Information Retrieval

Contents

Introduction

Information Retrieval (IR) is finding material of unstructured nature that satisfies an information need from within large collections. Unstructured data typically refers to free text to query for some keywords or concepts.

The boolean retrieval model is used to respond to a query composed as a boolean expression, that is using `AND`, `OR` and `NOT` operators to join the query terms. The first idea to maintain the relation between terms and documents containing them could be to use a matrix, but in practice this data structures turns out to be huge and sparse.

To solve this issue we make use of an inverted index where for each term t we must store a list of all documents that contain t, each of these identified by a unique id. The set of terms is called the dictionary, while the documents are called posting and consequently a list is called a postings list.

The `AND` operation can now be implemented by intersecting the postings list, doing this in growing order is one of the possibile optimizations to speedup the computation.

Crawling

Web crawling is the process by which we gather pages from the Web graph to index them and support a search engine. It is possibile to recognize different features that a web crawler should implement:

- Quality, it should be biased toward fetching "useful" pages first.
- Efficiency, it should avoid duplication, or near duplication, of the content crawled.
- Netiquette, there exist implicit and explicit policies regulating the access to a server by a crawler.
- Freshness, the content crawled should ideally reflect an updated image of the web.

Architecture

The crawler begins with one or more URLs, that constitute a seed set, inside its URLs frontier. Continuously it picks a URL from the frontier, then fetches the corresponding web page. The fetched page is then parsed, to extract both the text and the links from the page. The extracted text is fed to a text-indexer, while the extracted links are then added to the URL frontier, which at all times consists of URLs whose corresponding pages have yet to be fetched by the crawler.

In the architecture of a crawler we can distinguish many different modules, all of these are executed by different, and possibly multiple, threads in a parallel and distributed computing fashion.

- Link extractor

```
while ( PageRepository not empty )
  pop page from PageRepository
  check for near duplication
  extract links
  push links in PriorityQueue
```

- Downloader

```
while ( AssignedRepository not empty )
  pop URL from AssignedRepository
  download and archive page
  push page in PageRepository
```

- Crawler manager

```
while ( PriorityQueue not empty )
  pop URLs from PriorityQueue
  foreach URL
    check if recently crawled
    preprocess URL (DNS, robots, etc)
    push URL in AssignedRepository
```

Mercator

To guarantee the desired features of quality and politeness, it is mandatory for a crawler to implement some priority policies in the extraction of the URLs to be parsed. In the Mercator architecture this is done by using two set of FIFO queues: the front queues F and the back queues B.

After being extracted, a new URL is analyzed with some heuristics able to determine its priority value i, that is an integer between 1 and $|F|$. After this the value is assigned to the i-th front queue.

The URLs are then extracted from the front queues in a way biased by the priority of the queue, after this each one of the extracted URLs is pushed into one back queue according to its host. Each of the back queues B contains URLs from a single host, the mapping from hosts to back queues is maintained by an auxiliary table T. If one of the back queues gets empty a new host is immediately reassigned.

The URLs from the back queues are then extracted and inserted in a min-heap. This data structure contains one and only one element for each queue and it's ordered by the timestamps t_e, that represent the first eligible moment to repeat a request to a certain host. The crawler main loop extracts the root of the min-heap, waits for the indicated time t_e to expire and then adds the URL to the `AssignedRepository`.

Bloom Filter

To check if a page has been parsed or downloaded before a first control can be done on the URL, obviously, given the size of the web graph, storing all the crawled URLs in a dictionary is not a option.

A Bloom Filter (1970) is a probabilistic data structure used to check the elements of a set, it ensures a positive result if an element is in the set, but could also provide false positives. A binary array of size m is created, and then a family of k hash functions is used to map an object to a position in the array. To check if a certain element u is in the array is equivalent to the proposition $\wedge_{i=1}^{k} h_i(u)$.

Under the assumption of simple uniform hashing, the probability that a fixed position in the array contains zero is $p = (1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$, and so the probability of a false positive ϵ is equal to:

$$\begin{aligned} P(\forall i.B[h_i(u)] \neq 0) = \\ P(B[h_i(u)] \neq 0)^k = \\ (1 - P(B[h_i(u)] = 0))^k = \\ (1 - e^{-kn/m})^k \end{aligned}$$

Fixed the size of the array m and the number of elements in the set n, from this formula we can derive an optimal value for the number of hash functions $k_* = \ln 2 \frac{m}{n}$, that implies an error rate $\epsilon_* = (0.6185)^{\frac{m}{n}}$.

Various applications

The distributed computation of set intersection between two sets A and B can be done by computing the Bloom Filter of the first set, than computing Q as the set of all the $b \in B$ elements that are present according to the bloom filter and then by intersecting A and Q, where the advantage is given by the fact $|Q| < |B|$. The bit cost of the transmission of the Bloom Filter of A and Q is $\Theta(m_A) + (|A \cap B| + \epsilon|B|) \log |U|$, so it less expensive than sending the whole A set at a cost of $|A| \log |U|$.

The distributed computation of approximate set difference can be derived by the previous algorithm, when A can correctly compute the difference via $A - B = A - Q$, and B can obtain an approximation via $B - A \approx B - Q$.

Another possibile approach to compute the set difference is using Patricia Trees derived from A and B. Comparing each node in a top-down fashion, if the node are equals the visit backtracks, otherwise it proceeds to all children, when a leaf is reached then the corresponding element of B is declared to be in $B - A$. This solution is obviously unfeasible in practice because of the cost of replicating the subsets in a node, but this could be avoided by using the same algorithm on a Merkle Tree, that stores instead the hash of each subset in the corresponding node. It's possible to improve again the space occupancy by computing a Bloom

Filter of the nodes of the Merkle Tree MT_A, and then visiting top-down the MT_B tree each node is checked against $BF(MT_A)$. This improvement comes at the cost of possible false positives during the matching of the nodes between the two trees. Assuming $m_A = \Theta(|A| \log\log |B|)$ and the optimal $k_A = \Theta(\log\log |B|)$, the bit cost of the transmission is $O(|A| \log\log |B|)$, that is the bit size of the Bloom Filter.

Spectral Bloom Filter

A Spectral Bloom Filter is a variation of a standard BF that makes use of an integer array instead of a binary one, where each position of the array counts the number of occurrences of an object $s \in S$ in a multiset M. The space usage is slightly larger, but in a constant order, and opens to new possible use cases like aggregate and iceberg queries. The error probability is the same of the standard bloom filter, so $\epsilon \approx (1-p)^k$.

The insertion and deletion operations are implemented by simply incrementing, or decrementing, each counter derived by the application of each hash function to the element. The query result is instead given by the minimum of all the counters, this minimum selection allows to select the counter where the minimum number of collision have occurred.

It can be noticed that if the minimum of all the counters relative to an object is repeated multiple times, it is less likely that the same item is subject to a false positive error. This situation, called recurring minimum, can be used to improve the overall performances of the SBF. We can operate then with two separate SBF, where the second one is smaller and is used to store single minimum elements.

```
INSERTION
insert x in SBF1
if x single minimum in SBF1:
  if x in SBF2:
    insert x in SBF2
  else:
    set counters of x in SBF2 as the min value of x in SBF1

DELETION
delete x from SBF1
if x single minimum in SBF1:
  if x in SBF2:
    delete x from SBF2

LOOKUP
if x recurring minimum in SBF1:
  return min x in SBF1
else if x in SBF2:
  return min x in SBF2
```

```
else
  return min x in SBF1
```

Parallel Crawling

The web is too big to be crawled by a single crawler, so the work should be divided avoiding duplication of work. Using static assignment is difficult to load balance the URLs assigned to a crawler, also the situation of fault-tolerance where one downloader could be removed or created in a dynamic way makes the static assignment prone to errors.

A possibile solution is the use of the consistent hashing technique. Given two[1] hashing functions h_s : Server $\to x$ and h_u : URL $\to x$, we use an orientated circular mapping where the items are dynamically partitioned in arcs between the servers and, assuming clockwise orientation, each server needs to communicate only with its successor in case of mutation in the topography.

In average each server of the m server has assigned $\frac{n}{m}$ URLs, it's possibile to prove that this happens with hight probability.

Locality-Sensitive Hashing

In data analysis a frequent issue is, given a set S of items, each one with d features, to find the largest group of similar items. The similarity is a function that, taken two items, returns a value in the interval $[0, 1]$.

The brute-force approach is not useful because of the practically infinite number of possible groups, even limiting the group size to a constant L requires a huge amount of computational power. Introducing a certain level of approximation it's possibile to consider a clustering algorithm like the famous machine learning algorithm K-means.

In k-means, fixed a number of k clusters are fixed k random points. Until convergence the points are assigned to the nearest centroid that are then recomputed.

The LSH technique proposes instead to generate a fingerprint for every item, and then to transform the similarity between items into the equality of fingerprints. This approach is randomized and correct with high probability, also it guarantees local access to data reducing the number of I/O operations needed.

Given the hamming distance $D(p, q)$ between two binary vectors p and q, we define the similarity s as the probability, given an index i, that $p[i] = q[i]$, and this is equal to $s = (1 - \frac{D(p,q)}{d})$.

Now consider a set I of k random integers selected from $\{1, \ldots, d\}$, we call $h_I(p)$ the projection of p into the I positions. The probability that two fixed

[1] The slides of the course mention only one function, also in some of the exercises the same function is used for both objects.

projections of a pair of vectors are equal is $P(h_I(p) = h_I(q)) = s^k$.

Using L different projections we can state that p is similar to q with high probability if $\exists i.h_{I_i}(p) = h_{I_i}(q)$. The probability of this event is:

$$\begin{aligned} P(p \text{ matches } q) = \\ P(\exists i.h_{I_i}(p) = h_{I_i}(q)) = \\ 1 - P(\forall i.h_{I_i}(p) \neq h_{I_i}(q)) = \\ 1 - P(h_{I_i}(p) \neq h_{I_i}(q))^L = \\ 1 - (1 - s^k)^L \end{aligned}$$

So strictly dependent on the actual similarity s, between p and q. It's possibile to notice that while the k value reduces the false positives, the L reduces the false negatives.

In the practice of grouping similar items this technique is applied by generating L sets I_i, then computing for each item in the set its sketch, that is the L-ple containing all the h_{I_i} projections. Generating a graph of items where each node has an edge with any node with at least one equal projection in the sketch, permit to define the groups as the connected components in the graph. Given that this can be implemented via only scan and sort primitives, the number of IO operations to do this is $\tilde{O}(\frac{n}{b})$.

In the case of online queries instead, it's possible to create L hash tables where each table has 2^k elements, similar elements to the queried one are the elements that collide with it.

Comparing LSH with K-means we could use this resume:

Algorithm	Optimality	Cost	Cost per iteration	Number of cluster
LSH	Global with high probability	Short sketch comparison	Sort $\|S\|$ items	Learned
K-nn	Local	D-features comparison	$K \times S \times d$	Parameter

Document duplication

The Web contains multiple copies of the same content that search engines try to avoid indexing, to keep down storage and processing overheads. Theoretically any hash function could be used to store a fingerprint of a web document to be compared for each new document, obviously there are some efficiency and space constraints that makes some solutions more interesting than others.

The rolling hash technique described by Karp-Rabin fingerprints is commonly used. Given a prime number p, the fingerprint of an m-bit string A is $f(A) = A \mod p$, it's possibile to easily compute any shift of A. The probability of a collision between any pair A and B is equivalent to the probability that p divides $A - B$, that is practically zero.

This simplistic approach fails to capture a crucial and widespread phenomenon on the web: near duplication. In many cases, the contents of one web page are identical to those of another except for a few characters.

Shingling

Given the set of all the possibile q-grams of a document and their fingerprints called shingles, the shingling technique is used to reduce the near-duplicate document detection problem to intersection of the set of shingles of two distinct documents. We declare that page A and B are near duplicated if the intersection of S_A and S_B is large according to an arbitrary measure, like the Jaccard similarity defined as follows:

$$J(S_A, S_B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}$$

This process requires a big amount of space, and the full cost of computing the intersection over the whole shingling sets.

Min-hashing

A possible way to approximate the Jaccard similarity between two sets is using L random permutations to generate a sketch given by the minimum of the set in each considered permutation.

$$< \min \pi_1(S_A), \ldots, \min \pi_L(S_A) >$$

To share the same minimum value in the same permutation π_i, the minimum must have been taken from the intersection between all the possible values, so it's immediate that $p = P(\min \pi_i(S_A) = \min \pi_i(S_B)) = J(S_A, S_B)$.

We are now able to approximate the Jaccard similarity of the two sets by counting the number of equal components between two sketches and normalizing it via L, this is sound because of this observation:

$$\frac{\mathbb{E}(\#\text{equal components})}{L} = \frac{L * p}{L} = J(S_A, S_B)$$

The space occupancy of this technique can be reduced at the cost of introducing approximate results by projecting each sketch into a set of k integers for L' times, and then group them by equal component as in LSH.

Cosine distance

Another possible way to compute the sketch of a vector is using the cosine distance. Given L random lines, we can compute the i-th element of the sketch as the result of the hash function $h_i(p) = \text{sign}(r_i \cdot p)$, where r_i is the i-th random line.

Two sketches share the same value iff. the line r doesn't lie in the angle α between the original vectors of the sketches, so the probability of this event is $P(h_i(p) = h_i(q)) = \frac{\pi - \alpha}{\pi} = 1 - \frac{\alpha}{\pi}$.

By dividing the number of not shared components between two sketches, divided by the number of components L we obtain an approximation of $\cos(\alpha)$ for small α angles.

Compressed storage of the web-graph

The directed graph representing the web has three peculiar characteristics:

- Skewed distribution, the probability that a node has x links follows the power law $\frac{1}{x^\alpha}$, where experimentally $\alpha \approx 2.1$.
- Locality, usually most of the hyperlinks point to URLs in the same host.
- Similarity, if two URLs are close in lexicographic order, then they tend to share many hyperlinks

Permuting the host, reversing the dot order, it's possible to create a sequence of adjacency lists that uses the locality and the similarity properties to generate contiguous areas. This situation can be exploited to reduce space occupancy using the copy-list technique. In this technique each list has associated:

- Out-degree, the number of links exiting the page.
- A reference list, in respect to there is the compression.
- A copy list, that is a bit-string of the same length of the out-degree of the reference.
- Extra nodes, that stores the links not shared with the reference.

The bits in the copy list can be compressed using run-length encoding in a copy-block. This is done by storing the first bit on the list and then the length of each region of consecutive equal bits minus one, the integer representing the last region of the list could be dropped because the number of bits is constrained by the out-degree.

Index construction

In sorting a list of strings, the indirect list containing pointers breaks locality at an higher level, when the number of elements is huge this becomes a considerable problem.

Blocked sort-based indexing

To construct an inverted index the collection of documents is scanned to generate the term-docID pairs, then the pairs are sorted by term and the docIDs are grouped in the posting list. For small collections, all this can be done in memory, the problem requires more attention in a scenario where large collections have to be indexed. To make index construction more efficient, we represent unique terms as termIDs, this requires a first scan to compile the vocabulary, and a second one to construct the inverted index.

With main memory insufficient, we need to use an external sorting algorithm like multi-way MergeSort. This algorithm sorts N items with a main memory of size M and disk-pages of size B, the first pass is to produce $\frac{N}{M}$ sorted runs, and then merge them for a total cost of $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M})$ IO operations.

Single-pass in-memory indexing

Blocked sort-based indexing has excellent scaling properties, but it needs a data structure for mapping terms to termIDs. For very large collection, this data structure does not fit into memory. A more scalable alternative is SPIMI, that using terms writes each block's dictionary to disk, and then starts a new dictionary for the next block.

The tokens in the document are analyzed one by one, and the relative posting list is filled using a doubling algorithm. When the memory is full the block is written to the disk, and a new one is started. At the end of the process the block is merged to generate the full index. The overall algorithm is faster because it doesn't require external memory sorting nor the creation of a term-termID vocabulary.

Distributed indexing

When the index dimension becomes to big the only solution is to use distributed indexing algorithms for index construction. Two obvious alternative index implementations are partitioning by terms and partitioning by documents.

Partitioning by terms means that each node of the cluster contains a partition of the terms and all of their posting lists. A query is routed to the nodes corresponding to its query terms. In principle, this allows greater concurrency, but in practice this behaviour turns out to be non-trivial for multiword queries and load-balancing.

A more common implementation is to partition by documents, each node contains the index for a subset of all documents. Each query is distributed to all nodes, with the results from various nodes being merged before presentation to the user. This partitioning simplifies the communication, but requires to contact all the nodes to compute any global operation.

Dynamic indexing

When dealing with dynamic collections new approaches are needed to correctly index them. One simple solution, called auxiliary index, consist in using multiple indexes, a main one and other smallest where to insert newly arrived documents and periodically re-index all the collection into one main index; also deletion can be handled in this situation by invalidating a bit vector. Storing each postings list as a separate file, then the merges imply consists of extending each postings list of the main index by the corresponding postings list of the auxiliary index, this is unfeasible because the difficulties for a file system in handling a big number of files. This consolidation process is costly.

A better solution is provided by logarithmic merge, where a series of exponentially increasing indexes are allocated in the disk and in memory an index is present larger as the smallest on disk. If an index I_i becomes too big its content is merged with the successive I_{i+1}, and so on if after merging also this index becomes too big, this maintains the invariant about the fact that each one of the indexes on the disk is either empty or full.

Each text participates to no more than $\log \frac{C}{M}$ merge operations because at each merge the text moves to a next index and they are at most $\log \frac{C}{M}$, where C is the total size of the collection.

Compression of documents

In a modern search engine the raw documents are needed for various applications, one of the more common is the dynamic extraction of a snippet depending on the query. The most important tradeoff in data compressione is given by the inverse proportionality of the compression rate and the decompression speed. Recently many technologies have been developed, among the others we recall Snappy and Brotly by Google and LZFSE by Apple.

LZ77

The LZ77 algorithm, used by gzip, compresses data by exploiting repeated substrings in a document. Given a document it generates a set of triplets of the form $\langle$ distance, length, next-char $\rangle$ by scanning the document using a fixed size buffer window. So, at position i the algorithm checks if a substring $A[i, k]$ is repeated at least once in the substring $A[i - b, i - 1]$, where b is the buffer size, if so it generates a triplet and skips to the next non compressed character. The decompression reverses this process, by applying the sorted triplets to the initially empty string ϵ.

Z-delta compression

This algorithm is used to compress a new file f_n by using another known file f_k. Reprising the idea of the LZ77 algorithm the file f_k and f_n are concatenated,

then by scanning from f_n the set of triplets f_d is generated, it's possibile to prove that f_d it's an optimal coverage, contaning the minimal number of triplets. Now it's possibile to obtain f_n concatenating f_k to the empty string ϵ and then by applying the triplets as in the LZ77 algorithm.

The Z-delta algorithm can also be used to compress a cluster of files by constructing a graph where all the files are represented by a node. Any directed edge $< i, j >$ is weighted by the number of triples needed to compress f_j using f_i as common knowledge. By adding a dummy node ϵ an optimal compression of the cluster is found by computing a minimum directed spanning tree rooted in the dummy node. It should be noticed that the number of edges is quadratic in respect to the number of files, and it's costly to generate them. Since we are intereseted in using only edges between similar files, we can use locality sensitive hashing as an heuristic in the construction of the graph.

rsync

This algorithm is used to synchronize the content of two files in a client-server scenario, let's suppose that the old one f_o is in the client, and the new one f_n is in the server and they're possibly very different, so we can't assume common knowledge between them. Given a block size b the clients generates a non-overlapping sequence of blocks, then each of these is hashed and sent to the server. Once that the server has received the hashes it inserts them in a dictionary. After this it scans f_n with a window of size b and rolling hashes each buffer, if it doesn't match any of the ones in the dictionary the character in the first position of the buffer is sent to the client, otherwise it sends the index of the block.

zsync

The zsync algorithm is useful to reduce the work load of the server and could be considered as a symmetrical approach to rsync. This time is the server that computes the $\frac{n}{b}$ blocks, and then sends their hashes to the client. Via rolling hash the client looks for the received hashes in f_o and replies to the server with a bitmask of length $\frac{n}{b}$ where $B[i]$ is set to one iff. the block H_i is present in f_o. Given the bitmask the server is able to z-delta compress the block that the client doesn't have according to the ones it owns, enabling the client to construct the full file.

Document parsing

After the documents are crawled their content has to be tokenized, that is generating a stream of tokens ("words") from a document. The tokenization is a critical module in a search engine, in OSS frameworks is common to leave the implementation to the user. After this the tokens are normalized by using one or more linguistic models, and once normalized they can be indexed. The first

issue in parsing a document is identifying features like the format, the language and the character set; all of these are classification problems dealt via heuristics.

Tokenization

A token is an instance of a sequence of characters in some particular document, that are grouped together as a useful semantic unit for processing. Each such token is now a candidate for becoming an index entry, that is a term, after further processing.

The major question of the tokenization phase is what are the correct tokens to use? An easy answer like splitting by space isn't appropriate in the majority of the use cases. The language identification is so fundamental because of the different ways to separate words that could become terms.

Stop words

A common practice was to don't index stop words, because they are frequent and meaningless. But the current trend is away from doing this, good compression techniques and good query optimization techniques enables the search engine to cheaply store stop words.

Normalization

The normalization is a language-dependent process that transforms tokens into a canonical form, this is done usually by removing hyphens and periods. Another issue regards case-folding, a best practice is to minimize all the characters and leave the context to the other words. Other than syntactically we have to handle via a thesauri cases of synonyms and homonyms, historically this was done brute force using an handmade thesauri.

Stemming and lemmatization

The stemmer is a module able to recognize variations of the same word, this is done by using root prefixes of the tokens. A possible stemmer can be built by using the Porter's algorithm.

Lemmatization is the process to reduce variant forms to a base form, for example the transformation of a conjugated verb to its infinite form.

Statistical properties of text

Tokens are not distributed uniformly, but they follow the so called Zipf law. The Zipf law states that few tokens are very frequent, a middle sized set has medium frequency and many are rare, formally: $f_s(k) = \frac{c}{k^s}$. This empirical law has been found true in all the known languages, most of the more frequent are stop words. The Zipf law is a power law, so its log-log plot is approximately a straight line.

Also the number of distinct tokens grows sublinearly according to the Heaps law n^β where $\beta \approx 0.5 < 1$ and n is the number of total tokens. The interesting words from an IR point-of-view are, according to Luhn law, the ones with medium frequency.

Keyword extraction

One interesting aspect in the process of keyword extraction is the analysis of the collocations. A collocation is a sequence of two or more words that correspond to some conventional way of saying where their constituent words are not substitutable or modifiable and can't fully infer the meaning of the collocation without the others.

Frequency sorting of all the adjacent pairs in a document is not useful because of the high frequency of prepositions, articles and other stop words. Better result can be obtained by using Part-of-Speech tagging, and by allowing only certain pairs to be ranked.

This solutions doesn't consider flexibility, often words are not adjacent to each other. A possible approach is computing the mean and the variance of the distance within a window of all the possible pairs of words. We can conclude that if the mean is large the collocation is not interesting, whilst if the mean is very small the pair should be treated as a collocation.

Bi-grams

Pearson's chi-squared test (χ^2) is a statistical test applied to sets of categorical data to evaluate how likely it is that any observed difference between the sets arose by chance. It tests a null hypothesis stating that the frequency distribution of certain events observed in a sample is consistent with a particular theoretical distribution. The events considered must be mutually exclusive and have total probability 1.

In inferential statistics, the null hypothesis H_0 is a general statement or default position that there is nothing new happening, like there is no association among groups, or no relationship between two measured phenomena.

In our scenario taking as a null-hypothesis that two words are independent, and so $P(A \circ B) = P(A)P(B)$ it's possible to perform Pearson's chi-squared test to assess whether observations consisting of measures on two variables, expressed in a contingency table, are independent of each other

A contingency table O, that counts the occurrences of all the possible pairs fixed A and B, has to be computed. The so called degree of freedom is dependent on the size of the contingency matrix, as in $df = (\text{rows} - 1)(\text{columns} - 1)$.

	$w_1 = A$	$w_1 \neq A$

	$w_1 = A$	$w_1 \neq A$
$w_2 = B$	O_{11}	O_{12}
$w_2 \neq B$	O_{21}	O_{22}

Than we define $E_{ij} = N * p_i * p_j$, where $p_i = \sum_j \frac{O_{ij}}{n}$ and $p_j = \sum_i \frac{O_{ij}}{n}$, and consequently to this:

$$\chi^2 = \sum_{i,j} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

The χ^2 has to be compared against the critical value of the distribution, dependent of the degrees of freedom and a certain confidence level arbitrary chosen, if the value is smaller than the threshold the null hypothesis is plausible, and so it's not a good pair for a collocation.

Rapid Automatic Keyword Extraction

The algorithm works on single, not much long, documents and provides fast and unsupervised keyword extraction exploiting the fact that keywords frequently contain multiple words but rarely contain punctuation or stop words.

Given a set of word delimiters, a set of phrase delimiters and a list of stop words, in the first phase the algorithm splits the document by words, then by phrases and then by stop words: the remaining words are considered candidate keywords

The ordered sequence of candidate keywords is scanned to compute the table of co-occurences where the item M_{ij} counts the number of pairs that contains both the i-th and the j-th word. Given this table is possible to compute the frequency of a word as M_{ii}, and the degree of a word as the sum over its row $\sum_j M_{ij}$. We define the score of a word as the ratio $\frac{deg(w)}{freq(w)}$.

A set of adjoining keywords is a candidate if it appears at least twice in the document, if so the score for the new keyword is the sum of its member keyword scores. Sorting by score enables the user to select the k-th most significative keywords.

Search

Prefix-string

Given a dictionary D of K strings, of total length N, store them in a way that we can efficiently support prefix searches for a patter P over them.

A trie is a useful data structure for p-search, but every implementation suffers of big space issues. Using a 2-level indexing solution can mitigate the problem, dividing the sorted elements in pages and constructing a trie on a sampling we can use the trie as a router. This solution requires typical only one IO operation to visit the trie, and also requires less space given that the trie is built over a subset of strings. A disadvantage to consider is the tradeoff in speed vs space given by the bucket size used for the sampling.

With front-coding it's possible to further reduce space usage. To compress a sorted list of strings, the prefix is substituted by one byte that indicates the number of prefix shared characters with the previous word. Obviously random access is not possible, but in our scenario the scanning of a bucket was already required for p-search in a 2-level indexing solution. If the strings don't use a termination character, the length of the string also has to be stored.

Tolerant search

The area of spell correction can be used for two principal use cases: the correction of an indexed document and the correction of the user queries. Also this correction could occour by analyzing isolated words or by also considering the context of a sentence.

For the rest of the discussion we assume the presence of a lexicon, to correct isolated word in queries. Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q. The definition of closeness can vary using different distance measures.

Brute force

A brute force approach enumerates for each word in q in Q, all the possible $\bar{q}$ words with edit distance d from q, if $\bar{q}$ is in the dictionary then it's proposed as a suggestion to the user. The edit distance, or Levensthein distance, is computable via a dynamic programming algorithm where first the matrix E is constructed, and then the value $E[n][m]$ is the edit distance of the two strings.

$$E_{ij} = \begin{cases} i & j = 0 \\ j & i = 0 \\ \min \begin{cases} E_{i-1,j-1} + W_! \\ E_{i,j-1} + W_i \\ E_{i-1,j} + W_d \end{cases} & i \neq 0 \wedge j \neq 0 \end{cases}$$

One-error correction

The enumeration and the dynamic programming solutions are too costly, to improve the performance we have to assume the maximum edit distance as one. An useful approach is to trasform the problem exploiting the fact that

enumeration deletions is easy, so given the dictionary of correct words D_1 we can easily generate the dictionary D_2 that contains $\Theta(l|D_1|)$ words on average.

Now for each word $q \in Q$ we can operate as follows:

- If $q \in D_1$ the word is correct.
- For each $\bar{q}$ given by dropping a character in q, if $\bar{q} \in D_1$, then q had one char more and $\bar{q}$ has to be suggested.
- If $q \in D_2$, then q has one char less and so is not correct.
- For each $\bar{q}$ given by dropping a character in q, if $\bar{q} \in D_2$, then q may contain a mismatch and so it's not correct.

This algorithm request linear time respect to the number of words $q \in Q$, but suffers of a space problem and could generate false matching results.

Overlap distance

The overlap distance is an approximation of edit distance, used as an heuristic to generate a set of candidate words from a dictionary.

Assume that each word is anticipated by $k-1$ special character `$`, this ensures that the number of k-grams will be equal to the length of the string that generated it. All the words in a dictionary have to be partitioned in k-grams, and then an inverted index is built over the k-grams.

A query Q generates $|Q|$ k-grams, while each error affects k k-grams. So if Q and a given token have edit distance i, they have to share for sure at least $|Q| - ki$ k-grams.

Given a query word Q, by splitting it in k-grams we are able to find the corresponding inverted lists, and so keep track of how many k-grams Q shares with each word in these lists. Fixed i it's now possible to check all the strings in the list, those that satistfy the filter condition are flagged as candidate strings.

Wildcard queries

The wildcard queries $\alpha\star$ have been already considered by prefix-search, but also the form $\star\beta$ can be solved with the same techniques by constructing the data structures over the reverse of each term.

More interesting is the case of the form $\alpha \star \beta$. It's still possible to use the two previous approaches to p-search α and s-search β and then intersecting the results. If the resulting lists are big the intersection could be expensive in terms of time.

The permuterm index proposes an approach that pays with space to reduce time complexity. Each term in a dictionary is indexed under all the possible rotation of the word, so for example the term `word` is indexed by `word$,ord$w,rd$wo,d$wor,$word`. Given a query in the form $P \star Q$, the search is reduced to a p-search of the rotated query $Q\$P\star$.

Soundex

Under the name of soundex goes a class of heuristics to expand a query into phonetic equivalents. The soundex algorithm are language specific and mainly used for names; historically the first soundex was used to write down the names of the non-English immigrants in Coney Island.

Posting list compression

Given that the postings are encoded as integers it's a good idea to encode small numbers in less space than large numbers, this situation is furthermore very common if the posting list is preprocessed via gap-encoding.

The γ-code is a universal code for integers which uses a fixed model. Given an integer x and L the length of its optimal representation $B(x)$, x is represented as a binary sequence composed of two parts: a sequence of $L-1$ zero, followed by the binary representation $B(x)$. The decoding is easy, count the c consecutive number of zeros up to the first 1, then fetch the following $c+1$ characters and interpret the sequence as the integer x.

PForDelta code is a method that supports extremely fast decompression and achieves a small size in the compressed output whenever the values follow a Gaussian distribution. Fixed $a, b > 0$, all the integers in the interval $[a, a+2^b-1]$ are encoded with b bits by translating them in the interval $[0, 2^b-1]$, all the other integers are instead prefixed with b 1 bits before a standard integer representation. The encoding does not occur in streaming, but instead the b bytes ones are used as an escape symbol for the explicit coded integers, stored in another partition. The value b is a trade-off, for bigger values there will be less elements in the "extra values" partition, but the "normal" partition size is increased.

In the t-nibble code, the binary representation $B(x)$ is left-padded with zeros up to the minimum number of bits multiple of $t-1$. Then groups of t bits are generated by dividing the intermediate representation in buckets of $t-1$ bits, prefixing with 1 the first bucket and with 0 all the other. Given a sequence of bits, the decompression only require to scan shifting of t bits, concatenating the intermediate bits. The so called variable-byte code is a specialization of t-nibble, with $t = 8$.

The Elias-Fano code requires that the elements are strictly increasing, so the posting list must be explicitly kept and can't be gap-encoded. Fixed n the number of integers, m as the value of the maximum number plus one, we can compute $z = \log_2 n$ and $w = \log_2 \frac{m}{n}$. For each integer represented in $z+w$ bits the last w bits are concatenated in a list L, so $|L| = nw = n \log \frac{m}{n}$. The z remaining bits can possibly have 2^z representations, we then construct another sequence H by counting the occurrences of each representation in a negative unary[2] representation, so $|H| = 2^z + n$. Overall the cost of the representation

[2]The negative unary representation of a number consist in the repetition a 1 per unit,

is $|L| + |H| = n(2 + \log \frac{m}{n})$, wasting only two bits per number respect to the optimal code.

There exist data structures able to decompress numbers in constant time. We propose instead the following algorithm to get the k-th number of the list:

- Set r as the number of 0 bits up to the k-th 1 bit in H
- Get the k-th group of bits in L
- The value of the integer is the concatenation of the r-th prefix representation and the group of bits found in L.

Query Processing

Phrase queries

A phrase query is a query where multiple words are considered as an atomic unit, to be able to support such queries, it is no longer sufficient for postings lists to be simply lists of documents that contain individual terms.

One approach is to consider every pair of consecutive terms in a document as a phrase, to generate biwords. Each of these biwords is treated as a vocabulary term, and inserted as an entry in the dictionary. The query processing of a two-word phrase is immediate, furthermore longer phrases can also be processed by breaking them down in overlapping pairs and using the AND operator. Without examining the documents it's not possibile to verify the result of this boolean operation, that can possibile cause false positive results. To optimize the results it's possibile to use PoS tagging to construct an extended biwords index.

Another approach consists in storing in the postings list also the position in which the term occurs. Other than for phrase queries this approach can be used to solve free text queries, in which the results are biased according to the close proximity of each other. This is done according to the assumption that users prefer docs in which query terms occur within close proximity to each other. The combination of these two schemes is often used in real word search engine.

Usually when dealing with phrase queries the search engine makes use of an iterative process, called soft-AND. The engine tries to run the query as provided by the user, if the results are too many it tries to run multiple smaller queries and join them. If even in this case the set of results is too small the search engine must use other techniques, like vector space querying and then ranking the results.

Zone indexes

Up to now a document has been considered as a sequence of terms, but this view can be enlarged by considering other features like the author, the title,

terminated by a 0. That is 0, 10, 110, 1110, ...

the language and so on. This accessory features of a document are called its metadata.

A zone is a region of the document that can contain an arbitrary amount of text. The information about the zones can be stored in the dictionary or in the postings, in any case building an inverted index also on the zones allows the user to query directly on them.

Tiered index

Caching can be useful to speedup query resolution, there are two possible approaches: to cache the query results, exploiting query locality, or to cache pages of the postings lists, exploiting term locality.

Another possible solution consists in breaking up postings into a hierarchy of lists, sorted by importance. At query time only the top tier is used, unless it fails to provide a minimum number of documents, if so it recurse onto the lower tier.

Skip pointers

In a skip list the number of skips is an important trade-off, the more the skips the shorter the spans, in the opposite case is instead more likely to skip but lots of comparisons are required to evaluate skip pointers. A simple heuristic for posting list of length L is to use $\sqrt{L}$ evenly-spaced skip pointers.

Document ranking

By considering the matrix containing the relation of terms and documents, we can consider the i-th column as a binary vector representing the i-th document. Computing the intersection of the terms in two documents, that is counting the number of components both equal to one in the binary vectors, is a not so useful measure of similarity because it does not depend on the size of the sets. Two possible approaches to normalize this intersection are the Dice coefficient $D = 2\frac{|X \cap Y|}{|X|+|Y|}$ and the Jaccard coefficient $J = \frac{|X \cap Y|}{|X \cup Y|}$, that also respect the triangular inequality. An issue with both these measures resides in the fact that the importance of a term is not considered.

tf-idf

To start a discussion over the importance of a term in a document, a first approach could be counting the number of occurrences of term t in the document d, we call this value term frequency $tf_{t,d}$. Taking into account only the term frequency could be misleading because of the high frequency of stop words, to balance this situation we can use the inverted document frequency $idf_t = \log \frac{n}{df_t}$, where n is the number of documents in the indexed collection, while df_t is the

number of documents in the collection containing the term t. The product of these two measures is considered a good weight for a term in a document.

$$w_{t,d} = tf_{t,d} \times idf_t$$

The weight is zero if the term is not present in the document or if is not significative because it appears in most of the documents. By computing the weights for all the entries in a term-document matrix we are able to represent a document as a real valued vector, also the query will be considered as a very short document manageable in this vector space model.

Cosine score

One idea to determine the similarity between the weight vectors could be using the euclidean distance in the space, it should be noticed that this approach is dependent on the length of the vector and so it's not a good idea. A better approach could lead to measure the angle between two vectors, but we have to consider that being in a very high dimensional space the computation will be hard.

Computing the cosine of the angle is easier, and it is also a good similarity measure, considering that the document vectors are positive definite. The cosine of the angle between two document vectors can be computed as:

$$\cos(d, q) = \frac{\langle d, q \rangle}{||d|| \cdot ||q||}$$

It should be noticed that if the vectors d and q are already normalized the computation of the cosine it's reduced to the dot product only.

If the document is very well written the algorithm works very well, but it's easy exploitable by spammers using term repetition in a document, afflicting *tf* but not *idf*. The vector space solution is useful for bag-of-words queries and it's a good metaphor for similar-document queries, but it's not a good technique to use with operators.

The whole matrix can't be stored because of space issues, but we can store the vector by using inverted lists, also the query vector will be largely sparse we have a lot of space for optimizations. The positions of the terms in a document must be stored (possibly compressed), as seen in phrase queries, so the cardinality of the set of positions of a term in a document is *tf*, while the length of the posting list is *df*, so it's immediate the computation of the weight of a term in a document.

Since only the terms included in a query are interesting a possible algorithm initializes an empty vector of scores and then for every term t in the query it retrieves it's posting list after computing it's weight $w_{t,q}$. Then for each document d in a posting list, the score s_d is updated as in:

$$s_d = s_d + w_{t,d} \times w_{t,q}$$

To conclude the algorithm the scores must be normalized using the length of the documents, so $s_d = \frac{s_d}{l_d}$, taking now the top k components returns the top k similar documents to the query.

Approximate retrieval

Although the optimization over the sparse vector, the computation of the cosine distance is still costly, so we would like to reduce the computation to a set of documents A much smaller than the collection and then return the top-k docs in A as an approximation of the top-k in the whole collection. The same approach is also useful for other (non-cosine) scoring functions.

Index elimination

For a multiterm query it is clear that we only consider documents containing at least one of the query terms. We can take this a step further using additional heuristics:

- Consider documents containing terms whose idf exceeds a threshold.
- Consider only documents containing at least $|Q| - i$ query terms.

Champion Lists

To construct a champion list, each term is preprocessed to find the m best documents according to their tf-idf metric, to work well empirically must be $m > k$, also the value of m could be different for each term according to their importance. At query time the computation of the scores is done only on the champion lists and not on the posting list.

Fancy-hits

The fancy-hits heuristic is a variation of the champion list that makes use of an additional scoring scheme, the PageRank. In a preprocessing phase the docIDs are assigned decreasingly with respect to their PageRank, and also we compute for each term its champion list, called from now on FH, and the list of documents not in FH, called IL.

At query time the same process is repeated for each term, first thing the score is computed for all the documents in the fancy-hits list. Then to improve the results the IL list is scanned, computing the score up to a certain stopping criterion.

A sophisticated stopping criterion makes use of a value defined as the sum of the tf-idf and the PageRank of a document. Taken the minimum document x in

the champion list, we can assert that its tf-idf value is anyway greater than any of the documents in the IL list, and so:

$$\forall d \in \text{IL}. \quad s_d \leq \text{tf-idf}_x + PageRank_d$$

Since the PageRank is decreasing there will exist an element $\bar{d} \in$ IL such that $s_{\bar{d}} < \text{tf-idf}_x$, given that this property will be valid for also all the subsequent values the scan can be stopped.

Clustering

We can try to solve geometrically by clustering the documents, first of all $\sqrt{n}$ leaders are randomly extracted between all the documents and all the remaining documents are assigned to the nearest leader. At query time the result is obtained by seeking the k nearest docs from among the followers of the nearest leader, an obvious variation considers the nearest m leaders.

Exact top-k documents

The problem consists in, given a query Q, finding the exact top K documents for Q, using some ranking function r. The simplest strategy is to find all the documents interested by the query, compute the score for each document and then sorting to return the best top-K. This is obviously too costly as described in the reasons that introduced the approximate retrieval techniques, we want then to find an admissible heuristic approach the avoid computing the score on documents that won't make it into the top-K.

WAND

The WAND technique is a pruning method which uses a heap over the real documents scores. We could prove that the document IDs in the heap at the end of the process are the exact top-K. The algorithm follows a branch and bound approach, we maintain a running threshold Θ score, pruning away all the documents surely under the threshold and computing the exact scores only for the remaining.

Inside each postings list the documents have to be sorted by document IDs, also we have to assume a special iterator on the postings that can move a pointer to the first document ID greater than a certain value. This iterator allows only right-moves, and can be implemented by using skip pointer or the Elias-Fano compression technique.

$r(t, d)$ is a generic score function of the term t in the document d, also we want to compute the score of a document as $s(d) = \sum_{t \in T} r(t, d)$. Preprocessing the index we will make use of an upper-bound per term, such that $r(t, d) \leq u(t)$.

The threshold Θ is initialized to 0, and we want it always to hold that $\forall d \in$ top-K.$r(d) \geq \Theta$, updating it according to the score of the worst document currently in the top-K.

The algorithm parallel scans the postings list, at the first step of a generic iteration the terms are sorted according to the value of the document ID pointed. For each pointer we sum the terms upper bounds, finding the maximum score that a document can possibly obtain, the first document that possibly could have a score greater than the threshold is taken as pivot. All the documents between the previous pointers and the pivot have no possibility to enter the top-K, so their score computation is skipped. If the pivot is present in enough postings its score is computed, and if it's actually greater than the threshold it's inserted in the heap. If a document enters in the top-K the smallest has to exit, and the threshold is updated to the new smallest member.

In practice we can reduce score computation of the 90%.

Blocked WAND

The WAND algorithm makes use of an upper bound over the full list of a term. It's possibile to improve this upper bound by partitioning the set of documents, and storing for each bucket in the posting list the maximum score as upper bound.

The algorithm performs now a second check after the movement of the pivot. Reached the pivot the sum of the current bucket upper bound of the relevant terms is computed, if the sum is smaller then Θ we can discard all the documents whose right-end of the bucket is the leftmost one. Otherwise we have to compute the actual score and continue as the original algorithm.

Relevance feedback

The idea of relevance feedback is to involve the user in the information retrieval process so as to improve the final result set. This idea was faulty when first created in the 60s, but it's now getting interest because of the conversational queries where the user gives constant feedback.

The basic procedure consists in return in an iterative fashion a certain number of results, then asking to mark some of the returned documents as relevant or non relevant, the system is able now to refine the original query and so improve the result set. At the time the relevant/irrelevant documents where chosen by hand. The user should spend a certain amount of time by marking the results, also the final result will depend on the user choices, but also it's biased by the set returned by the search engine.

We are still in the vector space model, so the documents and the query are vectors.

In the Rocchio algorithm the query q it's updated by summing the relevant documents D_r and subtracting the non relevant ones D_n.

$$q = \alpha q + \frac{\beta}{|D_r|} \sum_{d \in D_r} d - \frac{\gamma}{|D_n|} \sum_{d \in D_n} d$$

The parameter α, β, γ are used to control the balance between trusting the query, trusting the relevant documents and not trusting the nonrelevant ones.

In pseudo-relevance feedback the choice is automated by the system, so the top-K results are assumed as relevant without asking any feedback to the user. This approach may works in practice, but can go wrong for some queries because of the bias introduced by the IR system.

While in relevance feedback the users give additional input on documents, used to recompute the scores, in query expansion the user give additional input on query words or phrases, possibly suggesting additional query terms. Autocompletion is an example of query expansion, as the suggestion of similar queries to the one proposed by the user.

Quality of a search engine

Information Retrieval has developed as a highly empirical discipline, requiring careful evaluation of the techniques used. Metrics as indexing speed, search speed and expressiveness of the language are just factors that contributes to the key measure, that is the user happiness. User groups are usually asked to evaluate the performance of a search engine, and factors like the UI/UX have proved to be crucial for the so called "user happiness".

The standard approach to IR system evaluation revolves around the notion of relevant and non-relevant documents, it should be noticed that the relevance is assessed relative to an information need and not to a query.

We are now able to partition the document set using the notions of relevance and retrieval, consequently defining two metrics. The Precision P is the fraction of retrieved documents that are relevant, so $P = \frac{\#(\text{Relevant} \cap \text{Retrieved})}{\#\text{Retrieved}}$. The Recall R is the fraction of the relevant documents that are retrieved, so $P = \frac{\#(\text{Relevant} \cap \text{Retrieved})}{\#\text{Relevant}}$.

By using the following contingency table it's possible to redefine P and R.

	Relevant	Not Relevant
Retrieved	true positive	false positive
Not retrieved	false negative	true negative

From the definitions given it's obvious that the Recall can't be estimated, because

it would require possibly a vast portion of the web, while we do not have this problem for the precision measure.

It's not trivial if an IR system should be optimized according to precision or to recall, it depends of the context of the application. We can combine the two values in a single measure, called the F measure, this trades off precision versus recall via the weighted harmonic means of these two.

$$F = \frac{1}{\alpha \frac{1}{P} + (1-\alpha)\frac{1}{R}}$$

If $\alpha > 1/2$ the F measure gives more weight to the precision respect that in the recall, we call $F1$ the balanced version of F with $\alpha = \frac{1}{2}$.

Web ranking

Considering the web graph we are interested in finding a mechanism for ranking the nodes, representing the pages. Under the assumption that an hyperlink between pages denotes author perceived relevance, it's possible to model the web graph using a random walker that given a page randomly choose where to go next, analyzing the random walker behaviour leads to the most visited pages that are so considered the most important. This can be done by modelling the web graph as a Markov process, to do so we need to get the transition probability matrix P from the adjacency matrix A, as in:

$$P_{i,j} = \frac{A_{i,j}}{\sum_z A_{i,z}}$$

Given the transition matrix P and a probability distribution x_t, representing the probabilities of being in a certain state at the moment t, we can easily update the probabilities by using vector-matrix multiplication: $x_{t+1} = x_t P$

Moreover we can prove that given the initial distribution x_0 it's possible to compute the probabilities at time t always using the same multiplication: $x_t = x_0 P^t$

A fixed point of the function defined by P is called the steady state distribution and it's equivalent to the eigenvector of the matrix P for the maximum eigenvalue 1. In our scenario the stationary distribution represents the proportion of time that a random walker spends visiting each node in an infinite walk, we assume it as a metric of its importance.

The existence of this eigenvector depends on these two conditions:

- The graph must be irreducible, that is $\forall i, j. \exists$. a path from i to j.
- The graph must be aperiodic, that is the MCD of all cycles length is one.

Given these the eigenvector exists unique, it does not depend on the initial distribution x_0 and represents the steady state distribution

Google PageRank

The web graph does not ensure the previous conditions, the intuition of Google inventors was to enforce these constraints.

They assumed the existence of a "teleport" operation, that allows the random walker to visit any node from any other. Fixing a constant $\alpha \in [0, 1]$ the random walker randomly jumps to any other node in the graph with probability $(1 - \alpha)$, while it chooses one of its proper neighbors with probability α. Superimposing the teleport graph to the web graph we obtain a complete graph, that is so irreducible and aperiodic, and so it has a unique steady state distribution.

We can formalize this using linear algebra by fixing a vector e where N is the total number of nodes in the web graph:

$$e = \begin{bmatrix} \frac{1}{\sqrt{N}} \\ \dots \\ \frac{1}{\sqrt{N}} \end{bmatrix}$$

The transition matrix P can be expanded as $P_\star = \alpha P^T + (1 - \alpha)ee^T$, and we can compute its left eigenvector as:

$$r(i) = \alpha \sum_{j \in B(i)} \frac{r(j)}{\#out(j)} + (1 - \alpha)\frac{1}{N}$$

The computation of this eigenvector is obviously extremely costly, and it's in practice done by extracting a random distribution and applying it to the augmented matrix $P^\star$ using the power law up to obtain $r \approx x_{2^k} = x_0 P_\star^{2^k}$ for $k \approx 7$. This is done by periodically preprocessing the web graph and using at query time the computed PageRank.

Personalized PageRank

A possibile variation on the original PageRank algorithm can be obtained by biasing the teleport step in the direction of a certain subset of pages, without considering the whole web graph. This can be done by simply mutating the e vector with a preference vector which jumps to preferred pages.

HITS

The Hypertext Induced Topic Search is a scoring system largely derived from the literature of the scientific papers ranking. Historically contemporaneous to

the PageRank and theoretically more interesting, it didn't made it in the long run because of the high computational cost at query time.

The mechanism is in fact query dependent and considers only a subset of the web graph. The set of the pages intersected by the query is called the root set, augmenting this with all the pages that links or are linked by the root set generates the so called base set. The base set is the interesting subset for the algorithm, that will operate using its adjacency matrix A.

The mechanism produces two scores per page[3]:

- Authority score, where a page has a good authority for a certain topic if it's pointed by many good hubs for a topic.
- Hub score, where a page has a good hub score for a topic if it points to many authoritative pages for that topic.

Reminding that the successors of a node can be determined by its adjacency matrix A, and symmetrically its predecessor by the transpose matrix A^T, the authority/hub mechanism is formalized as follows:

$$\begin{cases} a = A^T h \\ h = Aa \end{cases} = \begin{cases} a = A^T Aa \\ h = AA^T h \end{cases}$$

So h is the eigenvector of AA^T relative to the eigenvalue 1, the same for a and $A^T A$. This result is sound because of the symmetry of the multiplication of a matrix for its transposed. Also it's possible to weight the edges in A without losing any of the useful linear algebra properties.

Packing to fewer dimensions

Given the term-document matrix $A \in \mathbb{R}^{m \times n}$, we can associate every document to a column of the matrix, as in $d_i = A^{(i)}$. In the vector space model we already argued that the computation of the similarities between documents is an hard problem because of the high-dimensionality of the document vectors. We want to discuss how to speedup the cosine similarity with two techniques that are usually used subsequently, first reducing the number of dimensions by random projection, and then by further reducing via LSI.

LSI

The Latent Semantic Index is a data-dependent solution that creates a k-dim subspace by eliminating redundant axes by pulling together related dimension, aiming to resolve also synonymy and polysemy issues. To reduce the matrix A the documents are pre-processed by using a linear algebra technique called

[3]It could be useful to note that in the paper ranking literature the authority score is referred to papers, while the hub score to surveys.

Singular Value Decomposition (SVD). The decomposition creates a new smaller vector space, allowing to faster handle queries.

From the matrix A we generate a term-term correlation matrix $T = AA^T$ and a document-document correlation matrix $D = A^T A$. We can notice that $T_{i,j} = \langle t_i, t_j \rangle$ representing in our vector space model the similarity between the terms t_i and t_j, the same can be argued about $D_{i,j}$ in respect to the documents. Suppose that the matrix A had a rank $r \leq m, n$, we can now define U as the matrix containing the r linearly independent eigenvectors of T, and V as the same matrix with respect to D. We have now all the necessary bricks to construct the SVD decomposition of $A = USV^T$, where S is a diagonal matrix containing in decreasing order the eigenvalues of A[4].

By taking $k \ll r$ we can select the k biggest eigenvalues and their relative eigenvectors, reducing all the three matrices needed for the SVD and computing so $A_k = U_k S_k V_k^T$. A_k is provable as the best k-rank approximation of A, that is:

$$\|A - A_k\| = \min_{B \in \mathbb{R}^{m \times n}} \|A - B\|$$

Since we are interested in the similarities between documents we can define a new matrix $X = SV^T$, and its corresponding rank k approximation $X_k = S_k V_k^T$. This is sound because of $D = A^T A = X^T X$, so that X may substitute A when computing the document similarity. X_k has one column per document but the features are reduced from m terms to k concepts.

The similarity of the documents d_i and d_j can now be approximated by using only k multiplications if considering the following approximation:

$$\begin{aligned} & s(d_i, d_j) \\ &= \langle d_i, d_j \rangle \\ &= \langle A^{(i)}, A^{(j)} \rangle \\ &= \langle X^{(i)}, X^{(j)} \rangle \\ &\approx \langle X_k^{(i)}, X_k^{(j)} \rangle \end{aligned}$$

We can further investing the meaning of the concepts by noting that $(U_k)_{i,j}$ is the strength of the association between the term t_i and the concept t_j, the same holds for V_k and the document d_i.

Random projection

Another approach to reduce the dimensionality of the document vectors consists in extracting a random number of features.

[4]Formally S contains the singular values of A, but we consider them to be equals to the eigenvalues in our context because of some implicit assumptions that we will not discuss.

The Johnson-Linderstrauss lemma can be used to bound the error in computing the euclidean distance between two randomly projected vectors. Given a set P of n points in m-dimensions and $\epsilon > 0$, there exists a projection function $f\colon \mathbb{R}^m \to \mathbb{R}^k$, where $k = O(\epsilon^{-2} \log n)$, such that $\forall u, v \in P$ it holds

$$(1-\epsilon)\|u-v\|^2 \leq \|f(u)-f(v)\|^2 \leq (1+\epsilon)\|u-v\|^2$$

The lemma applies to the euclidean distance, while in our discussion we always used the cosine one, that we can actually also bound with some linear algebra manipulation:

$$\langle f(u), f(v)\rangle \leq \epsilon(\|u\|^2 + \|v\|^2) + (1-\epsilon)\langle u, v\rangle$$

Therefore, if u and v are normalized vectors, then the cosine distance changes by at most 2ϵ.

The Johnson-Linderstrauss lemma proves an existential results, not giving any clue on how to actually find or compute the f function. We assert, without proving it, that the projection matrix that implements the function f can be constructed using any random distribution with mean $\mu = 0$ and variance $\sigma = 1$, such as for example $P(F_{i,j} = -1) = P(F_{i,j} = 1) = \frac{1}{2}$.

Semantic annotation

The vector space model described up to here is mainly term-based, and it's too much subject to issues derived from polysemy and synonymy; more specifically two documents are considered similar only if they share some words. Using topic-based annotation we pursue the goal to construct algorithms that are able to resolve an information need, overcoming these issues.

An alternative approach consist in using graphs to organize the informations of a knowledge base, we will refer to this concept as knowledge graph[5]. In the graph each node is an entity representing a semantic unit, the edges can also be labelled indicating the kind of relation between the concepts.

Wikipedia is one of the simplest possible knowledge graphs, this is because the links are not typed but signify only a generic relation. The anchor includes a text that possibly differs with the name of the entity pointed, this pair of strings constitutes a mention-topic pair. By collecting all of these pairs relative to a fixed page, we can extract the common sense, that is how the people refer to a certain topic. Another useful kind of information that can describe a Wikipedia page is made of the categories it appears into, forming a directed acyclic graph.

[5]Knowledge Graph is the name given by Google to its own implementation. Given its diffusion and the simplicity of the name, the term is often used as a synecdoche to refer the general concept.

TagMe

We propose the definition of TagMe, as it's presented on its live online implementation[6]:

> TAGME is a powerful tool that is able to identify on-the-fly meaningful short-phrases (called "spots") in an unstructured text and link them to a pertinent Wikipedia page in a fast and effective way.

The system analyses a text, obtaining a set of entities that are then associated to a set of nodes in a knowledge graph.

To define if a given token is a feasible anchor, we can use as an indicator the link probability $lp(a) = \frac{\#\text{a as anchor}}{\#\text{a in the text}}$. Let's suppose for now that a text contains only two possible anchors a and b.

For each anchor we have now a set of possible pages, disambiguating an anchor means selecting only one page from this set. The first step consists in pruning from the set the less common pages: after setting a threshold τ, we can compare it against the commonness of a page p, with the respect to an anchor a, defined as $P(p|a) = \frac{\#\text{a linked to p}}{\#\text{a as anchor}}$.

We define the similarity between two pages $rel(p, q)$, as the Jaccard similarity between the set of pages pointing to p and the set of those pointing to q.

We can now define the score of a candidate page p_a for an anchor a, given that the text also contains the anchor b, as:

$$vote_b(p_a) = \frac{\sum_{p_b \in C(b)} rel(p_b, p_a) P(p_b|b)}{|C(b)|}$$

Where $C(b)$ is the candidate set of pages for the mention b.

The constraint of having only two anchors can be relaxed, in the case of multiple mentions we can average over the scores of all the possibile pairs of anchors. Using the results of the scoring system the top-ϵ are returned, and if the set is still to big or the results are uncertain we can filter again using the commonness indicator. Empirical values used are $\tau = 2\%$ and $\epsilon = 30\%$.

Another possible mechanism that can be used in the process of disambiguation is considering the words surrounding the anchor, trying to match them with the words in the documents supposedly pointed by the anchor.

[6] https://tagme.d4science.org/tagme/