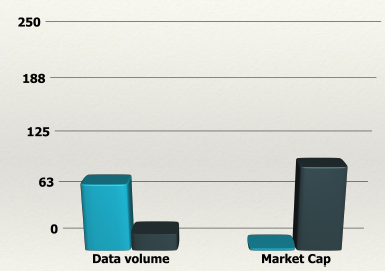
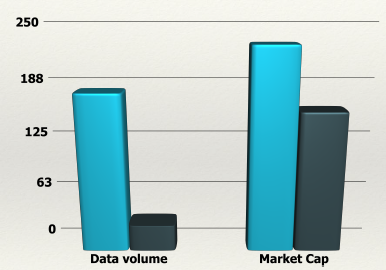
# Riassunto di Information Retrieval

Cos’è l’information retrieval?

L’information retrieval è la ricerca di materiale (di solito documenti) di natura non strutturata che soddisfa un bisogno all’interno di una grossa collezione.

Il primo esempio che ci viene in mente è la ricerca nel web, ma esistono molti altri esempi di cose che utilizzano l’information retrieval, quali la ricerca di email e di documenti nel notebook.

I dati nel information retrieval possono essere strutturati o no, ma quali sono quelli dominanti? Dipende.

Considerando il grafico a destra, possiamo notare come a metà degli anni novanta c’erano più dati non strutturati (colonna azzurra) siano più presenti in termine di volume rispetto a quelli strutturati (colonna nera) mentre nel market cap la situazione e duale.

Considerando le stesse cose ma nel 2009 (che possiamo vedere nel secondo grafico), possiamo notare come la situazione sia cambiata: infatti la mole di dati non strutturati è largamente cresciuta rispetto a quella di dati strutturati.

## Assunzioni base dell’information retrieval

Una collezione indica un insieme di documenti, per il momento assumiamo che sia statica.

Il goal è quello di recuperare i documenti con l’informazione che è rilevante in base al bisogno degli utenti e aiuta a risolvere un compito o task.

## Modello di ricerca classico

Nella foto a destra possiamo vedere il classico modello di ricerca dell’information retrieval, qui possiamo notare le seguenti cose:

* Si parte dalla definizione del task, qui può presentarsi la cosiddetta misconception, cioè un’interpretazione errata dell’obiettivo che vogliamo compiere;
* Dopodichè definiamo il nostro bisogno, qua rischiamo di incappare nella misformulation, un’errata interpretazione del bisogno;
* task e bisogno vanno a formare la query che andrà in un motore di ricerca, quest’ultimo cercherà in una collezione un sottoinsieme di documenti in cui la query viene soddisfatta;
* I risultati ottenuti verranno utilizzati per ridefinire la query e quindi cominciare un’altra ricerca.

## Come possiamo misurare la bontà di una ricerca?

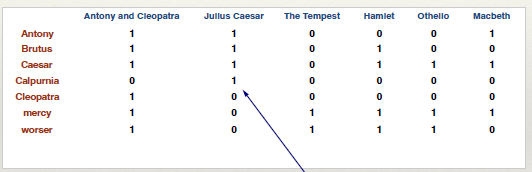
La bontà di una ricerca può essere misurata con i concetti di:

* precisione, cioè la frazione di documenti recuperati che sono rilevanti per il bisogno dell’utente;
* recall, la frazione di documenti rilevanti recuperati in una collezione.

# Recupero binario e matrici di incidenza

Un metodo per recuperare i documenti è la matrice di incidenza, una matrice binaria in cui si si segna True quando una parola è presente e False quando non lo è.

Per fare un esempio: bisogna cercare tutti i passaggi di Shakespeare che contengono le parole Brutus e Caesar ma non Calpurnia, in questo caso la matrice di incidenza è la seguente:



Le singole linee della matrice di incidenza sono i vettori di incidenza, su di essi è possibile effettuare operazioni binarie bit a bit, considerando il precedente esempio:

100100 è il vettore di incidenza indicante la presenza delle parole Brutus e Caesar ma non di Calpurnia, da qui recuperiamo tutti i documenti in cui è presente True, in questo caso “Antony and Cleopatra” e “Hamlet”.

## Quanto conviene effettivamente questa metodologia di ricerca?

In caso di piccole collezioni, non c’è nessuno problema, nelle grandi collezioni però compilare la matrice risulta dispendioso in tempo ma soprattutto in spazio. Oltre a ciò, la matrice può risultare molto sparsa.

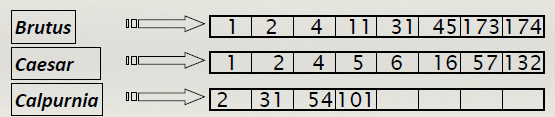
Esiste una soluzione migliore?

# Indici invertiti

Gli indici invertiti rappresentano la soluzione al precedente problema e sono la struttura dati chiave dell’information retrieval moderno, come funzionano?

Considerando ogni termine t:

* si inseriscono in una lista tutti i documenti che contengono il termine t;
* in questo modo possiamo identificare ogni documento con un id univoco.

Ragionando in questo modo, queste liste potrebbero risultare di lunghezza variabile dato che non è detto che le parole occorrono tutte nello stesso numero, è possibile utilizzare array di lunghezza fissa?

Considerando le tre liste a destra, che succede se aggiungiamo la parola Caesar al documento 14?

Rischiamo di non ottenere un risultato corretto dato che la query in questo caso non considererebbe tutti i documenti.

## Come si può rappresentare un indice invertito?

La struttura dati per rappresentare gli indici invertiti è la posting list, come la rappresentiamo?

* nello storage, il modo migliore è memorizzare ogni lista una dopo l’altra;
* In memoria, utilizziamo un dizionario, la chiave è il termine t, il valore invece è la lista di documenti in cui t è presente.

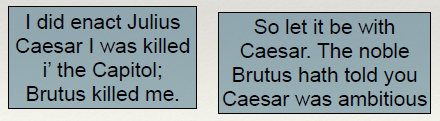
## Come si costruisce un indice invertito?

Prima della costruzione effettiva, effettuiamo quattro fasi di text processing:

* A partire da dei documenti, si esegue una fase detta tokenizzazione in cui si estraggono le parole dalle sequenze di caratteri. Quello che si ottiene viene chiamato token stream;
* La fase di normalizzazione ha il compito di mappare il testo e i termini della query che sono nella stessa forma, ad esempio “USA” e “U.S.A.” che vogliono dire la stessa cosa;
* La fase di stemming permette di matchare tra loro parole con una radice comune, come ad esempio “autorizzazione” e “autorizzare”;
* La fase di stop words permette di omettere o meno le parole molto comuni, come ad esempio gli articoli.

### Esempio

Consideriamo i due seguenti documenti:



| Tokenizzazione | Ordinamento | Dictionary & Posting |
| --- | --- | --- |
| Si ottiene una sequenza di coppie parole-docId. | Si ordina prima per termine, poi per id. | Si uniscono entry multiple di un singolo documento.  Si divide in Dictionary e Posting.  Si aggiunge la frequenza dell’informazione del documento. |
|  |  |  |

## Query processing con gli indici invertiti

### Clausola AND

Considerando Brutus e Caesar come parole, l’AND tra esse funzione nel seguente modo:

* localizza Brutus e Caesar nel dizionario e recupera i rispettivi posting;
* unisce i posting attraverso l’intersezione.

La complessità di questa operazione è O(x+y) dove x e y sono le lunghezze dei due posting.

# Modello di recupero booleano

Il modello di recupero booleano è in grado di rispondere a query sotto forma di espressioni booleane, cioè che utilizzano operatori come AND, OR and NOT.

Le query booleane vedono il documento come un insieme di parole, inoltre sono precise dato che i documenti matchano o meno le condizioni.

Questo metodo è il più semplice sistema di information retrieval ed è stato il fiore all’occhiello per ben trent’anni, tant’è che è ancora utilizzato nella ricerca delle email e in altri ambiti.

## Ottimizzazione delle query

L’ordine delle parole è importante e influenza la complessità dell’algoritmo, come possiamo trovare l’ordine migliore?

Si parte dall’insieme più piccolo e si procede via via con quelli più grossi.

Più in generale, è possibile stimare la grandezza di ogni clausola OR come la somma degli operandi.

## Phrase query e indici posizionali

Le phrase query sono particolari query la cui risposta è una frase. Un esempio di phrase query sono gli indici bi-parola, particolari indici formati da paia di termini che formano una frase. Ogni bi-parola è memorizzata come un termine del dizionario e rende immediato il query processing.

Utilizzando gli indici bi-parola è possibile effettuare phrase query con frasi più lunghe, ad esempio:

“stanford university palo alto” diventa “stanford university” AND “university palo” AND “palo alto”

Dall’altra parte però questo tipo di query può portare a falsi positivi.

Il problema dei falsi positivi è assai noto negli indici bi-parola, cosiccome il blowup quando il dizionario diventa troppo grande, ciò infatti rende unfeasible gli indici stessi.

Tuttavia, gli indici bi-parola non sono uno standard bensì una parte di una soluzione composta.

Un’altra soluzione è l’utilizzo degli indici posizionali, particolari indici in cui si memorizza per ogni termine il numero di occorrenze e la posizione in ogni documento.

## Processing di una phrase query

* Si estraggono le entry degli indici invertiti per ogni termine;
* Si uniscono le liste doc:pos in modo da enumerare tutte le posizioni.

Gli indici posizionali possono essere utilizzati come delle query, cosa che non è possibile con gli indici bi-parola.

Quanto spazio occupa un indice posizionale? Dal momento che sono un’espansione dei posting, lo spazio in più occupato è sostanziale! In ogni caso, data la loro grande utilità, esistono dei metodo per comprimerle.

In generale, un indice posizionale è grande dalle due alle quattro volte rispetto a quello non posizionale ed è una percentuale che va dal 35% al 50% del testo originale.

## Schemi combinati

È possibile combinare i due approcci citati in precedenza in modo da rendere efficienti cose che risultano inefficienti nel posting posizionale.

Ciò comunque ha un costo in termini di spazio, infatti a fronte di query più veloci, bisogna memorizzare circa il 26% di spazio in più.

# Come si parsifica un documento?

Il parsing di un documento dipende da tanti fattori, in primis dal formato per poi andare al linguaggio utilizzato al set di caratteri.

Riguardo il formato e il linguaggio, indicizzare un documento potrebbe risultare complicato dal momento che questi possono includere documenti con termini provenienti da differenti linguaggi.

Un esempio è una mail scritta in francese con un documento tedesco in allegato oppure una clausola scritta in inglese.

Come è possibile far fronte a ciò? Esistono delle librerie open source che gestiscono il tutto.

Un’altra complicazione riguarda il documento stesso, infatti bisogna definire l’unità documento in un certo modo. L’unità può essere ad esempio un file, un’email o un gruppo di questi.

## Token

Un token è un’istanza di una sequenza di caratteri ed è il risultato della fase di tokenizzazione descritta in precedenza.

Ogni token è un possibile candidato per un entry di un indice, ma quali di questi possiamo considerare validi?

Dipende, esistono infatti tanti casi in cui emettere token, ad esempio: “San Francisco” come possiamo tokenizzarlo? Utilizzando un solo token oppure due?

Passando ora ai numeri, esistono differenti formati che rappresentano cose altrettanto differenti, come possiamo gestirli?

I vecchi sistemi di information retrieval non riuscivano a gestire i numeri, tuttavia a volte possono risultare utili al fine di rendere la ricerca più precisa o per rappresentare errori. Un’altra idea è indicizzarli come metadati in maniera separata.

E per quanto riguarda la lingua usata?

Anche questo è un problema da gestire opportunamente in quanto ogni linguaggio ha le sue regole, basta pensare al tedesco in cui i nomi composti non sono segmentati o al giapponese che non utilizza spazi.

## Termini

I termini indicano le cose indicizzate da un sistema di information retrieval.

Al fine di rendere la ricerca più efficiente, conviene rimuovere le cosiddette stop words, quel sottoinsieme di parole che risultano più comuni all’interno del dizionario.

Intuitivamente queste parole riguardano perlopiù articoli e preposizioni.

La tendenza comunque è ben lontana da ciò, infatti esistono tecniche di compressioni tali da non rendere l’inclusione delle stopwords un problema, oltre che a tecniche di ottimizzazione delle query stesse.

Un’altro passo è la normalizzazione dei termini, cioè indicizzare le parole aventi la stessa forma, ciò permette di ottenere come risultato un termine che è una entry del dizionario.

E per quanto riguarda la normalizzazione tra linguaggi differenti?

Per quanto riguarda cose come le date, è possibile effettuare una conversione, ma in generale queste fasi dipendono dal linguaggio e perciò dal modo in cui questo viene riconosciuto.

Proprio per questo motivo è cruciale normalizzare i testi indicizzati così come i termini della query in modo identico.

Per quanto riguarda il case folding, come gestirlo? in questo caso si rende tutto minuscolo con dovute eccezioni quali gli acronimi.

Esiste anche un espansione asimmetrica per permettere la ricerca di parole anche al singolare/plurale, un approccio assai più potente ma meno efficiente.

La gestione di sinonimi e omonimi si può gestire scrivendo le equivalenze “a mano” e modificando opportunamente la query, in alternativa è possibile espanderla aggiungendo altre parole.

## Lemmatizzazione e Stemming

La lemmatizzazione permette di ridurre le forme varianti in quella base, cosa che avviene spesso per i verbi, nel genitivo sassone e nella conversione plurale-singolare.

Lo stemming indica la riduzione di una termine alla radice prima dell’indicizzazione.

Un comune algoritmo per effettuare lo stemming è l’algoritmo di Porter, esso è composto dalle convenzioni più cinque fasi di riduzione, ogni fase viene applicata sequenzialmente e consiste in un set di comandi.

Un esempio di convenzione è la seguente: tra tutti i comandi, seleziona quello che applica il suffisso più lungo.

I metodi citati in precedenza sono specifici per il linguaggio e/o l’applicazione, questi “plugin” vanno ad aggiungere al processo di indicizzazione.

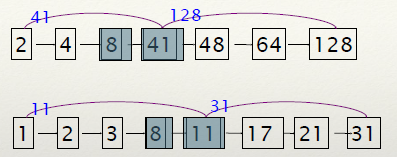
Tornando a parlare dello stemming, quanto risulta utile? Dipende dal linguaggio, nel caso dell’inglese si hanno risultati misti mentre risulta veramente utile per lo spagnolo, il tedesco e il finlandese.

## Puntatori e liste skip

Come già detto in precedenza, la complessità del merge tra posting è O(x+y), si può fare di meglio? Certo e possiamo farlo con i puntatori skip, ciò permette di evitare di scorrere tutto il posting per effettuare il merge.

Dove piazziamo i puntatori skip e come si utilizzano? Posizioniamo i puntatori skip agli estremi e a metà del posting, dopodiché effettuiamo il classico algoritmo, andando a destra o a sinistra in base al confronto.

Per fare un esempio:



Considerando i posting sopra, prima di tutto si salta verso 41 e 11 e si effettua il confronto. Dato che 11 è più piccolo, si va a sinistra e, in entrambi i casi, troviamo il numero 8, il primo nodo comune.

Non esiste un modo unico per piazzare i puntatori skip, ogni metodo ha i suoi pregi e i suoi difetti.

Un metodo utile per piazzare i puntatori skip è il seguente: dato un posting di lunghezza L, si posizione uno skip ogni sqrt(L) nodi, questo metodo è facile quando l’indice è statico, in caso di dinamicità risulta difficile a causa degli aggiornamenti.

# Compressione degli indici

La compressione degli indici ci permette tanti vantaggi:

* prima di tutto l’utilizzo di meno spazio su disco e quindi inserire più dati all’interno;
* incrementare la velocità di trasferimento dati dal disco alla memoria, ciò vuol dire che il tempo di lettura di dati compressi più la loro decompressione deve essere minore della lettura degli stessi dati ma non compressi.

Nel caso degli indici invertiti, la compressione:

* rende il dizionario più piccolo da mantenere in memoria, tanto che è possibile mantenere in memoria anche i posting;
* Per quanto riguarda i posting, lo spazio su disco si riduce così come il loro tempo di lettura. Questo permette ai motori di ricerca di mantenere una parte significativa del posting in memoria.

## Proprietà statistiche dei termini

Consideriamo N come il numero di documenti che vale 800k, L è 200 ed è il numero medio di token per documento mentre M è il numero di termini, circa 400k.

## Compressione con o senza perdita di dati?

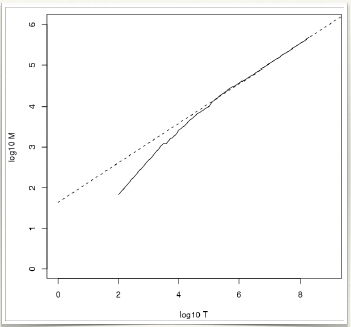
La fase di preprocessing si può considerare come una sorta di compressione a perdita di dati dato che va a eliminare parte dell’informazione.

## Dimensione del vocabolario o della collezione?

Quanto è grande un vocabolario e quando parole distinte ci sono? Dal momento che non è possibile assumere un upper bound, si può dedurre che la grandezza del vocabolario cresce insieme a quella della collezione.

### Regola di Heaps

Secondo la regola di Heaps:

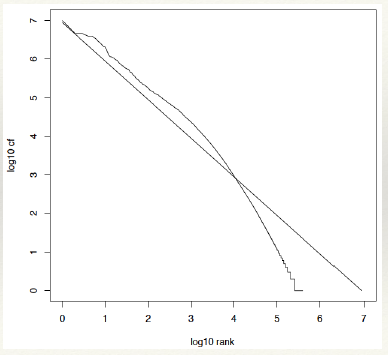


dove M è la dimensione del vocabolario e T il numero di token in una collezione.

I valori *c* e *b* sono costanti, tipicamente il primo varia da 30 a 100 mentre il secondo è intorno a 0.5.

In uno spazio logaritmico, la regola di Heaps predice una linea che si interseca a 0.5, questo è il più semplice caso possibile.

### Regola di Zipf

La regola di Zipf permette di ricavare l’i-esimo termine più frequente, esso ha una frequenza proporzionale a 1/i.

dove c è una costante di normalizzazione.

cfi invece è la frequenza della collezione, cioè il numero di occorrenze del termine ti all’interno della collezione.

Dal momento che il termine più frequente occorre cf1 volte, tutti i seguenti occorreranno cf1/i volte come conseguenza. Possiamo quindi riscrivere la formula come segue:

## Compressione del dizionario

La struttura di ricerca di un dizionario riprende quella degli alberi binari, qui le foglie sono gli stessi termini a cui è associata la frequenza e il puntatore al posting.

Di per sé, il termine occupa 20 byte mentre frequenza e puntatore al posting 4 ciascuno, quindi con circa 400k termini si occupano 11.2 megabyte di memoria.

Definire però una lunghezza fissa per i termini è uno spreco di spazio, quindi come si procede?

Si memorizza il dizionario come una lunga stringa di caratteri, precisamente il puntatore alla prossima parola indica anche la fine di quella corrente.

Questa soluzione permette di salvare fino al 60% di spazio, infatti:

* frequenza e puntatore al posting occupano assieme 4 byte;
* i puntatori ai termini ne occupano 3;
* ogni termine ne occupa mediamente 8.

Considerando sempre circa 400k termini, si ottengono 7.6 megabyte, un guadagno di circa la metà rispetto al caso precedente.

## Blocking

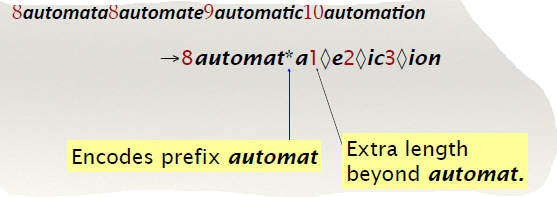
Memorizzare i puntatori a ogni k-esimo termine ha bisogno di un byte in più, ciò permette di risparmiare k byte.

Dall’altra parte il block rallenta la ricerca nel dizionario facendo più confronti.

| No blocking | Blocking |
| --- | --- |
|  |  |

## Front coding

Il front coding ordina le parole aventi un prefisso comune e memorizzando le differenze, quindi memorizzando solo il necessario e risparmiando spazio.



## Compressione dei posting

I file dei posting sono più grandi del dizionario almeno di un fattore 10. L’obiettivo è memorizzare il posting in modo compatto, come possiamo fare? Per i reuters si potrebbero utilizzare 32 bit per id oppure un numero di bit in base al numero di documenti.

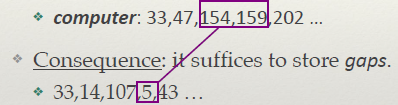
Quest’ultima soluzione risulta più conveniente.

Bisogna però tenere conto di due conflitti:

* alcuni termini occorrono solo in un documento o in pochi;
* altri invece occorrono in ogni documento.

In base a ciò, sarebbe inutile utilizzare lo stesso numero di bit per memorizzarli, quindi si memorizzano normalmente nel primo caso e usando un vettore bitmap nel secondo.

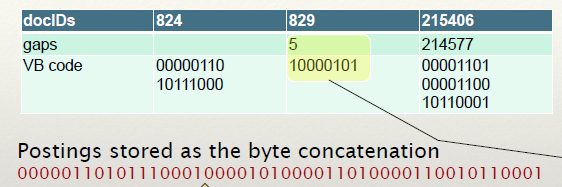
E’ inoltre possibile memorizzare solamente i gap tra un id e l’altro in modo da ridurre la dimensione e quindi memorizzarli usando meno bit.



L’idea è quindi quella di utilizzare solo i bit strettamente necessari per memorizzare un dato, ciò è possibile grazie al variable length encoding, un metodo che permette di utilizzare codici corti sui numeri piccoli.

Considerando invece un gap G, il numero minimo di bit utili per memorizzarlo è , uno di questi è il bit (c) che indica la continuazione.

Se G è minore o uguale a 127, lo memorizziamo su 7 bit e settiamo (c) ad 1. altrimenti si memorizza G con meno bit e si utilizzano quelli che restano per memorizzare altri usando lo stesso algoritmo. Alla fine si setta (c) a 1 per l’ultimo byte e a 0 per tutti gli altri.



E’ possibile utilizzare differenti unità di allineamento come le parole o i nibble, questo perché i byte sprecano spazio nel caso ci siano dei gap piccoli, infatti da questo punto di vista i nibble funzionano meglio.

## Codici unari e gamma

Rappresentare un numero n come una sequenza di n unì con una 0 alla fine non è certamente una gran soluzione dal momento che risulta esponenziale nella conversione.

Coi codici gamma però possiamo comprimere meglio rappresentando G come una coppia lunghezza-offset:

* l’offset è il gap G in binario senza il bit più significativo;
* la lunghezza è la lunghezza dell’offset e viene codificata in unario.

I codici gamma hanno le seguenti proprietà:

* la lunghezza dell’offset e della lunghezza sono logaritmiche, quindi G viene codificato su bit;
* Tutti i codici gamma hanno lunghezza dispari, hanno un prefisso unico decodificabile e possono essere utilizzate in ogni distribuzione, oltre a non necessitare di parametri.

Nella pratica i codici gamma non sono tanto utilizzati dato che la compressione e la manipolazione sono operazioni lente, oltre al fatto che le macchine hanno parole di lunghezza fissa.

Proprio per questo motivo si preferisce il variable length encoding.

## Group variable integer code

Questa tecnica consiste nel codificare quattro interni in blocchi dai 5 ai 17 byte:

* il primo di questi contiene 4 campi lunghi due bit;
* dopodiché ci sono i byte relativi ai quattro numeri, ogni numero ne utilizza dai 4 ai 16.

Questo algoritmo è almeno tanto veloce quanto il variable byte encoding, infatti decodificare i gap è molto semplice mentre il primo viene decodificato con una lookup table.

## Simple9

Simple9 è un algoritmo che permette di memorizzare numeri in 32 bit con una selettore di formato, come fa?

* si utilizzano blocchi di encoding di 4 byte;
* il nibble più significativo descrive il layout dei 28 bit che seguono, precisamente si segue il seguente ragionamento: .

Questo algoritmo decodifica in modo efficiente utilizzando una bitmask.

# Ranked retrieval

Dal momento che gli utenti non sono capaci di scrivere query booleane e non vogliono aspettare troppo tempo per ottenere un risultato, occorre trovare metodi più semplici e veloci.

Oltre a ciò bisogna inoltre tenere conto che, in base alle query, o abbiamo troppi risultati oppure questi sono troppo pochi.

A fronte di questi problemi, come procediamo? La soluzione è il ranked retrieval, qui infatti utilizziamo query formate solamente da testo in linguaggio naturale, la dimensione dei risultati prodotti non è un problema e basta mostrarne i primi k all’utente.

L’idea è quella di restituire tutti i documenti che possono risultare più utili all’utente, come facciamo a ordinarli in questo modo? Assegnamo un punteggio da 0 a 1 a ogni documento, esso indica la bontà del match fatto dalla query.

Come possiamo assegnare il punteggio ai documenti?

### Coefficienti di Jaccard

Una possibile soluzione è utilizzare i coefficienti di Jaccard, con essi si misura l’overlap tra due insiemi A e B nel seguente modo:

* ;
* ;

Se A e B sono insiemi disgiunti, la funzione dà 0 come risultato.

Il problema di Jaccard è che non considera la frequenza dei singoli termini, inoltre non considera i termini rari, essi infatti hanno grande valore informativo.

### Matrice termine-documento

La matrice termine documento prende spunto dalle matrici binari descritte all’inizio del riassunto, esse contano le occorrenze di ciascun termine all’interno di ogni documento.

Questa rappresentazione però non considera l’ordine delle parole all’interno di un documento, questo modello è detto “borsa di parole” ed è una sorta di passo indietro dato che gli indici posizionali permettono ciò.

### Frequenza del termine

Consideriamo la frequenza di un termine t in un documento d come tf(t,d), cosa possiamo dire riguardo a ciò?

Se in un documento un termine occorre 10 volte, esso ha più rilevanza rispetto a un documento in cui lo stesso termine occorre una volta. Tuttavia non è 10 volte più rilevante, questa è la frequenza raw e non è quella che vogliamo ottenere!

La rilevanza infatti non cresce proporzionalmente con la frequenza dei termini.

Come possiamo calcolare la rilevanza sulla base di ciò che abbiamo detto?

Si utilizza la frequenza logaritmica, ciò permette una rilevanza direttamente proporzionale alla frequenza ma non in modo lineare.

Calcoliamo la rilevanza come segue:



Il punteggio sarà quindi la somma di tutte le rilevanze di ogni termine t presente sia nella query, sia nel documento:



### E per quanto riguarda i termini rari?

Dato il grande contenuto informativo dei termini rari rispetto a quelli frequenti, come possiamo dare un peso maggiore ai primi? Usiamo la frequenza del documento!

Precisamente, df(t) indica il numero di documenti che contengono il termine t, essa indica una misura di informazione inversa sullo stesso t.

Da ciò, possiamo calcolare la frequenza di documento inversa nel seguente modo:



Che effetto ha la idf nel ranking? idf non ha effetto nelle query formate da un termine, esso funziona nelle query con almeno due termini.

Per far fronte a ciò, si combinano tf e idf per il calcolo della rilevanza:



La rilevanza diventa quindi direttamente proporzionale sia al numero di occorrenze di un termine, sia alla sua rarità. Per quanto riguarda lo score, la formula rimane la stessa.

### Weight matrix

Implementando ciò, la matrice di conteggio diventa una matrice di pesi! Inoltre possiamo considerare i documenti come vettori, infatti i termini indicano gli assi dello spazio, quindi ci possono essere migliaia di dimensioni. Tuttavia la maggior parte di questi vettori è molto sparsa, quindi molte entry sono 0.

E usare le query come vettori? In questo modo potremo calcolare il ranking dei documenti in base alla prossimità.

Che concetto di distanza utilizziamo? la distanza euclidea non va bene perchè non adatta a vettori di lunghezza differente. In generale, l’idea di distanza in questo caso non è una buona idea dato che può risultare significativa anche se la distribuzione è la simile.

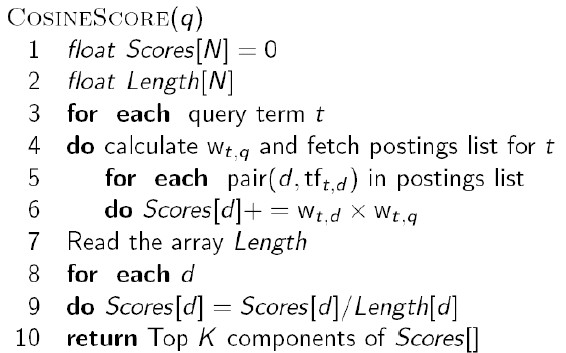
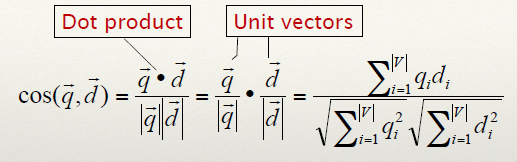
Cosa possiamo fare?

Possiamo provare a utilizzare l’angolo tra due punti come misura, quindi più i documenti risultato simili, più l’angolo tra questi sarà minore.

Per rendere la proporzionalità più diretta, consideriamo il coseno dell’angolo anziché l’angolo stesso.

E’ inoltre possibile normalizzare i vettori in modo da facilitare il confronto, per farlo si divide il vettore per la sua lunghezza.

Ci sono poi le proprietà dei vettori, quindi possiamo utilizzare per facilitare il calcolo del coseno, il quale diventa come segue:



Il coseno tra query e documento diventa quindi il punteggio da assegnare al documento stesso. L’algoritmo in alto a destra calcola lo score in base al termine ma può essere facilmente adattato per calcolarlo sul documento.

L’unico problema sta nel memorizzare la rilevanza w(t,d) dato che farlo nei posting diventa costoso per via della virgola mobile. Nel caso del tf-idf, è sufficiente memorizzare il tf nel posting e idf in testa a ognuno di essi.

Infine si estraggono i migliori K oggetti utilizzando una coda di priorità.

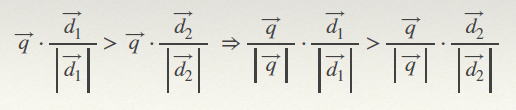
## Ranking del coseno efficiente

Per effettuare un ranking efficiente, l’idea è quella di trovare i K documenti della collezione più vicini, questi corrispondono a quelli con il valore query-documento più alto.

Come possiamo effettuare ciò?

Si risolve il problema del K-NN per un vettore query, ciò si può fare in modo efficiente per spazi con poche dimensioni ma non quando questo numero è alto.

Abbiamo inoltre un caso particolare: le query non pesate! In questo caso infatti non c’è bisogno di normalizzare il vettore e, assumendo che ogni termine occorra una sola volta:



## Selezione vs ordinamento

Per risolvere il kNN, esistono due metodi:

* l’ordinamento consiste nel calcolo del ranking di ogni documento, ordinarli in base a esso e scegliere i K migliori;
* la selezione invece calcola il ranking di ogni documento trattenendo i k risultati migliori.

Un modo per selezionare i migliori K documenti è utilizzare un heap, un albero binario dove alla radice si trova l’oggetto di valore massimo.

Considerando un numero J di documenti, la costruzione dell’heap richiede 2\*J operazioni mentre il K migliore è logaritmico.

Tuttavia è presente un collo di bottiglia nel calcolo del coseno, c’è un modo per evitarlo? E’ possibile evitare tutta la “computazione inutile” ma ciò può portare a risultati sbagliati.

## Pruning

Un modo per velocizzare il calcolo del coseno è fare del pruning sul heap, evitando quindi computazioni inutili.

I risultati che si ottengono non sono comunque i K documenti migliori bensì quelli che ci vanno più vicini, ciò però non è importante.

L’approccio generico è quello di trovare un set di contendenti A maggiore di K ma molto minore di N, questo set non deve necessariamente contenere i migliori K documenti ma almeno una parte di questi.

Dopo aver ottenuto A, prendiamo i K documenti migliori da esso e li restituiamo.

Possiamo considerare il set A come un pruning dei non contendenti.

## Index elimination

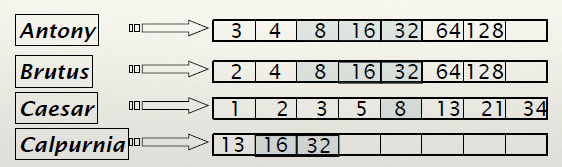
Il calcolo del coseno considera solamente i documenti che contengono almeno un termine della query.

Inoltre prende solo in considerazione i termini che modificano il ranking in modo sostanziale, ad esempio: con la query “catcher in the rye”, solo catcher ed rye sono termini importanti mentre in e the no, questi ultimi infatti hanno un idf basso e non contribuiscono in modo sostanziale.

Ciò permette di eliminare dal set A tutti i termini con idf basso.

Inoltre consideriamo solo i documenti che contengono più termini della query, come? calcolando lo score solo nei documenti che hanno un certo numero di parole.

Per fare un esempio, considerando i posting sotto, se consideriamo solo i documenti con più di tre o quattro termini, computeremo lo score solo per i documenti 8, 16 e 32.



Le liste di campioni sono la precomputazione del termine t di ogni dizionario utilizzando un numero r di documenti col peso più alto per t.

Quindi, a tempo della query, computiamo solamente gli score per i documenti della lista di campioni e da qui prendere i K migliori.

## Score indipendenti dalla query

### Score di qualità statica

Vogliamo che i migliori documenti siano rilevanti e autoritativi:

* la prima qualità è data dal coseno e quindi dalla query;
* la seconda invece non dipende da essa bensì dalle proprietà del documento, come ad esempio il numero di citazioni.

Come si può modellare l’autorità? Si assegna a ogni documento un score tra 0 e 1, ciò quindi si può modellare con una funzione g(d) a piacere in cui d indica il documento.

Combinando lo score della query con quello del documento, otteniamo il net-score, essa può essere una qualunque funzione di combinazione lineare tra due segnali, come ad esempio la somma:

## Come calcoliamo i migliori K documenti col netscore?

La prima idea è la seguente:

* si ordinano tutti i posting in base a g(d);
* in questo modo si può calcolare l’intersezione tra posting e il coseno in modo concorrente.

L’ordine per g(d) risulta importante in quanto i migliori documenti occorrono prima durante lo scorrimento dei posting, ciò si traduce in risultati migliori anche quando il tempo richiesto è molto poco.

E’ inoltre possibile combinare questo ordinamento con le liste di campione, quindi mantenere per ogni termine una lista di r documenti con gli score g(d)+tf-idf(t,d) più alti. Da queste liste si estraggono infine i K risultati migliori.

# Indici a livelli

Per migliorare la ricerca, possiamo mantenere due liste di posting chiamate livello alto e livello basso.

Consideriamo il livello alto come una lista di campioni, quando scorriamo i posting per una query, scorriamo solamente questa lista e selezioniamo i K documenti migliori.

Dopodichè si procede col livello basso.

Questa tecnica può essere utilizzata anche solo col coseno, senza quindi ordinare con g(d).

## Posting ordinati a impatto

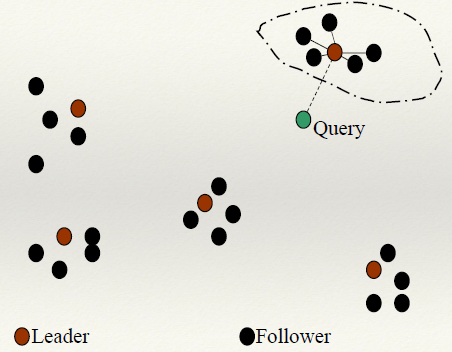
Noi vogliamo calcolare gli score solamenti nei documenti con un wf(t,d) abbastanza alto, come fare?

* Si ordina ogni posting per wf(t,d);
* dopodiché si computa lo score.

Esistono due idee per calcolare lo score in questo caso:

* la prima idea è l’early termination e consiste nello scorrere i posting di un termine t fino a quando non si hanno r documenti oppure wf(t,d) è più basso di un threshold dato. Dopodiché si uniscono tutti i set di documenti e infine si calcolano gli score.
* la seconda idea invece è quella di ordinare i posting in modo decrescente rispetto a idf, infatti i termini con l’idf più alto sono quelli che contribuiscono di più. Da ciò, scorriamo i posting e ci fermiamo quando gli score dei documenti non presentano cambiamenti relativi.

## Cluster pruning

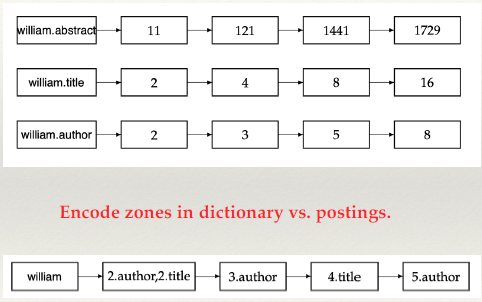
Il cluster pruning permette di non considerare tutti i sottoinsiemi di documenti che sono “lontani” dalla nostra query, come si fa?

* Si parte con una fase di preprocessing in cui si prendono documenti casuali che prendono i nomi di leader. Dopodiché per ogni documento si calcola il leader più vicino e si aggiunge al suo cluster, esso prende il nome di follower. Mediamente ogni leader ha follower;
* La fase di query processing ha il compito di calcolare il leader L più vicino alla query e, da qui, i K documenti più vicini tra i follower di L.

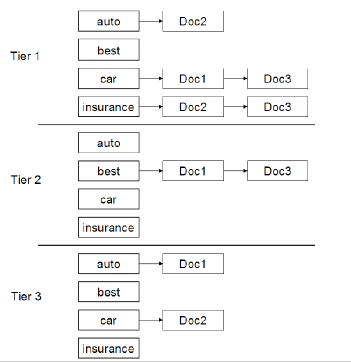
Si utilizza il random sampling perché velocizza l’algoritmo e perché i leader scelti seguono la distribuzione dei dati.

## Indici parametrici e di zona

Dal momento che i documenti sono composti da più parti con una certa semantica (alcune di queste formano i cosiddetti metadati), possiamo sfruttare ciò per rendere la ricerca più precisa, come?

Usiamo dei posting per ogni valore del campo mentre i campi della query vengono solitamente trattati come delle congiunzioni.

Una zona è una regione del documento che contiene una quantità arbitraria di testo, come ad esempio il titolo o l’abstract, come si possono gestire? Costruendo degli indici invertiti!

Una volta ottenuti gli indici invertiti di ogni zona, si codificano tutti in un dizionario, come possiamo vedere nella foto a destra.

## Indici a (più) livelli

Tornando a parlare degli indici a livelli, possiamo rompere i posting in una gerarchia di liste in ordine di importanza.

Queste liste possono essere ordinate usando g(d) o un’altra metrica.

Per gestire la query, si utilizzano i livelli più importanti per trovare i K documenti migliori, andando verso quelli meno importanti in caso di fallimento nei primi.

## Query di prossimità

Dato che gli utenti preferiscono documenti i cui termini della query siano più simili a esso, una soluzione sono le free text query, cioè degli insiemi di termini che stanno in una query box, tipico della ricerca web.

Le free text query possono corrispondere a più query, bisogna trovare quindi un modo per parsificarla, come? In questo modo:

* Eseguiamo la query come una phrase query;
* Se ci sono meno di K documenti che contengono la frase, si eseguono due phrase query contenenti metà della frase ciascuno;
* Se anche qui ci sono meno K documenti, si esegue la query originale in spazio vettoriale;
* Si calcola lo score dei documenti ottenuti.

## Score aggregati

C’è un modo per combinare coseno, qualità statica, prossimità, eccetera in una sola funzione? Sì ma non tutte le combinazioni risultano utili allo scopo, qual'è la migliore? Dipende, in molte applicazioni questa è l’expert-tuned, sta però diventando comunque il machine learning in questi contesti.

# Valutazione dei motori di ricerca

Come si valuta un motore di ricerca? Per farlo ci sono diverse metriche:

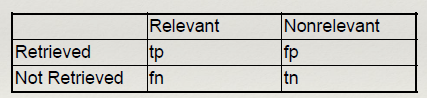
* La velocità di indicizzazione in documenti all’ora;
* La dimensione media dei documenti;
* La velocità di ricerca in termini di latenza;
* L’espressività del linguaggio delle query, quindi l’abilità di esprimere informazioni complesse e la loro velocità di esecuzione;
* L’interfaccia utente;
* Altre feature minori.

Tutti questi criteri sono in un modo o nell’altro misurabili, ma c’è una misura chiave: la user happiness, cioè il grado di soddisfacibilità di un utente!

Come possiamo misurarla? Dipende, nel caso di una ricerca su internet può essere quanto tempo ci mettiamo a trovare quello che ci serve, tuttavia ciò dipende dalla situazione.

La user happiness è proprio per questo motivo una misura elusiva, se ci concentrassimo sulla rilevanza delle informazioni? Come potremmo fare per misurarle? La misuriamo con tre elementi:

* un benchmark sulla collezione di documenti;
* uno sulla suite di query;
* una valutazione (di solito binaria) tra ciò che è rilevante e ciò che non lo è, questo per ogni query e per ogni documento.

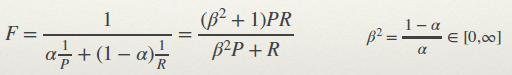
Fatto ciò, come si procede? Si compila una tabella 2x2 in cui nelle righe ci sono i documenti recuperati e non, nelle colonne invece la rilevanza, quindi possiamo indicare ogni cella come vediamo nella foto a destra.

Da questa tabella calcoliamo le seguenti misure come segue:

La misura precisione/recall permette di valutare un sistema di information retrieval, in un buon sistema infatti la precisione decresce al crescere del recall. Il recall infatti è una funzione non decrescente sul numero di documenti recuperati.

Questa misura tuttavia ha un continuo bisogno di valutazioni (binarie) da parte di persone ed è troppo dipendente dalla collezione tant’è che non si possono convertire i risultati da un dominio all’altro.

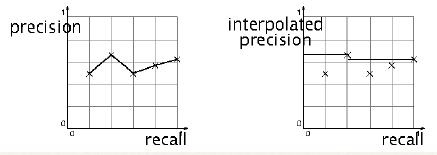
Una misura combinata è la seguente:



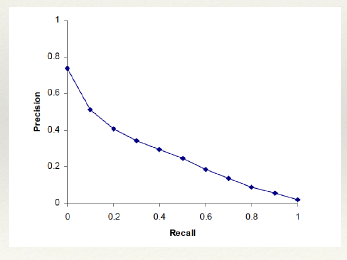
La F è molto utilizzata nella sua versione bilanciata, cioè ad esempio quanto beta è uguale a 1. Inoltre la media armonica è conservativa.

## Come si valutano i risultati?

Per valutare i risultati, si utilizza la curva di precisione-recall, considerarlo per una sola query non sensibilizza abbastanza, quindi serve un certo numero di query per avere un’idea sulle performance medie. C’è però un problema: il calcolo della precisione-recall piazza solo qualche punto sul grafo, c’è un modo per trovare i punti intermedi?

L’idea è quella di interpolare tra coppie di punti, tuttavia questo approccio potrebbe non essere preciso, quindi si fa quel che si può.

Gli utenti però vogliono misure sommarie! Quindi che si fa?

* si utilizza la precisione su un livello di retrieval fisso, molto appropriato per le ricerche su internet ma in media va male con un k arbitrario;
* si utilizzano 11 punti di precisione in cui 11 livelli di recall variano tra 0 e 1 in base al documento, dopodichè si interpola e si calcola la media;
* si valutano le performance di tutti i livelli di recall.

Una buona curva a 11 punti è quella nella foto a destra.

## Misure basate sul ranking

* la precisione su K indica la percentuale di documenti rilevanti su K;
* la precisione media è la media di ogni precisione su K per ogni K;

Nella precisione media:

* se un documento non viene mai recuperato, possiamo assumere che la sua precisione corrisponda a zero;
* c’è equità tra le query;
* assumiamo che l’utente sia interessato a trovare più documenti rilevanti per ogni query;
* richiede più giudizio rilevante in una collezione di testi.

## Valutazioni di grandi motori di ricerca

Nei grandi motori di ricerca ci sono migliaia di documenti e migliaia di query, di conseguenza cose come il recall sono difficili e onerosi da calcolare. Proprio per questo motivo vengono utilizzati approcci differenti, come ad esempio misure di rewarding in base al ranking e/o misure di non rilevanza.

Un metodo per valutare questo tipo di motori di ricerca è l’A/B testing, il quale consiste nel testare una singola innovazione.

Supponendo che la maggior parte degli utenti utilizzi un sistema vecchio, questo metodo dirotta l’1% del traffico sui sistemi nuovi che includono l’innovazione, quindi utilizzare una misura "automatica" basata sui click della prima ricerca.

Dopo ciò si può verificare se l’innovazione migliora la user happiness o meno.

## Presentazione dei risultati

I risultati possono essere mostrati all’utente attraverso degli snippet, mettendoli in ordine di rilevanza e fornendo un piccolo riassunto del contenuto, come avviene nei motori di ricerca.

I riassunti possono essere:

* statici quando il documento è sempre lo stesso indipendentemente dalla query;
* dinamici quando spiegano perché quel dato documento è stato recuperato.

Il riassunto statico è solitamente un sottoinsieme di documenti, nella più semplice delle euristiche si prendono le prime 50 parole, in quelle più complesse invece si estraggono delle frasi chiave oppure si utilizza il Natural Language processing.

I riassunti dinamici invece presentano più finestre in cui sono contenute informazioni differenti, una tecnica per computarli è la seguente:

* trovare un piccola finestra nel documento che contenga i termini della query, ciò richiede un lookup veloce nella cache del documento;
* calcolare lo score di ogni finestra, utilizzando varie feature e combinarle in una funzione;
* giudicare il riassunto;

# Quanto possiamo matchare una ricerca dell’utente?

Dal momento che il nostro obiettivo è capire la query e non vedere se due stringhe sono uguali, un puro keyword-match non sarebbe di aiuto. Certo abbiamo cose come la spelling correction e lo stemming, ma cosa serve effettivamente per capire il match tra query e documento?

* Con la query expansion, il feedback di rilevanza ci permetterebbe di farlo se il match tra documenti e parole è abbastanza vicino. Inoltre possiamo usare la similarità tra parole, sinonimi e una metrica per misurarle;
* Con la document expansion, utilizziamo degli anchor per provare a risolvere il match utilizzando dei sinonimi, tuttavia non è adatto a pagine web nuove, meno popolari, o non linkate a collezioni.

## Ricerca per log query expansion

La query expansion in ambito context free è problematica dato che non sempre le parole assumono un dato significato in contesti diversi, basti pensare a questi esempi:

* se scrivo le query “wet ground” e “wet earth”, i risultati saranno simili dato che ground in questo caso ha lo stesso significato di earth;
* se scrivo invece “ground coffee” e “earth coffee” i risultati saranno differenti, infatti ground non ha il significato di earth in questo caso.

Come possiamo fare ciò? Quel che si può fare è riscrivere la query in un modo specifico rispetto al contesto, sarà poi l’utente a fare più o meno ricerche in base ai suoi bisogni.

## Generazione di thesaurus

Per poter fare ciò, occorre analizzare una collezione di documenti, qui una nozione fondamentale è quella di similarità tra due parole, infatti:

* due parole sono simili se occorrono insieme ad altre parole simili;
* due parole sono simile se occorrono con una relazione grammaticale con le stesse parole.

La prima definizione che abbiamo scritto è quella più robusta, la seconda invece è quella più accurata, perchè?

Il modo più semplice per produrre un thesaurus di co-occorrenza (prima definizione) si basa sulla similarità tra due termini in dove A è la matrice termine-documento. Nella matrice A, il termine w(i,j) indica il peso del termine ti sul documento dj.

La matrice C è il prodotto tra A e il suo trasposto e ogni cella C(i,j) indica il numero di documenti che contengono sia il termine i, sia il termine j.

## Come rappresentiamo le relazioni tra termini?

Con la codifica standard dei termini, ogni termine rappresenta una dimensione, inoltre termini differenti non presentano similarità:



C’è un modo per apprendere direttamente le relazioni tra termini? Dal momento che lo score di un sistema IR base si calcola col prodotto , possiamo provare ad apprendere i parametri W in questo modo.

Il problema però rimane il solito, cioè la sparsità, specie quando W è grande.

Esiste un modo migliore per farlo? L’idea sarebbe quella di apprendere una rappresentazione densa e con poche dimensioni di una parole in in modo che il prodotto esprima similarità tra parole. Anche qui possiamo includere una traduzione da matrice a dizionario, ottenendo e rendendo la matrice W piccola!

## Rappresentazione basata sulla similarità

Possiamo ottenere molti valori rappresentando una parola con il significato delle sue vicine, questo approccio è molto utilizzato nel NLP statistico!

## Low dimensional vector

L’idea di questa soluzione è quella di memorizzare la maggior parte delle informazioni in un vettore denso con un fisso e piccolo numero di dimensioni.

Come riduciamo la dimensionalità? Si parte da un vettore grande e sparso e da ciò si ottiene il low dimensional vector attraverso il word embedding.

## Latent semantic indexing/analysis

Questo è il cosiddetto metodo tradizionale e consiste in una sorta di PCA per una matrice rettangolare arbitraria, oppure in una proiezione casuale per trovare dei vettori ortogonali.

## Neural Embedding

Un buon metodo per la predizione delle parole nel loro contesto è quello di costruire un vettore denso per ogni tipo di parola.

Ma essenzialmente, qual è l’idea?

L’idea è quella di definire un modello che permetta di prevedere una parola centrale wt e trovare tutte le altre in base al contesto.

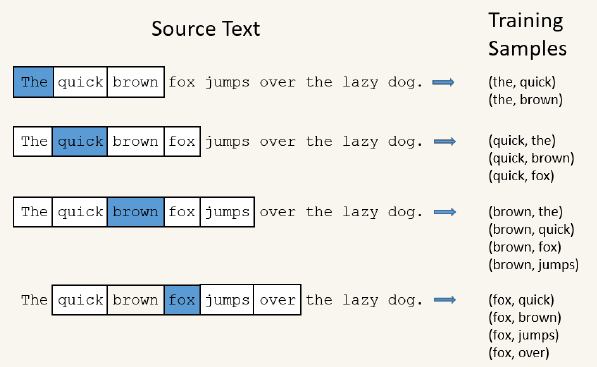
Possiamo quindi calcolare una loss function dalla funzione per il calcolo delle parole:

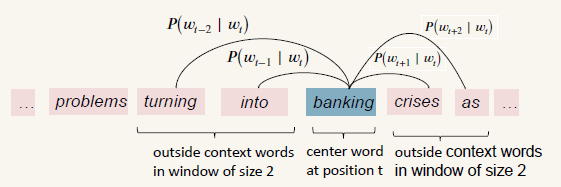
Una nuova idea è quella di apprendere direttamente questi vettori basandosi sull’abilità di previsione, una famiglia di algoritmi che segue questo ragionamento è il word2vec che consiste in:

* due algoritmi, cioè lo skip-gram, il quale predice parole di contesto dato un target, e il continuous bag of words, che sostanzialmente fa il contrario (predice il target in base a delle parole di contesto date in input);
* due metodi di training, il softmax gerarchico e il sampling negativo. A questi si aggiunge anche il Naive softmax ma non è molto utilizzato.

## Word2Vec e Skip-gram

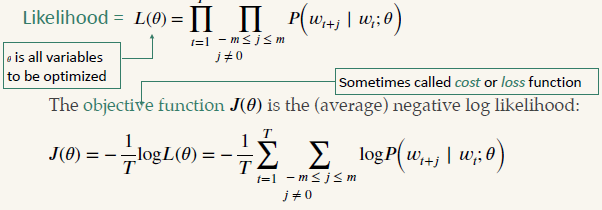
L’idea dello skip-gram è quello di calcolare le parole considerando una finestra di parole (di lunghezza arbitraria) di cui wt è quella centrale.

A ogni iterazione dell'algoritmo si ricavano delle coppie di parole che andranno a formare i training sample.



Quello che facciamo è predirre le parole di contesto all’interno di una finestra grande m data una parola centrale wj, questo per ogni posizione t.

Da qui possiamo calcolare la likelihood e la funzione obiettivo:



L’obiettivo è quindi minimizzare J(omega), ma prima di fare ciò, come si calcola P(wi+j,wi,omega)?

Utilizziamo due vettori per la parola w:

* v(w) si utilizza quando w è centrale;
* u(w) quando è di contesto.

Quindi, considerando una parola c centrale e una o di contesto:



Precisamente al denominatore si utilizza l’esponenziale per rendere tutto positivo, il prodotto dot invece serve per calcolare quanto sono simili le parole o e c. Al numeratore si effettua una normalizzazione sull’intero vocabolario per ottenere una distribuzione di probabilità.

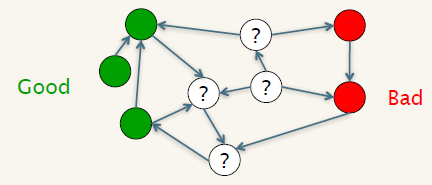
Questa funzione qui è un esempio di softmax:

* max perchè amplifica la probabilità dell’input xi più grande;
* soft perchè assegna delle probabilità agli input xi più piccoli.

## Relazione lineare nel word2vec

Queste rappresentazioni risultano molto buone nella codifica della similarità e delle sue dimensioni, infatti il testing sulle dimensioni può essere risolto sottraendo i vettori nell’embedding space.

# Link e ipertesti

Come gestiamo link e ipertesti? I link infatti sono ovunque e sono una potente sorgente di autenticità e autorità. Bisogna però trovare un modo per trovare quali link sono buoni e quali sono cattivi. Consideriamo il seguente esempio: consideriamo tre tipologie di nodi:

* i nodi buoni, i quali non puntano mai a nodi cattivi;
* i nodi cattivi, i quali possono essere puntati solo da nodi cattivi;
* i nodi sconosciuti.

Come possiamo capire il tipo dei nodi sconosciuti? Come possiamo notare dalla foto a destra, possiamo organizzare i link come in un grafo, quindi applicare una strategia di ricerca in cui implementiamo la nostra logica.

Il nostro interesse primario riguarda l’analisi dei link in modo da:

* effettuare lo scoring e il ranking;
* effettuare un clustering basato sui link, quindi dare una struttura topologica ai documenti;
* classificare i documenti usando i link come feature, infatti i documenti che puntano su uno stesso documento sono probabilmente nella stessa classe;
* effettuare il crawling.

## Link nel web

Possiamo considerare il web come un grafo diretto, infatti le pagine web possono linkarne altre ma queste ultime potrebbero non linkare la pagina di partenza. Ci sono due assunzione da considerare:

* un link tra pagine denota un conferimento di autorità;
* un testo ancorato di un link a una pagina ne descrive il target.

## Indicizzare un testo ancorato

L’indicizzazione di un documento D include i testi ancorati dei link che puntano a D, ciò può portare a effetti inaspettati come il fallimento nello spam, tuttavia possiamo dare uno score ai testi ancorati con dei pesi in base all’autorità degli anchor della pagina web.

## Connectivity server

I connectivity server sono un metodo per ottenere tutte le informazioni relativi ai link senza costi, essi supportano:

* query veloci sul web graph;
* memorizzano i mapping dagli URL;
* permettono l’analisi dei link, del web graph e il controllo sul crawl.

## Liste di adiacenza

La lista di adiacenza indica l’insieme di tutti i vicini di un nodo, quindi assumendo che ogni URL sia rappresentato con un intero, possiamo gestire i link in questo modo.

Questo approccio però richiede 64 bit per ogni link, esistono un modo migliore? Sì ed è Boldi/Vigna, il quale richiede mediamente 3 bit per ogni link.

Come possiamo comprimerle?

* possiamo sfruttare la similarità tra liste;
* la località, infatti tanti link da una pagina portano a pagine “vicine”;
* Usare il gap encoding nelle liste ordinate;
* Distribuzione di valori gap.

Consideriamo una lista di URL ordinati lessicograficamente, ognuno ha una lista di adiacenza. L’idea principale è quella che, a causa dei template, la lista di adiacenza di un nodo è simile a quelle dei 7 URL precedenti nell’ordinamento lessicografico.

## Gap encoding

Data una lista di interi ordinata rappresentata usando i gap tra elementi adiacenti, la compressione avviene usando dei codici:

* un codice gamma meno il numero di bit;
* un codice delta;
* un codice s, utile per le potenze.

I principali vantaggi di Boldi/VIgna riguardano la sola dipendenza nella località dell’ordine, l’adiacenza delle query può essere risolta in modo efficiente ed è facile da implementare.

## Page ranking nell’analisi dei link

Una soluzione per fare il page ranking in questo contesto è la citation analysis, tuttavia non si adatta bene al web dato che ci sono milioni di partecipanti, ognuno coi suoi interessi. Inoltre c’è troppo spam e il ranking non fa altro che aumentare questo fenomeno. Per quest’ultimo problema si può entrare in una link farm, un gruppo di siti web che link ad altri.

Come si assegna lo score del pagerank?

Immaginiamo un utente che naviga causalmente un sito web (ragionando tipo una random walk):

* Si comincia da una pagina random;
* a ogni step si va dalla pagina corrente verso le altre in modo equiprobabile.

Nelle esecuzioni lunghe ogni pagina ha un indice di visita, quindi possiamo utilizzare quello come score.

Un ragionamento del genere però porta a dead-end e inoltre non ha senso nel lungo termine.

In caso di dead end, una soluzione è il teleporting, il quale consiste nel saltare a una pagina random. Nelle situazioni normali invece si utilizza il teleporting solo con una probabilità del 10%, nel restante 90% si procede come già spiegato.

Come gestiamo l’indice di visita? un possibile approccio è l’utilizzo delle Markov chain, il loro comportamento infatti permette di astrarre la random walk.

Il nostro interesse ricade particolarmente nelle Markov chain ergodiche, cioè quello in cui non sono presenti cicli tra due stati.

Le Markov chain ergodiche permettono di implementare l’indice di visita a lungo termine in ogni suo stato, infatti lungo questo periodo visitiamo ogni stato in proporzione allo stesso indice.

## Vettori di probabilità

Un vettore di probabilità x è un vettore che indica a che punto si trova la random walk in ogni punto, precisamente indica che ci si trova allo stato i con probabilità xi.

Come possiamo ricavare il prossimo step da un vettore di probabilità? utilizziamo la matrice di probabilità di transizione P, quindi:

x=x\*P

## Hyperlink-induced Topic Search (HITS)

L’approccio HITS è il seguente: invece di utilizzare una lista ordinata di pagine, utilizziamo due set:

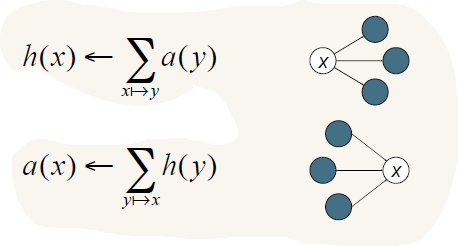
* le pagine hub, cioè buone liste di link su un soggetto;
* le pagine di autorità, quelle che occorrono in modo ricorrente sui buoni hub per il soggetto.

Una pagina hub è buona per un topic se punta a molte pagine autoritative per quello stesso topic.

Una pagina autoritativa è buona se è puntata da tante pagine hub buone per lo stesso topic.

E’ possibile estrarre dal web un set base di pagine che potrebbero essere buon hub e autorità, da qui si identificano un sottoinsieme di pagine indicanti le pagine migliori.

Come si ottiene il set base?

* utilizziamo indici di testo per ottenere tutte le pagine che contengono una text query data in input, questo è il root set;
* A questo set si aggiungono ogni pagina che punta a una pagina della root set o che è puntata da quest’ultima.

Per ogni pagina x del set base, calcoliamo un hub score h(x) e un authority score a(x):

* come inizializzazione, poniamo h(x) e a(x) a 1;
* Dopodiché aggiorniamo questi valori iterativamente.

Le pagine col hub score più alto saranno le top hub, quelle invece col miglior authority score saranno le top authority.

## Scaling

Onde evitare che h() e a() diventino troppo grandi, possiamo scalarli a ogni iterazione. Il fattore di scaling non è importante, quello che ci interessa infatti sono i valori relativi dello score.

I valori relativi degli score convergono in poche iterazioni dal momento che h() e a() sono steady state. In generale, circa 5 iterazioni vanno più che bene per avere stabilità.

### Dimostrazione di convergenza

Data una matrice di adiacenza A n x n, possiamo riscrivere le funzioni h() e a() in forma matriciale:

h e a sono eigenvector delle rispettive matrici, per computarle si utilizza il power interation (che appunto, converge).