Information Retrieval

Lorenzo Cristofori, Claudiu Daniel Hromei

October 2019

Indice

1 Boolean retrieval

Information retrieval significa trovare materiale (generalmente documenti) di natura non strutturata (generalmente testo) che soddisfi il bisogno di informazione in grandi collezioni (solitamente memorizzate nei computers), *ovvero garantire all'utente, in seguito ad una sua ricerca, i documenti e le informazioni che rispondono alla sua richiesta.* Inizialmente l'information retrieval era una attività per poche persone, ad esempio bibliotecari. Ora però sta diventando sempre più velocemente il metodo dominante di accesso alle informazioni.

1.1 An example of information retrieval problem

Supponiamo di voler determinare quali opere di Shakespeare contengano le parole Brutus AND Caesar AND NOT Calpurnia, all'interno del libro dei lavori di Shakespeare. Un modo di farlo è leggere tutto il testo vedendo quali opere contengono sia brutus che caesar ma escludendo quelle che contengono calpurnia. Questo processo viene comunemente chiamato grep. Ma per molti scopi, si ha bisogno di altro:

- Processare grandi collezioni di documenti velocemente.
- Permettere operazioni più flessibili. Ad esempio, è impossibile realizzare la query "Romans NEAR countrymen" con grep, dove NEAR puo essere definito come "a 5 parole di distanza" o "nella stessa frase".
- Permettere di ordinare i dati recuperati.

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
...							

► **Figure 1.1** A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t, and is 0 otherwise.

Il modo di evitare di analizzare linearmente i testi per ogni query è quello di indicizzare i documenti in anticipo. Supponiamo di memorizzare per ogni documento quali parole tra quelle usate da Shakespeare contenga. Il risultato è un matrice di incidenza termini-documenti in figura 1.1. I termini sono le unità indicizzate. Quindi, in base a come guardiamo la matrice, possiamo avere un vettore per ogni termine che indica in quali documenti appare, o un vettore per ogni documento che mostra i termini che compaiono al suo interno.

Per rispondere alla query "Brutus AND Caesar AND NOT Calpurnia", prendiamo i vettori di brutus, caesar e calpurnia, complementando l'ultimo, e facciamo un AND bit a bit.

$$110100\ AND\ 110111\ AND\ 101111 = 100100$$

La risposta sarà quindi "antony and cleopatra" e "hamlet". Il Boolean retrieval model è un modello di information retrieval nel quale ogni query puo essere posta come un'espressione booleana di termini. Il modello vede ogni documento come un semplice insieme di parole.

Il nostro goal è costruire un sistema per svolgere retrieval task ad hoc. Il sistema deve fornire i documenti della collezione che sono rilevanti per l'information need di un utente

generico, comunicato al sistema usando una query. Un'information need è un argomento che un utente desidera conoscere di più, è differente dalla query, la quale è ciò che l'utente inserisce nel computer nel cercare di comunicare il proprio information need. Un documento è rilevante per un utente se contiene dell'informazione di valore rispetto al proprio information need. Per valutare l'efficacia di un sistema IR un utente vuole conoscere due misure chiave riguardo ai risultati restituiti ad una query:

- *Precision*: quale è la frazione dei risultati che risulta rilevante per l'information need?
- *Recall*: quale è la frazione dei documenti rilevanti della collezione restituiti dal sistema?

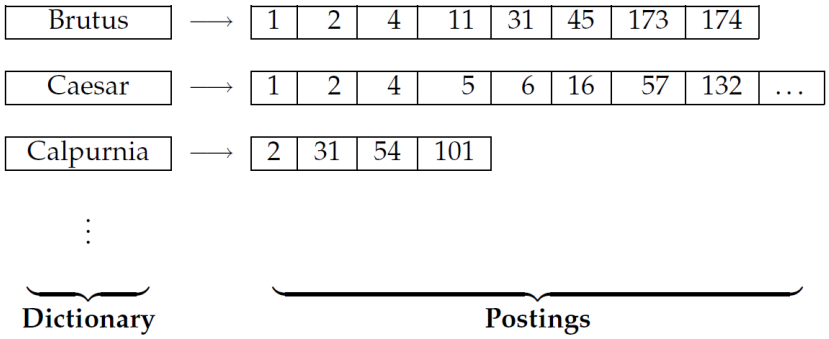
Supponiamo di avere 1 milione di documenti. Ci riferiamo all'insieme di tutti i documenti come collezione. Supponiamo che ogni documento contenga circa 1000 parole. Non possiamo costruire una matrice termini-documenti in un modo naive. L'osservazione cruciale è che la matrice è estremamente sparsa, ossia, ha pochi elementi non-zero. Un'ottima rappresentazione è salvare solo cio che si verifica, le posizioni di 1. Questa idea è centrale nel primo grande concetto dell'information retrieval, l'*inverted index*. L'idea base dell'inverted index è salvare un dizionario di termini dove, per ogni termine, avremo una lista che memorizza i documenti nei quali è presente il termine. Ogni oggetto della lista è comunemente chiamato posting. La lista viene quindi chiamata posting list, e tutte le posting lists prese insieme vengono chiamate the postings.

1.2 A first take at building an inverted index

Per ottentere dei benefici di velocità dall'indexing a tempo di recupero, abbiamo bisogno di costruire l'indice in anticipo. Gli step principali sono:

- Collezionare i documenti da indicizzare
- Tokenizzare il testo, trasformando ogni documento in una lista di tokens.
- Fare un preprocessing linguistico, producendo una lista di tokens normalizzati

- Indicizzare i documenti nei quali compaiono i termini costruendo un inverted index, costituito da un dizionario e dai postings.



▶ **Figure 1.3** The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

In una collezione di documenti possiamo assumere che ogni documento abbia un numero seriale unico, chiamato identificativo del documento (docID). L'input dell'indexing è una lista di tokens normalizzati per ogni documento, possiamo quindi pensarli come una lista di coppie term e docID. Il cuore dello step di indexing è ordinare questa lista in ordine alfabetico. Occorrenze multiple dello stesso termine nello stesso documento si eliminano. Istanze dello stesso termine sono raggruppate e il risultato viene splittato in dizionario e i postings. Il dizionario, inoltre, memorizza alcune misure, come il numero di documenti che contengono uno stesso termine (document frequency). I postinigs vengono poi ordinati secondo il docID, è alla base per un efficente elaborazione della query.

1.3 Processing Boolean queries

Come possiamo processare una query usando un inverted index e un modello Booleano base di retrieval? query: Brutus AND Calpurnia

- Trovare Brutus nel dizionario
- Recuperare i suoi postings

- Trovare Calpurnia nel dizionario
- Recuperare i suoi postings
- Fare l'intersezione tra le due postings list

L'operazione di intersezione è quella cruciale: bisogna riuscire a trovare velocemente i documenti che contengono entrambi i termini (questa operazione è conosciuta come merging postings lists).

1.3 Processing Boolean queries

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then p1 ← next(p1)
 9                else p2 ← next(p2)
10  return answer
```

► **Figure 1.6** Algorithm for the intersection of two postings lists p_1 and p_2.

Esiste un metodo semplice ed efficente per intersecare le postings lists usando l'algoritmo di merge. Manteniamo i puntatori nelle due liste e scorriamo le due postings lists contemporaneamente, in tempo lineare nel numero totale di elementi postings. Per usare questo algoritmo è fondamentale che le postings siano ordinate secondo un ordine globale. Possiamo estendere le operazioni di intersezione per processare query più complicate come:

Query: (Brutus OR Caesar) AND NOT Calpurnia

La *query optimization* è l'organizzazione del lavoro per rispondere ad una query in modo che il sistema debba eseguire il minor lavoro possibile.

```
INTERSECT(⟨t_1, ..., t_n⟩)
1  terms ← SORTBYINCREASINGFREQUENCY(⟨t_1, ..., t_n⟩)
2  result ← postings(first(terms))
3  terms ← rest(terms)
4  while terms ≠ NIL and result ≠ NIL
5  do result ← INTERSECT(result, postings(first(terms)))
6     terms ← rest(terms)
7  return result
```

► **Figure 1.7** Algorithm for conjunctive queries that returns the set of documents containing each term in the input list of terms.

1.4 The extended Boolean model versus ranked retrieval

Il modello Booleano di retrieval si contrappone ai modelli di retrieval ranked, come il vector space model, nei quali gli utenti usano largamente query testuali libere. Un'espressione Booleana sui termini con un insieme di risultati non ordinati risulta limitativa per molte information need che le persone possono avere. Si sono estesi i modelli di retrieval Booleani con operatori di term proximity. Un operatore di prossimità è un modo di specificare che due termini in una query devono occorrere insieme in un documento.
In alcuni casi gli utenti, specialmente professionisti, preferiscono modelli Booleani. Le query Booleane risultano precise: un documento o risponde ad una query o non lo fa. Questo offre agli utenti un grande controllo e una trasparenza su cosa viene recuperato. Inoltre in alcuni contesti, come per materiali legali, il modello Booleano può essere accompagnato da un sistema di ranking adeguato come un ordine cronologico decrescente. Il problema generale che si riscontra nei sistemi Booleani è che un'operazione di AND porta una alta precision ma una bassa recall mentre un'operazione di OR porta una bassa precision e una alta recall.
Rispetto al modello Booleano vorremmo poter fare cose aggiuntive:

- determinare in un modo migliore quali termini inserire nel dizionario e fornire un recupero che sia tollerante alla scelta errata delle parole.

- cercare frasi che denotino un concetto, inoltre fare query che mostrino un qualche senso di prossimità tra le parole.
- evidenziare il fatto che una parola occorre piu volte in uno stesso documento rispetto a parole che hanno una sola occorrenza in quello stesso documento (un modello Booleano memorizza solo la presenza e l'assenza di parole).
- ottenere un metodo per ordinare i risultati, poiché le query Booleane recuperano solo un insieme di documenti.

2 The term vocabulary and postings lists

Ricordiamo i principali steps nella costruzione del inverted index.

- Collezionare i documenti da indicizzare
- Tokenizzare il testo.
- Fare un preprocessing linguistico del testo
- Indicizzare i documenti secondo i termini che occorrono in questi.

Esaminiamo alcuni metodi della tokenizzazione e del preprocessing linguistico.

2.1 Tokenization

Data una sequenza di caratteri e definita l'unità documento, il tokenization è il task di tagliare "a pezzi" la sequenza, chiamati tokens. Questi ultimi vengono spesso collegati ai termini delle parole, ma è importante fare una distinzione tra type e token. Un token è un'istanza o una sequenza di caratteri di un dato documento che vengono raggruppati insieme a costituire un unità semantica utile per l'esecuzione. Un type è una classe di tutti i token che contengono la stessa sequenza di caratteri. Un term è un type che è stato incluso nel dizionario di un sistema di IR. Per esempio il type "cani" è come una classe di tutte le parole cani che possiamo incontrare in qualsiasi documento, mentre il token "cani" è la specifica sequenza di caratteri incontrata in un determinato documento.

La domanda principale della fase di tokenizzazione è: quali sono i tokens corretti da usare? Questo problema risulta essere fortemente dipendente dalla lingua ed è perciò importante poter conoscere la lingua di un certo documento.

2.2 Dropping common terms: stop words

A volte, alcune parole estremamente comuni che risultano fornire un piccolo contributo nell'aiutare a risolvere lo user need vengono escluse dal vocabolario. Queste parole vengono chiamate *stop words*. Una strategia comune per identificarle è quella di ordinare i termini secondo la *collection frequency*. Eliminare le stop words riduce in modo significativo il numero di postings che il sitema deve memorizzare.

2.3 Normalization (equivalence classing of terms)

Dopo aver trasformato i nostri documenti (e quindi anche le query) in tokens, ci sono molti casi nei quali due sequenze di caratteri non sono esattamente uguali ma per le quali vorremmo ci fosse una corrispondenza. Ad esempio, se si cerca USA vorremmo poter recuperare anche i documenti contenenti "U.S.A.". Quindi *Token normalization* è il processo di canonizzazione dei tokens in modo da corrispondere anche occorrenze superficialmente diverse. Il modo standard di normalizzare è quello di creare implicitamente delle classi di equivalenza rimuovendo, per esempio, i trattini all'interno dei documenti. Un altro modo è quello di mantenere delle relazioni tra tokens non normalizzati, costruendo quindi dei sinonimi come van e truck. Queste relazioni tra i termini possono essere memorizzate in due modi, un modo comune di indicizzare i tokens non normalizzati e di mantenere una lista di query espansa con più elementi del dizionario per uno stesso termine della query. Un termine query è quindi una disgiunzione di postings lists. L'alternativa è di fare l'espansione durante la costruzione dell'indice. Quando il documento contiene truck lo indiciziamo come van (o viceversa).

2.4 Stemming and lemmatization

Per ragioni grammaticali nei documenti possono essere usati diverse forme di una stessa parola, come organize, organizes, organizing. L'obbiettivo sia dello stemming che della lemmatization è quello di ridurre le parole nella loro forma base. Lo stemming si riferisce ad un duro processo euristico secondo il quale si elimina la parte finale delle parole(suffisso). La lemmatization, generalmente usa adeguadamente dizionario e analisi morfologica per riportare una parola nel suo cosi detto lemma. Confrontando lo stemming e la lemmatization per la parola saw, il primo restituirà s mentre la seconda può restituire see o saw, nel caso in cui il token sia visto come verbo o come nome. L'algoritmo più comune di stemming per l'inglese è il Porter's algorithm.

2.5 Positional postings and phrase queries

Ci piacerebbe poter fare query del tipo Stanford University in modo che documenti del tipo "The inventor Stanford Ovshinsky never went to university." non vengano recuperati. Consideriamo due approcci per supportare le phrase queries.

```
POSITIONALINTERSECT(p1, p2, k)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then l ← ⟨ ⟩
 5             pp1 ← positions(p1)
 6             pp2 ← positions(p2)
 7             while pp1 ≠ NIL
 8             do while pp2 ≠ NIL
 9                do if |pos(pp1) − pos(pp2)| ≤ k
10                      then ADD(l, pos(pp2))
11                      else if pos(pp2) > pos(pp1)
12                              then break
13                   pp2 ← next(pp2)
14                while l ≠ ⟨ ⟩ and |l[0] − pos(pp1)| > k
15                do DELETE(l[0])
16                for each ps ∈ l
17                do ADD(answer, ⟨docID(p1), pos(pp1), ps⟩)
18                pp1 ← next(pp1)
19             p1 ← next(p1)
20             p2 ← next(p2)
21        else if docID(p1) < docID(p2)
22                then p1 ← next(p1)
23                else p2 ← next(p2)
24  return answer
```

► **Figure 2.12** An algorithm for proximity intersection of postings lists p_1 and p_2. The algorithm finds places where the two terms appear within k words of each other and returns a list of triples giving docID and the term position in p_1 and p_2.

2.5.1 Biword indexes

Un approccio per supportare le phrase queries è considerare ogni coppia di termini consecutivi nei documenti come una frase. Ogni coppia di parole diventa quindi un term nel vocabolario. Tra tutte le possibili queries, i nomi e le frasi nominali svolgono un ruolo centrale nel descrivere i concetti nei quali sono interessate le persone. Questa osservazione può essere integrata nel modello di indicizzazioni biword in questo modo. Primo, tokeniziamo il testo e eseguiamo una part-of-speech-tagging. Possiamo quindi raggruppare i

termini in nomi, nomi propri, intervallati da articoli e preposizioni. Ora si considera ogni stringa di termini della forma NX^*N, dove N è un nome, X un articolo o preposizione.

2.5.2 K-word Indexes

Allo stesso modo potremmo pensare di costruire un Inverted Index usando K parole piuttosto che 2. Questo procedimento è simile a quello precedente e ci permette di essere molto più precisi nel caso di phrase query lunghe, tuttavia la costruzione dell'indice, il suo mantenimento in memoria e la ricerca delle posting diventa esponenziale. Questo perché dovremmo indicizzare N^K elementi, dove N è il numero di parole normalizzate (quindi dopo lemming/stemming) distinti e K è il numero di parole nello stesso token che vorremmo indicizzare.

3 Index construction

In questo capitolo vedremo come costruire un inverted index. Chiamiamo questo processo index construction o indexing.

3.1 Hardware basics

Quando si costruisce un sistema di information retrieval molte delle decisioni vengono prese in base alle caratteristiche hardware del computer sul quale il sistema dovrà girare. Cominicamo quindi con un piccolo riassunto dell'hardware di un computer.

- L'accesso dei dati in memoria è molto più veloce dell'accesso dei dati nel disco. Conseguentemente vogliamo salvare più dati possibile nella memoria, in special modo quelli ai quali accediamo più frequentemente. Questo processo di memorizzazione dei dati più frequenti nella memoria viene chiamato *caching*.

- Quando leggiamo o scriviamo sul disco, è necessario del tempo per muovere la testina nella parte del disco interessata. Questo tempo viene chiamato *seek time*. Per massimizzare la velocità di trasferimento dei dati, porzioni di dati che devono essere letti insieme dovrebbero essere salvati in locazioni contigue del disco.

- I sistemi operativi generalmente leggono e scrivono blocchi interi. La parte di memoria nella quale i blocchi vengono letti o scritti viene chiamata *buffer*.
- I trasferimenti di dati dalla memoria del disco sono gestiti dal bus di sistema, non dal processore. Si può velocizzare questo processo salvando dati compressi nel disco.
- I computer usati nell'IR hannno tipicamente molti gigabyte di memoria.

3.2 Blocked sort-based indexing

Nella costruzione di un indice non posizionale inizialmente analizziamo tutta la collezione formando tutte le coppie term-docID. Quindi ordiniamo le coppie con il term come chiave dominante e il docID come chiave secondaria. Infine organizziamo i docIDs per ogni termine in una postings list e calcoliamo delle statistiche come la term e la document frequency. Per rendere la costruzione dell'indice più efficente rappresentiamo i terms come termIDs, dove ogni termID è un numero seriale unico. Ad ogni modo, nel caso non avessimo abbastanza memoria abbiamo bisogno di un algoritmo di ordinamento esterno, ossia che usi il disco. Una possibile soluzione è il *blocked sort-based indexing algorithm* o *BSBI*.

- Si divide la collezione in parti (variabili, cioè il numero non è fissato, ma dipende dalla lunghezza della collezione) di uguale dimensione.
- Si analizzano le varie parti della collezione producendo le coppie termID-docID.
- Si ordina ogni parte per termID-docID in memoria.
- Si salvano i risultati parziali ordinati nel disco
- Si uniscono tutti i risultati intermedi nell'indice finale.

```
BSBIndexConstruction()
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4     block ← ParseNextBlock()
5     BSBI-Invert(block)
6     WriteBlockToDisk(block, f_n)
7  MergeBlocks(f_1, ..., f_n; f_merged)
```

► **Figure 4.2** Blocked sort-based indexing. The algorithm stores inverted blocks in files $f_1, \ldots, f_n$ and the merged index in f_{merged}.

La dimensione del blocco viene scelta in modo che entri nella memoria per permettere un veloce ordinamento. Il blocco viene quindi invertito e scritto sul disco. L'inversione si compone di due step. Si ordinano le coppie termID-docID, si salvano tutte le coppie termID-docID con lo stesso termID in una postings list, dove un posting è semplicemente un docID. Nella fase finale l'algoritmo fonde contemporaneamente tutti i blocchi in un unico grande indice.

La complessità temporale è $\Theta(TlogT)$ poiché lo step con maggiore complessità temporale è l'ordinamento e T è un upper bound al numero di elementi da ordinare. Tuttavia il tempo di indexing è dominato dal tempo richiesto nel parsare i documenti (ParseNextBlock) e dallo step finale di merge (MergeBlocks).

3.3 Single-pass in-memory indexing

Blocked sort-based indexing ha delle eccellenti proprietà di scalatura, ma necessita di una struttura dati per mappare terms e termIDs. Per collezioni molto grandi, questa struttura dati potrebbe non entrare in memoria. Un'alternativa maggiormente scalabile è *single-pass in-memory indexing* o *SPIMI*.

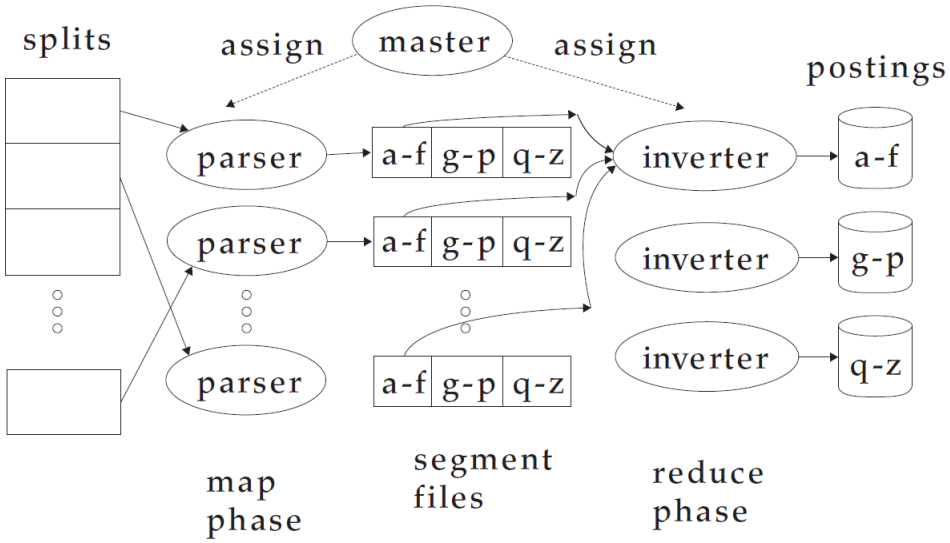
```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

► **Figure 4.4** Inversion of a block in single-pass in-memory indexing

SPIMI usa i terms piuttosto che i termIDs, scrive ogni blocco del dizionario nel disco, quindi comincia un nuovo dizionario per il prossimo blocco. SPIMI puo indicizzare collezioni di ogni dimensione fino a quando c'è spazio disponibile nel disco. Una parte dell'algoritmo prevede di trasformare i documenti in stream di coppie di term-docID, chiamati tokens. SPIMI-Invert viene chiamato ripetutamente nel token stream fino a che viene processata l'intera collezione. La differenza tra BSBI e SPIMI è che quest'ultimo aggiunge un posting direttamente alla postings list. Al contrario del primo ogni postings list è dinamica ed è immediatamente disponibile per memorizzare postings. Questo ha due vantaggi: è veloce perche non è necessario ordinare, e viene salvato in memoria. Dato che non possiamo sapere quanto una postings list di un termine sia grande quando viene incontrato per la prima volta il term, si alloca spazio per una postings list corta e si raddoppia lo spazio ogni volta che questa si riempie. Quando la memoria è piena si scrive l'indice del blocco nel disco. Prima di salvare in memoria è necessario ordinare i terms in ordine lessicografico per facilitare l'ultimo step di merge. Inoltre per costruire la nuova struttura del dizionario per ogni blocco ed eliminare lo step di ordinamento, SPIMI ha un terza componente molto importante: la compressione. Sia i postings che il dizionario dei terms possono essere salvati in modo compatto nel disco.

3.4 Distributed indexing

Le collezioni possono essere cosi grandi da non permettere la costruzione dell'indice in una singola macchina. In questi casi vengono attuati degli algoritmi di indexing distribuito. Il metodo di costruzione di indici distribuita è una applicazione di *MapReduce*. MapReduce è progettata per un grande cluster di computer. Un requisito per un indexing distribuito robusto è la divisione del lavoro in chunks che possono essere facilmente assegnati e riassegnati (nel caso fallissero). Un nodo master dirige il processo di assegnazione e riassegnazione di tasks a nodi worker individuali. Le fasi di map e reduce dividono il lavoro in chunk che le macchine possano processare in tempi veloci. Inizialmente, i dati di input, ad esempio una collezione, vengono divisi in n parti. Le varie parti vengono assegnate dal master node secondo la seguente politica: se una macchina finisce di processare una parte le viene assegnata la seguente. In generale, la fase di map consiste nel mappare le parti dei dati di input in coppie chiave-valore. La macchina che esegue la fase di map viene chiamata parser.



► **Figure 4.5** An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).

Ogni parser scrive il proprio output in file locali intermedi, detti *segment files*. Per la fase di reduce, vogliamo che tutti i valori di una stessa chiave vengano memorizzari insieme, cosi che possano essere letti e processati velocemente. Salvare tutti i valori di

una data chiave in una lista è il task degli inverters. Il master assegna ogni partition term ad un diverso inverter. Ogni partition term viene processata da un diverso inverter. Finalmente, la lista dei valori viene ordinata per ogni chiave e scritta nella postings list ordinata finale.

I parser e gli inverters non sono tipi diversi di macchine. Il master semplicemente identifica delle macchine inattive e assegna loro dei tasks. Una stessa macchina può quindi essere un parser nella fase di map e un inverter nella fase di reduce. Per minimizzare il tempo di scrittura da parte degli inverters nella fase di reduce, ogni parser scrive i propri file segmento nel proprio disco locale. Nella fase di reduce, il master comunica agli inverter la posizione del file relativo al segmento. Ogni file segmento richiede, in questo modo, un lettura sequenziale.

3.5 Dynamic indexing

Fino ad ora, abbiamo assunto che la collezione dei documenti fosse statica. Questo è vero per collezioni che non cambiano frequentemente o mai. La maggior parte delle collezioni però vengono modificate frequentemente con l'aggiunta, la cancellazione o la modifica di documenti. Questo significa che devono essere aggiunti al dizionario i nuovi termini, e devono essere aggiornate le postings list dei termini già presenti. Il modo più semplice di fare questo è quello di ricostruire periodicamente l'indice da zero. Questa è una buona soluzione se il numero di modifiche risulta piccola nel tempo e il ritardo nel rendere un documento recuperabile è accettabile.

Se esiste un requisito sulla velocità di aggiunta dei documenti, una soluzione può essere quella di mantenere 2 indici: un indice principale grande, mantenuto sul disco, e un indice ausiliario piccolo che memorizza i nuovi documenti, mantenuto in memoria. Le ricerche vengono quindi fatte in entrambi gli indici e le due risposte vengono mergiate. I riferimenti per i documenti cancellati vengono mantenuti in un vettore di bit (1 se attivo, 0 se cancellato). Inoltre i documenti possono essere aggiornati cancellando i vecchi e reinserendo i nuovi. Ogni volta che l'indice ausiliario diventa troppo grande, viene unito all'indice principale. Per ridurre i costi del merge si potrebbe pensare di salvare ogni indice in un file differente, questo nella realtà non è possibile per problemi di limiti dei

file-system, quello che viene fatto quindi è un compromesso tra tenere ogni indice in un file differente e mantenere un file unico di tutti gli indici. Secondo questo schema, processiamo i T postings T/n volte poiché viene fatto durante ognuna delle T/n fasi di merge, dove n è la dimensione dell'indice ausiliario e T il numero totale di postings. Otteniamo un complessità totale di $\Theta(T^2/n)$.

```
LMERGEADDTOKEN(indexes, Z_0, token)
 1  Z_0 ← MERGE(Z_0, {token})
 2  if |Z_0| = n
 3    then for i ← 0 to ∞
 4         do if I_i ∈ indexes
 5              then Z_{i+1} ← MERGE(I_i, Z_i)
 6                   (Z_{i+1} is a temporary index on disk.)
 7                   indexes ← indexes − {I_i}
 8              else I_i ← Z_i    (Z_i becomes the permanent index I_i.)
 9                   indexes ← indexes ∪ {I_i}
10                   BREAK
11         Z_0 ← ∅

LOGARITHMICMERGE()
1  Z_0 ← ∅    (Z_0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMERGEADDTOKEN(indexes, Z_0, GETNEXTTOKEN())
```

► **Figure 4.7** Logarithmic merging. Each token (termID,docID) is initially added to in-memory index Z_0 by LMERGEADDTOKEN. LOGARITHMICMERGE initializes Z_0 and *indexes*.

Possiamo fare meglio? si! Introducendo $log_2(T/n)$ indici $I_0, I_1, I_2, ..$ di dimensione $2^0n, 2^1n, 2^2n, ..$ (cioè dimensione doppia rispetto all'indice precedente). Questo schema viene chiamato *logarithmic merging*. Come prima, accumuliamo postings nell'indice ausiliario in memoria, che chiamiamo Z_0. Quando si raggiunge il limite di n postings, Z_0 viene trasferito nel nuovo indice I_0, di dimensione 2^0n, creato nel disco. La prossima volta che il nuovo Z_0 sarà pieno, verrà trasferito anch'esso in memoria e "mergiato" con I_0 creando un indice Z_1 di dimensione 2^1n. Infine anche Z_1 viene salvato come I_1, o mergiato con I_1 creando Z_2 (nel caso in cui I_1 esistesse già); e cosi via. Il tempo totale di costruzione dell'indice diventa $\Theta(Tlog(T/n))$ poiche ogni posting viene elaborato una sola volta per ognuno dei $log(T/n)$ livelli. Come nel caso dello schema dell'indice ausiliario, dobbiamo ancora mergiare grandi indici, ma questo accade meno frequentemente e gli indici coinvolti nel merge sono in media più piccoli.

4 Index compression

In questo capitolo vedremo alcune tecniche di compressione del dizionario e degli indici che risultano essenziali in un sistema IR che sia efficente. Ci sono molti benefici nel fare compressione.

Il primo è incrementare l'uso della cache. Ad esempio, mettere in cache la postings list di un term query t molto frequente permette di gestire le computazioni necessarie a rispondere alla query direttamente in memoria. Inoltre, con la compressione siamo in grado di mettere molte più informazioni nella memoria principale. Piuttosto che spendere tempo durante la disk seek (operazione di ricerca *fisica* delle celle di memoria sul disco) nel processare una query, possiamo accedere alla postings list in memoria e decomprimerla direttamente. Il secondo vantaggio della compressione è una maggiore velocità di trasferimento da disco a memoria. Il tempo totale impiegato nel trasferire un chunk compresso dal disco e decomprimerlo è solitamente minore del tempo necessario a trasferire lo stesso chunk non compresso. In questo capitolo verranno date delle statistiche di caratterizzazione delle distribuzioni dei terms e postings di grandi collezioni che vogliamo comprimere. Vedremo quindi come comprimere un dizionario, usando il dictionary-as-a-string method e il blocked storage.

4.1 Statistical properties of terms in information retrieval

In generale, le statistiche in tabella mostrano che il preprocessing inficia sulla dimensione del dizionario e ancora di più nel numero di postings non posizionali. Stemming e case folding riducono ognuno il numero di termini del 17% e il numero di non positional postings del 4% e del 3% rispettivamente. Risulta inoltre importante gestire le parole più frequenti. La regola dei 30 stati ci dice che le 30 parole più frequenti comprendono il 30% dei tokens scritti nel testo.

	(distinct) terms			nonpositional postings			tokens (= number of positi entries in postings)		
	number	Δ%	T%	number	Δ%	T%	number	Δ%	T%
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	−2	−2	100,680,242	−8	−8	179,158,204	−9	−9
case folding	391,523	−17	−19	96,969,056	−3	−12	179,158,204	−0	−9
30 stop words	391,493	−0	−19	83,390,443	−14	−24	121,857,825	−31	−38
150 stop words	391,373	−0	−19	67,001,847	−30	−39	94,516,599	−47	−52
stemming	322,383	−17	−33	63,812,300	−4	−42	94,516,599	−0	−52

Le tecniche di compressione che andremo a trattare saranno di tipo lossless, ossia, senza perdita di informazione. Un miglior rapporto di compressione si può ottenere con metodi lossy compression in cui vi è quindi perdita di informazione. Case folding, stemming e eliminazione delle stop words sono forme di lossy compression. Allo stesso modo lo sono metodi come space vector o latent semantic indexing. Prima di introdurre metodi per la compressione del dizionario vogliamo stimare il numero di termini distinti M di una collezione.

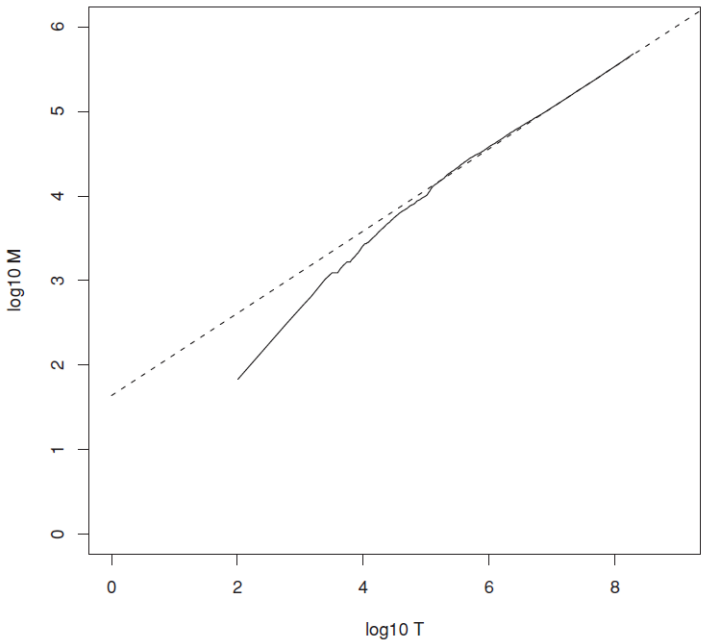
4.1.1 Heaps' law: Estimating the number of terms

Il modo migliore per stimare il valore di M è usando la legge di Heap, che stima la dimensione del dizionario come funzione della dimensione della collezione:

$$M = kT^b$$

Dove T è il numero di tokens nella collezione. Valori tipici per i parametri k e b sono: $30 \leq k \leq 100$ e $b \approx 0.5$. Il paramentro k è leggermente variabile in quanto dipende molto

dal tipo di collezione e da come questa viene processata.



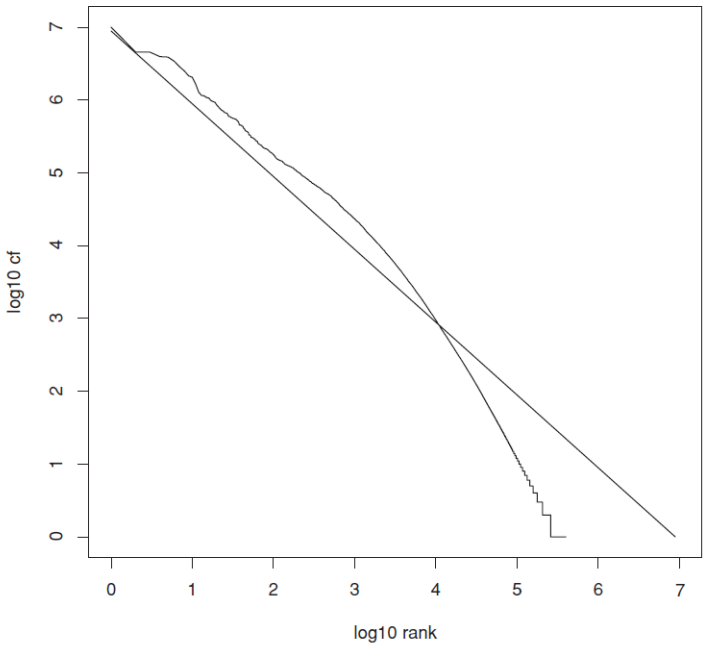
► **Figure 5.1** Heaps' law. Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1. For these data, the dashed line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least-squares fit. Thus, $k = 10^{1.64} \approx 44$ and $b = 0.49$.

4.1.2 Zipf's law: Modeling the distribution of terms

Vogliamo inoltre sapere come i termini siano distribuiti nei documenti. Un modello molto usato per la distribuzione dei termini in una collezione è la *Zipf's law*. Essa ci dice che, se t_1 è il termine più comune della collezione, t_2 è il prossimo più comune, e cosi via, allora in una collezione, la frequenza cf_i dell'i-esimo termine più comune è proporzionale a $1/i$:

$$cf_i \propto \frac{1}{i}$$

Quindi se il termine più comune ha cf_1 occorrenze, il secondo più comune ne avrà la metà, il terzo un terzo delle occorrenze e cosi via. L'intuizione è che la frequenza delle parole diminuisce molto velocemente con il rank. Possiamo scrivere la $Zipf's law$ come $cf_i = ci^k$ o come $log(cf_i) = log(c) + klog(i)$ con $k = -1$ e c costante. Risulta quindi una power law di esponente $k = -1$.



► **Figure 5.2** Zipf's law for Reuters-RCV1. Frequency is plotted as a function of frequency rank for the terms in the collection. The line is the distribution predicted by Zipf's law (weighted least-squares fit; intercept is 6.95).

4.2 Dictionary compression

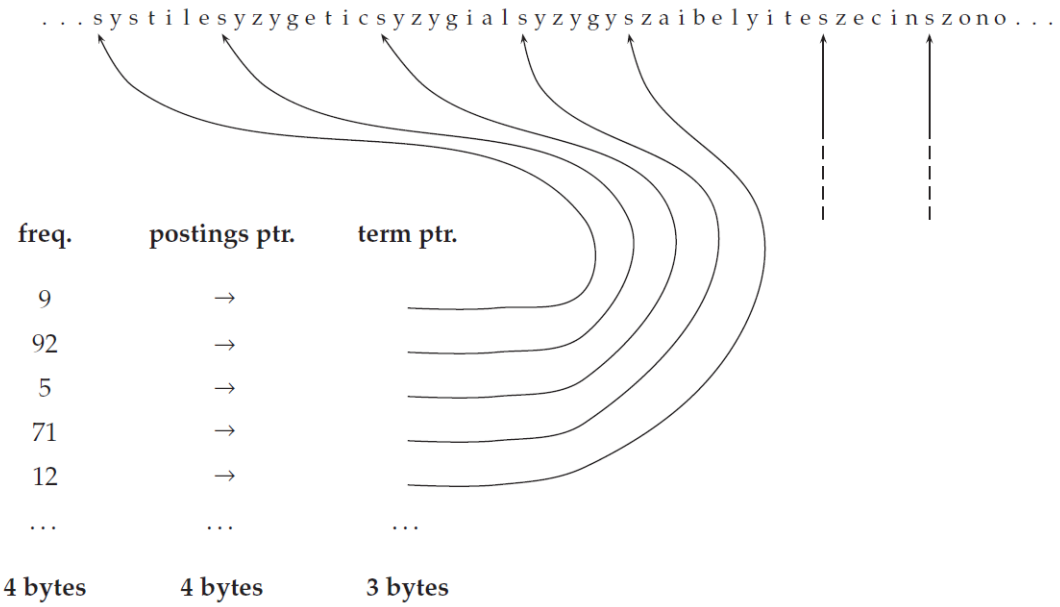
Questa sezione presenta una serie di strutture dati per i dizionari con indici di compressione crescenti. Perché comprimere il dizionario se questo occupa solo un piccola percentuale dello spazio occupato da un intero sistema IR? Uno dei primi fattori è dovuto al tempo necessario per processare una query, visto come il tempo di disk seeks necessari a processare la query. Se parti del dizionario sono memorizzate nel disco sara necessari molti più disk seeks. Inoltre se è vero che dizionari di grandi collezioni di documenti possono essere memorizzati nella memoria di un pc questo non è più vero se si parla di applicazioni differenti come ad esempio dispositivi portatili come i smartphones.

4.2.1 Dictionary as a string

La più semplice struttura dati per memorizzare un dizionario è quella di ordinare il dizionario in ordine lessicografico e memorizzarlo in un array con stringhe di lunghezza

fissa. Scegliamo di allocare 20byte per ogni termine, 4byte per memorizzare la document frequency e 4byte per il puntatore alla relativa postings list. Con questa struttura per memorizzare il dizionario di una collezioni di grandezza M saranno necessari $(20 + 4 + 4)M$ bytes. Nel caso di *Reuters-RCV1* si ha $28 \times 400,000 = 11.2MB$. Puntatori da 4byte generano 4GB di indirizzi, in grandi collezioni risulta necessario allocare più spazio per i puntatori.

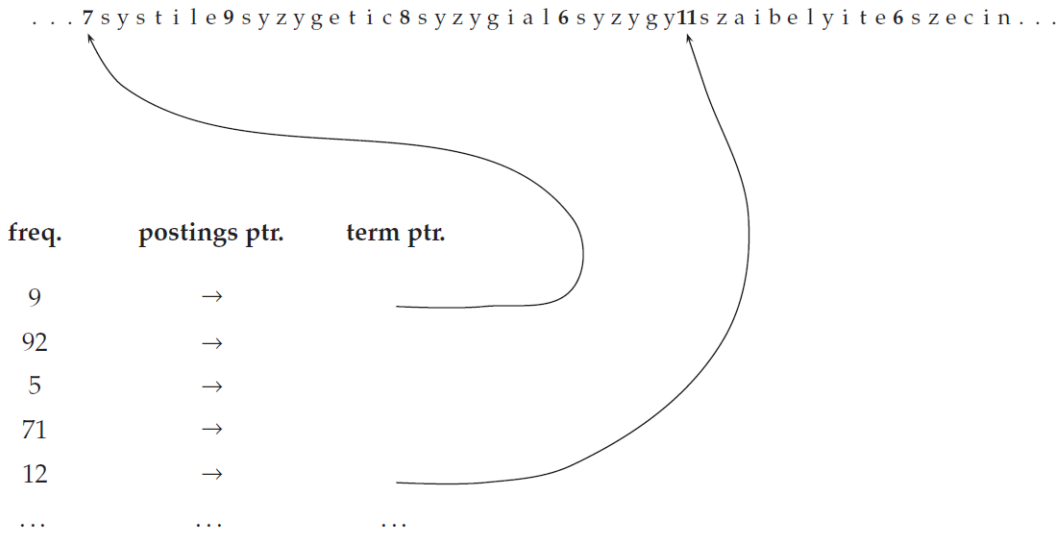
L'uso di stringhe a lunghezza fissa risulta chiaramente dispendioso. In media un term inglese è composto da 8 caratteri, quindi in media stiamo sprecando 12 caratteri. Inoltre in questo modo non si ha la possibilità di memorizzare terms con più di 20 caratteri. Possiamo risolvere questi problemi memorizzando i terms del dizionario come un'unica lunga stringa di caratteri. Il puntatore al prossimo term viene usato anche per marcare la fine del term corrente. Adesso però abbiamo bisogno di memorizzare i puntatori ai terms. nel caso di *Reuters-RCV1* si ha $400,000 \times 8 = 3.2 \times 10^6$ posizioni, quindi devono essere di $log_2(3.2 \times 10^6) \approx 22bits$ o $3bytes$. In questo nuovo schema avremo $400,000 \times (4 + 4 + 3 + 8) = 7.6MB$ (ricordando i 4 bytes per la tf e 4 per il puntatore alla posting list). Abbiamo quindi ridotto lo spazio necessario di un terzo, da 11.2 a $7.6MB$.



► **Figure 5.4** Dictionary-as-a-string storage. Pointers mark the end of the preceding term and the beginning of the next. For example, the first three terms in this example are systile, syzygetic, and syzygial.

4.2.2 Blocked storage

Possiamo operare una ancora maggiore compressione raggruppando i termini della stringa in blocchi di dimensione k salvando, per ogni blocco, il puntatore al primo elemento. Memorizziamo inoltre la lunghezza del term all'inizio di questultimo come byte addizionale. Eliminiamo cosi $k-1$ puntatori ai terms e aggiungiamo k byte per memorizzare la lunghezza di ogni termine. Ad esempio, con $k=4$, salviamo $(k-1)\times 3 = 9byte$ ma abbiamo bisogno di $k=4byte$ aggiuntivi per memorizzare la lunghezza dei terms. Aumentando la dimensione k dei blocchi avremo una compressione migliore. Tuttavia, è necessario avere un buon tradeoff tra compressione e velocità di recupero dei term.



► **Figure 5.5** Blocked storage with four terms per block. The first block consists of systile, syzygetic, syzygial, and syzygy with lengths of seven, nine, eight, and six characters, respectively. Each term is preceded by a byte encoding its length that indicates how many bytes to skip to reach subsequent terms.

Una fonte di ridondanza nel dizionario che non abbiamo ancora sfruttato è il fatto che le voci consecutive in un elenco in ordine alfabetico condividano alcuni prefissi. I prefissi in comune sono identificati come sottosequenze dei terms e vengono riferiti usando caratteri speciali.

One block in blocked compression ($k = 4$) ...
8automata**8**automate**9**automatic**10**automation

⇓

...further compressed with front coding.
8automat∗a**1**⋄e**2** ⋄ ic**3**⋄ion

► **Figure 5.7** Front coding. A sequence of terms with identical prefix ("automat") is encoded by marking the end of the prefix with ∗ and replacing it with ⋄ in subsequent terms. As before, the first byte of each entry encodes the number of characters.

Altri schemi per una migliore compressione usano funzioni di hashing minimali, che mappano M termini in $[1, ..., M]$ senza collisioni. Tuttavia, non è possibile inserire nuovi termini perchè questi causerebbero collisioni, è necessario quindi per ogni nuova aggiunta creare da zero una nuova funzione di hash. Quindi risulta impossibile usare questo metodo per ambienti dinamici.

► **Table 5.2** Dictionary compression for Reuters-RCV1.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
∼, with blocking, $k = 4$	7.1
∼, with blocking & front coding	5.9

4.3 Postings file compression

Per escogitare una rappresentazione migliore per i postings file, osserviamo che i postings di termini molto frequenti sono molto vicini tra loro. L'idea chiave è che i gap tra i postings sono piccoli e richiedono quindi poco spazio per essere memorizzati. Infatti i gap di termini frequenti come "the" e "for" sono solitamente 1. Al contrario i gap di parole che compaiono poche volte nella collezione possono essere prossimi ai DocID che le contengono. Per una rappresentazione parsimoniosa della distribuzione di questi gaps, abbiamo bisogno di un metodo di encoding variablile che usi pochi bit per gaps piccoli. Per codificare piccoli numeri in meno spazio dei grandi numeri, mostriamo 2 metodi: *bytewise compression* e *bitwise compression*.

4.4 Variable byte codes

Variable byte (VB) encoding usa un numero intero di byte per codificare un gap. Gli ultimi 7 bits del byte sono il "payload" e codificano parte del gap. Il primo bit del byte è un continuation bit. Viene impostato a 1 per denotare l'ultimo byte della codifica del gap, 0 altrimenti. Per decodificare un variable byte code, si leggono tutti i byte consecutivi con continuation bit pari a 0 e l'ultimo byte con continuation bit pari a 1. Quindi si

estraggono e si concatenano le varie parti da 7 bits. L'idea del VB encoding può essere applicata anche a unità diverse: parole da 32-bits, 16-bits o 4-bits. Per la maggior parte dei sistemi IR il variable byte codes offre un eccellente trade-off tra tempo e spazio.

docIDs	824		829	215406	
gaps			5	214577	
VB code	00000110	10111000	10000101	00001101 00001100	10110001

4.5 γ codes

Il VB codes usa un numero variabile di byte in base alla dimensione del gap. I bit-level codes adattano la lunghezza della codifica a livello dei bits. La più semplice codifica a livello bit è la codifica unaria. Assumendo che ci siano $G = 2^n$ gaps, con $1 \leq G \leq 2^n$ tutti ugualmente probabili. La codifica ottima usa n bits per ogni G. Alcuni gap non potranno essere codificati con meno di $log_2 G$ bits. Un metodo che non si discosta molto dall'ottimo è la codifica γ. La codifica γ implementa una codifica a lunghezza variabile dividendo la rappresentazione di un gap in due, lunghezza e offset. L'offset viene rappresentato in binario ma eliminando l'uno più significativo. La lunghezza rappresenta la lunghezza dell'offset e viene codificata in unario. Per il numero 13, la lunghezza dell'offset è 3, in unario 1110, il γ code di 13 è quindi 1110101, la concatenazione di lunghezza 1110 e dell'offset 101. Per decodificare un γ code si legge il codice unario fino allo 0 poi si leggono tanti bit quanta la lunghezza decodificata in unario e si aggiunge un uno all'inizio della sequenza. La lunghezza dell'offset è $log_2 G$ e la lunghezza della lunghezza è $log_2 G + 1$. La codifica γ ha due proprietà molto utili, è *prefix free*, ossia nessun γ code è prefisso di un altro. Questo vuol dire che c'è sempre una decodifica unica per un codice γ. Inoltre la codifica γ non ha bisogno di parametri.

5 Scoring, term weighting and the vector space model

5.1 slides

Per generare un ranking per ogni coppia query-documento si può assegnare uno score per ogni coppia, ad esempio in [0,1]. Lo score misura quanto una query è simile ad un documento, a questo punto basta ordinare i documenti secondo il valore di score. Nel calcolare gli score vorremmo che se un termine non compare in un documento quel documento abbia score uguale a 0 rispetto al termine. Inoltre vorremmo che più un termine di una query sia presente in un documento più sia alto lo score relativo. Mentre, al contrario, meno un termine è frequente in un documento più basso sarà lo score. Un modo per calcolare lo score può essere quello di vedere query e documenti come insiemi di termini e calcolare il *jaccard coefficient*. Siano A e B due insiemi:

$$jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

con $A \neq 0$ o $B \neq 0$ $jaccard(A, A) = 1$

$jaccard(A.B) = 0$ se $A \cap B = 0$

viene sempre assegnato un valore compreso tra 0 e 1.

Dato che i termini rari sono più informativi rispetto ai temrini comuni risulta necessario usare un modo più sofisticato per calcolare gli score. Usando ad esempio la *cosine similarity*.

$$cosine(A, B) = \frac{|A \cap B|}{\sqrt{|A \cup B|}}$$

5.2 Term frequency and weighting

Un documento o una zona (parte di un documento) che contiene piu volte un termine della query ha più a che fare con la query stessa e perciò bisognerebbe assegnargli uno score più alto. Se consideriamo query libere e quindi solo testuali (non booleane) queste verranno viste solo come un insieme di termini. Un meccanismo di scoring plausibile è quello di calcolare lo score come la somma per ogni termine della query dello score dei suoi

termini con il documento. Vogliamo quindi calcolare lo score per un termine t della query e un documento d. Il modo più semplice risulta essere quello di assegnare un peso che sia uguale al numero di occorrenze di t in d. In questa visione del documento, conosciuta in letteratura come *bag of words*, viene ignorato l'ordine delle parole nel documento mentre viene preservata la loro frequenza.

5.2.1 Inverse document frequency

La term frequency cruda soffre di un grave problema: tutti i termini vengono considerati di uguale importanza nel dare rilevanza alla query. Nella realtà alcuni termini hanno una piccola rilevanza o a volte quasi nulla. Per questo si introduce un meccanismo con il quale si attenua l'effetto dei termini che occorrono troppo spesso nella collezione. Una prima idea è quella di scalare la term frequency con la *collection frequency*, la frequenza del termine in tutta la collezione. Piuttosto che usare la collection frequency quello che si fa è usare la *document frequency* df_t, definita come il numero di documenti della collezione che contengono il termine t. Questo perché nel cercare di discriminare i documenti per assegnare uno scoring risulta migliore usare una statistica a livello documento piuttosto che a livello dell'intera collezione. Come viene usato il document frequency df di un termine t per scalare il suo peso? Dato N il numero totale dei documenti nella collezione, denotiamo la *inverse document frequency* idf come segue:

$$idf_t = log\frac{N}{df_t}$$

Quindi, il valore idf di un termine raro è alto mentre il valore idf di un termine comune risulta più basso.

5.2.2 Tf-idf weighting

Combiniamo quindi le definizioni di tf e di idf per produrre una pesatura adeguata per ogni termine di ogni documento. La pesatura *tf-idf* assegna al termine t un peso nel documento d dato da:

$$tf\text{-}idf_{t,d} = tf_{t,d} \cdot idf_t$$

in altre parole *tf-idf* assegna un peso al termine t che è:

- più alto quando t occorre molte volte in un numero limitato di documenti
- più basso quando t occorre poce volte nel documento, o occorre in molti documenti
- ancora più basso quando il termine t occorre virtualmente in tutti i documenti.

A questo punto possiamo vedere ogni documento come un vettore nel quale ogni componente corrisponde ad un termine del dizionario. Per i termini del dizionario che non hanno occorrenze nel documento il peso sarà 0. Possiamo rifinire l'idea iniziale per calcolare lo scoring di una query piuttosto che sommando il numero di occorrenze di ogni termine della query nel document, con i valori *tf-idf* associati.

$$score(q, d) = \sum_{t \in q} tf\text{-}idf_{t,d}$$

5.3 The vector space model for scoring

La rappresentazione di documenti come vettori di uno spazio vettoriale è conosciuta come *vector space model* e risulta fondamentale per operazioni di information retrival che vanno dallo scoring alla classificazione dei documenti e al clustering di questi. Il punto centrale in questo tipo di sviluppo è considerare la query esattamente come un documento dello stesso spazio vettoriale dei documenti della collezione.

5.3.1 Dot products

Denotiamo con $V(d)$ il vettore derivato dal documento d, con una componente per ogni termine nel dizionario dei termini. L'insieme dei documenti della collezione può quindi essere vista come un insieme di vettori in uno spazio vettoriale, nel quale compare un asse per ogni ternime. Come gia detto, questa rappresentazione perde informazioni riguardo l'ordine dei termini nei documenti. Come si può calcolare la similarità tra due documenti in questo spazio vettoriale? Un primo approccio può essere quello di considerare la distanza euclidea tra i due vettori documento (ricordando che possiamo rappresentare i due vettori in uno spazio n-dimensionale). Questa misura però presenta uno svantaggio in quanto documenti con contenuti molto simili possono risultare molto diversi solo perchè

uno può essere molto più lungo dell'altro (esempio: d1 = "Hello world!" e d2 = "Hello world!Hello world!Hello world!" anche se i due documenti dicono la stessa identica cosa, poiché il secondo è più lungo allora la distanza euclidea sarà molto grande). Il modo comune per compensare la lunghezza dei vettori, è quello di calcolare la similarità tra due documenti d_1 e d_2 usando la cosine similarity tra i due vettori $V(d_1)$, $V(d_2)$.

$$sim(d_1, d_2) = \frac{V(d_1) \cdot V(d_2)}{|V(d_1)||V(d_2)|}$$

dove il numeratore è il *dot product* (prodotto scalare) tra i vettori $V(d_1)$ e $V(d_2)$, mentre il denominatore è il prodotto delle loro lunghezze euclidiane. Il dot product $x \cdot y$ di due vettori è definito come $\sum_{i=1}^{M} x_i y_i$, cioè il prodotto delle loro componenti. Sia $V(d)$ il vettore del documento d con M componente, $V_1(d) \dots V_M(d)$. La lunghezza euclidiana di d è definita come $\sqrt{\sum_{i=1}^{M} V_i^2(d)}$. L'effetto del denominatore nella formula $sim(d_1, d_2)$ è quello di normalizzare le lunghezze dei vettori $V(d_1), V(d_2)$

5.4 Variant tf-idf functions

Per assegnare pesi per ogni termine documento vengono usate molte alternative al *tf* o *tf-idf*. Ne esistono varie:

1. per il TF: *natural*, *logaritm*, *augmentend*, *boolean*, *log-ave*
2. per il DF: *none*, *idf*, *prob idf*
3. per la Normalizzazione: *nessuna*, *cosine*, *pivoted unique*, *byte size*

5.4.1 Sublinear tf scaling (logarithm)

Sembra improbabile che un documento nel quale compare 20 volte un termine esprima venti volte di piu il significato (di quel termine) di un documento nel quale quel termine compare una sola volta. In accordo con questa osservazione sono state proposte alcune varianti al calcolo della term frequency. Una modifica comune è quella di fare il logaritmo della *term frequency* assegnando un peso dato da

$$wf_{t,d} = \begin{cases} 1 + log(tf_{t,d}) \ se \ tf_{t,d} > 0 \\ 0 \ altrimenti \end{cases}$$

Quando $tf_{t,d} = 1$ il risultato del logaritmo è 0. Per rimediare a questa cosa si è pensato di aggiungere 1 in modo da non avere numeri nulli.

6 Probabilistic information retrieval

Se avessimo una qualche conoscienza su quali documenti siano rilevanti e quali non lo siano per una certa query, potremmo stimare la probabilità di un termine t di apparire in un documento rilevante $P(t|R = 1)$. Esistono molti modelli di retrival con una base probabilistica. Introdurremo la teoria della probabilità quindi il *probability ranking principle*, ci concentreremo poi sul *binary indipendence model* tra i primi e piu usati modelli di ranking probabilistico. Quindi introdurremmo versioni estese che usano il conteggio dei termini come il *Okapi BM25* e *bayesian network model for IR*.

6.1 Review of basic probability theory

VEDILA SUL LIBRO

6.2 The Probability Ranking Principle

6.2.1 The 1/0 loss case

Assumiamo di avere un sistema di retrival in cui sia presente una collezione di documenti, si abbia una certa query dell'utente e venga restituita una lista di documenti ordinati. Per una query q e un documento d sia $R_{d,q}$ la viariabile aleatoria che indica se il documento d è rilevante per la query q. Usando un modello di ranking probabilistico i documenti restituiti all'utente vengono ordinati secondo la loro probabilità stimata di rilevanza rispetto alle query $P(R = 1|d, q)$. Questà è la base per il *probability ranking principle (prp)*. Nel caso semplice del PRP non ci sono costi di recupero ne altri tipi di interessi nell'utilità. Si perde un punto per ogni documento rilevante che non viene restituito e per ogni documento non

rilevante che viene restituito. Il goal è restituire i risultati milgiori possibili come i primi k documenti, per un generico valore di k. La PRP ci dice quindi, semplicememnte, di ordinare i documenti in ordine decrescente per valore di $P(R = 1|d, q)$. Se deve essere restituito un insieme di documenti, senza considerare l'ordine, la *bayes optimal decision rule*, la decisione che minimizza il rischio di perdita, è quella di restituire i documenti che sono più rilevanti che non rilevanti:

$$d \text{ è } rilevante \text{ } sse \text{ } P(R = 1|d, q) > P(R = 0|d, q)$$

6.2.2 The PRP with retrieval costs

Supponiamo ora un sistema di retrival che usa costi. Sia C_1 il costo nel non recuperare un documento rilevante e C_0 il costo di recupero di un documento non rilevante. Quindi se per uno specifico documento d e per tutti gli altri documenti $d^{'}$ non ancora recuperati vale:

$$C_0 \cdot P(R = 0|d) - C_1 \cdot P(R = 1|d) \leq C_0 \cdot P(R = 0|d') - C_1 \cdot P(R = 1|d')$$

allora d sarà il prossimo documento restituito secondo il PRP.

6.3 The Binary Independence Model

Il *Binary Indipendence Model* viene generalemente usato con il PRP. Questo introduce delle semplici assunzioni che permettono di stimare la probabilità $P(R|d, q)$ in modo pratico. Documenti e query vengono rapprensentati entrambi come vettori di incidenza dei termini. Ossia, un documento d è rappresentato da un vettore $x = (x_1 \dots x_M)$ dove $x_t = 1$ se il termine t comprare nel documento d e $x_t = 0$ se il termine t non compare nel documento d. Per rendere precisa la strategia di recupero probabilistico abbiamo la necessità di stimare come i termini nei documenti contribuiscano alla rilevanza, vorremmo sapere come la term frequency, la document frequency, la document length e altre statistiche influenzino la rilevanza dei documenti, e come queste possano essere combinate per stimare la rilevanza dei documenti. Ordianiamo quindi i documenti per il valore decrescente di rilevanza stimata. Assumiamo che la rilevanza di un documento sia indipendente dalla

rilevanza degli altri documenti. Nel BIM modellizziamo la probabilità $P(R|d,q)$ di rilevanza di un documento secondo il vettore di incidenza dei termini $P(R|x,q)$. Quindi usando la regola di Bayes abbiamo:

$$P(R=1|x,q) = \frac{P(x|R=1,q)P(R=1|q)}{P(x|q)}$$

$$P(R=0|x,q) = \frac{P(x|R=0,q)P(R=0|q)}{P(x|q)}$$

$P(x|R=1,q)$ e $P(x|R=0,q)$ sono le probabilità che un documento rispettivamente rilevante e non rilevante recuperato abbia come rappresentazione il vettore x. $P(R=1|q)$ e $P(R=0|q)$ indicano le probabilità a priori di recuperare un documento rilevante o non rilevante, rispettiavamente, per la query q.

6.3.1 Deriving a ranking function for query terms

Data una query q vorremmo ordinare i documenti recuperati secondo valori decrescenti della $P(R=1|d,q)$. Nel BIM questo viene modellizzato utilizzando la $P(R=1|x,q)$. Invece di calcolare questa quantità direttamente usiamo un'altra quantità piu semplice da calcolare. Possiamo ordinare i documenti secondo gli odds(rapporto in questo caso) della rilevanza (gli odds della rilevanza sono monotonici rispetto alla probabilità di rilevanza). Questo semplifica le cose in quanto possiamo eliminare i denominatori comuni:

$$Odds(R|x,q) = \frac{P(R=1|x,q)}{P(R=0|x,q)} = \frac{\frac{P(R=1|q)P(x|R=1,q)}{P(x,q)}}{\frac{P(R=0|q)P(x|R=0,q)}{P(x,q)}} = \frac{P(R=1|q)}{P(R=0|q)} \cdot \frac{P(x|R=1,q)}{P(x|R=0,q)}$$

Il fattore a sinistra dell'equazione piu a destra è una costante rispetto alla query (poiché è il rapporto tra le probabilità a priori) e quindi non è necessario che sia stimato. Invece bisogna stimare il secondo fattore, e non risulta semplice. Per stimare la probabilita di occorrenza di un vettore termine di incidenza possiamo usare la *Naive Bayes conditional independence assumption* per la quale la presenza o l'assenza di una parola in un documento è indipendente dalla prensenza o assenza di una qualiasi altra parola.

$$\frac{P(x|R=1,q)}{P(x|R=0,q)} = \prod_{t=1}^{M} \frac{P(x_t|R=1,q)}{P(x_t|R=0,q)}$$

allora:

$$Odds(R|x,q) = O(R|q) \cdot \prod_{t=1}^{M} \frac{P(x_t|R=1,q)}{P(x_t|R=0,q)}$$

dato che ogni x_t è 0 o 1, possiamo separare i termini ottenendo:

$$Odds(R|x,q) = O(R|q) \cdot \prod_{t:x_t=1} \frac{P(x_t=1|R=1,q)}{P(x_t=1|R=0,q)} \cdot \prod_{t:x_t=0} \frac{P(x_t=0|R=1,q)}{P(x_t=0|R=0,q)}$$

Sia $p_t = P(x_t = 1|R = 1, q)$ la probabilità di un termine di apparire in un documento rilevante per la query, e $u_t = P(x = 1|R = 0, q)$ la probabilità di un termine di apparire in un documento non rilevante. Questi valori possono essere visualizzati nella seguente tabella di contigenza con le colonne che sommano ad uno.

	document	relevant ($R = 1$)	nonrelevant ($R = 0$)
Term present	$x_t = 1$	p_t	u_t
Term absent	$x_t = 0$	$1 - p_t$	$1 - u_t$

Aggiungiamo ora un ulteriore assunzione di semplificazione, la probabilità di occorrenza dei termini che non occorrono nella query in documenti rilevanti e non rilevanti è uguale, ossia, se $q_t = 0$ allora $p_t = u_t$. Dobbiamo quindi considerare ora solo i termini che appaiono nella query:

$$Odds(R|x,q) = O(R|q) \cdot \prod_{t:x_t=q_t=1} \frac{p_t}{u_t} \cdot \prod_{t:x_t=0,q_t=1} \frac{1-p_t}{1-u_t}$$

Il primo prodotto è sui termini trovati nei documenti mentre il secondo prodotto è sui termini non trovati nei documenti. Possiamo manipolare questa espressione includendo nel secondo prodotto i termini della query trovati nei documenti, dividendo allo stesso tempo per i stessi termini nel primo prodotto, cosi da non alterare il valore. Quindi abbiamo:

$$Odds(R|x,q) = O(R|q) \cdot \prod_{t:x_t=q_t=1} \frac{p_t(1-u_t)}{u_t(1-p_t)} \cdot \prod_{t:q_t=1} \frac{1-p_t}{1-u_t}$$

La produttoria sinistra si riferisce ancora ai termini della query trovati nei docuementi, mentre la produttoria destra si riferisce ora a tutti i termini della query. Questo significa che la produttoria destra è costante per una certa query. Allora l'unica quantità che bisogna calcolare per elaborare un rank dei documenti è la produttoria di sinistra. Possiamo ordinare allo stesso modo i documenti usando il logaritmo, dato che questa è una funzione monotona. La quantità risultante usata per il ranking viene chiamata *Retrieval Status Value* (RSV):

$$RSV_d = log \prod_{t:x_t=q_t=1} \frac{p_t(1-u_t)}{u_t(1-p_t)} = \sum_{t:x_t=q_t=1} log \frac{p_t(1-u_t)}{u_t(1-p_t)}$$

quindi tutto si riduce al calcolo del RSV. Sia c_t:

$$c_t = log \frac{p_t(1-u_t)}{u_t(1-p_t)} = log \frac{p_t}{(1-p_t)} + \ log \frac{(1-u_t)}{u_t}$$

Il termine c_t è un rapporto odds logaritmico per i termini di una query. Abbiamo gli odds di un termine che appare se un documento è rilevante $(p_t/(1-p_t))$ e gli odds di un termine che appare se un documento non è rilevante $((1-u_t)/u_t)$. Il valore di c_t è 0 se ha gli stessi odds(probabilità) di apparire in un documento rilevante o non rilevante, e ha un valore positivo se ha odds (probabilità) maggiore di apparire in un documento rilevante pittosto che non rilevante. Le quantità c_t sono funzioni peso per i termini e lo score di un documento per una query è

$$RSV_d = \sum_{x_t=q_t=1} c_t$$

Proviamo ora a dare una stima dei c_t per una data collezione di documenti e una query.

6.3.2 Probability estimates in theory

Per ogni termine t, come cambierebbe c_t per l'intera collezione? La figura sottostante è una tabella di contingenza dei documenti nella collezione, dove df_t è il numero di documenti che contengono il termine t.

	documents	relevant	nonrelevant	Total
Term present	$x_t = 1$	s	$\mathrm{df}_t - s$	df_t
Term absent	$x_t = 0$	$S - s$	$(N - \mathrm{df}_t) - (S - s)$	$N - \mathrm{df}_t$
	Total	S	$N - S$	N

Allora sostituendo $p_t = s/S$ e $e_t = (df_t - s)/(N - S)$ abbiamo:

$$c_t = K(N, df_t, s, S) = log \frac{s/(S-s)}{(df_t - s)/((N - df_t) - (S - s))}$$

Per evitare valori nulli si applica uno smoothing particolare: aggiungiamo 4 volte 1/2 come segue(in questo modo la tabella sopra avrà N+2 invece di N documenti totali):

$$c_t = K(N, df_t, s, S) = log \frac{(s + \frac{1}{2})/(S - s + \frac{1}{2})}{(df_t - s + \frac{1}{2})/(N - df_t - S + s + \frac{1}{2})}$$

Per eventi categorici (presenza/assenza) si può stimare la probabilità di un evento dai dati semplicemente contando il numero di volte di quell'evento diviso il numero totale. Questa è la *frequenza relativa* da cui deriva la *maximum likelihood estimate* (MLE). Tuttavia usando semplicemente la MLE diamo maggiore importanza a eventi accaduti in passato e troppo poca a eventi mai visti, o comunque un numero talmente vicino allo 0 da "rompere" il modello (qualsiasi numero moltiplicato per 0 è 0). Perciò diminuire la probabilità degli eventi visti e aumentarla per quelli poco frequenti si chiama *smoothing*, usando solitamente un parametro α (aggiunto o sottratto) ad ogni evento. Ciò corrisponde all'uso di una distribuzione uniforme sul vocabolario nota come **Bayesian prior**. Poiché nel nostro caso crediamo poco a questa "uniformità" usiamo $\alpha = \frac{1}{2}$.
Questa è una forma di stima **maximum a posteriori** (MAP) in cui scegliamo il punto più probabile per le probabilità basandoci sulla prior e sull'evidenza.

Se assumiamo che il numero di documenti rilevanti sia molto minore rispetto al numero di documenti non rilevanti per una certa query possiamo fare delle ulteriori semplificazioni. Infatti u_t (la probabilità che un termine della query compaia in un documento non rilevante) è approssimabile a df_t/N. Quindi:

$$log \frac{1 - u_t}{u_t} = log \frac{N - df_t}{df_t} \approx log \frac{N}{df_t}$$

Allora:

$$c_t = log\frac{p_t(1-u_t)}{u_t(1-p_t)} \approx log\frac{p_t}{(1-p_t)} + log\frac{N}{df_t}$$

In oltre non avendo informazioni riguardo la rilevanza dei termini possiamo assumere che p_t sia costante su tutti i termini e che sia $p_t = 0.5$. Fatta questa assunzione, un termine ha la stessa probabilità di occorrere e non occorrere in un documento rilevante e per questo i fattori p_t e $(1-p_t)$ vegnono cancellati nell'espressione per l'RSV. Combinando questo con le approssimazioni fatte per il fattore u_t il ranking di un documento si riduce a determinare i termini del documento presenti nella query scalati per il valore di idf.

$$RSV_d = \sum_{t:x_t=q_t=1} log\frac{p_t(1-u_t)}{u_t(1-p_t)} \approx \sum_{t:x_t=q_t=1} log\frac{N}{df_t}$$

Ciò risulta efficace per documenti corti come titoli o abstract.

6.3.3 Okapi BM25

Okapi BM25 è un modello probabilistico molto usato e robusto nel retrival che affronta il problema incontrato dal BIM per i documenti lunghi. Infatti questo incorpora la term frequency e la length normalization. Come visto precedentemente con il BIM un semplice score per un documento d è la pesatura idf dei termini del documento che sono presenti nella query.

$$RSV_d = \sum_{t\in q} log\frac{N}{df_t}$$

A questo si può aggiungere un fattore moltiplicativo che tenga conto della term frequency.

$$RSV_d = \sum_{t\in q} \frac{(k_1+1)tf_{td}}{k_1+tf_{td}} log\frac{N}{df_t}$$

dove:

- k_1 è un paramentro di tuning che controlla lo scaling della term frequency.
- (k_1+1) è un fattore che rende lo score $= 1$ nel caso in cui $tf_{td} = 1$.

Ciò risulta simile al tf-idf ma in più è pesato. k_1 risulta essere un parametro di quanto un singolo termine può inficiare nello score di uno stesso documento, può quindi essere visto come un asintoto.

Termini presenti in documenti molto lunghi potranno avere generalmente alti valori di tf_td, questo può essere causato da documenti verbosi o da documenti che effettivamente sono molto vicini a quei termini. Per questo è necessario inserire un fattore di normalizzazione parziale.

$$B = (1 - b) + b\frac{L_d}{L_{ave}}$$

dove:

- L_d è la lunghezza del documento $L_d = \sum_{t \in d} tf_{td}$
- L_{ave} è la lunghezza media di un documento.
- $0 \leq b \leq 1$
- con $b = 1$ si ha una completa normalizzazione
- con $b = 0$ non si ha normalizzazione

Possiamo quindi inserire questo fattore nel calcolo del RSV di un documento.

$$RSV_d = \sum_{t \in q} \frac{(k_1 + 1)tf_{td}}{k_1((1 - b) + b\frac{L_d}{L_{ave}}) + tf_{td}} log\frac{N}{df_t}$$

- tf_{td} è la term frequency del termine t nel documento d
- L_d è la lunghezza del documento d
- L_ave è la lunghezza media di un documento all'interno dell'intera collezione.
- k_1 è un paramentro di tuning che controlla lo scaling del valore di tf_{td}
- b è un paramentro di tuning che controlla lo scaling della normalizzazione della lughezza del documento.

Nel caso in cui si abbia a che fare con query molto lunghe è opportuno scalare anche sui termini della query.

$$RSV_d = \sum_{t \in q} \frac{(k_1 + 1)tf_{td}}{k_1((1-b) + b\frac{L_d}{L_{ave}}) + tf_{td}} \frac{(k_3 + 1)tf_{tq}}{k_3 + tf_{tq}} log \frac{N}{df_t}$$

Dove:

- tf_{tq} è la term freqeuncy del termine t nella query q.
- k_3 è un paramentro di tuning che conrolla lo scaling della term frequency dei termini della query.
- Non si ha normalizzazione sulla lunghezza della query (poichè si ha una singola query)

7 Evaluation in information retrieval

7.0.1 Information retrieval system evaluation

Per misurare un sistema di information retrival ad hoc è necessario una collezione di test che consista di 3 parti:

- Una collezione di documenti
- Un insieme di information needs, esprimibili come query
- Un insieme di giudizi di rilevanza, normalmente binari

L'approccio per la valutazione di sistemi di information retrival giace nella nozione di rilevanza e non rilevanza dei documenti. I documenti nell'insieme di test sono accompagnati da un valore binario di rilevanza o non rilevanza. Queste decisioni sono chiamate *gold standard*. La rilevanza è relativa ad un information need, non alla query. Da cio dipende che un docuemento è rilevante per una certa query se risponde all'information need non solo se contiene tutte le parole dell query.

7.1 Evaluation of unranked retrieval sets

Le due misure più frequenti per l'efficacia di un sistema di IR sono la precision e la recall. Verranno prima definite nel caso semplice nel quale il sistema restituisce un insieme di documenti, poi verranno definite per situazioni di ranking.

La precision (P) è la frazione dei documenti recuperati che sono rilevanti

$$Precision = \frac{(items\ rilevanti\ recuperati)}{(items\ recuperati)}$$

La recall (R) è la frazione dei documenti rilevanti recuperati

$$Recall = \frac{(items\ rilevanti\ recuperati)}{(items\ rilevanti)}$$

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

quindi:

$$P = \frac{tp}{(tp + fp)}$$

$$R = \frac{tp}{(tp + fn)}$$

Una alternativa per giudicare un sistema di IR potrebbe essere l'uso della accuracy, la frazione delle classificazioni corrette. Accuracy = (tp+tn)/(tp+fp+fn+tn). Questo puo risultare plausibile se si hanno solo 2 classi, rilevante o non rilevante. L'accuracy però non risulta essere un buon sistema di valuazione per i sistemi di IR in quanto, ad esempio, i dati sono spesso molto sbilanciati, normalmente piu del 99.9% dei documenti sono non rilevanti, Un sistema che vuole massimizzare l'accuracy potrebbe quindi semplicemente ritenere tutti i documenti non rilevanti per tutte le query.

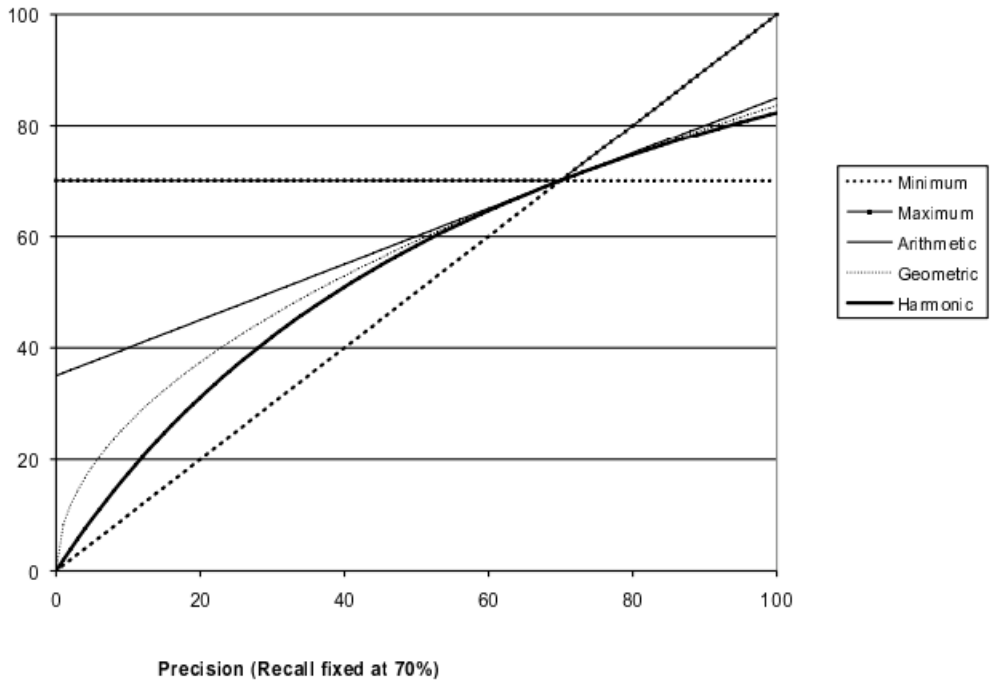
Una misura che da un buon trade-off tra precision e recall è la F *Measure* che è la media armonica pesata tra precision e recall:

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R} \quad \text{dove} \quad \beta^2 = \frac{1-\alpha}{\alpha}$$

Qui $\alpha \in [0, 1]$ e di conseguenza $\beta^2 \in [0, \infty]$. La *balanced F Measure* di default pesa ugualmente recall e precision e pone quindi $\alpha = 1/2$ o $\beta = 1$ e viene comunemente chiamata F_1 (o F1 score). Usando quindi $\beta = 1$ la formula diventa:

$$F_1 = \frac{2PR}{P + R}$$

Per valori di $\beta < 1$ si enfatizza la precisione, per valori di$\beta > 1$ si enfatizza la recall. Il motivo per il quale si usa la media armonica piuttosto che la media aritmetica (del pollo) è che, usando la media aritmetica, nel caso in cui si restituiscano tutti i documenti si avrebbe una recall del 100% e una media del 50%. Usando invece la media armonica, nel caso in cui ci sia 1 solo documento rilevante su 10,000 per la query la media sarebbe 0.02%. La media armonica è sempre minore o uguale alla media aritmetica e quella geometrica.



7.2 Rank-Based Measures

Distinguiamo due tipi di misure per la rilevanza con vari algoritmi

Rilevanza binaria

- Precision@K (P@K)

- Mean Average Precision (MAP)
- Mean Reciprocal Rank (MRR)

Rilevanza multi-livello

- Normalized Discounted Cumulative Gain (NDCG)

7.2.1 Precision@K

L'idea è quella di calcolare la precisione fino ad un certo punto, piuttosto che su tutti i documenti che il nostro sistema di retrieval ha recuperato. Quando un utente fa una query si aspetta che i primi 10 risultati, se non i primi 2/3, siano quelli più rilevanti che a lui interessano. Per questo motivo non andrà mai a cercare un documento alla 14-esima pagina (supponendo di formattare le pagine in modo da contenere 10 risultati l'una). L'algoritmo è alquanto semplice:

1. Scegli una soglia K
2. Calcola la precision fino a K
3. Ignora tutti gli altri documenti dopo K

Esempio

$P@3 = 2/3$

$P@4 = 3/4$

$P@7 = 5/7$

Similmente possiamo calcolare la Recall@K

TODO → scrivere interpolazione (slides 18-23 croce)

7.2.2 Mean Average Precision

Si tratta di calcolare la Precision@K di vari rankings/queries e poi calcolarne la media.

1. Considera l'ordinamento SOLO dei documenti rilevanti $K_1, K_2, .., K_R$
2. Calcola la P@K per ogni $K_1, K_2, .., K_R$
3. Average precision = media dei P@K

TODO → mettere immagine esempio (slides 25 e 26 croce)

7.2.3 Discounted Cumulative Gain

E' una misura molto popolare per valutare motori di ricerca web e i tasks relativi che enfatizza il recupero di documenti molto rilevanti. Tuttavia si fanno due grandi assunzioni: (i) i documenti veramente rilevanti sono più "utili", o potremmo dire soddisfano di più l'user need, di quelli marginali; (ii) più un documento si trova in basso meno è "utile" all'utente in quanto non verrà mai visualizzato. Questo perché l'utente medio è pigro e vuole vedere per primi i documenti che gli interessano.
La DCG usa la rilevanza "graduale" come misura, o il cosidetto guadagno (gain), nell'esaminare un documento. Il guadagno si accumula partendo dall'inizio del ranking e si riduce, a volte in maniera "scontata" (discounted) scendendo nella graduatoria, tipicamente di $1/log(rank)$
Quando la rilevanza $\in [0, r]$ con $r > 2$ la DCG a $1 \leq n \leq |docs|$ (la base del logaritmo arbitraria) è:

$$DCG = r_1 + \frac{r_2}{log_2(2)} + \frac{r_3}{log_2(3)} + .. + \frac{r_n}{log_2(n)}$$

o in forma più compatta

$$DCG_n = rel_1 + \sum_{i=2}^{n} \frac{rel_i}{log_2 i} = \sum_{i=1}^{n} \frac{2^{rel_i} - 1}{log(1+i)}$$

TODO → normalized slides 35 e seguenti

7.3 Evaluation of ranked retrieval results

Precision, Recall e Fmeasure sono misure calcolate su insiemi di documenti non ordinati. In contesti di IR l'insieme dei documenti rilevanti viene fornito come i k documenti più rilevanti,per ognuno di questi insiemi è possibile calcolare Precision e Recall ed ottenere

quindi una curva Precision-Recall. In questo grafico se il $(k+1) - esimo$ documento recuperato non è rilevante allora la Recall sarà la stessa calcolata per i k documenti recuperati mentre la Precision sarà minore. Al contrario, se quel documento è rilevante sia la recall che la precision aumenteranno. Per eliminare queste oscillazioni si può interpolare sulla Precision: la precision interpolata p_{interp} ad un certo livello r di Recall è definita come il più alto valore di precision calcolato per ogni livello di recall $r' \geq r$:

$$p_{interp}(r) = max_{r' \geq r} p(r')$$

La giustificazione di questo è che in genere siamo disposti a guardare qualche altro documento se questo incremente il numero di documenti rilevanti. La curva precision-recall può essere molto informativa ma sarebbe utile avere pochi numeri che rappresentino questa informazione. Per fare questo si usano gli *11-point interpolated average precision*. Per ogni information need (e non query!), viene calcolata la precision interpolata ad 11 livelli di recall $0.0, 0.1, \ldots, 1.0$. Per ogni livello di recall viene quindi calcolata la media aritmetica dei valori di precision interpolata per ogni information need della collezione di test. Un altro standard è la *Mean Average Precision* (MAP), che fornisce una singola misura di qualità su tutti i livelli di recall. Per un singolo information need, l'average precision è la media dei valori di precisione ottenuti per i k top documenti dopo che ogni documento rilevante è recuperato, e questo valore viene quindi mediato su tutti gli information need. Se l'insieme dei documenti rilevanti per un information need $q_j \in Q$ è $d_1, \ldots d_{mj}$ e R_{jk} è l'insieme dei risultati ordinati fino ad ottenere il documento d_k allora:

$$MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_{jk})$$

Quando un documento rilevante non viene recuperato, La precisione viene calcolata come 0. Usando la MAP, i livelli di recall non vengono scelti, non vi è quindi interpolazione. Il valore di MAP di una collezione di test è la media aritmetica delle medie dei valori di precision calcolate per ogni information need.

Le precedenti misure calcolano la precision a tutti i livelli di recall. In molte applicazioni, come nel web, questi problemi non riflettono quelli degli utenti. Si è piuttosto interessati ai risultati rilevanti nella prima pagina o nelle prime tre pagine, ossia nei primi 10-30 documenti, questo viene denotato come *Precision at k*. Questo approccio ha come vantaggio

il fatto che non è necessario stimare il numero di documenti rilevanti ma lo svantaggio di essere una delle misure meno stabili in quanto non ha una buona media, poiché il numero di documenti rilevanti per un certa query ha un forte impatto sul valore della precisione a k. Una misura alternativa è la $R - precision$, questa necessita di conoscere l'insieme dei documenti rilevanti Rel, grazie ai quali è possibile calcolare la precision dei top documenti rel restituiti. Se ci sono $|rel|$ documenti rilevanti per una certa query, vengono esaminati i migliori $|rel|$ risultati del sistema, e la precision e la recall vengono calcolate rispetto a quell'insieme di documenti.

Un ultimo approccio usato sopratutto in unione a tecniche di machine learing per il ranking è il *comulative gain* o meglio il *normalized discounted comulative gain*(NDCG). NDCG è stato pensato per situazioni di rilevanza non binaria. Come la precisione a k viene calcolato su un insieme di k risultati migliori. Per un insieme di query Q sia $R(j, d)$ lo score di rilevanza dato da annotatori al documento d rispetto alla query j, allora:

$$NDCG(Q, k) = \frac{1}{Q} \sum_{j=1}^{|Q|} Z_{kj} \sum_{m=1}^{k} \frac{2^{R(j,m)} - 1}{log_2(1 + m)}$$

dove $Z_{k,j}$ è un fattore di normalizzazione calcolato in modo tale che il valore di $NDCG$ per un ranking ottimo a k per la query j sia 1.

TODO → fondere 7.2 con 7.3

8 Computing scores in a complete search system

Abbiamo visto come pesare i termini nei documenti con adeguate metriche di scoring come nel vector space model l'algoritmo per la similarità del coseno. Alcune euristiche cercano di aumentare la velocità di questi calcoli con il rischio di non trovare esattamente i k documenti che rispondono alla query. Facciamo quindi un passo indietro rispetto alllo scoring del coseno per focalizzarici più in generale sul problema di calcolare scores nei sistemi di ricerca.

8.1 Efficient scoring and ranking

Per una query q = jealous gossip, due osservazioni risultano immediate:

1. Il vettore $v(q)$ ha solamente due componenti non zero.

2. In assenza di pesature per i termini delle query, queste componenti non zero risultano uguali - in questo caso entrambe uguali a 0.707

Per fornire un ranking pe questa query, siamo molto interessati ai scores relativi dei documenti nella collezione. Per questo risulta sufficente calcolare la cosine similarity per ogni vettore documento unitario. Per ogni documento d, la cosine similarity $V(q) \cdot v(d)$ è la somma pesata su tutti i termini della query q, dei pesi di questi termini in d. Scorriamo quindi la lista delle postings nell'indice inverso per i termini in q accumulando lo score totale per ogni documento. Questo schema calcola gli score pe rogni documento nelle postings list di ogni termine della query; Il numero totale di questi documenti dovrebbe essere considerevolmente minore di N. Dati questi scores, lo step finale prima di presentare i risultati all'utente è quello di prendere i K documenti con score più alto. Mentre è possibile ordinare l'intera lista degli scores un approccio migliore risulta essere quello di costruire un heap per restituire solo i top K documenti in ordine.

8.1.1 Inexact top k document retrieval

Consideriamo ora schemi per i quali vengono prodotti k documenti che sono probabilmente tra i k documenti con scoring più alto. Questo perchè in molte applicazioni è sufficente restituire k documenti che siano molto vicini ai k migliori. I k top documenti misurati con la cosine non sono in ogni caso i migliori k per una data query: la cosine similarity è solo una approssimazione alla rilevanza data dall'utente. Consideriamo ora una serie di idee progettate per eliminare un grande numero di documenti senza calcolare il loro valore di cosine. Le euristiche seguono questi due punti:

1. trovare un insieme A di documenti candidati, dove $k < |A| << n$. A non contiene necessariamente i k migliori documenti per valori di score rispetto alla query, ma ha molti documenti con scores vicini ai top k.

2. Restituire i k migliori documenti in A.

8.1.2 Index elimination

Per una query con più termini q, è chiaro che vengano considerati solo i documenti in cui sia presente almento un termine della query. Possiamo fare un ulterire passo in avanti usando altre euristiche:

1. Consideriamo i documenti che contengano termini la cui *idf* superi un certo valore di soglia. Questo ha un beneficio significante in quanto le postings list di termini con basso idf sono generalemente lunghe; cosi facendo l'insieme dei documenti dei quali bisogna calcolare la cosine si riduce drasticamente.

2. Consideriamo solo i documenti che contengono molti dei termini della query. Un effetto negativo di questo schema è che chiedendo documenti che contengono tutti i termini di una query si ha lo svantaggio di poter ottenere un numero di candidati inferiore al valore di k.

8.1.3 Champion lists

L'idea delle *Champion lists* è di precalcolare, per ogni termine t del dizionario, l'insieme degli r documenti con peso maggiore per t. Chiamiamo questo insieme di r documenti la *Champion lists* del termine t. Ora, data una query q creiamo l'insieme A come segue: Prendiamo l'unione delle Champion lists per ogni termine che compare in q. Restringiamo quindi il calcolo della cosine solo ai documenti in A. Il paramentro critco risulta quindi essere il valore di r.

8.1.4 Static quality scores and ordering

Possiamo quindi sviluppare l'idea delle champion lists, in una visione più generale di static quality scores. Abbiamo a disposizione una misura di qualità $g(d)$ per ogni documento d che è indipendente dalla query e quindi statica. Questa misura di qualità può essere vista come un numero compreso tra zero e uno. Il net score per un documento d è una qualche combinazione di $g(d)$ con lo score dipendente dalla query introdotto precedentemente.

$$net - score(q, d) = g(d) + \frac{V(d) \cdot V(d)}{|V(q)||V(d)|}$$

E' utile che i postings siano ordinati in un qualche ordine globale.
La prima idea è una diretta estensione delle champion lists: per un adeguato valore di r, manteniamo per ogni termine t una *champion list globale* dei r documenti con valori maggiori di $g(d) + tf - idf_{t,d}$.
Oppure per ogni termine t manteniamo due postings lists composte da due insieme disgiunti di documenti, ognuno ordinato per valori di $g(d)$. La prima lista, che chiamiamo high, contiene gli m documenti con valori maggiori di tf per t. La seconda lista, che chiamiamo low, contiene tutti gli altri documenti contententi t. Quindi processando una query, prima scorriamo le liste high dei termini della query, calcolando i net scores per ogni documento con nelle liste high di tutti i query terms. Se otteniamo gli scores di k documenti durante il processo terminiamo. Altrimenti, continuiamo considerando le liste low, calcolando gli scores dei documenti in queste liste.

8.1.5 Impact ordering

In tutte le postings list introdotte ordiniamo i documenti secondo un qualche ordine globale. Introduciamo ora una tecnica per il recupero degli inesatti top-K nella quale i postings non hanno un ordine globale. L'idea è quella di ordinare la posting list del termina t secondo il valore decrescente di $tf_{t,d}$. Due idee possono diminuire di molto il numero di documenti per i quali calcolare gli scores:

1. Quando si vede la posting list di un termine della query t ci si ferma dopo un numero fissato di documenti - o perchè si è raggiunti un numero r di documenti visti o perche il valore di $tf_{t,d}$ è sceso sotto un certo valore di soglia.

2. Quando calcoliamo gli scores consideriamo i termini della query in ordine decrescente per valore di *idf* cosi che i termini che probabilmente contribuiranno di più negli scores finali vengano considerati per primi.

Possiamo in oltre considerare una versione di ordine statico nella quale le postings lists sono ordinate secondo una combinazione additiva di score statici e dipendenti dalla query.

8.1.6 Cluster pruning

Nel cluster pruning abbiamo una fase di preprocessing nella quale i vettori documento vengono clusterizzati. Quindi a query time, vengono considerati solo un piccolo numero di cluster come candidati dei quali calcolare la cosine. La fase di preprocessing:

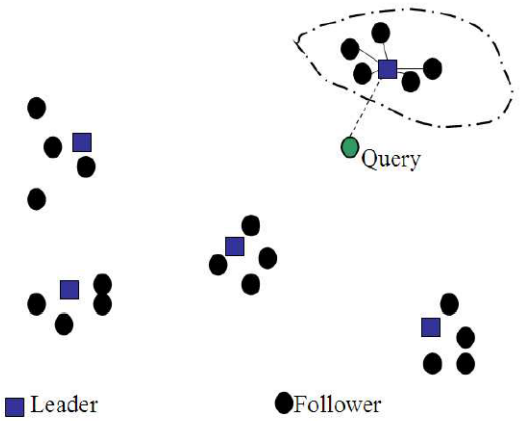
1. Prendi $\sqrt{N}$ documenti a caso nella collezione. chiamali *leader*.

2. Per ogni documento che non è un leader viene calcolato il leader a lui più vicino.

Ci riferiamo ai documenti che non sono leader come *followers*. Intuitivamente, nell'ipotesi di aver scelto $\sqrt{N}$ leader la partizione dei followers attesa sarà $N/\sqrt{N} = \sqrt{N}$. La fase di query processing:

1. Data una query q trova gli L leader più vicini a q. Questo può essere fatto calcolando le cosine similarities tra q ed ognuno dei $\sqrt{N}$ leaders.

2. L'insieme dei candidati A consiste dei leaders L unito con i suoi followers.



► **Figure 7.3** Cluster pruning.

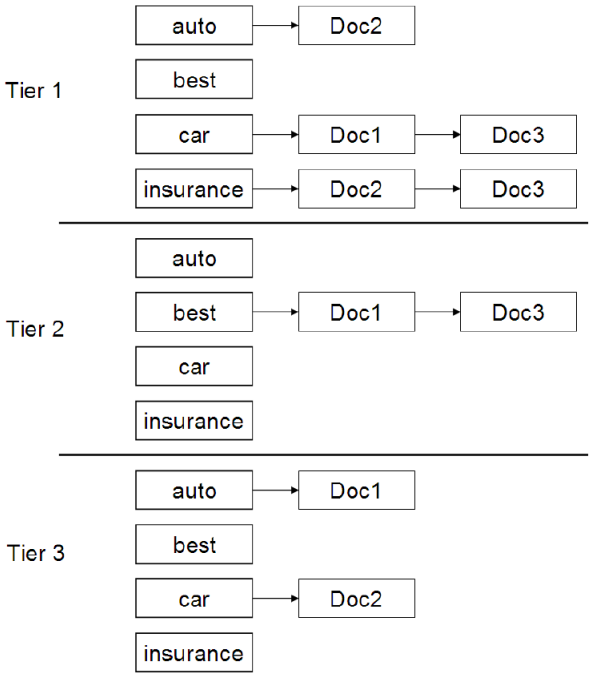
Una variazione al cluster pruning introduce due paramentri interi positivi b_1 e b_2. Nella fase di preprocessing si collega ogni follower ai propri b_1 leaders più vicini, invece che ad un singolo leader. A query time consideriamo i b_2 leader più vicini a q. Chiramente lo schema iniziale corrisponde a questa variazione quando $b_1 = b_2 = 1$.

8.2 Components of an information retrieval system

In questa sezione combineremo le idee viste precedentemente per descrivere un rudimentale motere di ricerca che recuperi e ordini documenti.

8.2.1 Tiered indexes

Come abbiamo detto precedentemente usando euristiche come la index elimination per i k-top recuperati possiamo trovarci nella situazione di recuperare un insieme A di contendenti che ha meno di K documenti. Una soluzione comune a questo problema è l'uso di un indice tired, che può essere visto come una generalizzazione di champion lists. Per fare questo si decidono delle soglie per ogni tier in modo che la postings list sia divisa in più parti (tier), si può pensare ad esempio di scegliere soglia di $tf = 20$ per il tier 1, soglia di $tf = 10$ per il tier 2.



8.2.2 Safe Ranking

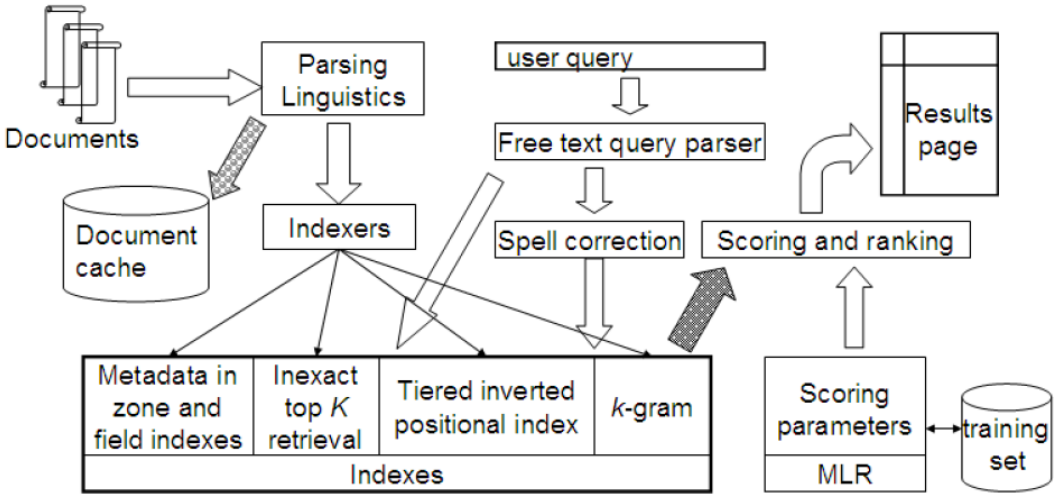
Il termine safe ranking è utilizzato per metodi che garantiscono che i K documenti restituiti siano i K documenti con scoring maggiore. Questo può portare a dover calcolare le

cosine per tutti i documenti. Per evitare questo vedremo metodi pruning che ci permettano di calcolare le cosine solo per una parte j dei documenti. WAND

8.2.3 Query term proximity

Sopratutto per query testuali nel web, l'utente preferisce documenti nei quali i termini della query compaiano vicini tra loro, poichè questo è evidenza che il documento sia focalizzato sul concetto espresso dalla query. Si considerino query con almeno 2 termini $t_1, t_2, ..., t_k$. Sia ω la grandezza della più piccola finestra in un documento d che contiene tutti i termini della query, misurata come il numero di parole nella finestra. Ad esempio in un documento composto dalla sola frase "The quality of mercy is not strained", la più piccola finestra per la query "strained mercy" sarebbe 4. Nel caso in cui il documento non contenga tutti i termini della query possiamo impostare ω ad un numero enorme. Possiamo inoltre utilizzare una variante per la quale si considerano solo i termini che non sono stop-word. Per implementare uno scoring di proximity-weighting dipendente da ω possiamo introdurla come ulteriore feature nella funzione di scoring da pesare usando tecniche di machine learning.

8.2.4 Putting it all togheter



► **Figure 7.5** A complete search system. Data paths are shown primarily for a free text query.

9 Relevance feedback and query expansion

Nella maggior parte delle collezioni vi è il problema di concetti espressi con più termini differenti. Questo problema, conosciuto come *sinonimia* ha un forte impatto sulla recall di molti sistemi di information retrival. Spesso l'utente affronta questo problema raffinando la query manualmente, in questo sistema vedremo tecniche di raffinamente della query di tipo totalemente automatico o che prevedono l'inserimento dell'utente del processo.

9.1 Relevance feedback and query expansion

L'idea del *relevance feedback* (RF) è quella di inserire l'utente nel processo di retrival cosi da perfezionare l'insieme dei risultati finali. In particolare, l'utente fornisce un feedback di rilevanza dei documenti in un insieme iniziale di risultati. La procedura base è:

- l'utente fornisce una query semplice.
- Il sistema fornisce un insieme iniziale di risultati.
- l'utente marka alcune dei risultati come rilevanti/non-rilevanti.
- Il sistema calcola una rappresentazione migliore dell'information need dell'utente.
- viene fornito un nuovo insieme di risultati rivisti.

Il processo sfrutta l'idea che potrebbe essere difficile formulare una buona query quando non conosci bene la collezione, ma è facile giudicare un particolare insieme di documenti, e quindi ha senso raffinare iterativamente la query in questo modo. La ricerca di immagini è un buon esempio di relavence feedback. Questo perché risulta essere un dominio nel quale l'utente può fare molta difficolta nell'esprimere il proprio information need ma al contrario risulta essere semplice indicare quali immagini siano rilevanti o meno.

9.1.1 The Rocchio algorithm for relevance feedback

The Rocchio algorithm è un algoritmo classico per implementare il relevance feedback. Modella un modo per incorporare il relevance feedback nel modello vettoriale.

La teoria sottostante. Vogliamo trovare un vettore q che massimizzi la similarità con i vettori dei documenti rilevanti e la minimizzi quella con vettori di documenti non rilevanti. Se C_r è l'insieme dei documenti rilevanti e C_{nr} quello dei documenti non rilevanti, vogliamo trovare:

$$q_{opt} = argmax_q[sim(q, C_r) - sim(q, C_{nr})]$$

Considerando la cosine similarity il vettore query ottimo q_{opt} per dividere i documenti rilevanti dai non rilevanti risulta essere:

$$q_{opt} = \frac{1}{|c_r|} \sum_{d_j \in C_r} d_j - \frac{1}{|C_{nr}|} \sum_{d_j \in C_{nr}} d_j$$

Ossia il vettore differenza tra i centroidi dei documenti rilevanti e non rilevanti.

Dati i risultati di una query q_{opt}, siano D_r e D_{nr} rispettivamente l'insieme dei documenti rilevanti e non rilevanti identificati dal relevance feedback, Rocchio deriva una query modificata q_m

$$q_m = \alpha q_0 + \beta \mu(D_r) - \gamma \mu(D_{nr})$$
$$= \alpha q_0 + \beta \frac{1}{|D_r|} \sum_{d_j \in D_r} d_j - \gamma \frac{1}{|D_{nr}|} \sum_{d_j \in D_{nr}} d_j$$

Dove α β e γ sono pesi predefiniti. Questi controllano l'equilibrio tra gli insiemi di documenti controllati e la query.

9.1.2 When does relevance feedback work?

Il successo del relevance feedback dipende da alcune assunzioni. Per prima cosa l'utente deve avere abbastanza conoscenza per fornire una query iniziale che si avvicini ai documenti che sta cercando. Alcuni tipi di problemi possono essere:

- Misspelling
- Cross-language information retrival.
- Mismatch, quando vengono usati termini esistenti nel dizionario ma non nella collezione di documenti.

In secondo luogo l'approccio relevance feedback richiede che i documenti rilevanti siano simili tra loro, in quanto implicitamente l'algoritmo di Rocchio modella i documenti rilevanti come un unico cluster.

9.1.3 Pseudo relevance feedback

Lo Pseudo relevance feedback fornisce un metodo per l'analisi locale automatica. Automatizza la parte manuale di pertinenza feedback, in modo che l'utente possa migliorare le prestazioni di recupero senza un'interazione estesa. Il metodo consiste nel fare il normale recupero per trovare un insieme iniziale dei documenti più rilevanti, per poi assumere che i k migliori siano rilevanti e, infine, fornire un relevance feedback come in precedenza sotto questa assunzione. Questo risulta funzionare bene in media ma può fornire risultati pessimi per alcune query a causa del *query drift*.

9.1.4 Query expansion

Per aumentare la recall possiamo utilizzare un altro metodo conosciuto come query expansion. Vengono utilizzati dei metodi globali per la riformulazione della query. I metodi globali non dipendono dalla specifica query e si basano sull'uso di sinonimi per arricchire la stessa, sistemi di memorizzazione per sinonimi sono chiamati tesauri. La costruzione di tesauri può però essere molto costosa quindi possono essere applicati metodi automatici per la loro creazione.

10 Web search basics

10.1 Near-duplicates and shingling

Il web contiene molte copie di contenuti uguali. I motori di ricerca cercano di evitare l'indicizzazione di diverse copie di stessi contenuti, sia per ridurre l'impatto in memoria che l'overhead nella fase di calcolo. Un approccio semplice per trovare duplicati risulta essere quello di calcolare, per ogni pagina web, un impronta digitale che sia riassunto succinto delle caratteristiche di quella pagina. Cosi, nel verificare l'uguaglianza di due fin-

gerprint possiamo desumere l'uguaglianza delle due pagine corrispondenti. Questo però non permette di catturare un fenomeno molto comune nel web: il *near duplication* che si ha quando due pagine sono uguali a meno di pochi caratteri. Una soluzione al problema di individuare i near duplicates giace in una tecnica chiamata *shingling*. Dato un intero positivo k e una sequenza di termini di un documento d, definiamo i $k - shingles$ di d come l'insieme di tutte le sequenze di k termini in d.

Intuitivamente due documenti verranno considerati near duplicates se i loro insiemi di shingles sono molto simili. Sia $S(d_j)$ l'insieme degli shingles di un documento d. Ricordando il *coefficente di Jaccard* che misura il grado di sovrapposizione tra due insiemi $S(d_1)$ e $S(d_2)$ come $\frac{|S(d_1)|\cap|S(d_2)|}{|S(d_1)|\cup|S(d_2)|}$; chiamiamolo quindi $J(S(d_1), S(d_2))$. Il test per calcolare la near duplication tra d_1 e d_2 è quello di calcolare il coefficente di Jaccard. Se questo supera una certa soglia allora i due documenti vengono considerati near duplicates. Tuttavia risulta molto costoso calcoare questo coefficente tra tutte le coppie di documenti.

Per evitare questo viene usato una funzione di hashing. Prima di tutto mappiamo ogni shingle in un valore di hash su un range molto ampio, ad esempio 64bits. Per $j = 1, 2$ sia $H(d_j)$ l'insieme di valori di hash da 64bits corrispondente a $S(d_j)$. Sia π una permutazione casuale da interi di 64bits a interi di 64bits. Denotiamo con $\Pi(d_j)$ l'insieme delle permutazioni dei valori di $H(d_j)$; allora per ogni $h \in H(d_j)$ esiste un valore corrispondente $\pi(h) \in \Pi(d_j)$.
Sia x_j^π il piu piccolo intero in $\Pi(d_j)$. Allora:

$$J(S(d_1), S(d_2)) = P(x_1^\pi = x_2^\pi)$$

Dimostazione. Diamo la dimostrazione in un modo più generale: consideriamo la famiglia degli insiemi i cui elementi sono pescati da un universo comune. Se guardiamo agli insiemi come a colonne di una matrice A, con un rigo per ogni elemento nell'universo. L'elemento $a_ij = 1$ se l'elemento i è presente nell'insieme S_j (rappresentato dalla colonna j). Sia Π una permutazione casuale delle righe di A; denotiamo con $\Pi(S_j)$ la colonna

risultante applicando Π alla colonna j. Finalmente, sia x_j^π la prima riga nella quale la colonna $\Pi(S_j)$ presenta un 1, allora si può provare che per ogni coppia di colonne j_1, j_2

$$P(x_{j_1}^\pi = x_{j_2}^\pi) = J(S_{j_1}, S_{j_2})$$

Consideriamo le due colonne j_1, j_2, le coppie ordinate di S_{j_1} e S_{j_2} definiscono 4 tipi di righe, 00, 01, 10, 11. Denotiamo rispettivamente il numero dei 4 tipi di righe precedenti come C_{00},C_{01}, C_{10}, C_{11}, allora:

$$J(S_{j_1}, S_{j_2}) = \frac{C_{11}}{C_{01} + C_{10} + C_{11}}$$

Per completare la dimostrazione mostriamo che la parte destra dell'equazione è uguale a $P(x_{j_1}^\pi = x_{j_2}^\pi)$, consideriamo una lettura crescente per numero di riga delle colonne j_1 e j_2 fino ad incontrare il primo numero diverso da 0. Poiche Π è una permutazione casuale la probabilità che la più piccola riga abbia un 1 in entrambe le colonne è esattamente quella a destra dell'equazione sopra.

Ripetendo il processo per 200 volte con permutazioni casuali indipendenti otteniamo l'insieme di 200 valori di x_i^π chiamato *sketch* $\psi(d_i)$ di d_i. Possiamo quindi stimare il coefficente di Jaccard tra d_i e d_j calcolando $|\psi(d_i) \cap \psi(d_j)|/200$. Se questo supera una certa soglia consideriamo d_i e d_j simili.

Come possiamo calcolare velocemente questo valore per tutte le coppie? Inizialmente possiamo usare i fingerprints per eliminare tutte le copie identiche. Quindi usiamo un algoritmo di union-find per creare cluster di documenti che sono simili. Cominciamo con la lista $< x_i^\pi, d_i >$. Per ogni x_i^π, possiamo ora generare tutte le coppie i, j per le quali x_i^π è presente in entrambi gli sketches. Ora possiamo calcolare per ogni coppia i, j con un overlap di sketches diverso da zero, il numero di x_i^π in comune. Applicando una certa soglia siamo in grado di sapere quali coppie hanno un overlap alto di sketches.

Un ultimo trick per risparmiare spazio necessario nella computazione di $\psi_i \cap \psi_j/200$. Per eliminare le coppie che probabilmente avranno pochi shingles in comune si possono ordinare gli x_i^π quindi creare degli shingles di questa sequenza per creare dei super-shingles per ogni documento. Se due documenti hanno un super-shingles in comune allora si procede con il calcolo del valore preciso, altrimenti no.

11 Link analysis

La link analysis nel web guarda ai links tra le pagine web come un conferimeno di autorità. Ma misurare la qualità di una pagina web guardando solo i link entranti nella pagina non risulta una soluzione robusta.

11.1 The Web as a graph

Lo studio della link analysis è costruito su due assunzioni:

1. il testo ancorato al link che collega alla pagina B è una buona descrizione della pagina B.

2. L'hiperlink dalla pagina A alla pagina B rappresenta una approvazione della pagina B da parte del creatore della pagina A.

11.2 PageRank

Ci focaliziamo ora sullo scoring e sulle misure di ranking date dalla sola analisi dei link tra le pagine. La prima tecnica, il page rank, assegna un valore compreso tra 0 e 1 per ogni nodo del grafo del web. Da ciò si ha che il valore di un nodo del grafo del web dipende solo dalla struttura dei link del web. Consideriamo una random walk nel web come segue. Ad ogni time step, si procede dalla pagina corrente A scegliendo a caso una pagina a cui A punta. Nel caso in cui un certo nodo A non abbia alcun arco uscente si presenta un problema che si può risolvere introducendo l'operazione di teleport. Nell'operazione di teleport si può arrivare da un nodo ad un qualsiasi altro nodo. La destinazione dell'operazione di teleport è modellata scelgiendo uniformemente a caso tra tutti i nodi del grafo e quindi con probabilità $1/N$ con N numero di nodi del grafo. Nell'assegnare u valore di PageRank ad ogni nodo del grafo l'operazione di teleport viene utilizzata in due modi: 1) in ogni nodo che non ha archi uscenti 2) in ogni nodo con probabilità $0 < \alpha < 1$ mentre viene eseguita la standard random walk con probabilità $1 - \alpha$. Useremo la teoria delle catene di Markov per argomentare il fatto che ogni nodo v

nel grafo viene visitato una frazione di volte $\pi(v)$ che dipende 1) dalla struttura del web, 2) dal valore di α.

11.2.1 Markov chains

Una catena di Markov è un processo stocastico a tempo discreto: un processo che occorre una serie di time-steps in ognuna delle quali viene fatta una scelta random. Consiste di N stati. Ogni pagina corrisponde ad uno stato nella catena di Markov che formuliamo. Una catena di Markov è caratterizzata da una matrice P NxN nella quale ogni elemento è compreso tra $[0, 1]$; gli elementi di ogni riga di P sommano ad 1. Ogni elemento $P_{i,j}$ da la probabilità che al prossimo time-step ci si trovi nello stato j, condizionato dal fatto di trovarsi nello stato i. Ogni elemento $P_{i,j}$ viene chiamato probabilità di transizione e dipende solo dallo stato corrente i. In una catena di Markov, la distribuzione di probabilità dei prossimi stati dipende solo dallo stato corrente e non da come ci si è arrivati. Possiamo vedere una random walk come una catena di markov, con uno stato per ogni pagin web ed ogni probabilità di transizione rappresenta la probabilità da una pagina web ad un'altra. Data una matrice di adiacenza A possiamo derivare la matrice di transizione probabilistica P come segue:

- Se una riga di A non ha elementi 1, si sostituisce ogni elemento della riga con $1/N$, per ogni altra riga si procede come segue
- si divide ogni 1 della matrice per il numero di 1 nella riga
- si moltiplica la matrice risultante per $1 - \alpha$
- si aggiunge α/N ad ogni elemento della matrice per ottenere P

Definition: Una catena di Markov è detta ergodica se esiste un intero positivo T_0 tale che per ogni coppia di stati i, j se si comincia al tempo 0 nello stato i allora per tutti $t > T_0$ la probabilità di trovarsi nello stato j è maggiore di 0. Per essere ergodica una catena di Markov deve soddisfare due condizioni; queste condizioni sono conosciute come *irriducibilità* e *aperiodicità*. La prima dice che esiste una sequenza di transizioni di probabilità diverse da 0 da ogni stato ad ogni altro stato, mentre la seconda dice gli

stati non possono essere partizionati in insiemi tali che le transizioni all'interno di questi occorrano ciclicamente all'interno di questi.

Theorem: Per ogni catena di Markov ergodica esiste un unico vettore di probabilità π stato stazionario che è il principale autovettore sinistro di P, tale che se $\mu(i,t)$ è il numero di visite nello stato i dopo t passi, allora:

$$lim_{t\infty} = \frac{\mu(i,t)}{t} = \pi(i)$$

dove $\pi(i) > 0$ è la probabilità di stato stazionario per lo stato i. Questo valore è il valore di PageRank della pagina web corrispondente.

11.2.2 The PageRank computation

Per calcolare i valore di PageRank ricordiamo che gli autovettori sinistri di una matrice stocastica di transizione P sono N-vettori π tali che:

$$\pi P = \lambda \pi$$

Le N componenti dell'autovettore principale π sono le probabilità degli stati stazionari della random walk con il teleporting, e quindi i valori di PageRank delle pagine corrispondenti. Diamo quindi un metodo elementare per il calcolo del PageRank conosciuto come metodo delle potenze. Se x è la distribuzione iniziale degli stati, allora la distribuzione dopo t steps sarà xP^t. Il metodo delle potenze simula la random walk: cominciamo da uno stato e avviamo la visista per un grande numero di steps, mantenendo le frequenze di visita dei vari stati. Dopo un grande numero di steps le variazioni saranno sotto una certa soglia. Chiamiamo la matrice delle frequenze ottenuta valori di PageRank. I valori di PageRank sono indipendenti dalla query posta dall'utente; Il PageRank è quindi una misura indipendente dalla query della qualità delle pagine web. Per questo motivo questo metodo viene usato in combinazione con altri che tengano conto della query posta.

11.2.3 PageRank analysis

Con il Page Rank si vuole assegnare una misura di rilevanza per una pagina che sia dipendente dalle misure di rilevanza delle pagine che puntano a quella pagina. in formule:

$$\pi(v) = (1 - \alpha) + \alpha \sum_{v_i \to v} \frac{\pi(v_i)}{o(v_i)}$$

dove:

- v è la pagina di cui si vuole calcolare il page rank
- $v_1, v_2, ..., v_n$ sono le pagine che puntano alla pagina v.
- $\pi(v_i)$ è il valore di pagerank della pagina v_i.
- $o(v_i)$ è il numero totale di pagine puntate dalla pagine v_i.
- α è il peso dato al valore di pagerank.

La formula del pagerank risulta essere ricorsiva, sorge quindi il problema di capire se questa converga e se sia dipendente o meno dai valori iniziali di pagerank delle pagine. Possiamo guardare al calcolo dei valori di pagerank come ad una random walk nel web, nella quale ci si segna le frequenze di visite delle pagine. Questo processo puo essere visto come una catena di markov a tempo discreto.
Una catena di markov a tempo discreto è un processo stocastico definito su un insieme discreto di $X = \{X_0, X_1, ...\}$ variabili aleatorie su un insieme finito di stati $S = \{s_1, s_2, ..., s_n\}$ nel quale

$$P(X_n = s_n | X_{n-1} = s_{n-1}, ..., X_0 = s_0) = P(X_n = s_n | X_{n-1} = s_{n-1})$$

Ossia, la probabilità di trovarsi in un certo stato X_n dipende solo dallo stato X_{n-1}.

In una catena di markov uno stato stazionario è uno stato π tale che, data una distribuzione iniziale $\pi(0)$ e una matrice di transizione M:

$$lim_{k \to \infty} \pi^{(k)} = \pi^{(0)} lim_{k \to \infty} M^k$$

equivalentemente:

$$\pi M = \pi$$

La distribuzione degli stati stazionari rappresenta i valori di pagerank. Vediamo quando e come è possibile calcolare questa distribuzione.

Il teorema di Perron dice che, data una matrice $A^{NxN} > 0$. Il raggio spettrale destro di A, $\rho(A) = max_i|\lambda_i|$, è positivo, viene chiamato radice di perron, è inoltre radice semplice. Per cui esiste un solo vettore p:

$$Ap = rp$$

Il medesimo discorso può essere fatto per il raggio spettrale sinistro, considerando la matrice trasposta A^T.

$$A^T p' = r'p'$$

Inoltre vale che:

$$lim_{k\to\infty}(\frac{A}{r})^k = \frac{pq^T}{qp^T}$$

dove:

- A è una matrice quadrata.
- $r = \rho(A)$.
- p è il vettore distro di perron
- q è il vettore sinistro di perron.

Tornando alla nostra matrice M, dato che questa è una matrice stocastica la coppia di perron di destra è $(1, e)$ e di sinistra è $(1, \pi)$, allora:

$$lim_{k\to\infty}(\frac{M}{1})^k = \frac{e\pi}{\pi^T e^T} = e\pi$$

Inserendo una distribuzione iniziale $\pi^{(0)}$

$$lim_{k\to\infty}\pi^{(k)} = \pi^{(0)} lim_{k\to\infty} M^{(k)} = \pi^{(0)} e\pi = \pi$$

Quindi lo stato stazionario raggiunto è indipendente dalla distribuzione iniziale $\pi^{(0)}$. Questo vale se valgono le condizioni del teorema di perron, ossia la matrice deve essere irriducibile. Non devono esistere stati pozzo cioè con probabilità di uscita dallo stato

uguale a 0. Per rendere la matrice irriducibile si introduce l'operazione di teleporting, introducendo nel calcolo la matrice di teleporting T.

$$T = \frac{1}{N} e e^T$$

dove $e^T = [1, 1, ..., 1]$ La matrice finale dove calcolare i valori di page rank sarà una matrice H:

$$H = \alpha P + (1 - \alpha) T$$

11.3 Hubs and Authorities

Mostriamo ora un metodo per il quale ogni pagina ha 2 scores uno detto *hub-score* e l'altro *authority-score*. Per ogni query vengono calcolate due liste una ordinata secondo i valori di hub e l'altra secondo i valori di authority. Questo approccio si basa sull'osservazione che le pagine web possono essere suddivise in due tipologie, pagine che detengono una certa importanza relativamente ad un dato argomento, dette pagine autorevoli, e pagine che, al contrario, parlano di un certo argomeno e citano pagine che sono autorevoli per quell'argomento, queste sono dette pagine hub. Una buona pagina hub è quella che punta a tante buone pagine autorevoli, una buona pagina autorevole è quella puntata da tante buone pagine hub. Questo porta ad un calcolo iterativo. Per una certa pagina v usiamo $h(v)$ per denotare il suo score di hub e $a(v)$ per denotare il suo score di authority.

$$h(v) = \sum_{v \to y} a(y)$$

$$a(v) = \sum_{y \to v} h(y)$$

Siano h e a i vettori di tutti i valori di hub e authority rispettivamente. Sia A la matrice di adiacenza del grafo del web. $A_{i,j}$ è 1 se esiste un link da i a j, 0 altrimenti, allora:

$$h = Aa$$

$$a = A^T h$$

Sostituendo si ha:

$$h = AA^T h$$

$$a = A^T A a$$

Introducendo l'autovalore la prima equazione diventa l'equazione degli autovettori di AA^T e la seconda l'equazione degli autovettori di $A^T A$:

$$h = (1/\lambda_h) A A^T h$$

$$a = (1/\lambda_a) A^T A a$$

Questo porta ad alcune conseguenze:

1. gli aggiornamenti iterativi, se scalati secondo gli appropriati autovalori, sono uguali al metodo per calcolare gli autovettori di AA^T e $A^T A$.

2. Nel calcolare questi valori non siamo costretti all'uso del metodo delle potenze.

Il calcolo finare ha quindi questa forma:

1. Calcolare AA^T e $A^T A$.

2. Calcolare i principali autovettori di AA^T e $A^T A$ per calcolare il vettori dei valori di hub e di authority.

3. restituire i migliori hub e i migliori autorevoli.