is the number of instances of a term in a single document only; although the frequency of the document is the number of separate documents in which the term appears, it depends on the entire corpus. Now let's look at the definition of the frequency of the inverse paper. The IDF of the word is the number of documents in the corpus separated by the frequency of the text.

While computing TF, all terms are considered equally important. However it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scale up the rare ones, by computing IDF, an inverse document frequency factor is incorporated which diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. IDF is the inverse of the document frequency which measures the informativeness of term t . When we

calculate IDF, it will be very low for the most occurring words such as stop words (because stop words such as “is” is present in almost all of the documents, and N/df will give a very low value to that word). This finally gives what we want, a relative weightage.

$$idf(t) = N/df$$

Now there are few other problems with the IDF , in case of a large corpus,say 100,000,000 , the IDF value explodes , to avoid the effect we take the log of idf .

$$idf(t) = \log(N/ df(t))$$

During the query time, when a word which is not in vocab occurs, the df will be 0. As we cannot divide by 0, we smoothen the value by adding 1 to the denominator.

Example: given a collection of 1000 documents and document frequency, compute IDF for each word?

Word	N	DF	IDF
the	1000	1000	0
some	1000	100	3.322
car	1000	10	6.644
merge	1000	1	9.966

- IDF provides high values for rare words and low values for common words.
- IDF is an indication of a term’s *discrimination* power.
 - Log used to dampen the effect relative to *tf*.
 - Make the difference between Document frequency vs. corpus frequency ?

TF*IDF Weighting

term frequency-inverse document frequency (*tf-idf*) is defined as $tf_{ij} * idf_i$ (Salton and McGill, 1986). It is a measure of importance of a term t_i in a given document d_j . It is a term frequency measure which gives a larger weight to terms which are less common in the corpus. The importance of very frequent terms will then be lowered, which could be a desirable feature.

TF-IDF is to statistically measure how important a word is in a collection of documents. A term occurring frequently in the document but rarely in the rest of the collection is given high weight.

- The tf-idf value for a term will always be greater than or equal to zero.

Experimentally, $tf*idf$ has been found to work well.

- It is often used in the vector space model together with cosine similarity to determine the similarity between two documents.
- When does TF*IDF registers a high weight? when a term t occurs many times within a small number of documents
 - Highest $tf*idf$ for a term shows a term has a high term frequency (in the given document) and a low document frequency (in the whole collection of documents);
 - the weights hence tend to filter out common terms.
 - Thus lending high discriminating power to those documents
- Lower TF*IDF is registered when the term occurs fewer times in a document, or occurs in many documents
 - Thus offering a less pronounced relevance signal
- Lowest TF*IDF is registered when the term occurs in virtually all documents

Example

Let's take an example to get a clearer understanding.

Sentence 1 : The car is driven on the road.

Sentence 2: The truck is driven on the highway.

In this example, each sentence is a separate document.

We will now calculate the TF-IDF for the above two documents, which represent our corpus.

Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	$\log(2/2) = 0$	0	0
Car	1/7	0	$\log(2/1) = 0.3$	0.043	0
Truck	0	1/7	$\log(2/1) = 0.3$	0	0.043
Is	1/7	1/7	$\log(2/2) = 0$	0	0
Driven	1/7	1/7	$\log(2/2) = 0$	0	0
On	1/7	1/7	$\log(2/2) = 0$	0	0
The	1/7	1/7	$\log(2/2) = 0$	0	0
Road	1/7	0	$\log(2/1) = 0.3$	0.043	0
Highway	0	1/7	$\log(2/1) = 0.3$	0	0.043

Similarity Measure

The basic aim of information retrieval is retrieval of most relevant documents for a given user query. Web searches are the perfect example for this application. Many algorithms were developed for this purpose, which take an input query and match it with the stored documents or text snippets and rank the documents based on their similarity score relative to the given query. Such algorithms rely on matching the indexed documents, which maintain the information concerning term frequencies and positions, against the individual query terms. A score is assigned to each document based on its similarity value.

We now have vectors for all documents in the collection, a vector for the query, how to compute similarity? A similarity measure is a function that computes the degree of similarity or distance between document vector and query vector.

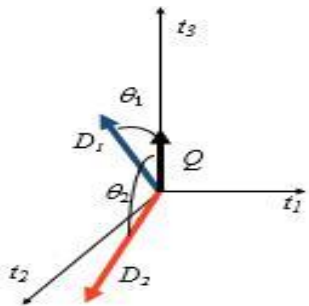
A similarity measure is a function which determines the degree of similarity between a pair of textual objects]. In general, these measures were used to calculate similarity between two queries, two documents and one document and one query. Similarity measures also used to rank the documents based on the similarity scores between the document and query. A similarity measure is a function which computes the degree of similarity between a pair of vectors or documents

– since queries and documents are both vectors, a similarity measure can represent the similarity between two documents, two queries, or one document and one query

- ❖ There are a large number of similarity measures proposed in the literature, because the best similarity measure doesn't exist (yet!)
- ❖ With similarity measure between query and documents

Using a similarity measure between the query and each document:

- It is possible to rank the retrieved documents in the order of presumed relevance.
- It is possible to enforce a certain threshold so that the size of the retrieved set can be controlled.
- the results can be used to reformulate the original query in relevance feedback (e.g., combining a document vector with the query vector)



Postulate: Documents that are “close together” in the vector space talk about the same things and more similar than others.

Two documents are similar if they contain some of the same terms. Possible measures of similarity might take into consideration:

- (a) The lengths of the documents
- (b) The number of terms in common
- (c) Whether the terms are common or unusual
- (d) How many times each term appears

Desiderata for proximity

1. If d_1 is near d_2 , then d_2 is near d_1 .
 2. If d_1 near d_2 , and d_2 near d_3 , then d_1 is not far from d_3 .
 3. No document is closer to d than d itself.
 - Sometimes it is a good idea to determine the maximum possible similarity as the “distance” between a document d and itself
- A similarity measure attempts to compute the distance between document vector w_j and query w_q vector.
 - The assumption here is that documents whose vectors are close to the query vector are more relevant to the query than documents whose vectors are away from the query vector.

Similarity Measure: Techniques

Euclidean distance

- It is the most common similarity measure.
- Euclidean distance examines the *root of square differences* between coordinates of a pair of document and query terms.

Dot product

- The dot product is also known as the scalar product or inner product
- the **dot product** is defined as the product of the magnitudes of query and document vectors
- It projects document and query vectors into a term space and calculate the cosine angle between these.

Euclidean distance

Euclidian distance measure is an ordinary distance measure to compute the distance between two points in two- and three-dimensional space. This measure is used in document clustering to group the documents into similar clusters based on the distance between the documents. Similarity between vectors for the document d_i and query q can be computed as:

$$\text{sim}(d_j, q) = |d_j - q| = \sqrt{\sum_{i=1}^n (w_{ij} - w_{iq})^2}$$

where w_{ij} is the weight of term i in document j and w_{iq} is the weight of term i in the query

Example: Determine the Euclidean distance between the document 1 vector (0, 3, 2, 1, 10) and query vector (2, 7, 1, 0, 0). 0 means corresponding term not found in document or query

$$= \sqrt{(0-2)^2 + (3-7)^2 + (2-1)^2 + (1-0)^2 + (10-0)^2} = 11.05$$

Inner Product Dot product

Similarity between vectors for the document d_i and query q can be computed as the vector inner product:

$$\text{sim}(d_j, q) = d_j \cdot q = \sum_{i=1}^n w_{ij} \cdot w_{iq}$$

where w_{ij} is the weight of term i in document j and w_{iq} is the weight of term i in the query q

- For binary vectors, the inner product is the number of matched query terms in the document (size of intersection).

For weighted term vectors, it is the sum of the products of the weights of the matched terms.

Properties of Inner Product

- Favors long documents with a large number of unique terms.
 - Again, the issue of normalization
- Measures how many terms matched but not how many terms are *not* matched.

Example

• Binary weight :

—Size of vector = size of vocabulary = 7

	Retrieval	Database	Term	Computer	Text	Manage	Data
D	1	1	1	0	1	1	0
Q	1	0	1	0	0	1	1

$$\text{sim}(D, Q) = 3$$

• Term Weighted:

	Retrieval	Database	Architecture
D1	2	3	5
D2	3	7	1
Q	1	0	2

$$\text{sim}(D_1, Q) = 2*1 + 3*0 + 5*2 = 12$$

$$\text{sim}(D_2, Q) = 3*1 + 7*0 + 1*2 = 5$$

Cosine similarity

Cosine similarity measure is a popular method for calculating the similarity value between the vectors. With document and queries being represented as vectors, similarity signifies the proximity between the two vectors. Cosine similarity measure computes similarity as a function of the angle made by the vectors. If two vectors are close, the angle formed between them would be small and if the two vectors are distant, the angle formed between them would be large. The cosine value varies from +1 to -1 for angles ranging from 0 to 180 degrees respectively, making it the ideal choice for these requirements. A score of 1 evaluates to the angle being 0°, which means the documents are similar. While a score of 0 evaluates to the angle being 90°, which means the documents are entirely dissimilar. The cosine weighting measure is implemented on length normalized vectors for making their weights comparable.

Cosine similarity measures the similarity between two vectors of an inner product space. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in roughly the same direction. It is often used to measure document similarity in text analysis.

The mathematical equation of Cosine similarity between two non-zero vectors is:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}},$$

Measures similarity between d_1 and d_2 captured by the cosine of the angle x between them.

$$\text{sim}(d_j, q) = \frac{\sum_i w_{ij} w_{iq}}{\sqrt{\sum_i w_{ij}^2} \sqrt{\sum_i w_{iq}^2}}$$

The denominator involves the lengths of the vectors
 $\text{Length } \vec{d_j} = \sqrt{\sum w_{ij}^2}$

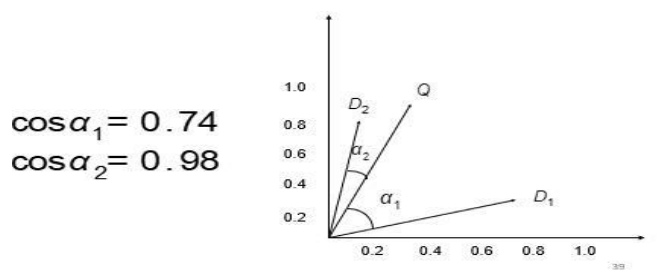
Example: Computing Cosine Similarity

Let say we have query vector $Q = (0.4, 0.8)$; and also document $D_1 = (0.2, 0.7)$. Compute their similarity using cosine?

$$\begin{aligned} \text{sim}(Q, D_1) &= \frac{(0.4 \times 0.2) + (0.8 \times 0.7)}{\sqrt{(0.4)^2 + (0.8)^2} \sqrt{(0.2)^2 + (0.7)^2}} \\ &= \frac{0.64}{\sqrt{0.8} \sqrt{0.53}} = 0.98 \end{aligned}$$

Example: Computing Cosine Similarity

Let say we have two documents in our corpus; $D_1 = (0.8, 0.3)$ and $D_2 = (0.2, 0.7)$. Given query vector $Q = (0.4, 0.8)$, determine which document is the most relevant one for the query?



Cosine Similarity vs. Inner Product

- Cosine similarity measures the cosine of the angle between two vectors.
- Inner product normalized by the vector lengths.

$$D_1 = 2T_1 + 3T_2 + 5T_3 \quad \text{CosSim}(D_1, Q) = 10 / \sqrt{(4+9+25)(0+0+4)} = 0.81$$

$$D_2 = 3T_1 + 7T_2 + 1T_3 \quad \text{CosSim}(D_2, Q) = 2 / \sqrt{(9+49+1)(0+0+4)} = 0.13$$

$$Q = 0T_1 + 0T_2 + 2T_3$$

D_1 is 6 times better than D_2 using cosine similarity but only 5 times better using inner product.

Exercise

A database collection consists of 1 million documents, of which 200,000 contain the term holiday while 250,000 contain the term season. A document repeats holiday 7 times and season 5 times. It is known that holiday is repeated more than any other term in the document. Calculate the weight of both terms in this document using three different term weight methods. Try with

- normalized and unnormalized TF;
- TF*IDF based on normalized and unnormalized TF

Chapter three

Indexing Structures

File structures: A fundamental decision in the design of IR systems is which type of file structure to use for the underlying document database. The file structures used in IR systems are flat files, inverted files, signature files, PAT trees, and graphs. Though it is possible to keep file structures in main memory, in practice IR databases are usually stored on disk because of their size. Using a flat file approach, one or more documents are stored in a file, usually as ASCII or EBCDIC text. Flat file searching is usually done via pattern matching. On UNIX, for example, one can store a document collection one per file in a UNIX directory, and search it using pattern searching tools such as `grep` (Earhart 1986) or `awk` (Aho, Kernighan, and Weinberger 1988).

An inverted file is a kind of indexed file. The structure of an inverted file entry is usually keyword, document-ID, and field-ID. A keyword is an indexing term that describes the document, document-ID is a unique identifier for a document, and field-ID is a unique name that indicates from which field in the document the keyword came. Some systems also include information about the paragraph and sentence location where the term occurs. Searching is done by looking up query terms in the inverted file.

Signature files contain signatures--it patterns--that represent documents. There are various ways of constructing signatures. Using one common signature method, for example, documents are split into logical blocks each containing a fixed number of distinct significant, that is, non-stoplist, words. Each word in the block is hashed to give a signature--a bit pattern with some of the bits set to 1. The block signatures are then concatenated to produce the document signature. Searching is done by comparing the signatures of queries with document signatures.

PAT trees are Patricia trees constructed over all strings in a text. If a document collection is viewed as a sequentially numbered array of characters, a string is a subsequence of characters from the array starting at a given point and extending an arbitrary distance to the right. A Patricia tree is a digital tree where the individual bits of the keys are used to decide branching.

raphs, or networks, are ordered collections of nodes connected by arcs. They can be used to represent documents in various ways. For example, a kind of graph called a semantic net can be used to represent the semantic relationships in text often lost in the indexing systems above.

Although interesting, graph-based techniques for IR are impractical now because of the amount of manual effort that would be needed to represent a large document collection in this form. Two well-known indexing methods are inverted files and signature files that have been proposed for large text databases. Inverted files are distinctly superior to signature files. Not only can inverted files be used to evaluate typical queries in less time than can signature files, but inverted files require less space and provide greater functionality. Both remain the subject of active research. However, although many researchers have evaluated the performance of one method or the other, there has been no detailed side-by-side comparison. Indeed, the absence of a comparison has meant that the question as to which is better has been a popular topic of debate in research institute, in which, in the context of text indexing, several of researchers have proposed variations on both of these indexing schemes [Sacks-Davis et al. 1987; Kent et al. 1990; Bell et al. 1993; Mofat and Zobel 1996]. The debate has been further added to by the steady flow of papers they have been asked to review in which the authors espouse or improve upon one of these forms of indexing without regard for the existence of the other. To resolve the question of which method is superior they have undertaken a detailed examination of inverted files and signature files, using both experimentation on realistic data and a refined approach to modeling of signature files. Their conclusion is that, for current architectures and typical applications of full-text indexing, inverted files are superior to signature files in almost every respect, including speed, space, and functionality.

3.2 Designing an IR system

Our focus during IR system design is: On improving the performance and effectiveness of the system. The effectiveness of the system is measured in terms of precision, and recall. Stemming, stopwords, weighting schemes, and matching algorithms In improving performance efficiency. The concern here is:

- storage space usage, access time
- Compression, data/file structures, space-time tradeoffs

Text Compression

Text compression is about finding ways to represent the text in fewer bits or bytes. Text files can be compressed to make them smaller and faster to send, and unzipping files on devices has a

low overhead. The process of encoding involves changing the representation of a file so that the (binary) compressed output takes less space to store and takes less time to transmit while retaining the ability to reconstruct the original file exactly from its compressed representation.

Advantages:

- save storage space requirement.
 - speed up document transmission time
 - Takes less time to search the compressed text
- ✓ Common compression methods
- **Statistical methods:** which requires statistical information about frequency of occurrence of symbols in the document

E.g. Huffman coding

Estimate probabilities of symbols, code one at a time, shorter codes for high probabilities

- **Adaptive methods:** which constructs dictionary in the processing of compression

E.g. Ziv-Lempel compression:

\emdash Replace words or symbols with a pointer to dictionary entries

Huffman coding

Huffman coding is a lossless data compression algorithm. In this algorithm, a variable-length code is assigned to input different characters. The code length is related to how frequently characters are used. Most frequent characters have the smallest codes and longer codes for least frequent characters.

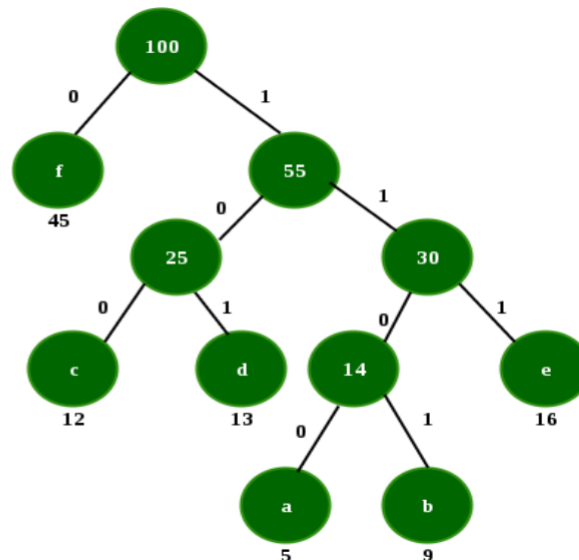
Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

- Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
- 2. Extract two nodes with the minimum frequency from the min heap.
- Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
- Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Example

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45



Code of A is 1100, Code of B is 1101, Code of C is 100, Code of D is 101, Code of E is 111
Code of F is 0

Exercise

1. Given the following, apply the Huffman algorithm to find an optimal binary code:

Character:	a	b	c	d	e	t
Frequency:	16	5	12	17	10	25

- Construct Huffman coding based on Given text: “for each rose, a rose is a rose”

Ziv-Lempel compression

The problem with Huffman coding is that it requires knowledge about the data before encoding takes place.

Huffman coding requires frequencies of symbol occurrence before codeword is assigned to symbols

\emdash Ziv-Lempel compression

- Not rely on previous knowledge about the data
- Rather builds this knowledge in the course of data transmission/data storage
- Ziv-Lempel algorithm (called LZ) uses a table of code-words created during data transmission;

each time it replaces strings of characters with a reference to a previous occurrence of the string

Lempel-Ziv Compression Algorithm

\emdash The multi-symbol patterns are of the form: $C_0C_1 \dots C_{n-1}C_n$. The prefix of a

pattern consists of all the pattern symbols except the last: $C_0C_1 \dots C_{n-1}$

Lempel-Ziv Output: there are three options in assigning a code to each symbol in the list

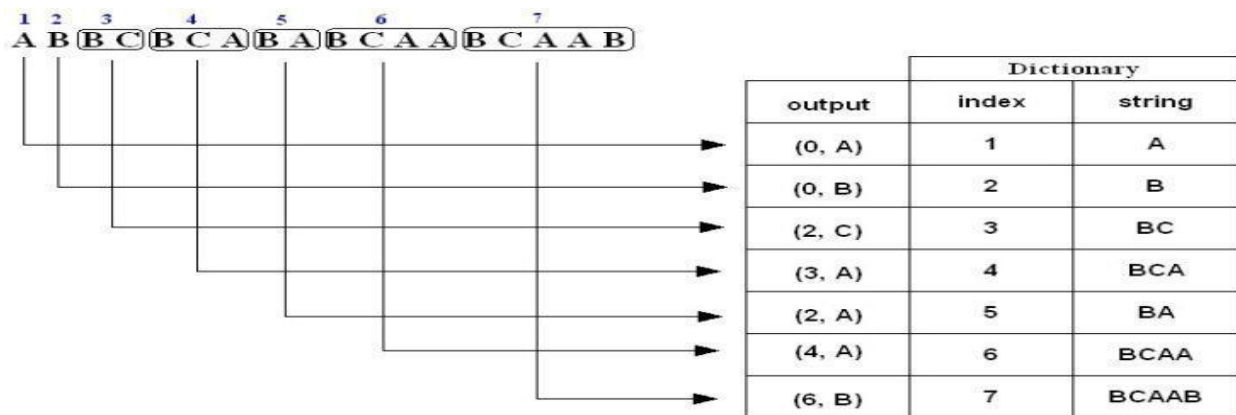
- If one-symbol pattern is not in dictionary, assign (0, symbol)

\emdash If multi-symbol pattern is not in dictionary, assign
(dictionaryPrefixIndex, lastPatternSymbol)

\emdash If the last input symbol or the last pattern is in the dictionary, assign
(dictionaryPrefixIndex,)

Example: LZ Compression

Encode (i.e., compress) the string **ABBCBCABABCAABCAAB** using the LZ algorithm



The compressed message is: **0A0B2C3A2A4A6B**

Exercise

Encode (i.e., compress) the following strings using the Lempel-Ziv algorithm.

\emdash Mississippi

\emdash ABBCBCABABCAABCAAB

\emdash SATATASACITASA.

3.2.1. Subsystems of IR system

The two subsystems of an IR system: Indexing and

Searching •**Indexing:**

It is an offline process of organizing documents using keywords extracted from the collection.

Indexing is used to speed up access to desired information from document collection as per users' query

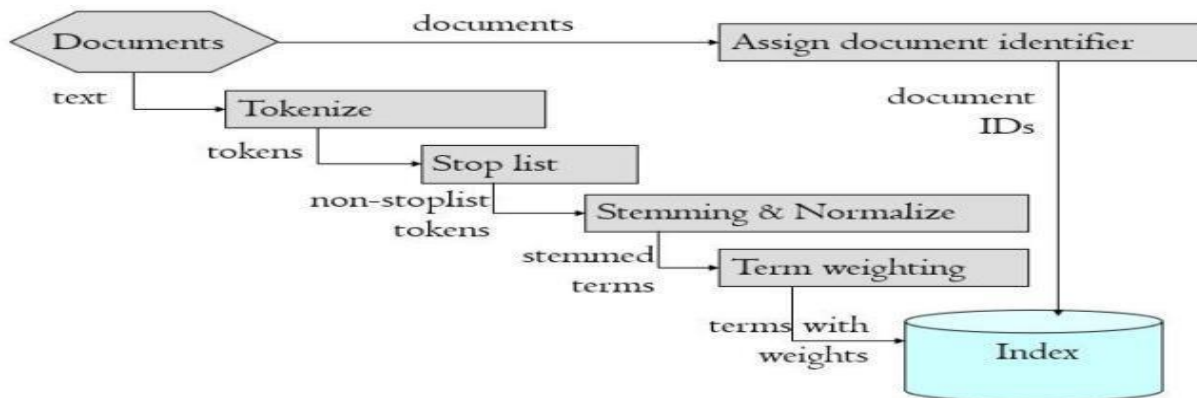


fig 3.1.indexing process

Searching

Is an online process that scans document corpus to find relevant documents that matches users query.

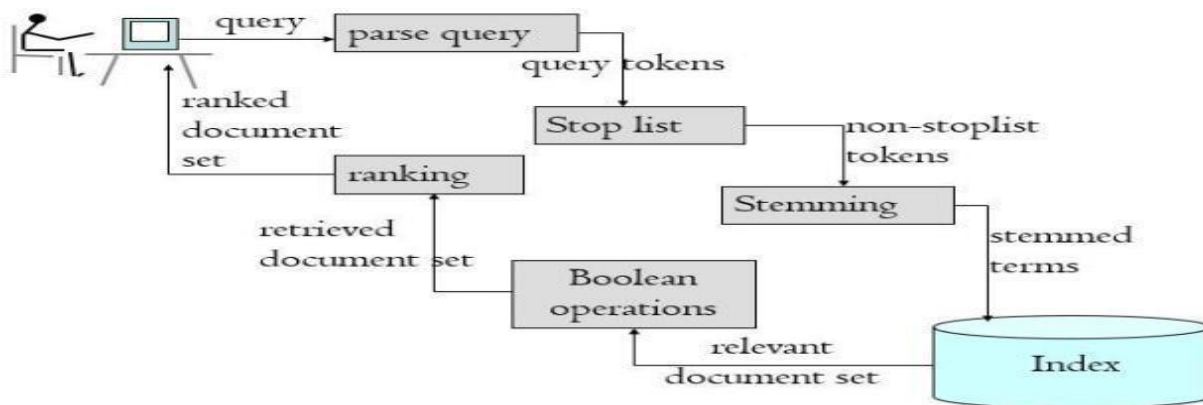


fig 3.2. searching process

Indexing and searching are inexorably connected. You cannot search that that was not first indexed in some manner or other. Indexing of documents or objects is done in order to be searchable. There are many ways to do indexing, to index one needs an indexing language. There are many indexing languages. Even taking every word in a document is an indexing language. Knowing searching is knowing indexing.

3.1. Inverted Files

An inverted file index [Fox et al. 1992] has two main parts: a search structure or vocabulary, containing all of the distinct values being indexed; and for each distinct value an inverted list, storing the identifiers of the records containing the value. Queries are evaluated by fetching the inverted lists for the query terms, and then intersecting them for conjunctive queries and merging them for disjunctive queries. To minimize buffer space requirements, inverted lists should be fetched in order of increasing length; thus, in a conjunctive query, the initial set of candidate answers are the records in the shortest inverted list, and processing of subsequent lists only reduces the size of this set. Once the inverted lists have been processed, the record identifiers must be mapped to physical record addresses. This is achieved with an address table, which can be stored in memory or on disk.

3.1.1. Compressed Inverted Files

The inverted lists themselves are sequences of record identifiers, sorted to allow fast query evaluation. Sorting of identifiers within inverted lists has another important benefit: the identifiers can be represented using variable-length codes that, for large text databases, compress the index by a factor of about six [Bell et al. 1993], to around 5% { 10% of the data size. This approach has the disadvantage that inverted lists must be decoded as they are retrieved, but such decompression can be fast. Moreover, by inserting a small amount of additional indexing information in each list a large part of the decompression can be avoided, so that on current hardware the limiting factor is transfer time, not decompression time [Moffat and Zobel 1996]. Indeed, the performance achieved by our implementation of inverted lists was one of the factors that spurred this investigation, and we assume inverted lists to be compressed throughout this paper.

An interesting feature of compressed inverted lists is that the best compression is achieved for the longest lists, that is, the most frequent terms. In the limit which, in the case of text indexing, is a term such as "the" that occurs in almost every record at most one bit per record is required. There is thus no particular need to eliminate common terms from the index: the decision as to whether or not to use the inverted lists for these terms to evaluate a query can be made, as it should be, at query evaluation time.

There are several widely held beliefs about inverted files that are either fallacious or incorrect once compression of index entries is taken into account:

- The assumption that sorting of inverted lists during query evaluation is an unacceptable cost. (This cost is illusory, because inverted lists should be maintained in sorted order.)

\emdash The assumption that a random disk access will be required for each record identifier for each term, as if inverted lists were stored as a linked list on disk. (They should be stored contiguously or at the very least in a linked list of blocks.)

\emdash The assumption that, if the vocabulary is stored on disk, $\log N$ accesses are required to fetch an inverted list, where N is variously the number of documents in the collection or the number of distinct terms in the collection. (This is only true if none of the nodes in the tree storing the vocabulary can be buffered in memory. Moreover, the base of the log is usually large, perhaps 1,000, so the true cost is one or at most two disk accesses.)

\emdash The assertion that inverted files are expensive to create. (Previous work in our research group has shown this to be fallacious [Moffat 1992; Moffat and Bell 1995], and the experiments reported below confirm that they are cheaper to build than signature file indexes.)

\emdash The assertion that inverted files are large, an oft-repeated claim being that they occupy between 50 and 300 percent of the space of the text they index [Haskin 1981] (With current techniques inverted files are stored in around 10% of the space of the text they index [Witten et al. 1994]).

The ordinary index would contain for each document, the index terms within it. But the inverted index stores for each term the list of documents where they appear. The benefit of using an inverted index comes from the fact that in IR we are interested in finding the documents that contain the index terms in the query. So, if we have an inverted index, we do not have to scan through all the documents in collection in search of the term. Inverted index may contain additional information like how many times the term appears in the document, the offset of the term within the document etc.

Example: Say there are three documents.

Doc I: Milk is nutritious

Doc2: Bread and milk tastes good

Doc3: Brown bread is better

After stop-word elimination and stemming, the inverted index looks like:

Terms	Documents containing the term
Better	3

4

Bread	2,3
Brown	3
Good	2
Milk	1,2
Nutritious	1
Taste	2

3.2.6. Types of an index file

Most surveys of file structures address themselves to applications in data management which is reflected in the terminology used to describe the basic concepts. There is one important distinction that must be made at the outset when discussing file structures. And that is the difference between the logical and physical organization of the data. On the whole a file structure will specify the logical structure of the data that is the relationships that will exist between data items independently of the way in which these relationships may actually be realized within any computer. It is this logical aspect that we will concentrate on.

The physical organization is much more concerned with optimizing the use of the storage medium when a particular logical structure is stored on, or in it. Typically for every unit of physical store there will be a number of units of the logical structure (probably records) to be stored in it. For example, if we were to store a tree structure on a magnetic disk, the physical organization would

be concerned with the best way of packing the nodes of the tree on the disk given the access characteristics of the disk.

Sequential File

A sequential file is the most primitive of all file structures. It has no directory and no linking pointers. The records are generally organized in lexicographic order on the value of some key. In other words, a particular attribute is chosen whose value will determine the order of the records. Sometimes when the attribute value is constant for a large number of records a second key is chosen to give an order when the first key fails to discriminate.

The implementation of this file structure requires the use of a sorting routine.

Its main advantages are:

- It is easy to implement;
- It provides fast access to the next record using lexicographic

order. Its disadvantages:

\emdash It is difficult to update – inserting a new record may require moving a large proportion of the file;

\emdash Random access is extremely slow.

Sometimes a file is considered to be sequentially organized despite the fact that it is not ordered according to any key. Perhaps the date of acquisition is considered to be the key value, the newest entries are added to the end of the file and therefore pose no difficulty to updating.

Example:

\emdash Given a collection of documents, they are parsed to extract words and these are saved with the Document ID.

Doc 1	I did enact Julius Caesar I was killed I the Capitol; Brutus killed me.
Doc 2	So let it be with Caesar. The noble Brutus has told you Caesar was ambitious

Sorting the Vocabulary

- After all documents have been tokenized, stopwords are removed, and normalization and stemming are applied, to generate index terms
- These index terms in sequential file are sorted in alphabetical order

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
I	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Sequential file

	Term	Doc No.
1	ambition	2
2	brutus	1
3	brutus	2
4	capitol	1
5	caesar	1
6	caesar	2
7	caesar	2
8	enact	1
9	julius	1
10	kill	1
11	kill	1
12	noble	2

Invertedfile

The importance of this file structure will become more apparent when Boolean Searches are discussed in the next chapter. For the moment we limit ourselves to describing its structure. An inverted file is a file structure in which every list contains only one record. Remember that a list is defined with respect to a keyword K, so every K-list contains only one record. This implies that the directory will be such that $n_i = h_i$ for all i, that is, the number of records containing K_i will equal the number of K_i -lists. So the directory will have an address for each record containing K_i

. For document retrieval this means that given a keyword we can immediately locate the addresses of all the documents containing that keyword.

Inverted file is technique that index based on sorted list of terms, with each term having links to the documents containing it. Building and maintaining an inverted index is a relatively low-cost risk.

On a text of n words an inverted index can be built in $O(n)$ time, n is number of terms

- Content of the inverted file: Data to be held in the inverted file includes:
- The vocabulary (List of terms)

- The occurrence (Location and frequency of terms in a document collection)

The occurrence: contains one record per term, listing Frequency of each term in a document

- TF_{ij} , number of occurrences of term t_j in document d_i
- DF_j , number of documents containing t_j
- max_i , maximum frequency of any term in d_i
- N , total number of documents in a collection
- $CF_{j,}$, collection frequency of t_j in n_j Locations/Positions of words in the text

Term Weighting: Term Frequency (TF)

\endash TF (term frequency) – Count the number of times a term occurs in document. f_{ij}
= frequency of term i in document j

\endash The more times a term t occurs in document d the more likely it is that t is relevant to the document, i.e. more indicative of the topic. If used alone, it favors common words and long documents. It gives too much credit to words that appears more frequently.

There is a need to normalize term frequency (tf)

<i>docs</i>	<i>t1</i>	<i>t2</i>	<i>t3</i>	RSV=Q.Di
D1	2	0	3	4
D2	1	0	0	1
D3	0	4	7	5
D4	3	0	0	1
D5	1	6	3	6
D6	3	5	0	3
D7	0	8	0	2
D8	0	10	0	2
D9	0	0	1	3
D10	0	3	5	5
D11	4	0	1	3
Q	1	2	3	
	<i>q1</i>	<i>q2</i>	<i>q3</i>	

Document Frequency

It is defined to be the number of documents in the collection that contain a term

DF = document frequency

Count the frequency considering the whole collection of documents. Less frequently a term appears in the whole collection, the more discriminating it is. df_i (document frequency of term i) = number of documents containing term i

– Having information about vocabulary (list of terms) speeds searching for relevant documents

Why location?

– Having information about the location of each term within the document helps for: user interface design: highlight location of search term. and proximity-based ranking: adjacency and near operators (in Boolean searching)

Why frequencies?

– Having information about frequency is used for: calculating term weighting (like IDF, $TF \cdot IDF$, ...), optimizing query processing In inverted file documents are organized by the terms/words they contain

<i>Term</i>	<i>CF</i>	<i>Document ID</i>	<i>TF</i>	<i>Location</i>
auto	3	2	1	66
		19	1	213
		29	1	45
bus	4	3	1	94
		19	2	7, 212
		22	1	56
taxi	1	5	1	43
train	3	11	2	3, 70
		34	1	40

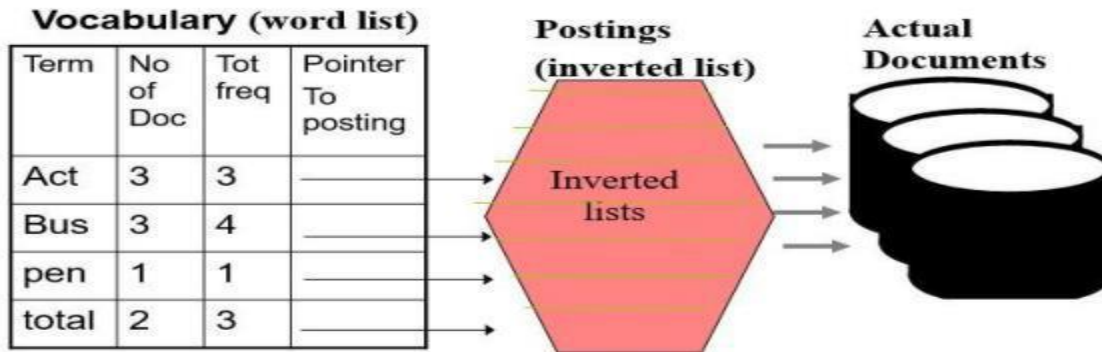
Table 4 Example of inverted file structure

This is called an **index file**

Text operations are performed before building the index. Organization of Index File An inverted index consists of two files:

\endash vocabulary file

\endash Posting file



Vocabulary file

A vocabulary file (Word list): stores all of the distinct terms (keywords) that appear in any of the documents (in lexicographical order) and for each word a pointer to posting file. In vocabulary file Records kept for each term j in the word list contains the following: term j , DF_j , CF_j and pointer to posting file.

Postings File (Inverted List): For each distinct term in the vocabulary, stores a list of pointers to the documents that contain that term. Each element in an inverted list is called a posting, i.e., the occurrence of a term in a document. It is stored as a separate inverted list for each column, i.e., a list corresponding to each term in the index file. Each list consists of one or many individual postings related to Document ID, TF and location information about a given term i

Advantage of dividing inverted file:

* **Keeping** a pointer in the vocabulary to the list in the posting file allows:

o the vocabulary to be kept in memory at search time even for large text collection, and Posting file to be kept on disk for accessing to documents

Separation of inverted file into vocabulary and posting file is a good idea.

Vocabulary: For searching purpose we need only word list. This allows the vocabulary to be kept in memory at search time since the space required for the vocabulary is small. • The vocabulary grows by $O(n\beta)$, where β is a constant between 0 – 1.

*Example: from 1,000,000,000 documents, there may be 1,000,000 distinct words. Hence, the size of index is 100 MBs, which can easily be held in memory of a dedicated computer.

– **Posting file** requires much more space. For each word appearing in the text we are keeping statistical information related to word occurrence in documents. Each of the postings pointer to the document requires an extra space of $O(n)$.

Example:

– Given a collection of documents, they are parsed to extract words and these are saved with the Document ID.

Doc 1

I did enact Julius
Caesar I was killed
I the Capitol;
Brutus killed me.

Doc 2

So let it be with
Caesar. The noble
Brutus has told you
Caesar was ambitious

- After all documents have been tokenized the inverted file is sorted by terms

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
I	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2



Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
has	1
I	1
I	1
I	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Step 1: Sorting the Vocabulary

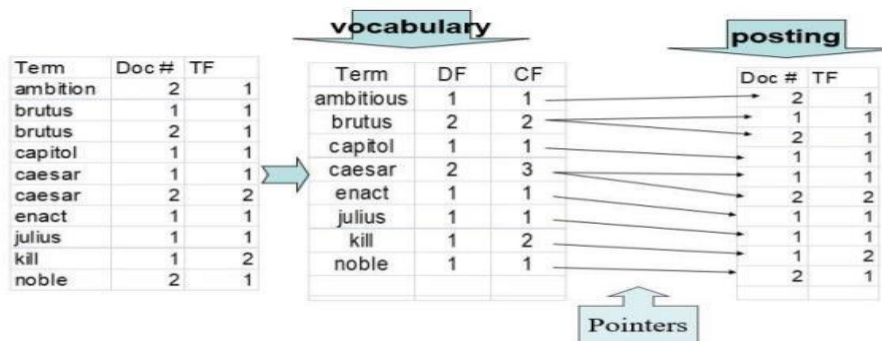
Step2: Remove stopwords, apply to stem & compute term frequency

- Multiple term entries in a single document are merged and frequency information added
- Counting number of occurrence of terms in the collections helps to compute TF

Term	Doc #		Term	Doc #	TF
ambition	2		ambition	2	1
brutus	1		brutus	1	1
brutus	2		brutus	2	1
capitol	1		capitol	1	1
caesar	1		caesar	1	1
caesar	2		caesar	2	2
caesar	2		caesar	2	2
enact	1		enact	1	1
julius	1		julius	1	1
kill	1		kill	1	2
kill	1		kill	1	2
noble	2		noble	2	1

Vocabulary and postings file

Step 3: The file is commonly split into a Dictionary and a Posting file



3.2. Tries, Suffix Trees and Suffix Arrays

3.2.1. Tries

Trie. ... In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings

Trie is the data structure very similar to Binary Tree. Trie data structure stores the data in particular fashion, so that retrieval of data became much faster and helps in performance. The name "TRIE" is coined from the word retrieve.

What are TRIE data structure usages or applications?

- **Dictionary Suggestions OR Auto complete dictionary**

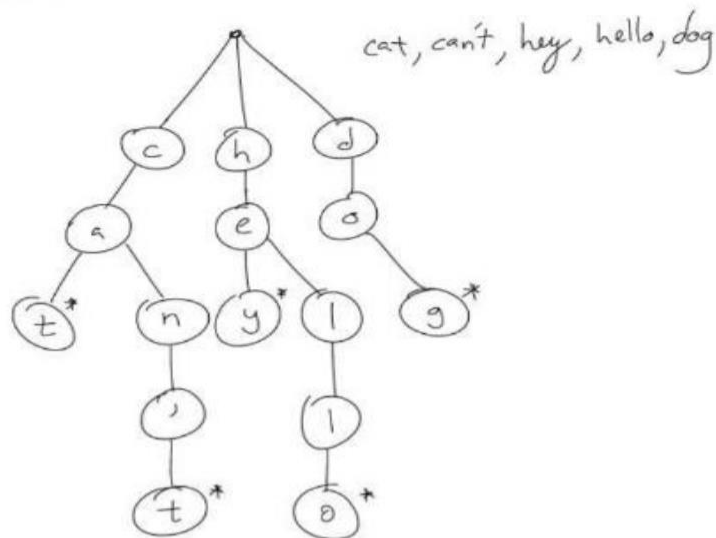
Retrieving data stored in Trie data structure is very fast, so it is most suited for application where retrieval are more frequently performed like Phone directory where contact searching operation is used frequently.

2. Searching Contact from Mobile Contact list OR Phone Directory

Auto suggestion of words while searching for anything in dictionary is very common. If we search for word "tiny", then it auto suggest words starting with same characters like "tine", "tin", "tinny" etc. Auto suggestion is very useful and Trie plays a nice role there, If say, Person doesn't know the complete spelling of the some word but know few, then rest of words starting with few characters can be auto suggested using TRIE data structure.

A prefix tree. or Erie (often pronounced "try"), is a tree whose nodes don't hold keys, but rather, hold partial keys. For example, if you have a prefix tree that stores strings, then each node would be a character of a string. If you have a prefix tree that stores arrays, each node would be an element of that array. The elements are ordered from the root. So if you had a prefix tree with the word "hello" in it, then the root node would have a child "h." and the "h" node would have a child, "e," and the "e" node would have a child node "l." etc. The deepest node of a key would have some sort of boolean flag on it indicating that it is the terminal node of some key. (This is important because the last node of a key isn't always a leaf node... consider a prefix tree with "dog" and "doggy" in it). Prefix trees are good for looking up keys with a particular prefix.

Example:



A trie (pronounced “try”) is a tree representing a collection of strings with one node per common prefix. Smallest tree such that:

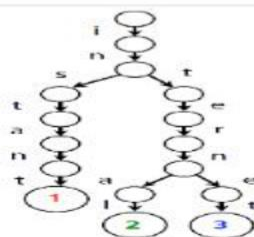
- ✓ Each edge is labeled with a character $c \in \Sigma$
- ✓ A node has at most one outgoing edge labeled c , for $c \in \Sigma$
- ✓ Each key is “spelled out” along some path starting at the root

Tries: example

Represent the following map with a Trie:

Key value

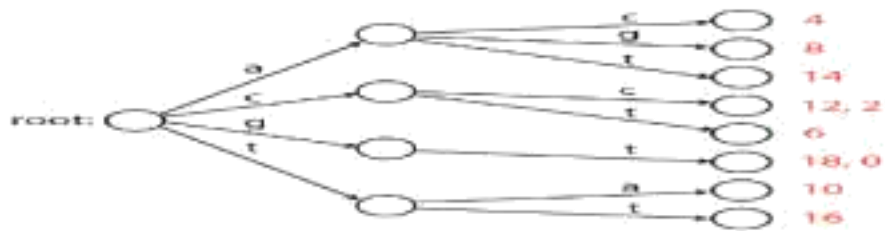
instant	1
internal	2
internet	3



ac	4
ag	8
at	14
cc	12
cc	2
ct	6
gt	18
gt	0
ta	10
tt	16

Tries: another example

We can index T with a trie. The trie maps substrings to offsets where they occur



3.2.2. Suffix Trees

Definition. Let $T = T[1..n]$ be a text of length n over a fixed alphabet E . A suffix tree for T is a tree with n leaves and the following properties:

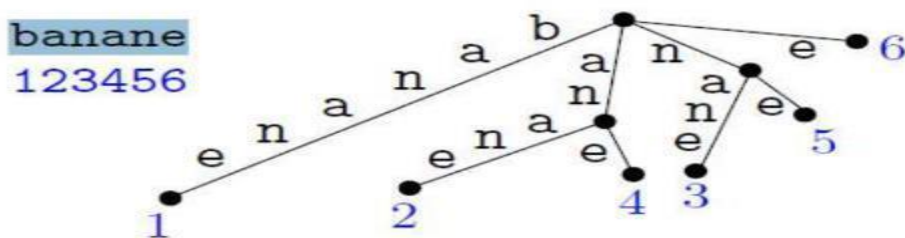
I. Every internal node other than the root has at least two children.

\endash Every edge is labeled with a nonempty substring of T .

\endash The edges leaving a given node have labels starting with different letters.

The concatenation of the labels of the path from the root to leaf i spells out the i -th suffix $T[i..n]$ of T . We denote $T[i..n]$ by T_i .

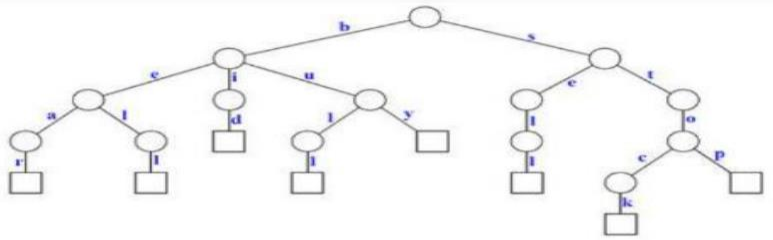
Example:



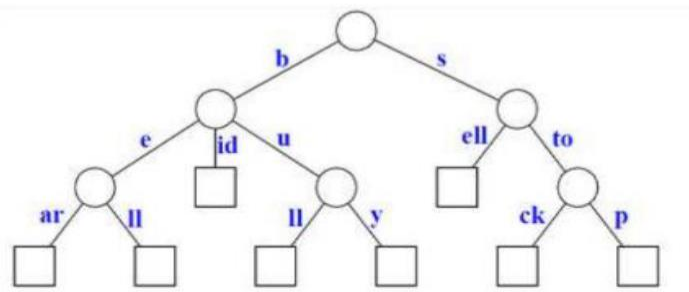
A Suffix Tree for a given text is a compressed trie for all suffixes of the given text.

{bear, bell, bid, bull, buy, sell, stock, stop}

Following is standard trie for the above input set of words



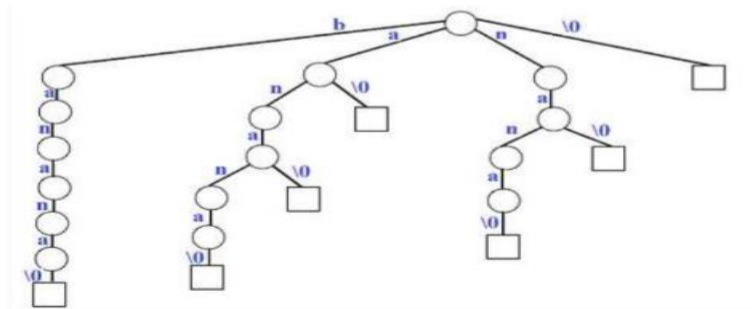
Following is the compressed trie. Compress Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.



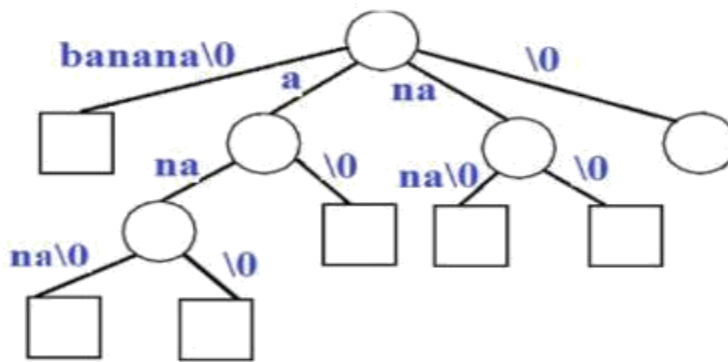
Let us consider an example text "banana\0" where '\0' is string termination character. Following are all suffixes of "banana\0"

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a trie, we get following.



If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text "banana\0"



3.2.3. Suffix Arrays

An array is an aggregate data structure that is designed to store a group of objects of the same or different types

3.3. Signature Files

In signature file indexes [Faloutsos 1992), each record is allocated a fixed-width signature, or bitstring, of w bits. Each word that appears in the record is hashed a number of times to determine the bits in the signature that should be set, with no remedial action taken if two or more distinct words should happen (as is inevitable) to set the same bit. Conjunctive queries are similarly hashed, then evaluated by comparing the query signature to each record signature; disjunctive queries are turned into a series of signatures, one per disjunct. Any record whose signature has a 1 -bit corresponding to every 1 -bit in the query signature is a potential answer. Each such record must be fetched and checked directly against the query to determine whether it is a false match-a record which the signature indicates may be an answer, but in fact is not-or a true match. Again, an address table is used to convert record numbers to addresses.

3.3.1. Types of Signature Files

Bitstring Signature Files

In signature file indexes [Faloutsos 1992), each record is allocated a fixed-width signature, or bitstring, of tv bits. Each word that appears in the record is hashed a number of times to determine the bits in the signature that should be set, with no remedial action taken if two or more distinct words should happen (as is inevitable) to set the same bit. Conjunctive queries are similarly hashed,

then evaluated by comparing the query signature to each record signature; disjunctive queries are turned into a series of signatures, one per disjunction. Any record whose signature has a 1-bit corresponding to every 1-bit in the query signature is a potential answer. Each such record must be fetched and checked directly against the query to determine whether it is a false match a record which the signature indicates may be an answer, but in fact is not or a true match. Again, an address table is used to convert record numbers to addresses.

Bitslice Signature Files

To reduce query-time access costs, the set of signatures can be transposed into a set of bitslices, so that a signature file contains one fixed-length slice for each bit position in the original string: the length of each slice is the number of records being fixed, that is, N bits for a database of N records. For bitsliced signature files, then, conjunctive query evaluation consists of: hashing the query into a signature: for some or all of the 1-bits in the query signature, fetching the corresponding bitslice: and the fetched slices together to form a bitmap of potential answers; and, for each 1-bit in the bitmap, retrieving the corresponding record and checking whether it is a true match. If each bitslice is sufficiently sparse, and the hash function used to set the bits sufficiently random, only a few false matches will remain after a subset of the bitslices have been processed. At this point it may be cheaper to cease processing of bitslices and start retrieving and checking records. That is, the number of bit slices actually fetched in a multi-term query might be only a little larger than the number of slices processed for a single-term query.

For a given bitsliced signature file index, the minimum number of bitslices that should be processed for a conjunctive query is indexed at s , typically in the range 6 to 8 [Sacks-Davis et al. 1987; Kent et al. 1990]. The index will have been created with parameters chosen to ensure that processing s bitslices will, in a probabilistic sense, reduce the number of false matches to a

specified level. If the number ϕ of query terms in a conjunctive query exceeds 5 then at least q slices should be fetched, since otherwise one or more of the query terms plays no part in the selection of answers. For a query of m disjuncts, at least sm slices are required, since these queries are processed as m independent queries and the answer sets merged. As for inverted files, either one or two disk accesses are then required to fetch each answer, depending on whether or not the address table is in memory. For text indexing, an application in which queries might have as few as one term, signatures are formed by letting each word in the record set s bits. To keep the number of false matches to a manageable level, signature width is such that each

bitslice is fairly sparse. For example, Kent et al. [1990] suggest that, to achieve good overall performance, approximately one bit in eight should be set. Note that false-match checking can be expensive in document databases, as it involves fetching a record (and thus either one or two disk accesses), parsing the record into words (often including stemming each word), and then evaluating the query directly against the list of terms. Fast query processing is thus only possible if the number of false matches is kept low. Combining these recommendations for signature density and s means that each distinct word in each record requires a notional space in the signature of about fifty bits, roughly the length of an average length word represented in ASCII. This allows an initial estimate of signature file size if each distinct term appears on average twice per record, then about 25 bits per word occurrence are required by the index, corresponding to approximately 50% of the space occupied by the input text. Note that standard bitstring signature files are claimed to be substantially more compact than this [Faloutsos 1992], since there is no disk access penalty for having a higher bit density. In fact, as will be shown below, the difference is negligible: for a given false-match rate bitstring signature files are only slightly smaller than

bitsliced signature files. Moreover, query processing costs are much greater, since the entire index must be scanned to determine candidate answers.

As for inverted file indexes, one processing heuristic is to select slices in increasing density, so that sparse slices are preferred to dense. Implementation of this technique requires that each slice be tagged with a density indication, which must be stored separately from the slice itself if the number of disk accesses is to be kept small. The selection process must also use knowledge of which query term corresponds to each bit in the query signature, since nothing is gained if all of the sparse slices correspond to the same query term.

Blocked Signature Files

A particular problem with bit sliced signature files is of scale: databases with large numbers of records have long slices. several of which must be retrieved in full regardless of the properties of the query. That is, despite that fact that the index is stored transposed and only a few bit positions in each signature must be inspected, index processing costs are guaranteed to rise linearly in the size of the database. For example, a database of one million records would have bit slices of one megabit each, and processing of even the simplest of queries would require transfer of approximately one megabyte of index data. This problem can be addressed by grouping records into blocks, so that each bit in each slice corresponds to B records, where B is the blocking factor. Slice length is reduced by a factor of B : to keep slice densities low signature width must be increased by a similar factor. To reduce the potential for block-level false matches in which a block contains all the query terms, but no record in the block is a match a different mapping from record number to block number can be used in each slice. Thus a record may be in block 6 in the

first slice. block II in the second slice, block 9 in the third slice, and so on. The number of different mappings is K . the number of slices (that is, signature width) is a multiple of K , and each mapping is applied to $w=K$ slices. This multi-organizational scheme [Kent et al. 1990] reduces but does not eliminate the potential for block-level false matches.

Queries are evaluated as for conventional bit sliced signature files: except that the bit slices must be decoded into record numbers as they are retrieved. It is possible to intersect bit slices by having a list of record numbers that are potential matches, and to use each subsequent bit slice to eliminate records that are not matches. by analogy with the query evaluation mechanism for inverted files: but the large number of records (about one in eight) that each slice implies is a match. and the disorder in each slice due to the complexities of the mappings. make this approach undesirable. It is also possible to decode each blocked bit slice into a full bit slice, then directly and the decoded slices. but the cost of decoding is high and dominates query evaluation time.

These problems were addressed in the atlas [Sacks-Davis et al. 1995] text database system as follows. The number of mappings K is set equal to s . the number of bits set per term, and the index is divided into K partitions, each of

$w=K$ slices. Then each term is allocated one slice in each partition, thus guaranteeing that each term uses all of the available mappings. The first few mappings (typically 3) are identical. so that the same blocking scheme is used in the first few partitions. Then, when queries are evaluated, the first slices fetched are from these partitions because such slices can be and without further processing.

Assuming that the selected slices are sufficiently sparse. the result of this operation is

a slice of length $N=8$ with a relatively small number of 1-bits. For each of these 1-bits, the corresponding record numbers are computed and the remaining slices probed to determine whether they have 1-bits for these records. The main disadvantage of this approach is that there is less flexibility in slice selection it decreases the likelihood of being able to use a sparse slice to reduce the number of candidates. During the development of atlas, this K-block approach was experimentally found to increase the number of block-level false matches, but greatly improved query response time. It is the implementation we assume in this paper.

The larger the blocking factor, the more slices must be fetched to eliminate block level false matches. Thus increasing the blocking factor increases the number of disk accesses, but reduces the amount of data to be transferred. It follows that choice of parameters needs to take actual disk characteristics into account.

CHAPTER FOUR

INFORMATION RETRIEVAL MODEL

At the end of this chapter every student must be able to:

- ✓ Define what model is
- ✓ Describe why model is needed in information retrieval
- ✓ Differentiate different types of information retrieval models
- ✓ know how to calculate and find the similarity of source documents to the given query
- ✓ Identify term frequency, document frequency, inverted document frequency, term weight and similarity measurements

4.1 What is Model?

Model- is the ideal abstraction of something which is working in the real world. There are two good reasons for having models of information retrieval. The first is that models guide research and provide the means for academic discussion. The second is that models can serve as a blueprint to implement an actual retrieval system. In IR, mathematical models are used to understand and reason about some behavior or phenomena in the real world. A model of an Information Retrieval predicts and explains what a user will find (relevant given the user query).

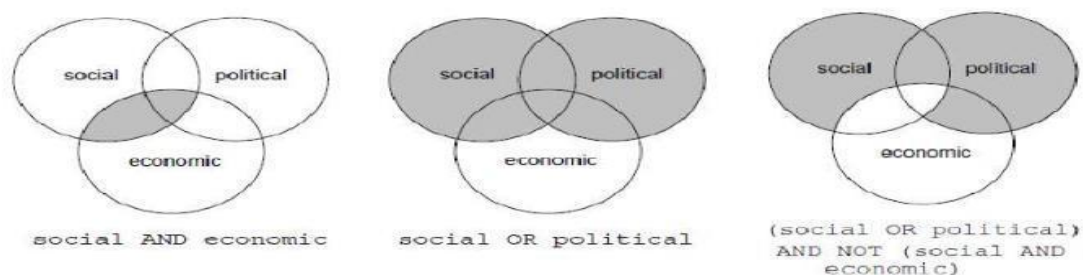
1. Boolean Model

The Boolean model is the first model of information retrieval and probably also the most criticized model.

The Boolean model is a simple retrieval model based on set theory and Boolean algebra. Since the concept of a set is quite intuitive, the Boolean model provides a framework that is easy to grasp by a common user of an IR system. Furthermore, the queries are specified as Boolean expressions which have precise semantics. Given its inherent simplicity and neat formalism, The Boolean model received great attention in past years and was adopted by many of the early commercial bibliographic systems

The model can be explained by thinking of a query term as a unambiguous definition of a set of documents. For instance, the query term economic simply defines the set of all documents that are indexed with the term economic. Using the operators of George Boole's mathematical logic,

query terms and their corresponding sets of documents can be combined to form new sets of documents. Boole defined **three basic operators**, the logical product called **AND**, the logical sum called **OR** and the logical difference called **NOT**. Combining terms with the AND operator will define a document set that is smaller than or equal to the document sets of any of the single terms. For instance, the query social AND economic will produce the set of documents that are indexed both with the term social and the term economic, i.e. the intersection of both sets. Combining terms with the OR operator will define a document set that is bigger than or equal to the document sets of any of the single terms. So, the query social OR political will produce the set of documents that are indexed with either the term social or the term political, or both, i.e. the union of both sets. The intersections of these discs and their complements divide the document collection into 8 non overlapping regions, the unions of which give 256 different Boolean combinations of 'social, political and economic documents. An advantage of the Boolean model is that it gives (expert) users a sense of control over the system. It is immediately clear why a document has been retrieved given a query. If the resulting document set is either too small or too big, it is directly clear which operators will produce respectively a bigger or smaller set. For untrained users, the model has a number of clear disadvantages. Its main disadvantage is that it does not provide a ranking of retrieved documents. The model either retrieves a document or not, which might lead to the system make rather frustrating decisions. For instance, the query social AND worker AND union will of course not retrieve a document indexed with party, birthday and cake, but will likewise not retrieve a document indexed with social and worker that lacks the term union. Clearly, it is likely that the latter document is more useful than the former, but the model has no means to make the distinction.



Source: - Information Retrieval Models (by Djoerd Hiemstr)

Example 1:

Assume there are four documents and we want to retrieve some documents for the query 'Information AND retrieval'. If the followings are our documents, which document can be retrieved for the given query? The documents are: Doc1 : Computer Information Retrieval Doc2: Computer Retrieval Doc3: Information Doc4: Computer Information Query: Information AND Retrieval Solution: put '1' in the following table if the terms exist in the document, and '0' if not. This is to identify the presence and absence of each term in the given documents.

	Computer	Information	Retrieval
Doc1	1	1	1
Doc2	1	0	1
Doc3	0	1	0
Doc4	1	1	0

Now since our query is: Information AND Retrieval, list down documents which contain both terms, Information AND Retrieval Doc1, Doc3, Doc4 AND Doc1, Doc2 Then take the intersection of the above documents (i.e. document which contain both the word information and retrieval). That is Doc1.

The Boolean Model: Example

Given the following three documents, Construct Term – document matrix
Boolean model for the query “gold silver truck”

- ✓ D1: “Shipment of gold damaged in a fire”
- ✓ D2: “Delivery of silver arrived in a silver truck”
- ✓ D3: “Shipment of gold arrived in a truck”

The table below shows the document –term (ti) matrix

	arrive	damage	deliver	fire	gold	silver	ship	truck
D1	0	1	0	1	1	0	1	0
D2	1	0	1	0	0	1	0	1
D3	1	0	0	0	1	0	1	1
query	0	0	0	0	1	1	0	1

The Boolean Model: Further Example

Given the following determine documents retrieved by the Boolean model based IR system

- Index Terms: K_1, \dots, K_8 .

- Documents:

\emdash $D_1 = \{K_1, K_2, K_3, K_4, K_5\}$

\emdash $D_2 = \{K_1, K_2, K_3, K_4\}$

\emdash $D_3 = \{K_2, K_4, K_6, K_8\}$

\emdash $D_4 = \{K_1, K_3, K_5, K_7\}$

\emdash $D_5 = \{K_4, K_5, K_6, K_7, K_8\}$

\emdash $D_6 = \{K_1, K_2, K_3, K_4\}$

- Answer: $\{D_1, D_2, D_4, D_6\} \subseteq (\{D_1, D_2, D_3, D_6\} \setminus \{D_3, D_5\}) = \{D_1, D_2, D_6\}$

- Query: $K_1 \cup (K_2 \cap K_3)$

The Boolean model predicts that each document is either relevant or nonrelevant. There is no notion of a partial match to the query conditions. For instance, let d_j be a document for which $d_j =$

- $(0,1,0)$. Document d_j includes the index term k_b but is considered non-relevant to the query $[q$

- $k_a \wedge (k_b \vee k_c)]$ ' The main disadvantages are that exact matching may lead to retrieval of too few or too many documents. Today, it is well known that index term weighting can lead to a substantial improvement in retrieval performance. Index term weighting brings us to the vector model.

2. Vector space model

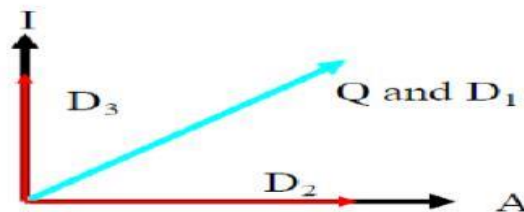
Vector Space Model is a classic model of document retrieval based on representing documents and queries as vectors of index terms. The vector model recognizes that the use of binary weights is too limiting and proposes a framework in which partial matching is possible. This is accomplished by assigning non-binary weights to index terms in queries and in documents. These term weights are ultimately used to compute the degree of similarity between each document stored in the system and the user query. By sorting the retrieved documents in decreasing order of this degree of similarity, the vector model takes into consideration documents that match the query terms only partially.

Gerard Salton introduced VSM in which partial matching is possible in the 1960s to oppose to the Boolean model where no partial matching method is used. So, the problem in the Boolean model (problem of no partial matching) is solved by Gerard Salton. The idea is: Meaning of a document is conveyed by the words used in that document. Documents and queries are mapped into term vector space. Each dimension represents tf-idf for one term. Documents are ranked by closeness to the query. Closeness is determined by a similarity score calculation.

In VSM, similarity measurements allow decisions to be made which documents are similar to each other and to keyword queries. VSM generates weighted term vectors for each document in the collection, and for the user query. Then the retrieval is based on the similarity between the query vector and document vectors. The output documents are ranked according to this similarity. The similarity is based on the occurrence frequencies of the keywords in the query and in the documents. The angle between query and document vectors is measured by using the cosine similarity measurement since both document and a user's query is represented as vectors in vector space model. Since relevance is modeled via similarity between a document and a query in vector space model (VSM), VSM assumes that if document vector V_1 is

closer to the query vector than another document vector V_2 , then the document represented by V_1 is more relevant than the one represented by V_2 .

Query: A I
 $D_1 - A \ I$
 $D_2 - A$
 $D_3 - I$



Idea: a document and a query are similar as their vectors point to the same general direction.

In VSM, to decide the similarity of the document to the given query, term weighting technique ($\text{tf} * \text{idf}$) is used. It is impossible to talk about term weight, if we do not have the concept of term frequency (tf) and inverse document frequency (idf). So, the following section describes about them.

Term frequency (TF): is the count of the term i in document j (the number of times a given term appears in that document). The term-frequency of a term is normalized by the maximum term frequency of any term in the document to bring the range of the term-frequency 0 to 1. Terms appearing in larger documents would always have larger term frequency, so, this count is usually normalized to prevent a bias towards longer documents.

Inverse Document Frequency (IDF): idf is used to measure whether the terms are common or rare across all documents. Higher idf value is obtained for rare terms whereas lower value for common terms.

The motivation for the usage of an IDF factor is that terms which appear in many documents are not very useful for distinguishing a relevant document from a non-relevant one. As with good clustering algorithms, the most effective term weighting schemes for IR try to balance these two effects. Definition Let N be the total number of documents in the system and n_i be the number of documents in which the index term k_i appears. Let $\text{freq}_{i,j}$ be the row frequency of term k_i in the document d_j (i.e., the number of times the term k_i is mentioned in the text of the document d_j). Then, the normalized frequency A_j of term k_i in document, d_j is given by

$$\text{Tf}_{i,j} = \frac{\text{freq}_{i,j}}{\text{Max}(\text{freq}_{i,j})} \quad \text{Equation 4.1}$$

Where

$\text{freq}_{i,j}$ is the count of how many times the term i appear in the document j .

When the maximum is computed over all terms which are mentioned in the text of document d_j . If the term K_i does not appear in the document d_j then $\text{fi},j=0$. Further, let idfi , inverse document frequency for k_i , be given by

$$\text{idf}(t) = \log \left[\frac{N}{n_i} \right] \quad \text{equ 4.2}$$

where

N = total number of document

Idf = inverse document frequency

$df(t)$ = Number of documents containing the term t

Then the product of equation 4.1 and equation 4.2 is called term weighting.

Term weighting (tw): Term weighting is calculated because we need to know which term is more important than the other term. Because every term in the given documents is not equally important, that is why we need to have term weight. Term weight of a given term is the product of the term frequency and inversed document frequency. It is given by:

Term weighting ($tf \cdot idf$) = $Tf_{i,j} * \log(N/df)$ Equation 4.3

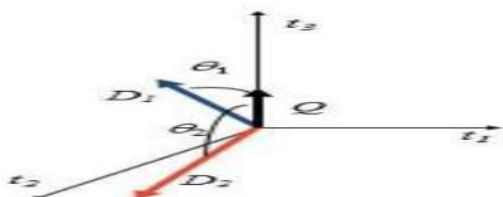
After calculating the term weight we have to calculate the length of the documents. The document length is given by: Document length = the square root summation of term weight square.

$$\text{Document length} = \sqrt{\sum \text{term weight}^2}$$

At the end we need to calculate similarity of the documents to the query. The widely used measure of similarity in vector space model is called the cosine similarity.

Similarity Measure

We now have vectors for all documents in the collection, a vector for the query, and how to compute similarity? A similarity measure is a function that computes the degree of similarity or distance between the document vector and query vector. Using a similarity measure between the query and each document: It is possible to rank the retrieved documents in the order of presumed relevance. Also, it is possible to enforce a certain threshold so that the size of the retrieved set can be controlled.



Similarity Measures method

- **Euclidean distance**

It is the most common similarity measure. Euclidean distance examines the root of square differences between coordinates of a pair of document and query terms. Similarity between vectors for the document d_i and query q can be computed as:

$$\text{sim}(d_i, q) = |d_i - q| = \sqrt{\sum_{i=1}^n (w_{ij} - w_{iq})^2}$$

where w_{ij} is the weight of term i in document j and w_{iq} is the weight of term i in the query.

- **Dot product**

The dot product is also known as the scalar product or inner product. the dot product is defined as the product of the magnitudes of query and document vectors. Similarity between vectors for the document d_i and query q can be computed as the vector inner product:

$$\text{sim}(d_i, q) = d_i \cdot q = \sum_{i=1}^n w_{ij} \cdot w_{iq}$$

where w_{ij} is the weight of term i in document j and w_{iq} is the weight of term i in the query q

- For binary vectors, the inner product is the number of matched query terms in the document
- For weighted term vectors, it is the sum of the products of the weights of the matched terms.

- **Cosine similarity (or normalized inner product)**

It projects to document and query vectors into a term space and calculate the cosine angle between these.

- Measures similarity between d_1 and d_2 captured by the cosine of the angle x between them.

$$\text{sim}(d_j, q) = \frac{\vec{d}_j \cdot \vec{q}}{|\vec{d}_j| |\vec{q}|} = \frac{\sum_{i=1}^n w_{i,j} w_{i,q}}{\sqrt{\sum_{i=1}^n w_{i,j}^2} \sqrt{\sum_{i=1}^n w_{i,q}^2}}$$

The denominator of the above formula can be replaced by the length of the document times length of the query. This means

$$\text{Sim}(\text{dj}, Q) = \frac{\sum(\text{term weight of docj} * \text{weight of query})}{\text{Length of docj} * \text{length of query}}$$

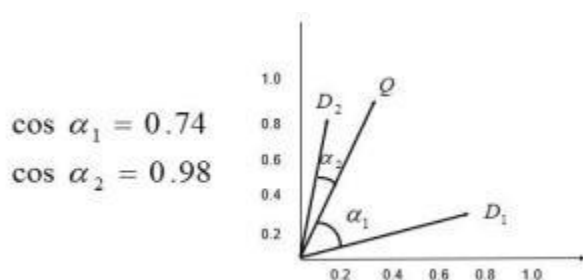
$$\text{Length of docj} = \sqrt{\sum_{i=1}^n w_{i,j}^2}$$

$$\text{Length } |\vec{d}_j| = \sqrt{\sum_{i=1}^n w_{i,j}^2}$$

Example 1: Computing Cosine Similarity Let say we have query vector $Q = (0.4, 0.8)$; and also document $D1 = (0.2, 0.7)$. Compute their **similarity using cosine?**

$$\begin{aligned} \text{sim}(Q, D_2) &= \frac{(0.4 * 0.2) + (0.8 * 0.7)}{\sqrt{[(0.4)^2 + (0.8)^2]} * \sqrt{[(0.2)^2 + (0.7)^2]}} \\ &= \frac{0.64}{\sqrt{0.42}} = 0.98 \end{aligned}$$

Example 2: Computing Cosine Similarity Let say we have two documents in our corpus; $D1 = (0.8, 0.3)$ and $D2 = (0.2, 0.7)$. Given query vector $Q = (0.4, 0.8)$, determine which document is more relevant one for the query?

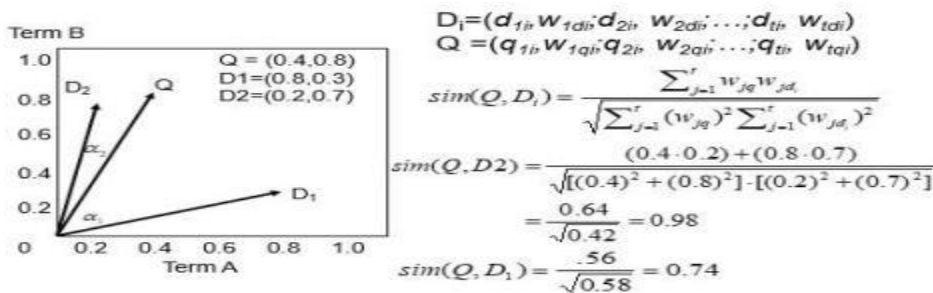


Example

Given three documents; $D1$, $D2$ and $D3$ with the corresponding TFIDF weight, Which documents are more similar using the three similarity measurement?

Terms	D1	D2	D3
<i>affection</i>	0.996	0.993	0.847
<i>Jealous</i>	0.087	0.120	0.466
<i>gossip</i>	0.017	0.000	0.254

Vector Space with Term Weights and Cosine Matching



Example: If the following three documents are given with one query, then, which document must be ranked first?

D1: New york times

D2: New York post

D3: Los Angeles times Query: new new times

Solution Step1: calculate inversed document frequency (IDF) for each term. $\text{Idf} = \log_2 N/\text{df}$

Step1: calculate inversed document frequency (IDF) for each term.

$$\text{Idf} = \log_2 N/\text{df}$$

$$\text{Idf}(\text{angeles}) = \log 3/1 = 1.584$$

$$\text{Idf}(\text{new}) = \log 3/2 = 0.584$$

$$\text{Idf}(\text{los}) = \log 3/1 = 1.584$$

$$\text{Idf}(\text{times}) = \log 3/2 = 0.584$$

$$\text{Idf}(\text{york}) = \log 3/2 = 0.584$$

$$\text{Idf}(\text{post}) = \log 3/1 = 1.584$$

Step2: calculate term frequency (tf)

Step2: calculate term frequency (tf)

	Angeles	Los	New	Post	Times	York
D1	0	0	1	0	1	1
D2	0	0	1	1	0	1
D3	1	1	0	0	1	0
Q	0	0	2/2=1	0	1/2=0.5	0

Step 3: calculate term weight (tw): $TW = tf * idf$

	Angeles	Los	New	Post	Times	York
D1	0	0	0.584	0	0.584	0.584
D2	0	0	0.584	1.584	0	0.584
D3	1.584	1.584	0	0	0.584	0
Q	0	0	2/2*0.584=0.584	0	0.5*0.584=0.292	0

Step 4: calculate document length or length of the document and length of the query

$$\text{Length of D1} = \sqrt{0.584^2 + 0.584^2 + 0.584^2} = 1.011$$

$$\text{Length of D2} = \sqrt{0.584^2 + 1.584^2 + 0.584^2} = 1.786$$

$$\text{Length of D3} = \sqrt{1.584^2 + 1.584^2 + 0.584^2} = 2.316$$

$$\text{Length of Q} = \sqrt{0.584^2 + 0.292^2} = 0.652$$

Step 5: calculate the similarity of each document to the query

$$\text{Similarity of (D1, Q)} = \frac{0 + 0.584 * 0.584 + 0.584 * 0.292}{1.011 * 0.652} = 0.776$$

$$\text{Similarity of (D2, Q)} = \frac{0 + 0.584 * 0.584}{1.786 * 0.652} = 0.292$$

$$\text{Similarity of (D3, Q)} = \frac{0 + \frac{0.584 * 0.292}{2.316 * 0.652}}{1} = 0.112$$

Therefore; since the value of $D1 > D2 > D3$, the document must be ranked as:
D1, D2, D3.

advantages of the vector space model are:

- \emdash its term-weighting scheme improves retrieval performance;
- \emdash its partial matching strategy allows retrieval of documents that approximate the query conditions: The cosine similarity measure returns value in the range 0 to 1. Hence, partial matching is possible.
- \emdash its cosine ranking formula sorts the documents according to their degree of similarity to the query. Ranking of the retrieved results according to the cosine similarity score is possible.

Although vector space model has the above advantage, it has also some

disadvantages. For instance: assumption of term independence, text VSM can't deal with lexical ambiguity and variability. e.g.: "he's affected by AIDS" and "HIV is a virus" don't have any words in common. So, in the Text VSM the similarity is 0: these vectors are orthogonal even though the concepts are related. on the other hand. similarity between "the laptop has a virus" and "HIV is a virus" is not 0: due to the ambiguity of the word "virus". VSM also un able to convey any relationship including the Boolean relationship, existing between the terms. It theoretically assumes that terms are statistically independent. There are no predictions about the techniques for effective and long documents are poorly represented because they have poor similarity values. In addition to this, the vector model has the **disadvantage** that index terms are assumed to be mutually independent (equation (TF-idf) does not account for index term dependencies).

3. Probabilistic model

Given a user information need (represented as a query) and a collection of documents (transformed into document representations), a system must determine how well the documents satisfy the query. Boolean or vector space models of IR: query-document matching done in a

formally defined but semantically imprecise calculus of index terms. An IR system has an uncertain understanding of the user query, and makes an uncertain guess of whether a document satisfies the query. Probability theory provides a principled foundation for such reasoning under uncertainty. Probabilistic models exploit this foundation to estimate how likely it is that a document is relevant to a query.

In a probabilistic method, one usually computes the “conditional” probability $P(D|R)$

that a given document D is observed on a random basis given event R , that d is relevant to a given query. If, as is typically the case, query and document are represented by sets of terms, then $P(D|R)$ is calculated as a function of the probability of occurrence of these terms in relevant vs. non-relevant documents. The term probabilities are analogous to the term weights in the vector space model (and may be calculated using the same statistical measures). A probabilistic formula is used to calculate $P(D|R)$, in place of the vector similarity formula,

e.g., cosine similarity, used to calculate relevance ranking in the vector space model. The probability formula depends on the specific model used, and also on the assumptions made about the distribution of terms, e.g., how terms are distributed over documents in the set of relevant documents, and in the set of non-relevant documents.

More generally, $P(D|R)$ may be computed based on any clues available about the document, e.g., manually assigned index terms (concepts with which the document deals, synonyms, etc.) as well as terms extracted automatically from the actual text of the document. Hence, we want to calculate $P(D|A, B, C \dots)$, i.e., the probability that the given document, D , is relevant, given the clues A, B, C , etc. As a further complication, the clues themselves may be viewed as complex. e.g., if the presence of term t is a clue to the relevance of document D , t may be viewed as a cluster of related clues, e.g., its frequency in the query, its frequency in the document, its idf, synonyms, etc. This has led to the idea of a “staged” computation, in which a probabilistic model is first applied to each composite clue (stage one), and then applied to the combination of these composite clues (stage two).

Several approaches that try to define term weighting more formally are based on probability theory. The notion of the probability of something, for instance the probability of relevance notated as $P(R)$, is usually formalized through the concept of

an experiment, where an experiment is the process by which an observation is made.

The set of all possible outcomes of the experiment is called the sample space. In the case of $P(R)$ the sample space might be {relevant, irrelevant}. and we might define the random variable R to take the values {0, 1}. where 0=irrelevant and 1=relevant.

Let's define an experiment for which we take one document from the collection at random: If we know the number of relevant documents in the collection, say 100 documents are relevant, and we know the total number of documents in the collection, say 1 million, then the quotient of those two defines the probability of relevance

$$P(R=1) = 100/1000000 = 0.0001.$$

Suppose further more that $P(D_k)$ is the probability that a document contains the term k with the sample space {0, 1}. (0=the document does not contain term k , 1=the document contains term k), then we will use $P(R, D_k)$ to denote the joint probability

distribution with outcomes {(0, 0), (0, 1), (1, 0) and (1, 1)}. and we will use $P(R, D_k)$ to denote the conditional probability distribution with outcomes {0, 1}. So, $P(R=1, D_k=1)$ is the probability of relevance if we consider documents that contain the term k .

The probabilistic retrieval model

Whereas Maron and Kuhns introduced ranking by the probability of relevance. it was Stephen Robertson who turned the idea into a principle. He formulated the probability ranking principle, which he attributed to William Cooper, as follows: Ranked retrieval] setup: given a collection of documents, the user issues a query, and an ordered list of documents is returned. Assume binary notion of relevance: R_d is a random dichotomous variable, such that

— $R_d = 1$ if document d is relevant w.r.t query q

- $R_d = 0$ otherwise

Probabilistic ranking orders documents decreasingly by their estimated probability

of relevance w.r.t. query: $P(R = 1 | d, q)$

PRP in brief

If the retrieved documents (w.r.t a query) are ranked decreasingly on their probability of relevance. then the effectiveness of the system will be the best that is obtainable <<If a reference retrieval system's response to each request is a ranking of the documents in the collections in order of decreasing probability of usefulness to the user who submitted the request. where the probabilities are estimated as accurately as possible on the basis of whatever data has been made available to the system for this purpose, then the overall effectiveness of the system to its users will be the best that is obtainable on the basis of that data>>.

Bayesian based probabilistic model

Assume that document 'y' is in the collection

Probability that if a non relevant document retrieved, it is 'y' is $p(y/NR) = 0.2$

Probability of non relevant documents in the collection is $p(NR) = 0.6$

Probability of 'y' in the collection is $p(y) = 0.4$

- What is the probability that y is non relevant document?
- Is the document is relevant or non relevant?

Solution

- $P(NR/Y) = p(y/NR) p(NR)/p(y)$

$$P(NR/Y) = 0.2 * 0.6 / 0.4 = \underline{0.3}$$

- $P(R/Y) + p(NR/Y) = 1$

$$P(R/Y) = 1 - p(NR/Y)$$

$$P(R/Y) = 1 - 0.3$$

$$\underline{P(R/Y) = 0.7}, \quad \text{hence the document is relevant}$$

CHAPTER FIVE

RETRIEVAL EVALUATION

5.1 Information retrieval system evaluation

Before the final implementation of an information retrieval system, an evaluation of the system is usually carried out. The type of evaluation to be considered depends on the objectives of the retrieval system.

We need a test collection consisting of three things: 1. A document collection 2. A test suite of information needs, expressible as queries 3. A set of relevance judgments, standard a binary assessment of either relevant or non relevant 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.

5.2. Why System Evaluation?

✓ It provides the ability to measure the difference between IR systems

— How well do our searches engines work?

— Is system A better than B?

— Under what conditions?

- Evaluation drives what to research

Identify techniques that work and do not work

There are many retrieval models/ algorithms/ systems

- Which one is the best?

- What is the best component for:

\emdash Similarity measures (dot-product, cosine, ...)

\emdash Index term selection (stop-word removal, stemming...)

- Term weighting (TF, TF-IDF,...)

Types of Evaluation Strategies

- System-centered studies
- Given documents, queries, and relevance judgments

Try several variations of the system

Measure which system returns the "best" hit list

- User-centered studies
 - Given several users, and at least two retrieval systems
 - Have each user try the same task on both systems
 - Measure which system works the "best" for users information need
 - More user-oriented measures
 - Satisfaction, informativeness
 - Other types of measures
 - Time, cost-benefit, error rate, task analysis
 - Evaluation of user characteristics
 - Evaluation of interface
 - Evaluation of process or interaction

Evaluation Criteria

What are some main measures for evaluating an IR system's performance?

\emdash Efficiency: time, space

— Speed in terms of retrieval time and indexing time

— Speed of query processing —

The space taken by corpus vs. index —

Index size: Index/corpus size ratio

- Effectiveness
 - How is a system capable of retrieving relevant documents from the collection?
 - Is a system better than another one?
 - User satisfaction: How "good" are the documents that are returned as a response to user query?

Difficulties in Evaluating IR System

¶ IR systems essentially facilitate communication between a user and document collections

¶ Relevance is a measure of the effectiveness of communication

- Effectiveness is related to the relevancy of retrieved items.
- Relevance: relates to problem, information need, query and a document or surrogate

¶ Relevancy is not typically binary but continuous.

- Even if relevancy is binary, it is a difficult judgment to make.

¶ Relevance judgments is made by

- The user who posed the retrieval problem
- An external judge
- Is the relevance judgment made by users and external person the same?

¶ Relevance judgment is usually:

- Subjective: Depends upon a specific user's judgment.
- Situational: Relates to user's current needs.
- Cognitive: Depends on human perception and behavior.
- Dynamic: Changes over time.

Evaluation of unranked retrieval sets Given these ingredients, how is system effectiveness measured?

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 a set of documents for a query.

Consider an example information request I (of a test reference collection) and its set R of relevant documents. Let $|R|$ be the number of documents in this set. Assume that a given retrieval strategy (which is being evaluated) processes the information request I and generates a documented answer set A .

Let $|A|$ be the number of documents in this set. Further, let $|R \cap A|$ be the number of documents in the intersection of the sets R and A . Figure 5.1 illustrates these sets. The recall and precision

measures are defined as follows. The recall is the fraction of the relevant documents (the set R) which has been retrieved i.e.,

$$Recall = \frac{|Ra|}{|R|}$$

¶

Recall (R) is the fraction of relevant documents that are retrieved

— The ability of the search to find all of the relevant items in the corpus

— Recall is percentage of relevant documents retrieved from the database in response to users query.

Recall= #(relevant items retrieved)/ #(relevant items) = P(retrieved/relevant)

Precision (P) is the fraction of retrieved documents PRECISION that are relevant

¶

The ability to retrieve top-ranked documents that are mostly relevant

¶

Precision is percentage of retrieved documents that are relevant to the query (i.e. number of retrieved documents that are relevant).

These notions can be made clear by examining the following contingency table:

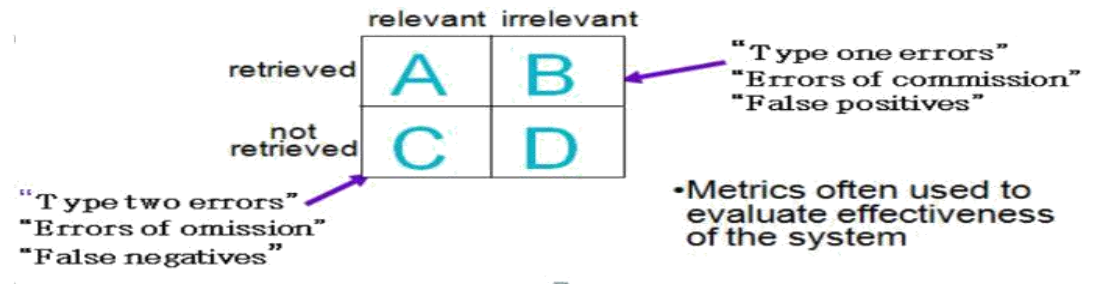
*	Relevant	Nonrelevant
Retrieved	true positives (tp)	false positives (fp)
Not retrieved	false negatives (fn)	true negatives (tn)

$$\text{precision} = \text{tp} / (\text{tp} + \text{fp})$$

$$\text{Recall} = \text{tp} / (\text{tp} + \text{fn})$$

Measuring Retrieval Effectiveness: Retrieval of documents may result in:

- **False negative (false drop):** some relevant documents may not be retrieved.
- **False positive:** some irrelevant documents may be retrieved.
- For many applications a good index should not permit any false drops, but may permit a few false positives.



Precision is the fraction of the retrieved documents (the set A) which is relevant i.e.,

$$Precision = \frac{|Ra|}{|A|}$$

Recall and precision, as defined above, assume that all the documents in the answer set A have been examined (or seen). However, the user is not usually presented with all the documents in the answer set A at once. Instead, the

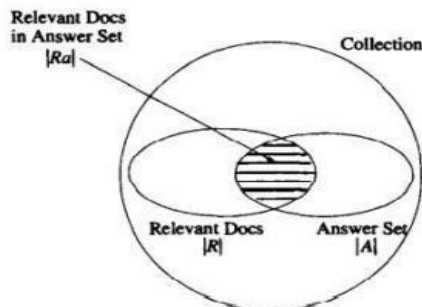


fig 5. 1 Precision and recall for a given example information request.

Documents in A are first sorted according to a degree of relevance (i.e., a ranking is generated). The user then examines this ranked list starting from the top document. In this situation, the recall and precision measures vary as the user proceeds with his examination of the answer set A. Thus, proper evaluation requires plotting a precision versus recall curve as follows. As before, consider a reference collection and its set of example information requests. Let us focus on a given example information request for which a query q is formulated. Assume that a set R_q containing the relevant

documents for q has been defined. Without loss of generality, assume further that the set R_q is composed of the following documents.

$$R_q = \{d_3, d_5, d_9, d_{25}, d_{39}, d_{44}, d_{56}, d_{71}, d_{89}, d_{123}\}$$

Thus, according to a group of specialists, there are ten documents which are relevant to the query q . Consider now a new retrieval algorithm which has just been designed. Assume that this algorithm returns, for the query q , a ranking of the documents in the answer set as follows.

Ranking for query q :

- | | | |
|----------------|----------------|---------------|
| 1. d_{123} • | 6. d_9 • | 11. d_{38} |
| 2. d_{84} | 7. d_{511} | 12. d_{48} |
| 3. d_{56} • | 8. d_{129} | 13. d_{250} |
| 4. d_6 | 9. d_{187} | 14. d_{113} |
| 5. d_8 | 10. d_{25} • | 15. d_3 • |

The documents that are relevant to the query q are marked with a bullet after the document number. If we examine this ranking, starting from the top document, we observe the following points. First, the document d_{123} which is ranked as number 1 is relevant. Further, this document corresponds to 10% of all the relevant documents in the set R_q

• Thus, we say that we have a precision of 100% at 10% recall. Second, the document d_{56} which is ranked as number 3 is the next relevant document. At this point, we say that we have a precision of roughly 66% (two documents out of three are relevant) at 20% recall (two of the ten relevant documents have been seen). Third, if we proceed with our examination of the ranking generated

we can plot a curve of precision versus recall as illustrated in Figure 5.2. The precision at levels of recall higher than 50% drops to 0 because not all relevant documents have been retrieved. This precision versus recall curve is usually based on 11 (instead of ten) standard recall levels which are 0%, 10%, 20%, ..., 100%. For the recall level 0%, the precision is obtained through an interpolation procedure as detailed below. In the above example, the precision and recall figures are for a single query. Usually, however, retrieval algorithms are evaluated by running them for several distinct queries. In this case, for each query a distinct precision versus recall curve is generated. To evaluate the retrieval performance of an algorithm over

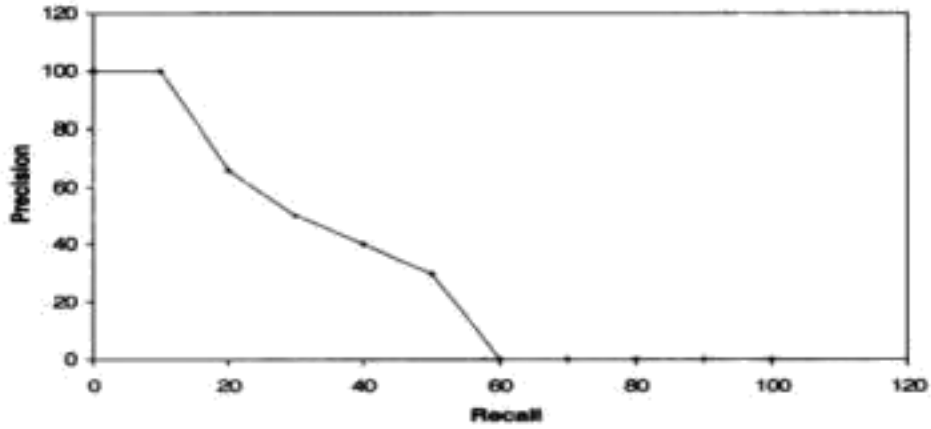


fig 5. 2 Precision at 11 standard recall levels

All test queries, we average the precision figures at each recall level as follows.

$$\bar{P}(r) = \sum_{i=1}^{N_q} \frac{P_i(r)}{N_q}$$

Example 1: Assume there are a total of 14 relevant documents, compute precision and recall?

Hits 1-10										
	R	NR	NR	NR	R	R	NR	R	NR	NR
Recall:	1/14	1/14	1/14	1/14	2/14	3/14	3/14	4/14	4/14	4/14
Precision:	1/1	1/2	1/3	1/4	2/5	3/6	4/7	4/8	4/9	4/10
Hits 10-20										
	R	NR	NR	NR	NR	R	NR	NR	NR	NR
Recall	5/14	5/14	5/14	5/14	5/14	6/14	6/14	6/14	6/14	6/14
Precision	5/11	5/12	5/13	5/14	5/15	6/16	6/17	6/18	6/19	6/20

Note:

R-relevant

NR-non relevant

Example 2:

Let total number of relevant documents = 6, compute recall and precision for each cut off point

n:

n	doc #	relevant	Recall	Precision
1	588	x	0.167	1
2	589	x	0.333	1
3	576			
4	590	x	0.5	0.75
5	986			
6	592	x	0.667	0.667
7	984			
8	988			
9	578			
10	985			
11	103			
12	591			
13	772	x	0.833	0.38
14	990			

Calculate

Exercise: An IR system returns 8 relevant documents, and 10 non relevant documents. There are a total of 20 relevant documents in the collection. What is the precision of the system on this search, and what is its recall?

¥emdash

R- Precision

Precision at the R-th position in the ranking of results for a query, where R is the total number of relevant documents. — Calculate precision after R documents are seen — Can be averaged over all queries

n	doc #	relevant	
1	588	x	
2	589	x	
3	576		
4	590	x	
5	986		
6	592	x	
7	984		
8	988		
9	578		
10	985		
11	103		
12	591		
13	772	x	
14	990		

R = # of relevant docs = 6

R-Precision = $4/6 = 0.67$

Problems with both precision and recall

- Number of irrelevant documents in the collection is not taken into account.
- Recall is undefined when there is no relevant document in the collection.
- Precision is undefined when no document is retrieved. Other measures
- Noise = retrieved irrelevant docs / retrieved docs
- Silence/Miss = non-retrieved relevant docs / relevant docs

— Noise = I — Precision; Silence = I — Recall

F-measure A single measure that trades off precision versus recall is the F measure, which is the weighted harmonic mean of precision and recall: One measure of performance that takes into accounts both recall and precision. Harmonic mean of recall and precision:

F measure = $2PR / P+R$

Query Languages and Operations

Information is the main value of Information Society. The recent developments in computing power and telecommunications, along with the constant drop of Internet access costs and data management and storing, created the right conditions for the global diffusion of the Web and, more generally, of new research tools able to analyze information and their contents. Depending on the particular application scenario and on the type of information that has to be managed and searched, different techniques need to be devised.

The dictionary definition of query is a set of instructions passed to a database to retrieve particular data. A query is the formulation of a user information need. A query is composed of keywords and the documents containing such keywords are searched for popular and Intuitive, Easy to express, Allow fast ranking. A query is formulation of a user information need.

Query Languages: A source language consisting of procedural operators that invoke functions to be executed.

6.1. Key word based queries

Queries are combinations of words. The document collection is searched for documents that contain these words. Word queries are intuitive, easy to express and provide fast ranking. The concept of word must be defined as a sequence of letters terminated by a separator (period, comma, blank, etc). Definition of letter and separator is flexible: e.g., hyphen could be defined as a letter or as a separator. Usually, “trivial words” (such as “a”, “the”, or “of”) are ignored.

popular Keyword-based queries are

The most elementary query that can be formulated in a text retrieval system is a word. Text documents are assumed to be essentially long sequences of words. Although some models present a more general view, virtually all models allow us to see the text in this perspective and to search words. Some models are also able to see the internal division of words into letters. These latter models permit the searching of other types of patterns. The set of words retrieved by these extended queries can then be fed into the word treating machinery, say to perform thesaurus expansion or for ranking purposes. A word is normally defined in a rather simple way. The alphabet is split into ‘letters’ and ‘separators,’ and a word is a sequence of letters surrounded by separators. More

complex models allow us to specify that some characters are not letters but do not split a word, e.g. the hyphen in ‘on-line.’ It is good practice to leave the choice of what is a letter and what is a separator to the manager of the text database. The division of the text into words is not arbitrary, since words carry a lot of meaning in natural language. Because of that, many models (such as the vector model) are completely structured on the concept of words, and words are the only type of queries allowed (moreover, some systems only allow a small set of words to be extracted from the documents).

The result of word queries is the set of documents containing at least one of the words of the query. Further, the resulting documents are ranked according to a degree of similarity to the query. To support ranking, two common statistics on word occurrences inside texts are commonly used: ‘term frequency’ which counts the number of times a word appears inside a document and ‘inverse document frequency’ which counts the number of documents in which a word appears. See Chapter 2 for more details. Additionally, the exact positions where a word appears in the text may be required for instance, by an interface which highlights each occurrence of that word.

Single-word queries:

- ✓ A query is a single word
- ✓ Simplest form of query.
- ✓ All documents that include this word are retrieved.
- ✓ Documents may be ranked by the frequency of this word in the document.

Phrase queries: A query is a sequence of words treated as a single unit. Also called “literal string” or “exact phrase” query, Phrase is usually surrounded by quotation marks, All documents that include this phrase are retrieved, Usually, separators (commas, colons, etc.) and “trivial words” (e.g., “a”, “the”, or “of”) in the phrase are ignored, In effect, this query is for a set of words that must appear in sequence, Allows users to specify a context and thus gain precision.

∅ Example: “United States of America”.

Multiple-word queries: A query is a set of words (or phrases). Two interpretations:

- ✓ A document is retrieved if it includes any of the query words.
- ✓ A document is retrieved if it includes each of the query words.

- Documents may be ranked by the number of query words they contain: A document containing n query words is ranked higher than a document containing $m < n$ query words. Documents containing all the query words are ranked at the top. Documents containing only one query word are ranked at bottom. Frequency counts may still be used to break ties among documents that contain the same query words.

Example:

— The phrase “Venetian blind” finds documents that discuss Venetian blinds.

— The set (Venetian, blind) finds in addition documents that discuss blind Venetians.

Proximity queries: Restrict the distance within a document between two search terms. Important for large documents in which the two search words may appear in different contexts. Proximity specifications limit the acceptable occurrences and hence increase the precision of the search.

- General Format: Word1 within mm units of Word2. Unit may be character, word, paragraph, etc.
- and » intersection
- or > union
- But difference

The use of But prevents creation of very large answers: not B computes all the documents that do not include B (complement), whereas A But B limits the universe to the documents that include A. Precedence: But, and, or: use parentheses to override: process left-to-right among operators with the same precedence.

Examples:

- Computer or server But mainframe: Select all documents that discuss computers, or documents that discuss servers but do not discuss mainframes.
- (Computer or server) But mainframe: Select all documents that discuss computers or

servers, do not select any documents that discuss mainframes.

- Examples: united within 5 words of american: Finds documents that discuss “United Airlines and American Airlines” but not “United States of America and the American dream”. Another example is: nuclear within 0 paragraphs of cleanup: Finds documents that discuss “nuclear” and “cleanup” in the same paragraph.

Boolean queries:

The oldest (and still heavily used) form of combining keyword queries is to use Boolean operators. A Boolean query has a syntax composed of atoms (i.e., basic queries) that retrieve documents, and of Boolean operators which work on their operands (which are sets of documents) and deliver sets of documents. Since this scheme is in general compositional (i.e., operators can be composed over the results of other operators), a query syntax tree is naturally defined, where the leaves correspond to the basic queries and the internal nodes to the operators. The query syntax tree operates on an algebra over sets of documents (and the final answer of the query is also a set of documents).

Describe the information needed by relating multiple words with Boolean operators.

¥emdash

Operators: and, or, But

OR the query (e1 OR e2) selects all documents which satisfy e1 or e2. Duplicates are eliminated.

AND the query (e1 AND e2) selects all documents which satisfy both e1 and e2.

BUT the query (e1 BUT e2) selects all documents which satisfy e1 but not e2. Notice that classical Boolean logic uses a NOT operation, where (NOT e2) is valid whenever e2 is not.

¥emdash

But corresponds to and not

¥emdash

Semantics: For each query word w_a corresponding set D_w is constructed that includes the documents that contain w .

¥emdash

The Boolean expression is then interpreted as an expression on the corresponding

document sets with corresponding set operators:

\emdash Computer But (server or mainframe): Select all documents that discuss computers, and do not discuss either servers or mainframes.

Natural Language

Pushing the fuzzy Boolean model even further, the distinction between AND and OR can be completely blurred, so that a query becomes simply an enumeration of words and context queries. All the documents matching a portion of the user query are retrieved. Higher ranking is assigned to those documents matching more parts of the query. The negation can be handled by letting the user express that some words are not desired, so that the documents containing them are penalized in the ranking computation. A threshold may be selected so that the documents with very low weights are not retrieved. Under this scheme we have completely eliminated any reference to Boolean operations and entered into the field of natural language queries. In fact, one can consider that Boolean queries are a simplified abstraction of natural language queries. A number of new issues arise once this model is used, especially those related to the proper way to rank an element with respect to a query. The search criterion can be re-expressed using a different model, where documents and queries are considered just as a vector of ‘term weights’ (with one coordinate per interesting keyword or even per existing text word) and queries are considered in exactly the same way (context queries are not considered in this case). Therefore, the query is now internally converted into a vector of term weights and the aim is to retrieve all the vectors (documents) which are close to the query (where closeness has to be defined in the model).

This allows many interesting possibilities, for instance a complete document can be used as a query (since it is also a vector), which naturally leads to the use of relevance feedback techniques (i.e., the user can select a document from the result and submit it as a new query to retrieve

documents similar to the selected one). The algorithms for this model are totally different from those based on searching patterns (it is even possible that not every text word needs to be searched but only a small set of hopefully representative keywords extracted from each document).

6.2. Pattern matching

Pattern matching is the act of checking a given sequence of tokens for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form of either sequences or tree structures.

Allow queries that match strings rather than word tokens. Requires more sophisticated data structures and algorithms than inverted indices to retrieve efficiently.

A pattern is a set of syntactic features that must occur in a text segment. Those segments satisfying the pattern specifications are said to match the pattern. We can search for documents containing segments which match a given search pattern. Each system allows specifying some types of patterns. The more powerful the set of patterns allowed, the more involved queries can the user formulate. Match: segments satisfying the pattern. Pattern Matching: Allow the retrieval of pieces of text that have some property.

The most used types of patterns are:

Words: a string which must be a word in the text

Prefixes: a string which must form the beginning of a text word

Suffixes: a string which must form the termination of a text word

Substrings: a string which can appear within a text word

Ranges: a pair of strings which matches any word which lexicographically lies between them

Allowing errors: a word together with an error threshold

Regular expressions: a rather general pattern built up by simple strings

Extended patterns: a more user-friendly query language to represent some common cases of regular expressions

■ Types

➤ Words

➤ Prefixes

e.q 'comput' -> 'computer', 'computation', 'computing', etc

➤ Suffixes

e.q 'ters' -> 'computers', 'testers', 'painters', etc

➤ Substrings

e.q 'tal' -> 'coastal', 'talk', 'metallic', etc

➤ Ranges

Between 'held' and 'hold' -> 'hoax' and 'hissing'

6.3. Structured queries

So far, we assumed documents that are entirely free of structure. Structured documents would allow more powerful queries. Queries could combine text queries with structural queries: queries that relate to the structure of the document. Mixing contents and structure in queries:

Contents \rightarrow words, phrases, or patterns and Structural constraints \rightarrow containment, proximity, or other restrictions on structural elements

Example: Retrieve documents that contain a page in which the phrase “terrorist attack” appears in the text and a photo whose caption contains the phrase “World Trade Center”. The corresponding query could be: same page (“terrorist attack”, photo (caption (“World Trade Center”))). Three main structures

- Fixed structure
- Hypertext structure
- Hierarchical structure

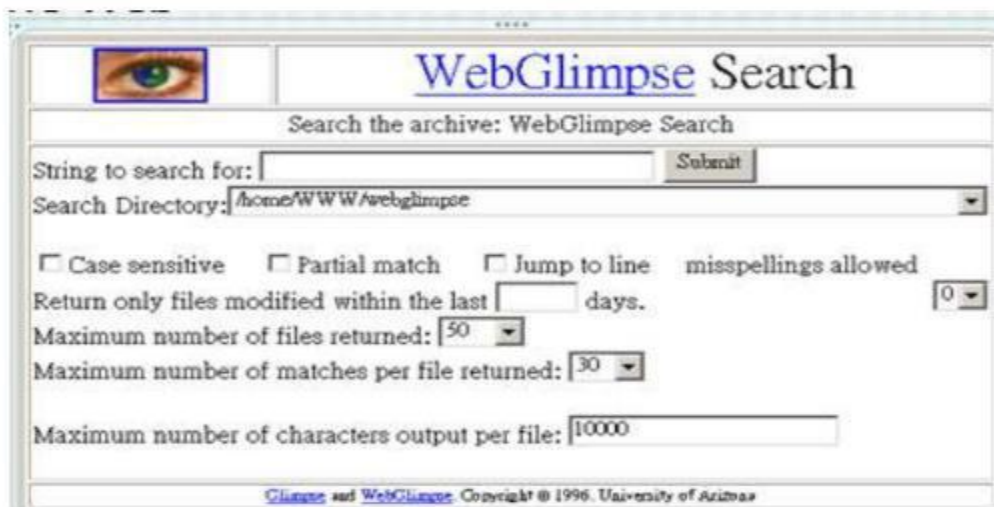
Fixed Structure: Document is divided to a fixed set of fields, much like a filled form. Fields may be associated with types, such as date. Each field has text and fields cannot nest or overlap. Queries

(multiple-words, Boolean, proximity. patterns, etc.) are targeted at particular fields. Suitable for documents such as mail messages, with fields for: Sender, Receiver, Date, Subject, Message body

EX: a mail has a sender, a receiver, a date, a subject and a body field Search for the mails sent to a given person with “football” in the Subject field

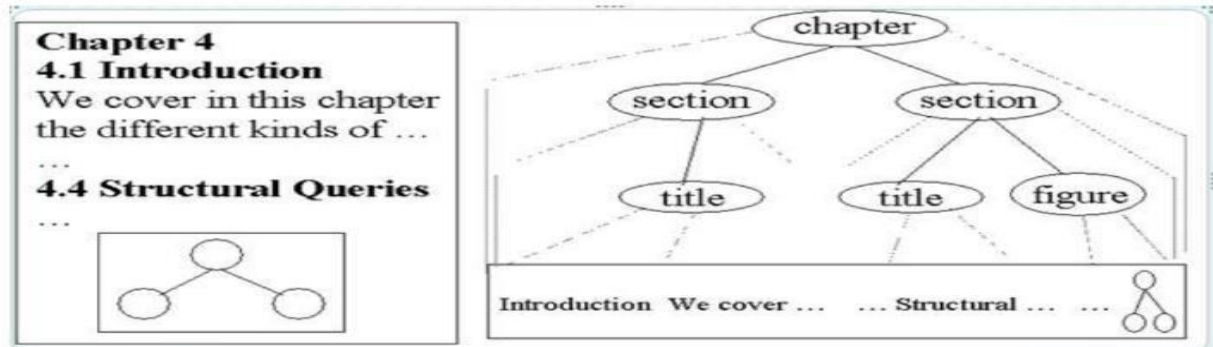
Hypertext: The most general document structure. The term hypertext was coined by American computer scientist Ted Nelson in 1965 to describe textual information that could be accessed in a nonlinear way. The prefix hyper describes the speed and facility with which users could jump to and from related areas of text. Each document is divided into regions (nodes), where a region could be a section, a paragraph, or an entire document; regions may be nested. The nodes are connected with directed links. A link is anchored in a phrase or a word in one node and leads to another node. Result is a network of document parts. Iypertext lends itself more to browsing than to querying.

WebGlimpse: combine browsing and searching on the Web

The image shows a screenshot of the WebGlimpse Search web interface. At the top left is a logo of a stylized eye. To its right is the title "WebGlimpse Search" in a large, blue, serif font. Below the title is a subtitle "Search the archive: WebGlimpse Search". The main form area contains a "String to search for:" text input field followed by a "Submit" button. Below this is a "Search Directory:" dropdown menu showing "/home/WWW/webglimpse". There are four checkboxes: "Case sensitive", "Partial match", "Jump to line", and "misspellings allowed". Below these is a text input field for "Return only files modified within the last" followed by a small spinner box set to "0" and the word "days". Below that are two more spinner boxes: "Maximum number of files returned:" set to "50" and "Maximum number of matches per file returned:" set to "30". At the bottom of the form is a text input field for "Maximum number of characters output per file:" set to "10000". At the very bottom of the window, there is a small copyright notice: "Glimpse and WebGlimpse Copyright © 1996, University of Arizona".

Hierarchical structure: Intermediate model between fixed structure and hypertext. The “anarchic” hypertext network is restricted to a hierarchical structure. The model allows recursive

decomposition of documents. Queries may combine Regular text queries, which are targeted at particular areas (the target area is defined by a “path expression”) and Queries on the structure itself; for example “retrieve documents with at least 5 sections



6.4 Relevance feedback

After initial retrieval] results are presented, allow the user to provide feedback on the relevance of one or more of the retrieved documents. The system use this feedback information to reformulate the query and Produce new results based on reformulated query. After that allows more interactive, multi-pass process.

The idea of relevance feedback (RF) is to involve the user in RELEVANCE FEEDBACK 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:

¥emdash

The user issues a (short, simple) query.

¥emdash

The system returns an initial set of retrieval results.

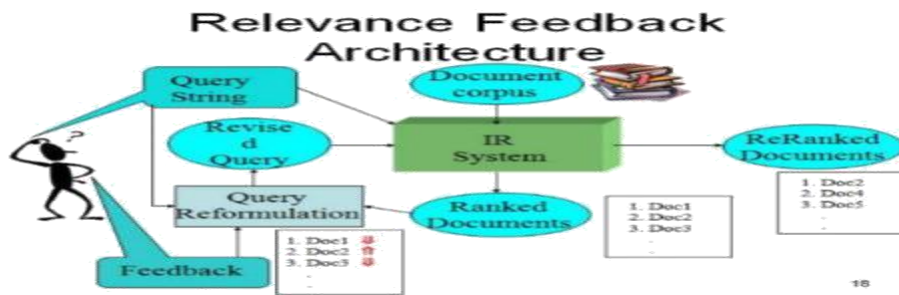
¥emdash

The system computes a better representation of the information need based on the user

¥emdash

The system displays a revised set of retrieval results.

Relevance feedback can go through one or more iterations of this sort. 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



Why is Feedback Not Widely Used

¥emdash

Users sometimes reluctant to provide explicit feedback.

¥emdash

Results in Jong queries that require more computation to retrieve, and search engines process lots of queries and allow little time for each one.

¥emdash

Makes it harder to understand why a particular document was retrieved.

QUERY REFORMULATION

A common search strategy is for the user to first issue a general query, then look at a few results, and if the desired information is not found, to make changes to the query in an attempt to improve the results. This cycle is repeated until the user is satisfied, or gives up. This is called query reformulation step.

THE NEED FOR REFORMULATION

Examination of search engine query logs suggests a high frequency of query reformulation. One study by Jansen et al., 2005 analyzed 3 million records from a 24 hour snapshot of Web logs taken in 2002 from the AltaVista search engine. (The search activity was partitioned into sessions separated by periods of inactivity, and no effort was made to determine if users searched for more than one topic during a session. 72% of the sessions were less than five minutes long, and so one-topic-per-session is a reasonable, if noisy, estimate.) The analysis found that the proportion of users who modified queries was 52%, with 32% issuing 3 or more queries within

the session. Other studies show similar proportions of refinements, thus supporting the assertion that query reformulation is a common part of the search process.

Good tools are needed to aid in the query formulation process. At times, when a searcher chooses a way to express an information need that does not successfully match relevant documents, the searcher becomes reluctant to radically modify their original query and stays stuck on the original formulation. Hertzum_ and Frokjaer, 1996 note that at this point “the user is subject to what psychologists call anchoring, i.e.. the tendency to make insufficient adjustments to initial values when judging under uncertainty”. This can lead to “thrashing” on small variations of the same query. Russell, 2006 remarks on this kind of behavior in Google query logs.

In order to show users helpful alternatives, researchers have developed several techniques to try to aid in the query reformulation process (although existing tools may not be sophisticated enough to aid the user with the information need shown above.)

SPELLING SUGGESTIONS AND CORRECTIONS

Search logs suggest that from 10-15% of queries contain spelling or typographical errors (Cucerzan and Brill. 2004). Fittingly, one important query reformulation tool is spelling suggestions or corrections. Web search engines have developed highly effective algorithms for detecting potential spelling errors (Cucerzan and Brill, 2004 Li et al., 2006).

Before the web, spelling correction software was seen mainly in word processing programs. Most spelling correction software compared the author's words to those found in a pre-defined dictionary (Kukich, 1992), and did not allow for word substitution. With the enormous usage of Web search engines, it became clear that query spelling correction was a harder problem than traditional spelling correction, because of the prevalence of proper names, company names, neologisms, multi-word phrases, and very short contexts (some spelling correction algorithms make use of the sentential structure of text). Most dictionaries do not contain words like blog, shrek, and nsync.

But with the greater difficulty also came the benefit of huge amounts of user behavior data. Web spelling suggestions are produced with the realization that queries should be compared to other queries, because queries tend to have special characteristics, and there is a lot of commonality in the kinds of spelling errors that searchers make. A key insight for improving spelling suggestions on the Web was that query logs often show not only the misspelling, but also the corrections that users make in subsequent queries. For example, if a searcher first types *schwarzeneger* and then corrects this to *schwarzenegger*, if the latter spelling is correct, an algorithm can make use of this pair for guessing the intended word. Experiments on algorithms that derive spelling corrections from query logs achieve results in the range of 88-90% accuracy for coverage of about 50% of misspellings (Cucercan and Brill, 2004, Li et al., 2006). For Web search engine interfaces, one alternative spelling is typically shown beneath the original query but above the retrieval results. The suggestion is also repeated at the bottom of the results page in case the user does not notice the error until they have scrolled through all of the suggested hits. But in the case of a blatantly incorrect typographical error, a user may prefer the correction to be made automatically to avoid the need for an extra click. To balance this tradeoff, some search engines show some hits with their guess of the correct spelling interwoven with others that contain the original, most likely incorrect spelling.

There are no published large-scale statistics on user uptake of spelling correction, but a presentation by Russell. 2006 shows that, for those queries that are reformulations, and for which the original query consisted of two words. 33% of the users making reformulations used the spelling correction facility. For three-word query reformulations, 5% of these users used the spelling suggestion.

In an in-person study conducted with a statistically representative subject pool of 100 people, Hargittai, 2006 studied the effects of typographical and spelling errors. (Here typographical means that the participant knows the correct spelling but made a typing mistake, whereas spelling error means the participant does not know the correct spelling.) Hargittai. 2006 found that 63% of the participants made a mistake of some kind, and among these, 35% made only one mistake, but 17% made four or more errors during their entire session. As might be predicted, lower education predicted higher number of spelling errors, but an interesting finding was that the higher the participant's income, the more likely they were to make a typographical error. Older participants

were more also likely to make spelling errors. The most surprising result, however, was that of the 37 participants who made an error while using Google search, none of them clicked on the spelling corrections link. This would seem to contradict the statistics from Russell. 2006. It may be the case that in Hargittai's data, participants made errors on longer queries exclusively, or that those from a broader demographic do not regularly make use of this kind of search aid, or that the pool was too small to observe the full range of user behavior.

AUTOMATED TERM SUGGESTIONS

The second important classes of query reformulation aids are automatically suggested term refinements and expansions. Spelling correction suggestions are also query reformulation aids, but the phrase term expansion is usually applied to tools that suggest alternative words and phrases. In this usage, the suggested terms are used to either replace or augment the current query. Term suggestions that require no user input can be generated from characteristics of the search engine, a directory browser, and an experimental interface with query suggestions. This interface showed upwards of 40 suggested terms and hid results listing until after the participant selected terms. (The selected terms were conjoined to those in the original query.) The study found that automatically generated term suggestions resulted in higher average precision than using the Web search engine, but with a slower response time and the penalty of a higher cognitive load (as measured by performance on a distracter task). No subjective responses were recorded. Another study using a similar interface and technology found that users preferred not to use the refinements in favor of going straight to the search results (Dennis et al.. 1998), underscoring the search interface design principle that search results should be shown immediately after the initial query, alongside additional search aids. collection itself (Schutze and Pedersen. 1994), from terms derived from the top-ranked results (Anick. 2003, Bruza and Dennis. 1997), a combination of both (Xu and Croft. 1996), from a hand-built thesaurus (Voorhees. 1994, Sihvonen and Vakkari. 2004), or from query logs (Cui et al.. 2003, Cucerzan and Brill. 2005, Jones et al.. 2006) or by combining query logs with navigation or other online behavior (Parikh and Sundaresan. 2008).

Usability studies are generally positive as to the efficacy of term suggestions when users are not required to make relevance judgments and do not have to choose among too many terms. Some studies have produced negative results, but they seem to stem from problems with the presentation interface. Generally it seems users do not wish to reformulate their queries by selecting multiple

terms, but many researchers have presented study participants with multiple-term selection interfaces.

For example, in one study by Bruza et al.. 2000, 54 participants were exposed to a standard Web

Research areas

I. Text Annotation Techniques

- \emdash Cross-lingual IR
- \emdash Web search engine for local languages
- \emdash Intelligent IR (content-based understanding)
- \emdash Application of NLP techniques for IR
- \emdash Document classification
- \emdash Document summarization
- \emdash Multimedia Retrieval
- \emdash Image Retrieval (Content-based, Document. etc.)

Query Expansion through Local Context Analysis

The local clustering techniques discussed above are based on the set of documents retrieved for the original query and use the top-ranked documents for clustering neighbor terms (or stems). Such clustering is based on term (stems were considered above) co-occurrence inside documents. Terms that are the best neighbors of each query term are then used to expand the original query q . A distinct approach is to search for term correlations in the whole collection – an approach called global analysis. Global techniques usually involve the building of a thesaurus that identifies term relationships in the whole collection. The terms are treated as concepts and the thesaurus is viewed as a concept relationship structure. Thesauri are expensive to build but, besides providing support for query expansion, are useful as a browsing tool as demonstrated by some search engines in the Web. The building of a thesaurus usually considers the use of small contexts and phrase structures instead of simply adopting the context provided by a whole document. Furthermore, with modern variants of

global analysis, terms that are closest to the whole query (and not to individual query terms) are selected for query expansion.

The application of ideas from global analysis (such as small contexts and phrase structures) to the local set of documents retrieved is a recent idea that we now discuss. Local context analysis [838] combines global and local analysis and works as follows. First, the approach is based on the use of noun groups (i.e., a single noun, two adjacent nouns, or three adjacent nouns in the text), instead of simple keywords, as document concepts. For query expansion, concepts are selected from the top-ranked documents (as in local analysis) based on their co-occurrence with query terms (no stemming). However, instead of documents, passages (i.e., a text window of fixed size) are used for determining co-occurrence (as in global analysis). More specifically, the local context analysis procedure operates in three steps.

- **First**, retrieve the top n -ranked passages using the original query. This is accomplished by breaking up the documents initially retrieved by the query in fixed-length passages (for instance, of size 300 words) and ranking these passages as if they were documents.

- **Second**, for each concept c in the top ranked passages, the similarity $\text{sim}(q, c)$ between the whole query q (not individual query terms) and the concept c is computed using a variant of tf-idf ranking.

- **Third**, the top m ranked concepts (according to $\text{sim}(q, c)$) are added to the original query q . To each added concept is assigned a weight given by $1 - 0.9 \times i/m$ where i is the position of the concept in the final concept ranking. The terms in the original query q might be stressed by assigning a weight equal to 2 to each of them. Of these three steps, the second one is the most complex and the one which we now discuss. The similarity $\text{sim}(q, c)$ between each related concept c and the original query q is computed as follows.

$$\text{sim}(q, c) = \prod_{k_i \in q} \left(\delta + \frac{\log(f(c, k_i) \times \text{idf}_c)}{\log n} \right)^{\text{idf}_i}$$

where n is the number of top-ranked passages consider

he function $f(c, k_i)$ quantifies the correlation between the concept c and the query term k_i and is given by

$$f(c, k_i) = \sum_{j=1}^n pf_{i,j} \times pf_{c,j}$$

where $pf_{i,j}$ is the frequency of term k_i in the j -th passage and $pf_{c,j}$ is the frequency of the concept c in the j -th passage. Notice that this is the standard correlation measure defined for association clusters (by Equation 5.5) but adapted for passages. The inverse document frequency factors are computed as

$$idf_i = \max(1, \frac{\log_{10} N / np_i}{5})$$

$$idf_c = \max(1, \frac{\log_{10} N / np_c}{5})$$

where N is the number of passages in the collection, np_i is the number of passages containing the term k_i , and np_c is the number of passages containing the concept c . Factor 5 is a constant parameter that avoids a value equal to zero for $\text{sim}(q, c)$ (which is useful, for instance, if the approach is to be used with probabilistic frameworks such as that provided by belief networks). Usually, 5 is a small factor with values close to 0.1 (10% of the maximum of 1). Finally, the idf_i factor in the exponent is introduced to emphasize infrequent query terms. The procedure above for computing $\text{sim}(q, c)$ is a non-trivial variant of tfidf ranking. Furthermore, it has been adjusted for operation with TREe data and did not work so well with a different collection. Thus, it is important to have in mind that tuning might be required for operation with a different collection. We also notice that the correlation measure adopted with local context analysis is of type association. However, we already know that a correlation of type metric is expected to be more effective. Thus, it remains to be tested whether the adoption of a metric correlation factor (for the function $f(c, k_i)$) makes any difference with local context analysis.

6.2.4. Automatic Global Analysis

The methods of local analysis discussed above extract information from the local set of documents retrieved to expand the query. It is well accepted that such a procedure yields improved retrieval performance with various collections. An alternative approach is to expand the query using

information from the whole set of documents in the collection. Strategies based on this idea are called global analysis procedures.

Until the beginning of the 1990s, global analysis was considered to be a technique which failed to yield consistent improvements in retrieval performance with general collections. This perception has changed with the appearance of modern procedures for global analysis. In the following, we discuss two of these modern variants. Both of them are based on a thesaurus-like structure built using all the documents in the collection. However, the approach is taken for building the thesaurus and the procedure for selecting terms for query expansion are quite distinct in the two cases

6.2.5. Query Expansion based on a Similarity Thesaurus

In this section, we discuss a query expansion model based on a global similarity thesaurus which is constructed automatically [655]. The similarity thesaurus is based on term-to-term relationships rather than on a matrix of co-occurrence. The distinction is made clear in the discussion below. Furthermore, special attention is paid to the selection of terms for expansion and to the reweighting of these terms. In contrast to previous global analysis approaches, terms for expansion are selected based on their similarity to the whole query rather than on their similarities to individual query terms. A similarity thesaurus is built considering term-to-term relationships. However, such relationships are not derived directly from the co-occurrence of terms inside documents. Rather, they are obtained by considering that the terms are concepts in a concept space. In this concept space, each term is indexed by the documents in which it appears. Thus, terms assume the original role of documents while documents are interpreted as indexing elements.

Given the global similarity thesaurus, query expansion is done in three steps as follows.

- First, represent the query in the concept space used for representation of the index terms.
- Second, based on the global similarity thesaurus, compute a similarity $\text{sim}(q, kv)$ between each term kv correlated to the query terms and the whole query q .
- Third, expand the query with the top r ranked terms according to $\text{sim}(q, kv)$. For the first step, the query is represented in the concept space of index term vectors as follows. Definition To the

$$\vec{q} = \sum_{k_i \in q} w_{i,q} \vec{k}_i$$

query q has associated a vector in the term concept space given by Where $w_{i,q}$ is a weight associated to the index-query pair $[k_i, q]$. This weight is computed

analogously to the index-document weight formula. For the second step, a similarity $\text{sim}(q, k_v)$ between each term k_v (correlated to the query terms) and the user query q is computed as

$$\text{sim}(q, k_v) = \vec{q} \cdot \vec{k}_v = \sum_{k_u \in q} w_{u,q} \times c_{u,v}$$

where $c_{u,v}$ is the correlation factor. As illustrated in Figure 5.2, a term might be quite close to the whole query while its distances to individual query terms are larger. This implies that the terms selected here for query expansion might be distinct from those selected by previous global analysis methods (which adopted a similarity to individual query terms for deciding terms for query expansion). For the third step, the top r ranked terms according to $\text{sim}(q, k_v)$ are added to the original query q to form the expanded query q' . To each expansion term k_v in the query q' is assigned a weight $w_{v,q'}$ given by

$$w_{v,q'} = \frac{\text{sim}(q, k_v)}{\sum_{k_u \in q} w_{u,q}}$$

6.2.6. Query Expansion based on a Statistical Thesaurus

In this section, we discuss a query expansion technique based on a global statistical thesaurus — Despite also being a global analysis technique, the approach is quite distinct from the one described above which is based on a similar thesaurus. The global thesaurus is composed of classes that group correlated terms in the context of the whole collection. Such correlated terms can then be used to expand the original user query. To be effective, the terms selected for expansion must have high term discrimination values [699] which implies that they must be low-frequency terms. However, it is difficult to cluster low frequency terms effectively due to the small amount of information about them (they occur in a few documents).

To circumvent this problem, we cluster documents into classes instead and use the low-frequency terms in these documents to define our thesaurus classes. In this situation, the

document clustering algorithm must produce small and tight clusters. A document clustering algorithm which produces small and tight clusters is the complete link algorithm which works as follows (naive formulation).

- Initially, place each document in a distinct cluster.
- Compute the similarity between all pairs of clusters.
- Determine the pair of clusters $[C_u, C_v]$ with the highest inter-cluster similarity.
- Merge the clusters C_u and C_v
- Verify a stop criterion. If this criterion is not met then go back to step 2.
- Return a hierarchy of clusters. The similarity between two clusters is defined as the minimum of the similarities between all pairs of inter-cluster documents (I.e., two documents not in the same cluster).

To compute the similarity between documents in a pair, the cosine formula of the vector model is used. As a result of this minimality criterion, the resultant clusters tend to be small and tight. Consider that the whole document collection has been clustered using the complete link algorithm. Figure 5.3 illustrates a small portion of the whole cluster hierarchy in which $\text{sim}(C_u, C_v) = 0.15$ and $\text{sim}(C_{u+v}, C_z) = 0.11$ where C_{u+v} is a reference to the cluster which results from merging C_u and C_v . Notice that the similarities decrease as we move up in the hierarchy because high-level clusters include more documents and thus represent a looser grouping. Thus, the tightest clusters lie at the bottom of the clustering hierarchy. Given the document cluster hierarchy for the whole collection, the terms that compose each class of the global thesaurus are selected as follows.

- Obtain from the user three parameters: threshold class (TC), number of

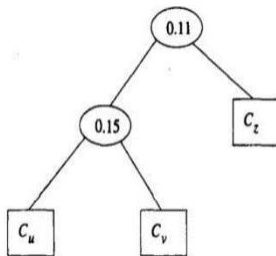


fig 6. 4 Hierarchy of three clusters (inter-cluster similarities indicated in the ovals) generated by the complete link algorithm

Documents in a class (NDC), and minimum inverse document frequency (MIDF).

- Use the parameter TC as a threshold value for determining the generate thesaurus classes. This threshold has to be surpassed by $\text{sim}(C_u, C_v)$ if the documents in the clusters C_u and C_v are to be selected as sources of terms for a thesaurus class..
- Use the parameter NDC as a limit on the size of clusters (number of documents) to be considered. For instance, if both C_u+v and C_u+v+z are preselected (through the parameter TC) then the parameter NDC might be used to decide between the two. A low value of NDC might restrict the selection to the smaller cluster C_u+v .
- Consider the set of documents in each document cluster preselected above (through the parameters TC and NDC). Only the lower frequency documents are used as sources of terms for the thesaurus classes. The parameter MIDF defines the minimum value of inverse document frequency for any term which is selected to participate in a thesaurus class. By doing so, it is possible to ensure that only low-frequency terms participate in the thesaurus classes generated (terms too generic are not good synonyms). Given that the thesaurus classes have been built, they can be used for query expansion. For this, an average term weight $w_{t,c}$ for each thesaurus class C is computed as follows. where $|C|$ is the number of terms in the thesaurus class C and $w_{i,c}$ document clusters that will be used to

$$w_{t,c} = \frac{\sum_{i=1}^{|C|} w_{i,c}}{|C|}$$

where $|C|$ is the number of terms in the thesaurus class C and $w_{i,c}$ is a pre-computed weight associated with the term-class pair $[t_i, C]$. This average term weight can then be used to compute a thesaurus class weight w_c as

$$w_c = \frac{w_{t,c}}{|C|} \times 0.5$$

The above weight formulations have been verified through experimentation and have yielded good results. Experiments with four test collections (ADI, Medlars, CACM, and ISI; see Chapter3 for details on these collections)

indicate that global analysis using a thesaurus built by the complete link algorithm might yield consistent improvements in retrieval performance.

The main problem with this approach is the initialization of the parameters TC, NDC, and MIDF. The threshold value TC depends on the collection and can be difficult to set properly. Inspection of the cluster hierarchy is almost always necessary for assisting with the setting of TC. Care must be exercised because a high value of TC might yield classes with too few terms while a low value of TC might yield too few classes. The selection of the parameter NDC can be decided more easily once TC has been set. However, the setting of the parameter MIDF might be difficult and also requires careful consideration.