Information Retrieval Cheatsheet

Edoardo Riggio

October 29, 2021

Operating Systems - SA. 2021
Computer Science
Università della Svizzera Italiana, Lugano

Contents

1 Introduction

1.1 Text Information Systems

Text Information Systems involve three main capabilities:

- Text Retrieval

  Information Retrieval is a field concerned with the structure, analysis, organization, storage searching, and retrieval of information.

- Text Analysis

  Analyze large amounts of text data in order to discover interesting patterns buried in text.

- Text Organization

  Annotate a collection of text documents with meaningful topical structures so that scattered information can be connected and navigated.

While text retrieval is part of information retrieval, text analysis and text organization are part of text mining.

Differently from queries done on DBMS, queries in search engines make use of natural language. It is much harder to compare the text query to the document text and determining what is a good match and what is not a good match. This is the core issue of information retrieval. There are many different ways of writing the same thing, thus an identical matching of words is not enough.

1.2 Relevance

A document is said to be relevant when it contains the information that a person was looking for when he/she submitted the query tot eh search engine.

In order to understand what the user is asking for in the query, we use something that is known as **NLP** (Natural Language Processing). NLP is concerned with developing techniques for enabling computers to understand the meaning of natural language text.

2 Text Access

Text data access is the foundation for text analysis. The general goal of text data access is to connect users with the right information at the right time.

Connection with users can be done in two ways:

- **Pull**

The user takes initiative in order to fetch relevant information from the system.

- **Push**

  The system takes initiative in order to offer relevant pieces of information to the users.

2.1 Access Mode: Pull vs Push

In **pull** mode, the user initiates the access process in order to find the relevant text data. When a user has such need, then this can be done in two different ways:

- **Querying**

  The user can use a query in order to obtain ad hoc information. This mode of research is done by using a few – yet specific – words.

- **Browsing**

  It is a way of accessing text data, and can be very useful for users when they do not know how to formulate an effective query.

In **push** mode, the system initiates the process to recommend a set of relevant information items to the user.

Broadly, there are two kinds of information needs:

- **Short-Term Needs**

  These needs are often associated with pull mode. This type of information need is temporary and usually satisfied through searching or browsing.

- **Long-Term Needs**

  These needs are often associated will pull mode. This type of information need can be better satisfied through filtering or recommendation of the system to the user.

Finally we have **browsing traces**. These traces happen whenever a user does any kind of browsing. They are used in order to model the behaviour of the user, and make more precise recommendations.

2.2 Search Engine Architecture

A **software architecture** consists of software components, the interfaces provided by those components, and the relationships between them.

The software architecture of a search engine is determined by the following requirements:

- **Effectiveness**
- **Efficiency**

An information retrieval process can be divided into four subprocesses:

1. **Indexing Process**

   This process is composed of several different stages. First we have **text acquisition**, in which the system identifies and stores documents for indexing.

   Second we have **text transformation**, in which the system transforms documents into index terms or features.

   Finally we have **index creation**, in which the system takes index terms and creates data structures to support fast searching.

2. **Query and Retrieval Process**

   It is composed of the following stages. **User interaction**, which supports the creation of a query and displays the results.

   Next we have **ranking and retrieval**, in which the query and indices are used in order to generate a ranked list of documents.

   Finally we have **evaluation**, which monitors and measures the effectiveness and efficiency of the information retrieval system.

3. **Relevance Feedback Process**

   It is composed of three parts. The first part is the **user evaluation**, where the user assesses the effectiveness of the system.
   Then we have **user feedback**, which supports refinement of the query and display of the result.
   Finally we have **ranking and retrieval**, where the system generates a ranked list of the documents.

2.3 Formal Definition of Information Retrieval

Information retrieval is composed of several elements, such as:

- **Vocabulary**

  Vocabulary is defined as:

  $$V = \{w_1, w_2, ..., w_n\}$$

  This represents the vocabulary of a language.

- **Document**

  A document is defined as:

  $$d_i = d_{i1}, ..., d_{iM}$$

  Where $d_{ij} \in V$.

- **Collection**

  A collection is defined as:

  $$C = \{d_1, ..., d_M\}$$

  this represents a collection of documents.

- **Query**

  A query is defined as:

  $$q = q_1, ..., q_M$$

  Where $q_j \in V$.

- **Relevant Documents**

  This set is defined as:

  $$R(q) \subseteq C$$

  And it is generally unknown and user-dependent. The query acts as a hint on which the document must be contained in $R(q)$.

2.4 How to Compute $R'(q)$

$R'(q)$ can be computed in one of two ways:

- **Document Selection**

  This can be described as:

  $$R'(q) = \{d \in C \mid f(d, q) = 1\}$$

  Where $f(d, q) \in \{0, 1\}$ and is an indicator function/binary classifier. Here the system must decide whether a document is relevant or not – i.e. **absolute relevance**. This is also known as **boolean retrieval**.

- **Document Ranking**

  This can be defined as:

  $$R'(q) = \{d \in C \mid f(d, q) > \Theta\}$$

  Where $f(d, q) \in \mathbb{R}$ is a relevance measure function, and Θ is a cutoff determined by the user. Here the system only decides if one document is more likely relevant than another – i.e. **relative measure**. This is also known as **ranked retrieval**.

3 Implementation of an IR System

An information retrieval system is mainly made up of four components:

1. **Tokenizer**

   This component takes in documents as raw strings and determines how to separate the large document into separate tokens.

2. **Indexer**

   This module processes documents and indexes them with appropriate data structures. This module can be ran offline.

3. **Scorer/Ranker**

   This module takes a query and returns a ranked list of documents.

3.1 Indexing

The main role of the indexer is to convert documents into data structures in order to enable fast search. The **inverted index** is the dominant indexing method for supporting basic search algorithms.

This data structure is composed of two parts:

- **Lexicon**

  It is a table of search-specific information, such as document frequency and where to find in the postings the per-document term counts

- **Posting File**

  It is a mapping that goes from any term integer ID to a list of documents IDs and frequency information of the term in those documents.

Before obtaining the inverted index we need to pre-process the documents in order to extract only the features we are interested in.

3.2 Zipf's Law

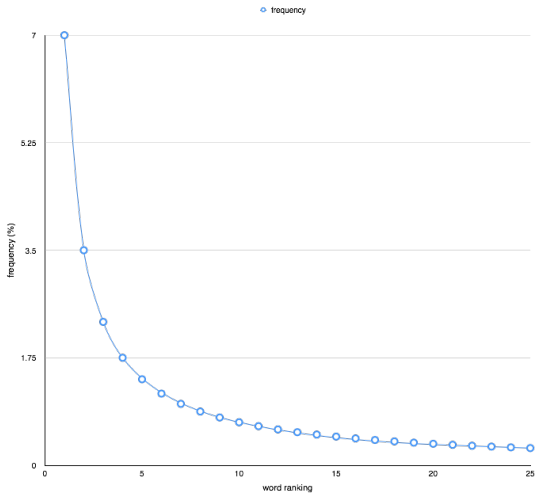
The distribution of words in documents is very skewed. This means that only a few words occur often, while many other words occur rarely.

Zipf's Law says that the rank r of a word times its frequency f, is approximately a constant k – assuming that the words are ranked in decreasing order of frequency. Thus we have the following formula:

$$r \cdot f \approx k$$

In the following case, for example, the words on the far right – with $x = [0, 7.5)$

– are high frequency words, thus useless, the words in the middle – with $x = [7.5, 15)$ – are intermediate frequency words, thus very useful, and the words in the far right – with $x = [15, 25]$ – are rare words, thus they might be useful.



3.3 Text Indexing

Text indexing can be divided into the following steps:

1. **Tokenization**
2. **Stopword Removal**
3. **Stemming**
4. **Detecting Phrases**
5. **POS Tagging and N-Grams**
6. **Processing Document Structure and Markup**
7. **Named Entity Recognition**
8. **Link Analysis**
9. **Build the Inverted Index**
10. **Compress the Inverted Index**

3.3.1 Tokenizing

To tokenize means to break down words into appropriate sequences of characters.

The first step is to use the **parser** in order to identify the appropriate parts of the document that need to be tokenized.

3.3.2 Stopping Removal

Some words have little to no meaning on their own and occur very frequently. These are treated as stopwords and removed. Although, sometimes, they could be important in combination with other words.

3.3.3 Stemming

Many morphological variations of words exist. In most cases, these have the same or very similar meanings. The goal of **stemmers** is to attempt to reduce morphological variations of words to a common stem.

For example, the words *consign*, *consigned*, *consigning* and *consignment* can all be reduced down to **consign**.

There are two main types of stemming approaches:

- **Algorithmic Approach**

  In this case it is a program that determines related words. This could give some false positives and many false negatives.

- **Dictionary-Based Approach**

  In this case a list of related words is used. Endings are removed based on a dictionary. This approach produces real words, not stems, and it's much more precise – but more expensive to run.

Some used stemmers are:

- **Porter Stemmer**

  This is an algorithm used since the 70s. It consists of a series of rules designed to remove the longest possible suffix from a word at each step. It produces stems, not real words. Sometimes this algorithm can be too aggressive or too weak. Porter2 stemmer was created in order to address some of the issues Porter had.

- **Krovertz Stemmer**

  This is a hybrid algorithmic-dictionary stemmer. The word is checked in a dictionary. If the word is present, then the word is either left alone or replaced with "exception", if the word is not present, the word is checked

for suffixes that could be removed. Finally, after the removal, the dictionary is checked again.

In this case words are generated, not stems. Furthermore, this stemmer has lower false positives, but higher false negatives.

3.3.4 POS Tagger and N-Grams

POS (Part Of Speech) tagging is the process of marking up a word in a text as corresponding to a particular part of speech, based on both its definition and its context. POS taggers use statistical models of text in order to predict syntactic tags of words.

Since POS tagging is too slow for some collections, **N-Grams** also exist. These are typically formed from overlapping sequences of words. Frequent N-Grams are more likely to be useful phrases. N-Grams follow Zipf's distribution.

3.3.5 Build the Inverted Index

In order to construct an inverted index, we use sort-based methods by following these steps:

1. Collect local tuples, such as term IDs, Doc IDs and frequency;
2. Sort the local tuples;
3. Perform pair-wise merge runs;
4. Output the inverted file.

While the dictionary part of the inverted index is of modest size, the postings part is huge and stored on disk.

3.4 Ranking Documents

The formula that is used in order to rank documents is the following.

$$f(q,d) = f_a(h(g(t_1,d,q),...,g(t_k,d,q)), f_d(d), f_q(q))$$

Where $f_d(q)$ and $f_q(q)$ are pre-computed. Moreover, a score accumulator is maintained for each d in order to compute h. Finally, for each query term t_i, the following inverted list is fetched.

$$\{(d_1,f_1),...,(d_n,f_n)\}$$

In order to improve the efficiency of the ranker, one could use caching, and keep only the most promising accumulators. There is no need for parallel processing.

4 Retrieval Models

The retrieval process is based on a retrieval model which matches a query with a document. There are at least two classes of retrieval models:

- **Set-Based Models**

  These are models like the Boolean model, and are defined by the following function:

  $$f(q, d) = \{0, 1\}$$

- **Similarity-Based Models**

  These are models such as the vector space model, the probabilistic model... They are defined by the following function:

  $$f(q, d) = \text{similarity}(q, d) = [0, \infty)$$

4.1 Examples of Retrieval Models

4.1.1 Boolean Model

A boolean model can only have two possible outcomes for query processing: true or false. It is an exact-match retrieval process, and the simplest for of ranking.

A query is usually specified using boolean operators (such as AND, NOT, OR...), and is used in DBMSs.

Some advantages of boolean retrieval are:

- The result is predictable;
- Many different features can be incorporated;
- Efficient query processing.

While the disadvantages are:

- The effectiveness of the query solely depends on the user;
- Simple queries do not usually work well;
- Complex queries are difficult both to think and to write.

4.1.2 Ranked Retrieval

In ranked retrieval, documents are presented according to how much they match the query. In a good ranking function relevant documents should be ranked on top of non-relevant ones.

4.2 Designing Retrieval Models

All retrieval models are based on the assumption of using a **bag-of-words** representation of text. A bag-of-words is a model in which a text is represented as the multiset of its words, disregarding grammar and even word order.

In order to design a retrieval function, we require a computational definition of relevance. Moreover, we need to use features such as:

- **Term Frequency (TF)**

  This represents how many times does one term appear inside of each document.

- **Document Length**

  If a term occurs in a long document many times, it is not as significant as a term that occurs the same number of times inside a short document.

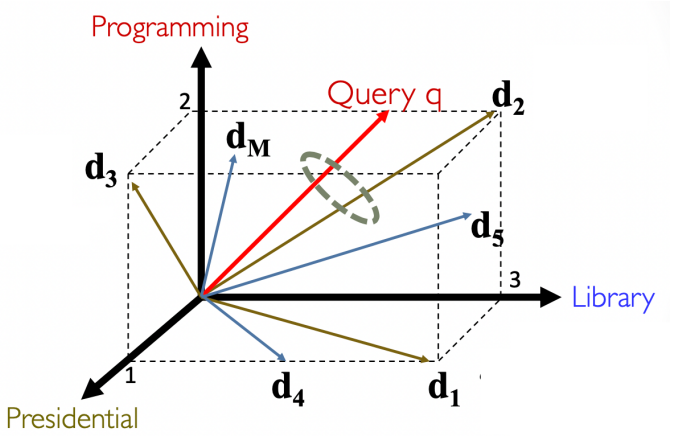
- **Document Frequency (DF)**

  This represents how often a term appears at least once in any document of the entire collection.

4.3 Vector Space Model

This is a simple, yet effective, way of designing ranking functions for information retrieval systems. This is a special case of similarity-based models, where we assume that relevance is roughly correlated to the similarity between a query and a document.

In this representation, each dimension of a highly dimensional space represents a term. We can plot the documents in the collection as vectors of term magnitude.



This is a framework, thus it needs to be defined. In order to use this framework we need the following:

- **Dimension Instantiation**

  The number of dimension can be defined by the number of words inside of a bag-of-words.

- **Vector Placement**

  In order to place the vector inside of the framework, we can use bit vectors. These bit vectors can only be either 0 or 1. We will have a 1 if the word w_i – one of the words of the axes – is present in the document or query, 0 otherwise.

- **Similarity Instantiation**

  The similarity between a query and a document can be defined as the dot product between these two vectors. In this case the formula would be:

$$\text{sim}(q, d) = \langle q, d \rangle = \sum_{i=1}^{N} x_i \cdot y_i$$

By using only these definitions and instantiations, the framework will be limited. For example, wouldn't it be correct to give more credit to terms that appear more times in a document or query? The current model does not allow it. This is why we can introduce some improvements to the current model, such as:

- **Term Frequency Vector**

  An improvement can be applied by representing a frequency vector. This vector will represent the number of times that the word w_i – one of the words of the axes – is contained in the query or in the document.

- **Term Frequency Weighting (TFW)**

  This improvement consists in using the same formula as the one used in the similarity instantiation, but with q and d represented by term frequency vectors.

- **Inverse Document Frequency**

  This improvement consists in giving a weight to words. The formula for computing y_i is given by:

$$y_i = \text{count}(w_i, q) * \log\left(\frac{M+1}{k}\right)$$

  Where $\text{count}(w_i, q)$ is the number of times the word w_i appears in document d, M is the total number of documents in a collection, and k is the total number of documents containing the word w.

The weight of each word inside of the collection of documents is computed by the second part of the previous formula, i.e.

$$\log\left(\frac{M+1}{k}\right)$$

4.4 TF Transformation

Since the inverse document frequency still has some problems, we can use TF transformation in order to normalize the number of times a word appears in a collection. This method is used in order to avoid the dominance of a single term above all the others.

This transformation, which is also known as BM25, is mathematically defined as follows:

$$f(q,d) = \sum_{w \in q \cap d} \text{count}(w,q) \cdot \frac{(k+1) \cdot \text{count}(w,d)}{\text{count}(w,d) + k} \cdot \log\left(\frac{M+1}{df(w)}\right)$$

Where $k \geq 0$ is a constant, $w \in q \cap d$ indicates all matched query words in the document, and $df(w)$ indicates the document frequency of the word w.

4.5 Document Length Normalization

Since a long document has a higher chance of matching any query, it must be penalised – but not too much – by using a document length normalizer. In order to not over-penalise a long document, we say that a document with more words must be penalised more, while a document with more contents must be penalised less.

In order to normalize the document lengths, we use the average document length as a pivot. The formula of the document length normalizer is the following:

$$1 - b + b \cdot \left(\frac{|d|}{\text{avgdl}}\right)$$

Where $b \in [0,1]$ is a constant and $avgdl$ is the average length of the documents in the collection.