Capitolo 1

Text and web mining

Wholly new forms of encyclopedias will appear, ready made with a mesh of associative trails running through them, ready to be dropped into the memex and there amplified.
(Vannevar Bush, 1945)

Applications of machine learning and optimization are countless. In the following chapters we consider a couple of them: text mining, which is an entire field in itself, and collaborative recommendations, a paradigmatic case in business to extract value from simple customers data.

When the data consist of a collection of non-numerical elements, for example documents, we can still use many of the techniques used to analyze numerical data but we need to adapt them, by suitably **preprocessing the documents and fine-tuning the ML methods**. Preprocessing transforms texts into vectors containing numeric values. Fine-tuning the ML methods has to deal with the fact that these

```
<html>
 <head>
  <title>Learning and Intelligent Optimization</title>
  <meta name="author" content="Roberto Battiti">
  <meta name="keywords" content="LION, ML, optimization, big data">
 </head>
 <body>
  <h1>The LION way is the future</h1>
  The reasons are explained in the
  <a href="intelligent-optimization.org"> LIONlab homepage </a>.
 </body>
</html>
```

Figura 1.1: An example of a web page written in HTML, the Hypertext Markup Language which is standard to describe the overall page structure.

vectors may possess a huge number of coordinates, that words have synonyms, that texts have a structure going beyond a bag of words, facts which require specific ad hoc metrics, feature selection and extraction.

Information retrieval deals mostly with searching for documents and for information within documents, and **web mining** is related to adapting methods to the context of the world wide web. The Web is an *unstructured* (or, at most, *semi-structured*) collection of data mostly in form of human-readable texts and images, connected by hyper-links. The Web is not a database: a complete description of data items structure (a *schema*) is missing, it is just a messy collection of human-readable data and human-exploitable hyper-links. There are efforts to help machines (computers) to automatically extract meaning from web pages through semantic support[1], but the task is daunting given the anarchic and continuously evolving structure of the web. "**Big data**" is a popular commercial term for a collection of data sets so large, complex and unstructured that it becomes difficult to handle using traditional data processing applications.

In addition to text, it is important to remember that web pages contain tags that modify the *appearance* and the *meaning* of the text, they contain links to other documents (**hyper-links**) and, in some cases, **meta-data** describing the meaning of the different parts. A short example of a page written in HTML is shown in Fig. 1.1. *No web page is an island entire of itself*: in fact hyperlinks will turn out to be an effective help in searching for, ranking, and classifying pages in the web. Of course, similar linked structures are present in other areas, like social networks, bibliographic references in research papers, etc.

[1] "Semantic" means related to the "meaning" of the data items, and the so-called semantic web is an effort to add meta-data —data about the meaning of data— to enable automated agents and other software to access the Web more intelligently, for example understanding that a field is the name of a person, another field the age, another one the address, etc.

1.1 Retrieving and organizing information from the web

Before taking the plunge into the more interesting web mining tasks like ranking, clustering and classification, let's start with a short introduction about how to collect the raw content of the web pages (**crawling**), structure it so that it is ready for further analysis (**indexing**). If you do not care about how the raw data is obtained and you are just interested in a high-level view of the web mining issues, you may read the following abstract, skip these sections entirely and jump to Section 1.2.

The collected documents are processed into an **index** suitable for answering queries and retrieving information. Unlike the context of RDBMS (relational databases), *the order of answers is fundamental*: the user wants to see relevant data first. In other words: one aims at maximizing the probability that the first few answers will satisfy the user's needs. The union of a web crawler and a web index is a **search engine**.

In some cases **topic directories** are built to simplify searching. They are tree-like structures (taxonomies), initially designed by hand. The process of organizing the documents can be automated by **clustering and unsupervised learning** methods. The purpose is the automated discovery of groups in the set of documents so that documents *inside* the same group are *more similar* than documents in different groups. As one may expect, similarity measures are a crucial issue when designing automated document clustering techniques.

1.1.1 Crawling

The processing of the web information starts with **crawling**, systematic methods to visit web pages and harvest the information contained therein. The basic crawling principle consists of visiting the web graph by starting from a given set of URLs, fetching and collecting the corresponding pages, scanning collected pages for hyperlinks to pages that have not been collected yet.

If you are familiar with *graphs*, nodes represent web pages, edges represent links and the task is to *visit the graph*, i.e., visit all nodes in a systematic manner while avoiding duplicate visits. While a basic crawler implementing a visit of the web graph can be put together with a little knowledge of the underlying communication protocol (HTTP), avoiding the many pitfalls requires careful design considerations:

- many web servers assume that a human mind is driving the web requests, therefore they assume that any attempt at fetching many pages per second is an attack, and respond by denying access;

- more and more pages on the Internet are dynamic in many subtle ways: their content depends on data previously input by the user, by pre-existing client cookies, even by the position the request is being originated from; therefore,

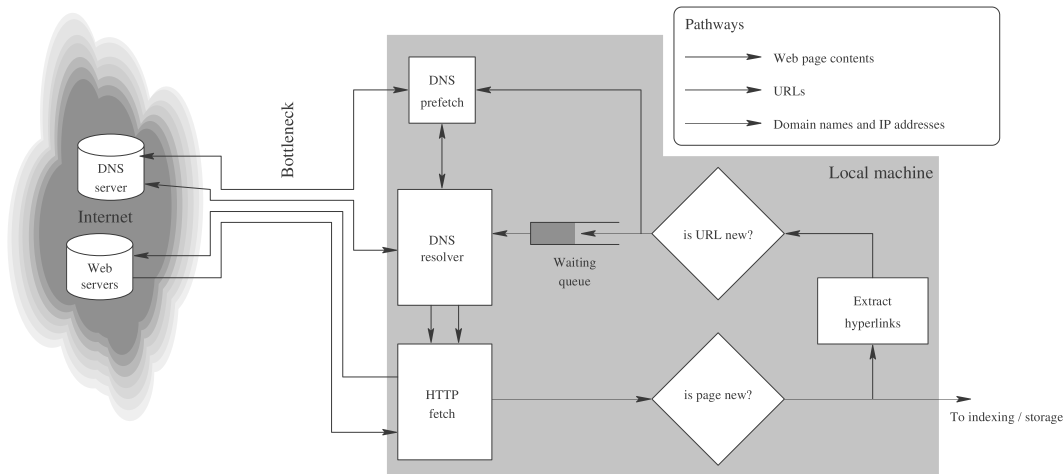

Figura 1.2: A basic crawler architecture: URLs are extracted from fetched pages and enqueued; domain names are pre-fetched to overcome the potential bottleneck.

any attempt to automatically collect all available information fails, and some user intelligence must be put into the system;

- resolving host names might take longer than fetching the data itself; in general, identifying the real bottleneck is not an easy task;
- the web is now dominated by virtual servers with their many-to-many relationships (Domain names to IP addresses, URLs to pages, not to mention mirrored or plagiarized info), so that identifying what has already been visited is becoming more and more difficult.

A minimal crawler architecture is shown in Fig. 1.2: a queue of still unvisited URLs is maintained; every time a page is fetched, it is scanned for new URLs. In order to overcome the aforementioned DNS bottleneck, a preliminary DNS request can be issued long before the URL is finally requested. Avoiding page and URL duplicates is also important, so various “is it new?” checkpoints can be placed throughout the workflow.

1.1.2 Indexing

Indexing is a required **preprocessing so that queries can be answered rapidly**. The simplest kind of queries, and by far the most used, involves one or more terms, in some cases combined by Boolean operators. For example one may search for: documents containing the word “Reactive” but not the word “Search”; documents containing the phrase “Reactive Search Optimization”; documents where “Reactive” and “Search” occur in the same sentence, etc.

Before building indices, documents undergo a sequence of cleaning steps, often including the following: HTML tags and other non-relevant markup items are filtered out (there are some exceptions: some meta-information should be retained, heading tags might provide information about the relevance and visibility of words); punctuation can be removed and replaced, if needed, by end-of-sentence markers; character casing is made uniform (e.g., all lowercase); the remaining text is *tokenized*, i.e., divided into words; very common words ("and", "I", "the"...), also known as *stopwords*, are removed; variant forms of the same word are collapsed to their *stem* (so that "play", "playing" and "played" all correspond to the same token).

While not all of the original information is preserved in the process, from the information retrieval point of view, the lost part is mainly noise, and a user who sends the query "Shakespeare play" to a search engine will expect results containing the words "Shakespeare plays" too, with proper casing and plural forms.

$$d_1 = \text{My}_1\ \text{care}_2\ \text{is}_3\ \text{loss}_4\ \text{of}_5\ \text{care}_6,\ \text{by}_7\ \text{old}_8\ \text{care}_9\ \text{done}_{10}.$$
$$d_2 = \text{Your}_1\ \text{care}_2\ \text{is}_3\ \text{gain}_4\ \text{of}_5\ \text{care}_6,\ \text{by}_7\ \text{new}_8\ \text{care}_9\ \text{won}_{10}.$$

tid	did	pos
my	1	1
care	1	2
is	1	3
⋮	⋮	⋮
new	2	8
care	2	9
won	2	10

tid	pos list
my	d_1/1
care	d_1/2,6,9 // d_2/2,6,9
is	d_1/3 // d_2/3
loss	d_1/4
of	d_1/5 // d_2/5
by	d_1/7 // d_2/7
old	d_1/8
done	d_1/10
your	d_2/1
gain	d_2/4
new	d_2/8
won	d_2/10

Figura 1.3: Two documents (top) and their direct (left) and inverted (right) index, from [**?**].

Fig. 1.3 shows two sample documents[2], d_1 and d_2, where the subscript denotes the position of the token in the document:

A direct index is a table mapping term ID `tid` to document's ID and position (`did`,`pos`). Such table, shown in the left-hand side of Fig. 1.3, makes searching for all documents containing a token very inefficient (one has to scan the entire table). An inverted index is a table obtained by "transposing" the previous one (right-hand side of Fig. 1.3), and giving for each token the list of documents that contain it.

[2] WILLIAM SHAKESPEARE — *The Life and Death of Richard the Second*, Act IV, Scene 1.

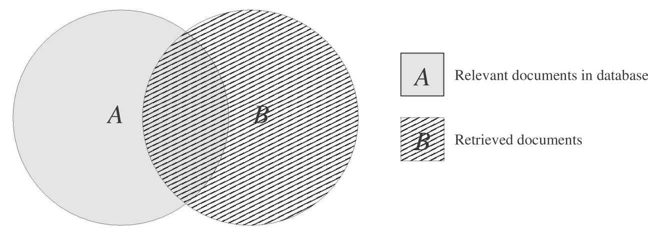

Figura 1.4: Information retrieval: relevant and retrieved documents.

1.2 Information retrieval and ranking

After the raw content of the web is properly saved and preprocessed (indexed), let's now consider the more interesting task of searching for documents, and searching for information within documents, also called Information retrieval (IR). In general, one wants to retrieve documents which are *relevant* to a query and which are of *good quality*. If one searches for "loss" and "care" one may retrieve a piece by Shakespeare, as well as documents about, let's say, "hair loss care," which are probably of inferior quality, if you are interested in literature and not in hair loss.

Standard definitions of performance measures when retrieving documents are as follows. If A is the set of relevant documents, and B the set of retrieved documents, see also Fig. 1.4 and Fig. **??** in Section **??**, one identifies:

- retrieved relevant items (true positives): $A \cap B$;
- retrieved irrelevant items (false positives): $B \setminus A$;
- unretrieved relevant items (false negatives): $A \setminus B$.

The **precision** of a retrieval system is defined as the fraction of retrieved documents that is relevant:

$$\text{precision} = \frac{|A \cap B|}{|B|}.$$

The **recall** of the system is defined as the fraction of relevant documents that is retrieved:

$$\text{recall} = \frac{|A \cap B|}{|A|}.$$

The recall measure usually is not so relevant for web searches, where the number of relevant documents typically is too large for a human to examine. For search engines, the order in which results are presented to the user is fundamental. In general, such order implies *ranking* the documents, so an adequate performance measure should favor those methods that place relevant documents in the highest ranks, and show them first in the user browser as response to a search.

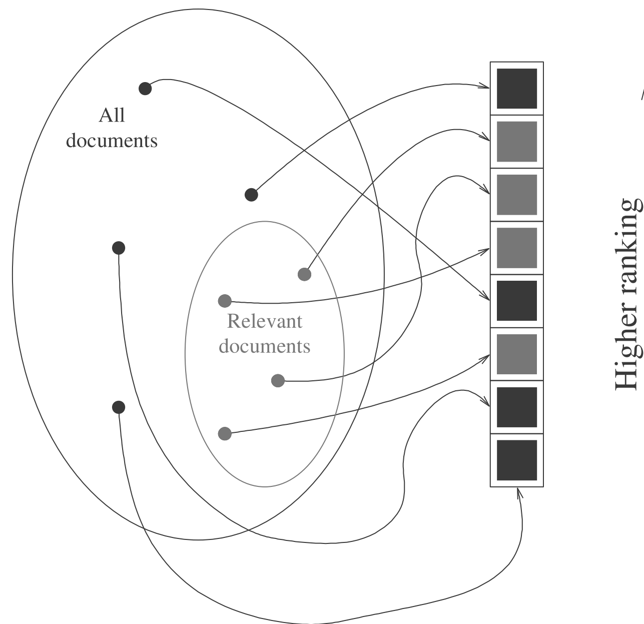

Figura 1.5: A ranking example.

Consider Fig. 1.5, where the darker dots represent relevant documents, while the ranking order is provided on the right. It is clear that the best ranking procedure should place the red elements in the top positions. Let's introduce more specialized definition of performance to take this into account.

Let D be a corpus of $n = |D|$ documents, and let q be a query. Define $D_q \subset D$ as the set of all relevant documents for query q. We assume that D_q represents the "desired" answer of the system. Let $(d_1^q, d_2^q, \ldots, d_n^q)$ be an ordering ("ranking") of D returned by the system in response to query q. Let $(r_1^q, r_2^q, \ldots, r_n^q)$ be defined as

$$r_i^q = \begin{cases} 1 & \text{if } d_i^q \in D_q \\ 0 & \text{otherwise.} \end{cases}$$

We can now define rank-dependent versions of the recall and precision figures, which will help us answer the question "how would we rate the performance of our system if we only took the top-k ranked answers?"

The recall at rank k is defined as the fraction of relevant documents found in the top k positions:

$$\text{recall}_q(k) = \frac{1}{|D_q|} \sum_{i=1}^{k} r_i^q,$$

and similarly for the precision:

$$\text{precision}_q(k) = \frac{1}{k}\sum_{i=1}^{k} r_i^q.$$

As usual, there are no free meals: when analyzing the ranked list, the recall can be increased by increasing k; but then, more and more irrelevant documents occur, driving down the precision (**precision-recall tradeoff**).

1.2.1 From Documents to Vectors: the Vector-Space Model

To use standard techniques designed for vector spaces to search, cluster, and classify documents, we first need to map each document to a vector (the **vector-space model**).

After preprocessing, our document is now a *bag of words*, actually a bag of tokens, and the most straightforward way to obtain a vector is to: i) fix a set of terms (tokens), ii) have a separate axis to represents each term (token), iii) set the value of the vector along the axis t to zero if the document does not contain the token t, to a number greater than zero if the document contains the token one or more times.

If $n(d,t)$ is the number of times that document d contains the term t, the **term frequency** $\text{TF}(d,t)$ of term t in document d is defined as a figure that increases monotonically with the relative frequency of t in d. Some possible definitions are the following:

$$\begin{aligned}
\text{TF}(d,t) &= \frac{n(d,t)}{\sum_\tau n(d,\tau)} \\
\text{TF}_{\text{SMART}}(d,t) &= \begin{cases} 0 & \text{if } n(d,t) = 0 \\ 1 + \log\bigl(1 + \log n(d,t)\bigr) & \text{otherwise.} \end{cases}
\end{aligned}$$

The TF_{SMART} formula is meant to avoid an exaggerated value along a dimension if a term is present too many times. This was a frequent case in the initial years of the web, when simple search engines were just counting term occurrences. It used to be that many pages contained for example the term "sex" repeated hundreds of times, aiming at reaching a high rank for many users searches. Actually, the more recent search engines combat this kind of spamming by using the hyperlink information, as we will see later.

Actually, often the most interesting terms are the ones which do not appear in many documents (**rare terms** like "C++", "Reactive Search Optimization", "stochastic" are probably more informative than "is", "nice", "free", "excellent"), and an **inverse document frequency** can be defined as a figure that monotonically *decreases*

as the overall frequency of a term in the whole document corpus increases:

$$\mathrm{IDF}(t) = \log \frac{1 + |D|}{|D_t|},$$

where D_t is the set of documents containing term t, and the logarithm is used to avoid an exaggerated multiplier for very rare terms. Let's note that the above methods to derive vectors are heuristic and not based on fundamental principles like information theory. If you think that the logarithm is not appropriate, feel free to experiment with other functions. After discounting the importance of weak terms appearing in too many documents, a specific document d in TF-IDF space (term-frequency inverse-document-frequency) is represented by vector

$$\boldsymbol{d} = (d_t)_{t \in \mathrm{terms}} \in \mathbb{R}^{\mathrm{terms}},$$

where component d_t is

$$d_t = \mathrm{TF}(d, t)\, \mathrm{IDF}(t).$$

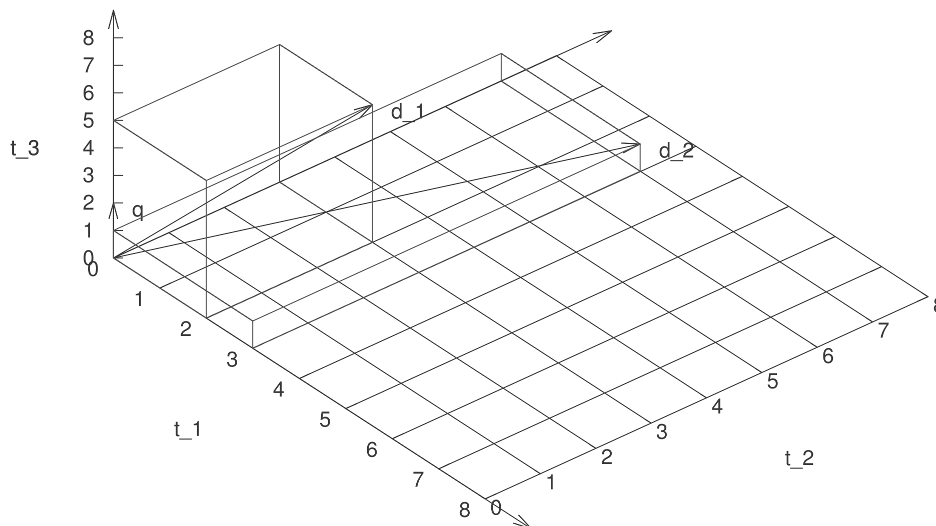

Figura 1.6: Geometric Interpretation.

A query q is a sequence of terms, therefore it admits a representation $\boldsymbol{q} = (q_t)$ in the same space as documents. Given the query $\boldsymbol{q}$ and the document $\boldsymbol{d}$, one can now measure their proximity, by considering vector-space similarity measures, as

shown in Fig. 1.6. Two frequently used proximity measures in TF-IDF space are listed below.

- Euclidean distance $\|\boldsymbol{d}-\boldsymbol{q}\|$. To avoid artifacts, vectors should be normalized,i.e., an n-fold replica of document $\boldsymbol{d}$ should have the same similarity to $\boldsymbol{q}$ as $\boldsymbol{d}$ itself:
$$\left\|\frac{\boldsymbol{d}}{\|\boldsymbol{d}\|}-\frac{\boldsymbol{q}}{\|\boldsymbol{q}\|}\right\|.$$
- Cosine similarity, i.e., the cosine of the angle between vectors $\boldsymbol{d}$ and $\boldsymbol{q}$:
$$\frac{\boldsymbol{d}\cdot\boldsymbol{q}}{\|\boldsymbol{d}\|\,\|\boldsymbol{q}\|},$$
see also equation (**??**).

An Information Retrieval system based on TF-IDF coordinates therefore works as follows. First build an inverse index with TF(t,d) and IDF(t) information. When given a query, map it onto TF-IDF space, sort documents according to the similarity metric, return the most similar documents. Searching methods can be extended in different ways, for example to search for phrases. The book [**?**] presents more details on the topic which cannot be presented in this short introductory chapter.

Please note that there is nothing magic in the traditional TF-IDF representation: it is just a heuristic recipe to give more weight to more informative words, so that the standard metrics given above can produce reasonable results. More sophisticated metric-learning or feature-selection methods based on information content (like mutual information) may well lead to superior results but require a greater knowledge by the user.

1.2.2 Relevance feedback

As mentioned above, after transforming the query into a vector, a vector-similarity measure can be used to identify a set of most similar documents to return to the user.

Unfortunately, the average web query is as few as one or two terms long, and it is not surprising that a lot of irrelevant documents can be retrieved. This is why either a serious measure to rank qualitatively superior documents is needed (like PageRank in Section 1.3) or at least a way to rapidly get **feedback from the user** and use it to form a better query.

Rocchio's Method is based on updating the vector used for the first query to make it more similar to vectors describing documents that the user identifies as relevant (likes), and less similar to the ones classified as irrelevant (dislikes), as illustrated in Fig. 1.7. For a mental image, think about documents that the user liked *attracting* the query vector, and documents that he disliked *repelling it*. In details, the query vector is updated as:

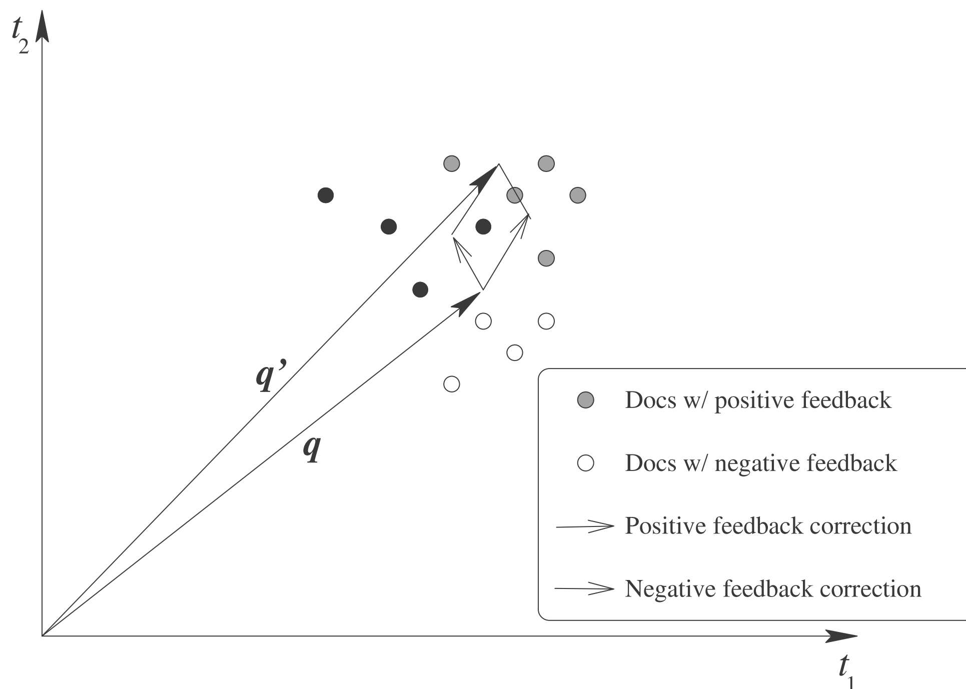

Figura 1.7: Rocchio's method.

$$\boldsymbol{q}' = \alpha \boldsymbol{q} + \beta \sum_{\boldsymbol{d} \in D_+} \boldsymbol{d} - \gamma \sum_{\boldsymbol{d} \in D_-} \boldsymbol{d},$$

where D_+ is a set of retrieved documents liked by the user, and D_- is a set of retrieved documents that the user dislikes. Parameters α, β and γ control the amount of modification. The careful reader may notice a resemblance with the way in which prototype vectors are updated in the self-organizing maps in Chapter **??**.

1.2.3 More complex similarity measures

In a TF-IDF vector space, we can define the "similarity" between two items as a decreasing function of distance — its inverse, for example, although it goes to infinity when comparing an object to itself, requiring some correction. If the elements admit a set representation, another similarity criterion is available, the *Jaccard coefficient*. Let A and B be two (finite) sets, the Jaccard coefficient of A and B is defined as

$$r'(A, B) = \frac{A \cap B}{A \cup B}.$$

Its aim should be clear: compare what is common to the two sets (their intersection) to their total size. It varies between 0 and 1, where $r'(A, B) = 0$ implies that the

two sets have no common element and $r'(A, B) = 1$ means that A and B are equal. An additional important property is that $1 - r'(A, B)$ is a *distance*, it obeys all the properties of a metric.

Let us adopt a more document-centric definition. If d is a document, let's define $T(d)$ as the set of tokens (terms) it contains. Note that, as always when referring to sets, elements have no multiplicity and we are just interested in a binary model where a term either occurs or does not. Then, the *Jaccard coefficient* of the two documents is

$$r'(d_1, d_2) = \frac{|T(d_1) \cap T(d_2)|}{|T(d_1) \cup T(d_2)|}.$$

The use of the Jaccard coefficient in a search engine can be motivated by the following fact: a query is usually seen by the user as a set of terms without repetitions, and no user would ever write a query such as "reactive reactive search" into Google expecting it to return documents containing the word "reactive" twice as frequently as the word "search".

The skeleton of an algorithm for computing the *Jaccard coefficient* $r'(\cdot, \cdot)$ is the following one.

- For each $d \in D$:
  - for each term $t \in T(d)$: put record (t, d) on file f_1.
- Sort f_1 in (t, d) order and aggregate into form (t, D_t).
- For each term t scanned from f_1:
  - for each pair $d_1, d_2 \in D_t$: put record (d_1, d_2, t) on file f_2.
- Sort f_2 on (d_1, d_2) and aggregate by adding on the third field.

Some possible tricks to reduce the search cost can be related to pre-computing the Jaccard coefficient for all pairs of documents and queries, which can require huge amounts of storage and CPU time, or reducing the set of pairs, by pre-associating every document or query to a list of a small and fixed number of the most similar documents. In addition, very frequent terms (having low IDF) can be omitted entirely from consideration.

In practice, in many cases one is interested in *approximating* the coefficient. An interesting randomized algorithm using random permutations is available.

If one uses probabilities, given sets A and B, one starts from this interesting equality:

$$\frac{|A \cap B|}{|A \cup B|} = \Pr(x \in A \cap B | x \in A \cup B).$$

If we can estimate the above probability, we can estimate the Jaccard coefficient. What we can do is to generate random elements in the set $S \subset \{1, \ldots, n\}$ and to estimate the probability by the ratio of events.

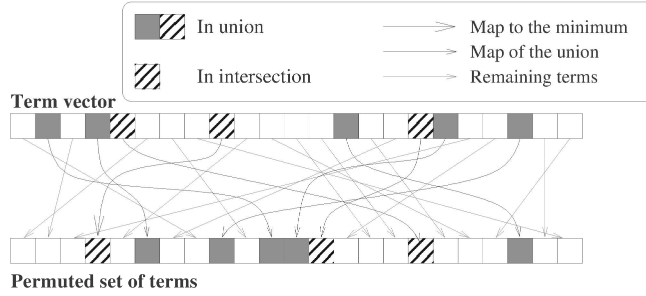

Figura 1.8: Building a permutation.

To select a random element from a set $S \subset \{1, \ldots, n\}$ one can select a random permutation π on n elements and pick the element in S such that its image in π is minimum:

$$x = \arg\min_{x \in S} \pi(x) = \arg\min \pi(S)$$

When applied to $A \cup B$, this method locates an element in the intersection if and only if

$$\min \pi(A) = \min \pi(B).$$

We can therefore estimate the above ratio by applying random permutations and checking if the two minima are coincident.

Let's demonstrate why permutations work. Given sets $A, B \subset \{1, \ldots, n\}$, to derive the probability that the two minima are coincident, let us count how many permutations $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ (of the $n!$ possible) have the property

$$\min \pi(A) = \min \pi(B)$$

by building one such permutation. From Fig. 1.8 it should be clear that:

- The image of $A \cup B$ (lighter and darker squares) can be chosen in $\binom{n}{|A \cup B|}$ different ways.
- Within such image, the minimum element can be chosen within the $|A \cap B|$ elements that form the intersection (thick arrow).
- The remaining elements in the image of $A \cup B$ can be permuted in $(|A \cup B| - 1)!$ ways (thin arrow).
- The elements not in $A \cup B$ can be permuted in any of $(n - |A \cup B|)!$ ways (light arrows).

After multiplying all these, one obtains:

$$\binom{n}{|A \cup B|} \cdot |A \cap B| \cdot (|A \cup B| - 1)! \cdot (n - |A \cup B|)! = n! \frac{|A \cap B|}{|A \cup B|}.$$

and after dividing by the total number of permutations one derived the desired equality.

A randomized but inefficient algorithm is as follows:

- generate a set Π of m permutations on the set of terms;
- $k \leftarrow 0$
- for each $\pi \in \Pi$:
  - if $\min \pi(T(d_1)) = \min \pi(T(d_2))$ then $k \leftarrow k + 1$;
- estimate $r'(d_1, d_2) \approx \dfrac{k}{m}$.

By combining a randomized algorithm with suitable data structures working on external storage and simultaneous computation of the coefficients for many documents, one manages to deal with the enormous amount of documents contained in the world wide web!

1.3 Using the hyperlinks to rank web pages

As mentioned before, there are so many web pages that the issue is not only that of retrieving a few set of pages relevant to the query, but that of retrieving a set of *high quality* and relevant pages. The issue predates the web and is encountered also when reading books, or papers. One would like to concentrate on good quality papers written on a subject, without wasting time on poor quality ones. In scientific communities, a paper is considered of good quality if it is *cited* by other good quality papers, meaning that some colleagues found the paper useful and acknowledged it by putting the paper reference in the list of citations. For a more mundane analogy, a candidate for employment is valued if many other valued people are prepared to recommend him. In general, see also Fig. 1.9, in a social network of relationships between people, a high reputation is obtained by having other highly-reputed people recommending you. It is not sufficient to convince many low-rank individuals to support you, there are no shortcuts!

After a seminal paper by Marchiori [**?**] highlighting the importance of *hyper-information* (information in the hyperlinks), Larry Page and Sergey Brin developed the PageRank algorithm, which follows the same basic social networks principles, by substituting "recommendations" and "citations" with hyperlinks [**?**] (the authors then became Google founders). They define a "measure of prestige" such that the prestige of a page is related to how many pages of prestige link to it. Let's note that this is a *recursive definition*. To measure the prestige of a page one needs to have the prestige of pages pointing to it, and so on. In short, their solution is: start with an initial distribution of prestige values, iterate the prestige calculation for the different

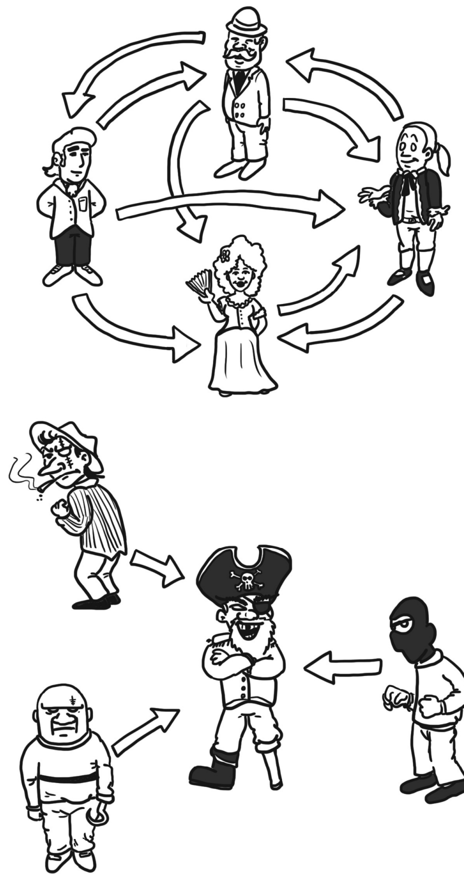

Figura 1.9: Prestige in social networks: recommendations from (or relationship with) high-rank individuals (above) are more effective to reach a high rank than recommendations by low-rank ones (below).

nodes, and stop when the values do not change too much after recalculation, as simple as that! At first reading, this process seems prone to pitfalls. What guarantee do we have that the process converges, hopefully to the same limiting distribution, not depending on the initial distribution of values?

Now: it is fascinating how the solution to this problem is related to basic linear algebra concepts of **eigenvalues and eigenvectors**, as well as concepts related to **Markov chains**. Let's summarize the main relationships.

First, let's see how iterating the prestige calculation after starting from an initial distribution is related to the classic **power iteration method for finding the dominant eigenvector** of a matrix. We proceed very rapidly, skipping mathematical details, only to give the flavor of the method.

The rank of a page is calculated by examining the incoming links (the hyperlinks of other pages pointing to the given page). Each incoming link from page i

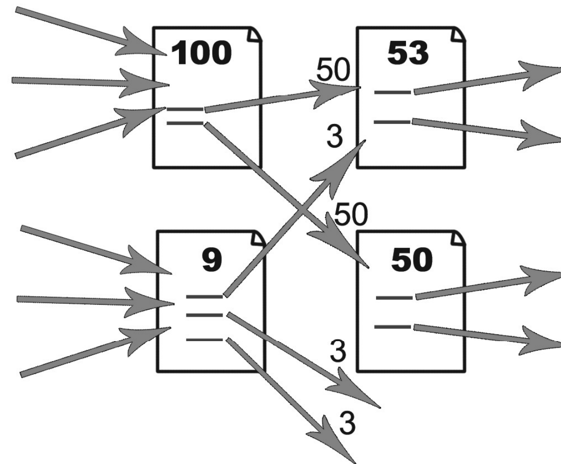

Figura 1.10: Recalculating the rank of a page in PageRank. The initial rank is distributed along the outgoing links (adapted from the original paper).

contributes a partial rank equal to the rank of i divided by the number of i's outgoing links, as shown in Fig. 1.10. The human motivation for the division is clear: a page of high rank but pointing to a very large number of pages is like a person of good quality but recommending a too large number of people. Without the division, the owner of a top-ranked page could influence all pages in the world just by putting an enormous number of outgoing links.

Given the above recalculation rule, once the network of hyper-links is given, the computation of new rank values $\mathbf{p}^k$ at iteration k is obtained by a *linear* transformation of the previous values through a matrix, which we denote as M, as follows:

$$\mathbf{p}^k = M\mathbf{p}^{k-1}.$$

The matrix M will depend only on the connectivity structure, on the links between pages. Now, after starting from the initial rank distribution $\mathbf{p}^0$ and executing k recalculations:

$$\mathbf{p}^k = M^k\mathbf{p}^0.$$

Assume that a basis of eigenvectors of M is available, let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the n eigenvalues, and let $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ be the corresponding eigenvectors. Suppose that λ_1 is the dominant eigenvalue, so that $|\lambda_1| > |\lambda_j|$ for $j > 1$.

The initial vector $\mathbf{p}^0$ can be written as a linear combination of the basis vectors:

$$\mathbf{p}^0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n.$$

If $\mathbf{p}^0$ is chosen randomly (with uniform probability), then $c_1 \neq 0$ with probability 1. Now, using linearity and the defining property of eigenvectors, one immediately

obtains:

$$\begin{aligned} M^k\mathbf{p}^0 &= c_1M^k\mathbf{v}_1 + c_2M^k\mathbf{v}_2 + \cdots + c_nM^k\mathbf{v}_n \\ &= c_1\lambda_1^k\mathbf{v}_1 + c_2\lambda_2^k\mathbf{v}_2 + \cdots + c_n\lambda_n^k\mathbf{v}_n \\ &= c_1\lambda_1^k\left(\mathbf{v}_1 + \tfrac{c_2}{c_1}\left(\tfrac{\lambda_2}{\lambda_1}\right)^k\mathbf{v}_2 + \cdots + \tfrac{c_n}{c_1}\left(\tfrac{\lambda_n}{\lambda_1}\right)^k\mathbf{v}_n\right). \end{aligned}$$

When the number of recalculations k, equal to the power of the matrix M^k, goes to infinity, all terms tends to zero apart from the one proportional to the dominant eigenvector. A simple iteration of matrix multiplication after starting from almost arbitrary initial conditions is indeed sufficient to extract the dominant eigenvector!

Let's now consider a different interpretation related to Markov chains, and imagine that you want to analyze a system representing the **movement of a web surfer** on the various web pages. Assume that a surfer is navigating through outgoing links forever, picking them uniformly at random. Let the starting page u be taken with probability p_u^0. Let E be the adjacency matrix of the web: $(u, v) \in E$ (or $E_{uv} = 1$) if and only if there is a link from page u to page v. What is the probability p_v^i of the surfer being at page v after i clicks?

Let's start with a single step. What is the probability p_v^1 that surfer is at page v after one step? Let

$$N_u = \sum_v E_{uv}$$

be the *out-degree* of page u (sum of u-th row of E). Suppose no parallel edges exist,

$$p_v^1 = \sum_{(u,v)\in E} \frac{p_u^0}{N_u}.$$

By normalizing E to have row sums equal to 1

$$L_{uv} = \frac{E_{uv}}{N_u},$$

we get

$$p_v^1 = \sum_u L_{uv}p_u^0 \qquad \text{or} \qquad \mathbf{p}^1 = L^T\mathbf{p}^0.$$

Let's now consider the situation after i steps:

$$\mathbf{p}^i = L^T\mathbf{p}^{i-1}.$$

If E is *irreducible* and *aperiodic* (none is actually true but the problem can be cured), then

$$\lim_{i\to\infty} \mathbf{p}^i = \mathbf{p},$$

where $\mathbf{p}$ is the principal eigenvector of L^T, a.k.a. its *stationary distribution*:

$$\mathbf{p} = L^T\mathbf{p} \qquad \text{(eigenvalue is 1).}$$

But p_u it the *prestige* of page u determined also by the previous interpretation. It is now clear how the prestige can be interpreted also as probability that a random surfer following links will be found at a given page.

Let's now deal with bad properties of real-world transition matrices. Surveys show that the Web is not strongly connected, and that random walks can be trapped into cycles. A possible fix is to introduce a "damping factor" corresponding to a user that occasionally stops following links: with an arbitrary probability d of going to a random page (even unconnected) at every step. The transition becomes:

$$\mathbf{p}^i = \left((1-d)L^T + \frac{d}{N}\mathbf{1}_N\right)\mathbf{p}^{i-1}.$$

The eigenvector of the matrix corresponding to the largest eigenvalue can be obtained as follows.

- Start with random vector $\mathbf{p} \leftarrow \mathbf{p}^0$;
- repeat:
  - update vector:
  $$\mathbf{p} \leftarrow \left((1-d)L^T + \frac{d}{N}\mathbf{1}_N\right)\mathbf{p};$$
  - from time to time, normalize it:
  $$\mathbf{p} \leftarrow \frac{\mathbf{p}}{\|\mathbf{p}\|_1}.$$

Normalization is done to avoid very large components and therefore numerical problems with finite-precision computation. Of course, for the application we are not interested in *absolute* prestige values but in *relative* ones. The absolute values depend on the chosen range (one may measure prestige on a range from 0 to 10, or on a range from 0 to 100, etc.) but what is relevant is that a page is, say, three times more prestigious than another one. Normalization is a simple way to discount multiples of the given normalized eigenvector.

In practical applications, the notion of prestige is so fuzzy that nobody will ever care about obtaining the actual eigenvector with high precision! To have a flavor of how long is required for convergence, in his original paper Page says that 52 iterations are enough for about 3×10^8 pages, quite an exciting result paving the way to significant business applications.

1.4 Identifying hubs and authorities: HITS

Let's now consider a different analysis of the web. In a scientific community all good articles are either seminal (i.e., many others reference to them) or surveys (i.e., they reference to many others). In the web, pages may be *authorities* or *hubs* [**?**]. For example portals are very good hubs, even if they do not contain significant information, and they are used only as starting points to reach good quality pages.

To reflect this distinction, let's introduce *two* score measures, called **hubness** and **authority**:

$$\boldsymbol{h} = (h_u), \qquad \boldsymbol{a} = (a_u).$$

Let's now summarize the HITS algorithm (Hyperlink-Induced Topic Search). In the HITS algorithm, the first step is to retrieve the set of results to the search query. Given query q, let R_q be the *root set* returned by an IR system. The computation is performed only on this result set, not across all Web pages. Authority and hub values are defined in terms of one another in a mutual recursion.

The *expanded set* is formed by adding all nodes linked to the root set:

$$V_q = R_q \cup \{u : ((u \to v) \vee (v \to u)) \wedge v \in R_q\}.$$

Let E_q be the induced link subset, $G_q = (V_q, E_q)$. The recurrent relationship is defined as follows. Let the hub score h_u be proportional to sum of referred authorities, let the authority score a_u be proportional to the sum of referring hubs.

$$\begin{aligned} \boldsymbol{a} &= E^T \boldsymbol{h} \\ \boldsymbol{h} &= E\boldsymbol{a}. \end{aligned}$$

The iterated method is therefore given by:

- initialize $\boldsymbol{a}$ and $\boldsymbol{h}$ (e.g., uniformly) ;
- repeat:
  - $\boldsymbol{h} \leftarrow E\boldsymbol{a}$;
  - $\boldsymbol{a} \leftarrow E^T\boldsymbol{h}$;
  - normalize $\boldsymbol{h}$ and $\boldsymbol{a}$.

The top-ranking authorities and hubs are reported to the user.

The principal eigenvector identifies the largest dense bipartite subgraph. To find smaller sets, the other eigenvectors must be explored. There are iterative methods that remove known eigenvectors from a system: they reduce the search subspace once an eigenvector is identified.

Although of theoretical interest, HITS is not commonly used by search engines, also because pre-computing hubness and authority values for different queries is not doable in practice, the algorithm has to run after the query is executed and this

makes the algorithm very heavy for general-purpose usage. Coming back to the PageRank algorithms, let's note that it is independent of page content and therefore a suitable combination with the content, depending on the query, must be executed. Google's way of combining query and ranking is unknown. Probably, empirical parameters and manual inspection are necessary.

1.5 Clustering

The motivations for clustering are related to the huge number of documents retrieved by web searches. To avoid overloading the user, identifying groups of closely related documents is useful, for example to show only a small number of representative prototypes.

Automatically identified clusters can also be a help for later manual classification à la Yahoo. In addition, if a user is interested in document d, he is likely to be interested in documents in the same cluster and therefore pre-computed clusters permit to obtain more documents similar to the one under examination on demand.

Let's note that queries can be ambiguous, especially *web* queries. For example, if one searches for `star`, one may look for movie stars, or for celestial objects, clearly two very different topics.

Mutual similarities in term vector space can help grouping similar documents together, i.e., to find "clusters" of documents. Let D be the corpus of documents (or other entities) to be grouped together by similarity. Items $d \in D$ are characterized either *internally* by some intrinsic property (e.g., terms contained, coordinates in TF-IDF space) or *externally* by a measure of distance $\delta(d_1, d_2)$ or similarity $\rho(d_1, d_2)$ between pairs. Examples are: Euclidean distance, dot product, Jaccard coefficient. After defining the metric, the usual bottom-up or top-down clustering techniques can be used. The methods have been explained in Chapter **??** and **??**.

Gist

The Web is a vast expanse of data, some of it structured, some partially structured or not at all. **Crawling and indexing** are systematic methods to visit web pages, harvest the information contained therein and prepare data structures for searching, information retrieval and ranking.

By transforming text into vectors of data (e.g., frequencies of selected words as in the **vector-space model**) some traditional ML techniques can be reused, but the richer amount of structure in web documents permits a more focused analysis.

Web-mining schemes find explicit relationships between documents (web links), infer implicit ones (by clustering), rank the most relevant pages in a network of connected sites or identify the most relevant and well-connected person in a network of people. Abstraction helps to use similar tools for networks of pages and networks of people. As a notable example, the use of hyperlinks and linear algebra tools (eigenvectors and eigenvalues), previously used to rank researchers in bibliometrics, leads to a very powerful technique to **rank web pages**, now at the basis of Google search-engine technology.

From now on, you will look at your hyperlinks, Facebook “Likes” and Twitter “Followers” (or at the social network software which will be the most popular when you read this book) with new analytic and aware eyes.