Cerca i Anàlisi de la Informació

Apunts basats en el curs 2021-22 impartit per la Marta Arias

Àlex Batlle Casellas

Chapter 1. Introduction, Preprocessing and Text Laws

Information Retrieval

The origins of information retrieval come from librarians, government agencies, census, etc. Gradually, the information that was contained in all of these sources was digitalized. With the appearance of the internet and the (closed or private and then world-wide) web, everybody could set up data on the internet (and hence sometimes information). These are the pillars of the field of information retrieval: the web and the information it contains.

This field uses techniques from many disciplines: computer architecture, data structures and algorithmics, networks, logic and discrete mathematics, databases, probabilistic and statistical laws, AI, or image and sound processing, to name a few. In information retrieval, we may not know where the information resides, or if it even exists. We do not have a fixed schema, or we may not even know what information we want to retrieve exactly.

We may use, for example, a couple of search procedures which illustrate what do we need to specify to set up a formal information retrieval environment.

- **Example 1.** Hierarchical or taxonomical search: in this search procedure, we have the information containers (documents) organized hierarchically, that is, grouped by increasingly general concepts that contain each other. For example, a giraffe is found in taxonomy (from more to less general): Animalia -> Chordata -> Mammalia -> Artiodactyla -> Giraffidae -> Giraffa.
- **Example 2.** Faceted search: in this search procedure, we combine characteristics or properties of the piece of information we are searching for. For example, "it is black and yellow, and lives near the equator" may return a number of different animals and plants.

From these two search examples, we can conclude that to define an **information retrieval model** (IR model) we need:

- A certain notion of what a **document** is (an abstraction of a real document).
- To define what is an **admissible query** (for example, an SQL query, or an HTTP request).
- A **relevance** measure of the documents within our reach, that is, a two-argument function for pairs (query, document) that measures how much information relevant to the query does the document carry.

To sum this all up, let's summarize what the information retrieval process is:

1. **Offline stage.** In this stage, we must crawl, pre-process and index the documents we find. The goal of this stage is to prepare data structures as to make online query processing fast. This stage can afford time-consuming computations, and must produce relatively compact data structures.
2. **Online stage.** In this stage, we receive and process queries. Then, we must retrieve relevant documents, rank them, format the answer to the query and return it to the user. This stage should be very fast (almost instantaneous) and may use additional information from the user, adverts, etc.

Pre-processing

Within document pre-processing, we may have the following (non-closed) list of potential actions:

- Parsing: extracting structure from text.
- Tokenization: decomposing character sequences into individual units to be handled.
- Enriching: annotating units with additional information.
- Lemmatization and stemming: reducing words to roots.

Tokenization

Tokenization decomposes text into individual units, for example words, and is similar to lexical analysis in compilers. Usual problems in tokenization include how to treat hyphens, letter casing, punctuation symbols, which words can be ignored (stopwords) and how to identify them, etc. Some words may mean different things when used as a noun or as an adverb, for example. In summary, tokenization is strongly language-dependent, has to be adapted to the application we want to serve, is crucial for efficient retrieval and requires many many hard-coding of language structures, rules and exceptions into retrieval systems. A step beyond tokenization may be **Named Entity Recognition**.

Enriching

Enriching means that each term is associated to additional information (metadata) that can be helpful to retrieve more adequate documents for certain queries. For instance, enriching may encode synonyms, related words, definitions, categories, part-of-speech (POS) tagging (such as adverb, verb, noun), etc. A step beyond enriching text is **Word Sense Disambiguation**.

Lemmatization and stemming

Stemming consists of removing suffixed and preffixes from words to make them relate to each other in the face of the computer. Lemmatization consists of reducing a word to its linguistic root. In comparison, stemming is simple and faster than lemmatization, but is often impossible in some languages due to structure and way of writing. Lemmatization, on the other hand, is much slower and may require much more processing, but is more accurate and can be used in a wider variety of languages.

Math review & Text Statistics

Mathematical reminder

As a reminder, say we have a random variable X and a (random or non-random) parameter Y, then $\Pr(X = x|Y = y)$ is the probability of $X = x$ conditioned to the event $Y = y$, and is equal to $\frac{\Pr(X=x,\, Y=y)}{\Pr(Y=y)}$. It is also important to remember **Bayes' Theorem**,

$$\Pr(X = x|Y = y) = \frac{\Pr(Y = y|X = x) \cdot \Pr(X = x)}{\Pr(Y = y)},$$

as well as the notion of **independence between random variables**, which is attained by definition if $\Pr(X = x,\ Y = y) = \Pr(X = x) \cdot \Pr(Y = y)$, or equivalently if $\Pr(X = x|Y = y) = \Pr(X = x)$.

The notion of **expectation**, expected value, or mean is also quite useful, and is defined as (X is a discrete variable, Y is a continuous one):

$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} x \cdot \Pr(X = x),$$

$$\mathbb{E}[Y] = \int_{\mathcal{Y}} y \cdot f_Y(y)\, dy.$$

Its major property as a mathematical operator is its linearity, and additionally, if X and Y are now independent random variables of any kind (continuous or discrete), then it follows that

$$\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y].$$

In this section we will also make use of some results around the widely studied **harmonic series**, $\sum_{n \geq 1} \frac{1}{n}$, and its very much related cousin of complex argument, the Riemann Zeta function,

$$\zeta(s) = \sum_{n \geq 1} \frac{1}{n^s}.$$

It may be useful to remember that when s is restricted to the real numbers $\mathbb{R}$, this series only converges (in a real-valued function sense) when $s > 1$.

Text Statistics

The key question in this section is: how are texts constituted? Obviously, some words are very frequent and some appear less. The basic questions here are:

- How many different words do we/texts use frequently?
- How much more frequent exactly are frequent words?

In fact, there are very precise **empirical laws** that govern this behaviour in most human languages.

Heavy tails

In the statistical analysis of natural and artificial phenomena, it is common to observe that the resulting empirical probability distributions "decrease slowly" compared to Gaussian distributions, or exponential family distributions in general. This means that the empirical distributions show heavy tails, which is to say, infrequent objects ammount to a substantial probability density. This happens in phenomena such as:

- Texts (Zipf's law)
- People names
- Website popularities
- Wealth of individuals, companies, regions, countries, ...
- Earthquake intensity

Power laws in text

Zipf-Mandelbrot Equation

In text statistics, the frequency of words specifically follows a power law, the **Zipf-Mandelbrot equation**, which is governed by corpus-dependent (that is, text- or document-dependent) parameters a, b, c. The equation tells us that the relative frequency of the n-th most common word is approximately $\frac{c}{(n+b)^a}$. This was postulated by Zipf in the 1930's with $a = 1$, and was further developed in subsequent studies.

Power laws in a certain document may be detected by trying to estimate the exponent of a harmonic sequence. To do this,

1. Sort the items by decreasing frequency.
2. Plot them against their position in the sorted sequence, that is, against their rank.
3. Adjust the axes to obtain a log-log plot.
4. Up to this point, there should appear a shape close to a straight line.
    a. Of course, there will be some rounding to integer absolute frequencies from the power law.
5. With the plot, we can approximate the law's parameters.

Naturally, longer texts tend to use wider lexicon. However, the longer the text already seen, the lesser the chances of finding new, not yet processed words.

Herdan's or Heaps' Law

The number of different words is usually described by a polynomial of degree less than 1 on the text length. Again, like in the Zipf-Mandelbrot case, this is easily seen by using log-log plots.

For a text of length N, say that we tend to find d words. How do we relate these two variables? If we impose a straight line in a log-log plot, we get

$$\log d = k_1 + \beta \cdot \log N \implies d = k \cdot N^{\beta}$$

The value of β varies with language and type of text. With finite vocabulary, we do not necessarily have a greater value for β for very large values of N.

Chapter 2. Information Retrieval Models

We are going to take a look at two different information retrieval models: the **boolean model** and the **vector model**. Both models are **bag of words** models, which means, that they **do not take into account the order of the words** within the document. The boolean document goes further and also disregards frequency, which means it just takes into account if a word is or is not within a document. In both models, a document is a collection of pairs $(w, m(w))$, where m is a measure of the appearance of the word w, true or false in the boolean model and numeric in the vector model. The boolean model has **boolean queries** and **exact answers**, and can be extended to **phrase queries**, which we will also study. Meanwhile, the vector model assigns **weights on terms and documents**, and its queries are **similarity queries**, giving **approximate answers and ranking of results**.

Boolean Model of Information Retrieval

Let us first define the notion of document, query and relevance in this model:

- A document is completely defined by the **set of terms** it contains. For a set of terms $\mathcal{T} = \{t_1, \ldots, t_T\}$, a document is then just a subset of $\mathcal{T}$, so we can represent any document D as a boolean vector of length T, $d_D = (d_1, \ldots, d_T)$, with $d_i = 1$ if and only if $t_i \in D$.
- A query can be of two kinds:
    - *Atomic* query: this is formed by a single term. The answer is the set of documents that contain it.
    - *Compound* query: this is formed via boolean operators on a collection of atomic queries. The boolean operators are `AND`, `OR`, `NOT`, and all the combinations that can be made with them.
- The relevance of a document for a particular query is defined recursively. For atomic queries, which are the base case, a document either is relevant (1) or is not relevant at all (0), depending on whether it contains the term in the query. For compound queries, a document is relevant if it satisfies the boolean function that the query represents (the combination of boolean operators that forms the query). So, relevance is binary, hence there is no ranking of documents by relevance: either a document is relevant or is totally irrelevant.

Slighly beyond the boolean model: Phrase Queries

Phrase queries are a type of conjunction query (this is, using the `AND` operator) that wants to receive results that maintain adjacency of its components. For example, the query to ask about "black holes" on an astrophysics database following the boolean model would be `black AND holes`, but we want information specifically about the post-stellar objects, so we need to maintain the consecutivity (adjacency) of words.

In order to be able to respond to this queries effectively, we have many different options to proceed, which put more effort on the online or the offline stage, depending on the volume of results that we expect:

- We can run a phrase query just like a conjunction query, and then filter out the results that do not satisfy the adjacency requirement. This puts the effort in the online stage, and may be slow if we have a big database with a lot of false positives.
- We can keep dedicated information in the index about adjacency of any two terms. This puts effort on the offline stage and might get arbitrarily memory-consuming if the database is big enough.
- We can keep dedicated information in the index about some interesting pairs of adjacent words that we know from the start will be queried about. This requires prior knowledge that we may not always have.

Vector Space Model of Information Retrieval

In this model, as we have already said, the order of words is still irrelevant, but now frequency is relevant. Not all words are equally important, as they were in the boolean model. Now, for a set of terms (which we will call a **vocabulary** from now on) $\mathcal{T} = \{t_1, \dots, t_T\}$, a document D is represented by a vector $d = (w_1, \dots, w_T) \in \mathbb{R}^T$. Hence, $w_i \in \mathbb{R}$ is the weight of word t_i in D, but also within all the database, which we also call **corpus**, and we will represent by $\mathcal{D}$ from now on.

This model, then, conceptually represents the database as a matrix. If the database had $|\mathcal{D}| = N$ documents, the matrix that represents it is an element of $\mathcal{M}_{N\times T}(\mathbb{R})$, but this matrix is never computed for real values of N and T, because it would be very sparse. Its rows are the vector representation of each document.

How to assign weight to a word: the *tf-idf* scheme.

To assign a weight to a word, we follow the following two principles: the more frequent the term t is in document D, the higher its weight should be; but also, if the word is very frequent in the whole database, its weight should be lower in all the documents it appears, to represent its low discriminativeness among documents.

So, if we represent D by the vector $d = (w_1, \ldots, w_T) \equiv (w_{D,1}, \ldots, w_{D,T})$, then the *tf-idf* scheme assigns each weight as a product of two terms:

$$w_{D,j} = \mathrm{tf}_{D,j} \ \cdot \ \mathrm{idf}_j.$$

These two terms are called, respectively, the **term frequency** and the **inverse document frequency**. They are calculated using the following formulas:

$$\mathrm{tf}_{D,j} = \frac{\mathrm{f}_{D,j}}{\max\{\mathrm{f}_{D,k}\}_{k=1,\ldots,T}}, \qquad \mathrm{idf}_j = \log_2 \frac{N}{\mathrm{df}_j},$$

where $\mathrm{f}_{D,j}$ is the absolute frequency of t_j within D, df_j is the number of documents that contain t_j within the database, and N is the size of the database (how many documents it contains).

This scheme to calculate weights can be arbitrarily tweaked to fulfill our needs. For example, we could boost the term frequency of word t_i if it appears in the title, in boldface, in the metadata of a document, etcetera. We could, also, define the inverse document frequency for words that do not appear in the index. This is a way to account for vocabulary and corpus (database) that changes with time. For example, Laplace correction: $\mathrm{idf}_j = \log_2 \frac{N+1}{\mathrm{df}_i+1}$.

Queries in the vector model

We now introduce the **cosine similarity** measure, which will be useful in this model. Given two documents, with vector representations $d, d' \in \mathbb{R}^T$, we want to know how similar they are. In order to do this, as the name might suggest, we calculate the normalized dot product between them,

$$\mathrm{sim}(d, d') = \frac{\langle d, d' \rangle}{\|d\| \cdot \|d'\|} = \left\langle \frac{d}{\|d\|}, \frac{d'}{\|d'\|} \right\rangle.$$

As all of our weights are non-negative, the cosine similarity is always non-negative, and is bounded above by 1 because of the Cauchy-Schwarz inequality. Hence, $\mathrm{sim} : \mathbb{R}^T \times \mathbb{R}^T \to [0, 1]$, and we have a similarity measure for any two documents from $\mathcal{D}$ via composition of the representation and the similarity functions.

Queries in this model take any form of written language. To process them, we transform them into representations in $\mathbb{R}^T$ just as we do with the documents in $\mathcal{D}$, and them we calculate the cosine similarity of the query with all of our documents, to end up with a vector of similarities which will help us rank the answer collection by relevance. Hence, we perform the following calculation: given a query $Q \in \mathcal{Q}$ (an element of the "query space") which we represent by the vector $q \in \mathbb{R}^T$ in the same way that we represent documents in $\mathcal{D}$, the resulting relevance vector r is the result of

$$r := \frac{1}{\|q\|} \begin{bmatrix} \frac{1}{\|d_1\|} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{\|d_N\|} \end{bmatrix} \begin{bmatrix} w_{D_1,1} & \cdots & w_{D_1,T} \\ \vdots & \ddots & \vdots \\ w_{D_N,1} & \cdots & w_{D_N,T} \end{bmatrix} \begin{bmatrix} w_{Q,1} \\ \vdots \\ w_{Q,T} \end{bmatrix}.$$

When we have computed r, we can sort it in decreasing order, and then, we can return the documents that correspond to the higher values as the answer to Q.

Please note that this is mathematically what we want to do, but implementations may vary and most importantly, they might be more efficient than multiplying matrices.

Evaluation of an Information Retrieval System

Let us define $\mathcal{A}$ as the set of documents that the system returns as an answer to query Q, and $\mathcal{R}$ as the set of relevant documents, the ones that the user actually wants to see in an answer. Of course, nobody knows $\mathcal{R}$, not even the user making the query. Starting with the boolean model, we also define the following two metrics for an IR system:

- The **recall** of the system for a query is

  $$\frac{|\mathcal{R} \cap \mathcal{A}|}{|\mathcal{R}|},$$

  this is, the proportion of relevant documents that have been returned within the answer to the query.
- The **precision** of the system for a query is

  $$\frac{|\mathcal{R} \cap \mathcal{A}|}{|\mathcal{A}|}$$

  this is, the proportion of answers that are actually relevant for the query.

It is very difficult to get both high recall and high precision. Note that this can be translated to machine learning vocabulary using the confusion matrix of the system:

		ANSWERED
		relevant not relevant
REALITY	relevant	tp fn
	not relevant	fp tn

Now, we can identify concepts: $|\mathcal{R}| = tp + fn$, $|\mathcal{A}| = tp + fp$, $|\mathcal{R} \cap \mathcal{A}| = tp$, and hence, recall is $\frac{tp}{tp+fn}$, and precision is $\frac{tp}{tp+fp}$. Usually, tn is huge in IR, so the machine learning *accuracy* is of little use here.

This was all for boolean models. Now, what happens when relevance is a real number? What happens in the vector model? In models where relevance is not boolean, we set a threshold to indicate what is the lower bound that we want to consider relevant, and then classify all the similarities with our query either as relevant or as irrelevant. Also, this decision of the threshold provokes a long or a short answer, as more permissive bounds will return more documents. Long answers usually tend to exhibit low precision, while short ones tend to exhibit low recall. We usually represent this values separately as functions of $k = |\mathcal{A}|$, but there exists a single precision and recall curve, similar to the ROC curve in predictive models, using the x-axis as recall, and the y-axis as precision, which is useful in defining other metrics:

- *AUC*, the area under the curve of said plot relative to the best possible area.
- The F*-measure* or harmonic mean, calculated via

  $$\frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}},$$

  which is closer to the minimum than the arithmetic mean.
- The αF*-measure*, calculated via

  $$\frac{1}{\frac{\alpha}{\text{recall}} + \frac{1-\alpha}{\text{precision}}},$$

  which for different values of α gives different importance to either recall or precision.

To calculate the effectiveness of our system, we could also take into account the documents previously known by the user, $\mathcal{K}$, and calculate measures of coverage and novelty,

$$\text{Coverage} = \frac{|\mathcal{R} \cap \mathcal{K} \cap \mathcal{A}|}{|\mathcal{R} \cap \mathcal{K}|}, \qquad \text{Novelty} = \frac{|\mathcal{R} \cap \mathcal{K}^c \cap \mathcal{A}|}{|\mathcal{R} \cap \mathcal{A}|}.$$

Relevance Feedback

Now we want to go beyond what the user asked for and what we returned as answers. We want to **re-evaluate** the relevance of our answers, and improve our answers to a query, using user's feedback. To do that, we follow the user relevance cycle:

1. Get a query $Q \in \mathcal{Q}$.
2. Retrieve relevant documents for Q.
3. Show the top k relevant documents to the user.

4. Ask the user to mark them as relevant or irrelevant.
5. **Use user's answers to refine Q.**
6. If desired, return to 2.

The tricky part here is how to refine the query and hence, its results. What we actually refine is the vector q representing the query Q. One way to do this is Rocchio's rule.

Rocchio's rule

Given a query's representation q and a corpus $\mathcal{D}$, split up the corpus $\mathcal{D} = \mathcal{R} \sqcup \mathcal{R}^c$ and build a new query q', using

$$q' := \alpha q + \beta \frac{1}{|\mathcal{R}|} \sum_{D \in \mathcal{R}} d_D - \gamma \frac{1}{|\mathcal{R}^c|} \sum_{D \in \mathcal{R}^c} d_D$$

with $\alpha > \beta > \gamma \geq 0$, and all vectors that appear in the equation being normalized (except for q' obviously). α represents the degree of trust placed on the original user's query, β is the weight of positive information, that is, the weight that we give to the documents that are relevant by user's criteria but do not appear within the answer, and γ is the weight of negative information, that is, the weight that we give to the documents that are irrelevant by user's criteria. Usually, we take $\gamma = 0$. In practice, we often see a good improvement of the recall in the first round of the feedback loop, a marginal improvement for the second round, and almost no improvement beyond this point. In web search, for example, precision is much more important than recall, so extra computation and user time may be unproductive.

This query improvement procedure that we have seen is a form of **query expansion**, because the new query may have non-zero weights where before were zeros. Some other existing techniques and further information may be found in this paper, in the Wikipedia article, and in Stanford's NLP group book on Information Retrieval, particulary in chapter 9.

Pseudo-relevance Feedback

The pseudo-relevance feedback tries not to ask anything from the user, and instead assumes it has done well on its top k answers. Then, performs the same loop that we defined previously, and stops, if it does not have an iteration limit, when the top k results do not change over an iteration.

We can use alternative sources of feedback and query refinement, such as link value (number of clicks), time spent looking/reading an item, a user's previous history, or other user preferences. We can even use information about how related are words in the query with other words within our knowledge to link and refine our query using those.

Chapter 3. Implementation of Information Retrieval Systems

To answer queries received by our IR system, we could use the following **bad** algorithm:

```
input query q;
for every document d in database:
    if d matches q:
        add id(d) to list L;
output list L (perhaps sorted);
```

This algorithm scales badly with the database size, and we want a code that is ideally independent of this quantity. Enter **inverted files**.

Inverted files

Usually, a **vocabulary**, **lexicon** or **dictionary**, usually kept in main memory, maintains all the indexed terms as a set or map containing a correspondence from a document $d \in \mathcal{D}$ to a list of words that are contained in the document, $\mathcal{D} \to \{\text{words}\}$. Inverted files are the inverse correspondence, of sorts, mapping a word to the documents that contain it, that is, $\{\text{words}\} \to \mathcal{D}$. These files are built at pre-processing time, not at query time, so we can afford to spend time and computational effort in their construction.

In practice, inverted files are often implemented with **incidence** or **posting lists**. We assign a document identifier (*docid*) to each document, and for each term that we are indexing, we save a posting list, which is a list of docid's pointing to documents where the term appears, potentially with additional information. For a medium-sized application and large collections of documents, posting lists are compressed and stored in disk sorted by docid, while the vocabulary may fit in RAM.

Implementing the Boolean model

Given an implementation of the storage (e.g. inverted files), the way to process an atomic query for the boolean model is quite trivial: just look for the posting list of the term, and done. We are interested in how to process compound queries.

Conjunctive query

We are given a query of the form `a AND b`. Hence, in layman's terms, we want to get the posting lists of both `a` and `b`, and intersect them. This operation takes time in $\mathcal{O}(|L(a)| + |L(b)|)$, where $L(t)$ is the posting list of term t, and of course, it can be improved if both lists are sorted, by performing a merge-sort like intersection. In python, this would be something like:

```
def merge_intersect(L1, L2):
    # Suppose both L1 and L2 are sorted
    i = 0
    j = 0
    result = []
    while i < len(L1) and j < len(L2):
        if L1[i].docid < L2[j].docid:
            i += 1
        elif L1[i].docid > L2[j].docid:
            j += 1
        else: # L1[i].docid == L2[j].docid
            result.append(L1[i])
            i += 1
            j += 1
    return result
```

Disjunctive query

We are given a query of the form `a OR b`. Hence, in layman's terms, we want to get the posting lists of both `a` and `b`, and perform their union. This operation takes time in $\mathcal{O}(|L(a)| + |L(b)|)$, where $L(t)$ is the posting list of term t, and of course, it can be improved if both lists are sorted, by performing a merge-sort like union. In python, this would be something like:

```
def merge_union(L1, L2):
    i = 0
    j = 0
    result = []
    while i < len(L1) and j < len(L2):
        if L1[i].docid < L2[j].docid:
            result.append(L1[i])
            i += 1
        elif L2[j].docid < L1[i].docid:
            result.append(L2[j])
            j += 1
        else: # equal document identifiers
```

```
            result.append(L1[i])
            i += 1
            j += 1
    return result
```

Exclusive query

We are given a query of the form `a BUTNOT b`. Hence, in layman's terms, we want to get the posting lists of both `a` and `b`, and compute $L(a) \setminus L(b)$, that is, their set difference. This operation takes time at most in $\mathcal{O}(|L(a)| + |L(b)|)$, where $L(t)$ is the posting list of term t, and of course, it can be improved if both lists are sorted. In python, this would be something like:

```
def merge_set_difference(L1, L2):
    i = 0
    j = 0
    result = []
    while i < len(L1):
        if L1[i].docid < L2[j].docid:
            result.append(L1[i])
            i += 1
        elif L1[i].docid > L2[j].docid:
            if j < len(L2)-1:
                j += 1
            else:
                result.append(L1[i])
                i += 1
        else: # equal docid's, which is what we don't want
            i += 1
            if j < len(L2)-1:
                j += 1
    return result
```

Query optimization

We now want to deal with queries in the boolean model and their speed. Specifically, we want to address how to optimize queries to run in the least time possible. To do this, we have, among others, the following options:

- We could **rewrite** the query using boolean algebra.
- We could choose **other algorithms** for intersection and union.
- We could use more **sophisticated data structures** (computed offline).

Query rewriting

What is the most efficient way to compute the result of a conjunctive query like `a AND b AND c`? We have three equivalent ways of expressing the query that could be useful, namely `(a AND b) AND c` and its variants. And a disjunctive query like `(a AND b) OR (a AND c)`? We also have an equivalent form which maybe is more efficient, `a AND (b OR c)`. This equivalent forms are called **execution plans** in the IR context, and their cost depends on the sizes of the atomic results as well as the sizes of intermediate lists. The worst case scenario sizes for intersection and union are the following:

$$|L_1 \cap L_2| \leq \min(|L_1|, |L_2|), \qquad |L_1 \cup L_2| \leq |L_1| + |L_2| - |L_1 \cap L_2| \leq |L_1| + |L_2|$$

For example, let's compute the cost of `a AND b AND c` using two different execution plans, for sizes $|L_a| = 1000, |L_b| = 2000, |L_c| = 300$. The minimum number of comparisons that we have to compute if using sequential scanning is, naturally, $|L_a| + |L_b| + |L_c| = 3300$.

- Execution plan: `(a AND b) AND c`. So, the instructions are:

INSTRUCTION	COMPARISONS	RESULT $\leq$
1. $L_{a \cap b} = \text{intersect}(L_a, L_b)$	1000+2000=3000	1000
2. $L_{\text{res}} = \text{intersect}(L_{a \cap b}, L_c)$	1000+300=1300	300
Total comparisons	3000+1300=**4300**	—

- Execution plan: `(a AND c) AND b`. The instructions are:

INSTRUCTION	COMPARISONS	RESULT $\leq$
1. $L_{a \cap c} = \text{intersect}(L_a, L_c)$	1000+300=1300	300
2. $L_{\text{res}} = \text{intersect}(L_{a \cap c}, L_b)$	300+2000=2300	300
Total comparisons	1300+2300=**3600**<4300	—

From these, we derive the following **heuristic for `AND`-only queries**: intersections must happen **from shortest list of results to longest list of results**, in order to minimize the number of comparisons.

Let's compute another example, this time with the query `a AND (b OR c)`, with sizes $|L_a| = 300, |L_b| = 4000, |L_c| = 5000$. The minimum number of comparisons if we perform a sequential scan is, again, the sum of the sizes of the lists, that is, 9300. This first execution plan has instructions:

INSTRUCTION	COMPARISONS	RESULT $\leq$

INSTRUCTION	COMPARISONS	RESULT $\leq$
1. $L_{b \cup c} = \text{union}(L_b, L_c)$	4000+5000=9000	9000
2. $L_{\text{res}} = \text{intersect}(L_a, L_{b \cup c})$	9000+300=9300	300
Total comparisons	9000+9300=**18300**	—

We could try with a different execution plan for the same query, namely `(a AND b) OR (a AND c)`. The instructions would be:

INSTRUCTION	COMPARISONS	RESULT $\leq$
1. $L_{a \cap b} = \text{intersect}(L_a, L_b)$	300+4000=4300	300
2. $L_{a \cap c} = \text{intersect}(L_a, L_c)$	300+5000=5300	300
3. $L_{\text{res}} = \text{union}(L_{a \cap b}, L_{a \cap c})$	300+300=600	600
Total comparisons	4300+5300+600=**9900**<18300	—

As we can observe, the combinatorics may get very complicated with more convoluted queries. One can find more information on general boolean query optimization and boolean retrieval in Stanford's NLP group book on IR, chapter 1, in this PhD thesis on Boolean query optimization, and also in this article reviewing boolean query optimization using both normal as well as fuzzy logic. For tangential (but interesting!) information on other types of query optimization, see this article on genetic algorithms to optimize general query systems.

Using other algorithms

To optimize the processing of a query, we could implement other algorithms. For example, to achieve intersection in sub-linear time, we could exploit the binary search algorithm in a sorted list, which we know is logarithmic in the input list size. To do this, given two lists, we take the shortest one, and while we linearly scan this one, we search every docid in the longest list to find coincidences. Using this algorithm, the time would be $\mathcal{O}(|L_{\min}| \cdot \log |L_{\max}|)$. When $|L_{\min}| \ll |L_{\max}|$, it holds that $|L_{\min}| \cdot \log |L_{\max}| < |L_{\min}| + |L_{\max}|$.

Using sophisticated data structures

We could add to some elements in the posting list a pointer to a subsequent element in the list. Using this, we could check if inequality comparisons between two elements are going to hold for a lot of elements, and skip some segments, hence avoiding some comparisons and reducing the general algorithm cost. The optimal number and length of pointers, for example when working with posting lists in RAM, is $\sqrt{|L|}$ pointers, pointing to $\sqrt{|L|}$ elements after.

Implementing the Vector model

Now we want to retrieve documents from a system in which the vector model is implemented. Given a fixed similarity-measuring function $\text{sim}(d, q)$ between a document and a query, we want to retrieve (1) the documents which have a similarity with our query that is above a certain threshold $\text{sim}_{\min}$, (2) the top r documents according to our measure, or (3) all documents together with their similarity measure with the query, all sorted by decreasing similarity with our query q. Our system must react very fast, and with a reasonable memory expense.

An obvious (non-)solution is the following:

```
for doc in D:
    sim_q[doc] = 0
    compute_vector(doc)
    for word in query: # q is in vector form already
        sim_q[d] += tf[word, doc]*idf[word]
    sim_q /= norm(doc)*norm(query)
sort results by similarity
```

The inverse document frequency terms and the norm of each document can be precomputed and stored in the index, but the norm of the query has to be computed online. This is too inefficient for a large $\mathcal{D}$. To improve this, we notice some facts:

- Most documents only include a **small proportion** of the considered terms.
- Queries are usually **human-sized**, this is, they have a small number of different terms.
- Only a very small proportion of the documents will be revelant.
- We have the **inverted file** available.

Following this facts, we use the inverted file to make a faster algorithm:

```
sim_q = {}
for word in query:
    L = posting_list[word] # comes from inverted file
    for doc in L:
        if not doc in sim_q:
            sim_q[doc] = 0
        sim_q[doc] += tf[word, doc]*idf[word]
for doc in sim_q: # for each document that we have visited
    sim_q[doc] /= norm(doc)*norm(query)
sort results by similarity
```

After a few outer loops, instead of having completely computed $\text{sim\_q}(d)$ for some of the documents $d \in \mathcal{D}$, **we have partially computed** $\text{sim\_q}(d)$ **for all the documents**. Hence, this is similar to a column-wise scan of the document-term matrix, instead of a row-wise scan.

Index compression

We now want to talk about **compressing the index** of an IR system. But why would we want to do that? Well, that is because a large part of the query-answering time is spent bringing posting lists from disk to RAM, hence, we want to minimize the amount of bits that we have to transfer from one to the other. The index compression schemes exploit some properties that we have already seen and used to our advantage: the docids are usually sorted in increasing order, the frequencies of a word in a document are usually very small numbers, and so, we can do better than using, for example, 32 bits for each frequency value (the memory used by an `int`).

Posting lists typically have small values for frequency as we stated, so we can compress those values using **self-delimiting codes for frequency**. As document ids are sorted, we can also compress the docids using **Gap compression** and **Elias-Gamma codes**.

Frequency compression

We will use unary self-delimiting codes to indicate the values of a frequency list. We let the symbol 0 be the separator between values of the list, and replace the last 1 in a value with a 0. Hence, a frequency list takes the aspect of 110 10 0 1110 0 11110 (for the list $[3, 2, 1, 4, 1, 5]$). This obviously has unique decoding, and no prefix of a code is a code.

Docid compression

Instead of compressing the id's themselves, we will compress the *gaps* between two consecutive id's, this is, we will compress the vector $[(\text{id}_1, f_1), (\text{id}_2 - \text{id}_1, f_2), \ldots, (\text{id}_k - \text{id}_{k-1}, f_k)]$. If gaps are small, we will use very few unary digits in total. But, as gaps are not biased towards 1, we need to use a variable-length self-delimiting binary code. Enter **Elias-Gamma** codes.

Elias-Gamma codes

The idea of Elias-Gamma codes (developed by Peter Elias in 1975) adhere to the following scheme: first say how long x is in binary, and then send x in binary. The following Python code does the trick:

```
def elias_gamma(n: int):
    post = translate_into_binary(n)
    res = '0'*(len(post)-1)+post
    return res
```

It is quite obvious that Elias-Gamma is a self-delimiting code: once you start reading a sequence of elements encoded in Elias-Gamma, you read a number of zeros, say z, and afterwards you decode the following $z+1$ bits as a binary encoded integer. Assymptotically, this code has approximate length $2\log_2 x$; one log comes from the binary encoding, and another one from the length.

An easy alternative to Elias-Gamma codes is variable byte codes. These make use of the first bit in a group (be it a byte, 8 bits, or a nibble, 4 bits) to indicate whether this is the last group of bits that belong to the encoded number. If the first bit is 0, then the group of bits is the last one, else it is a 1 and the group of bits is a continuation byte/nibble.

There also exist improvements on Elias-Gamma, namely Elias-Delta and Elias-Omega codes. These encode recursively the prefixes of Elias-Gamma codes, in order to assymptotically decrease the number of bits used to encode a number. One can find more information on them in the partial exam from the Fall 2021 course of CAI, and on the internet.

Getting fast the top r results

The last line of the vector model implementation and response to a query was

```
sort results by similarity
```

This, as we know from previous courses, can be done in quasi-linear time $R\log R$, being and R the number of documents that have a similarity over the minimum stipulated. We will usually want to present only the top r documents, with $r \ll R$, really fast. To do this, we use a minheap. Let $L = [L_1, \ldots, L_R]$ in any (possibly random) order. Then, the algorithm works as follows:

```
put [d_1, ..., d_r] in a minheap
for i in range(r+1, R+1):
    d_top = minheap.top()
    min_val = sim(d_top, q)
    if sim(d_i, q) > min_val:
        replace the smallest element in the heap with d_i
        re-heapify the minheap
```

One can see that after any iteration of the loop, the heap will contain the top r documents among the first i. A not-so-obvious claim is the fact that, if the similarities in L are randomly ordered, the expected running time of the algorithm is $\mathcal{O}\left(R + r \cdot \log r \cdot \log\left(\frac{R}{r}\right)\right)$. This can be decomposed as the sum of the following terms:

- Time to insert r elements in the heap, $\mathcal{O}(r)$.
- Expected value of the time to process d_i:

$$p_i := \Pr\left(d_i \text{ enters the heap}\right) = \Pr\left(d_i \text{ is among the largest in } d_1, \ldots, d_i\right) = \frac{r}{i},$$
$$\mathbb{E}[\text{time to process } d_i] = p_i \cdot \mathcal{O}(\log r) + (1 - p_i) \cdot \mathcal{O}(1)$$

Now, one can see that the expected running time will be

$$\begin{aligned}
\mathbb{E}[\text{total running time}] &= \mathcal{O}(r) + \sum_{i=r+1}^{R} \left[p_i \mathcal{O}(\log r) + (1 - p_i)\mathcal{O}(1)\right] \\
&= \mathcal{O}(r) + \sum_{i=r+1}^{R} \left[\frac{r}{i}\mathcal{O}(\log r) + \frac{i-r}{i}\mathcal{O}(1)\right] \\
&= \cdots (\text{using that the harmonic series } H(n) \approx \log n) \\
&= \mathcal{O}(r) + \mathcal{O}(r \log r(\log R - \log r)) + \mathcal{O}(R - r) \\
&= \mathcal{O}(R) + \mathcal{O}\left(r \log r \log\left(\frac{R}{r}\right)\right)
\end{aligned}$$

For $r \ll R$, we have gotten from $\mathcal{O}(R \log R)$ to $\mathcal{O}(R)$.

Creating the index

Now we want to know, given a document collection $\mathcal{D}$, how do we build the inverted file, exactly? In Python, we can build it in RAM like this:

```
F = {}
for doc in D:
    d = docid(doc)
    for word in doc:
        if word not in F:
            F[word] = {}
        if d not in F[word]:
            F[word][d] = 0
        F[word][d] += 1
```

To handle problems with disk latency and throughput, and writing the RAM inverted file to disk, refer to the course material. Just bear in mind that we have a more efficient way to write an inverted index to disk, doing it by chunks: we create the inverted file for a chunk of the document collection, until the RAM is full. Then, we transfer each list in RAM to the end of the corresponding list in the disk, clear the RAM index, and repeat until we have processed all documents. A RAMful of index is sometimes called a *barrel*, and many barrels can be built parallely using a cluster.

Chapter 4. Searching the web. Pagerank.

The web is huge. Like really, really big. It had over 60 billion indexed pages in the happy times of 2019. This means that content, that is, text alone, cannot discriminate between two indexed documents, as most queries would return millions of results of pages with high similarity. Hence, we have to do something different if we are to retrieve any useful information. We will use the structure of the web