UNIVERSITÀ DI PISA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Dipartimento di Informatica

Tesi di Laurea Specialistica in
INFORMATICA

Studio e Comparazione di Algoritmi per Query Recommendation in Motori di Ricerca Web

Relatori:

Dott. Fabrizio Silvestri

Ing. Franco Maria Nardini

Candidato:

Daniele Broccolo

Anno Accademico 2008-2009

Ai miei genitori,
Angelino e Giuseppina

Indice

Capitolo 1

Introduzione

1.1 Query Recommender System

La crescita del Web degli ultimi anni, con l'aumento delle informazioni presenti in esso, ha reso disponibile agli utenti della rete, una grande quantità di informazioni. Per un utente, riuscire ad orientarsi è diventato sempre più difficile rendendo necessario l'utilizzo di sistemi complessi per la ricerca dei contenuti. In questo contesto i motori di ricerca Web hanno assunto un ruolo sempre più importante e sono ormai diventati indispensabili, tanto che, uno studio del 2006 [35] ha stimato che ogni giorno nei vari motori di ricerca vengono sottomesse circa 213 milioni di richieste.

La sfida principale per chi si occupa di migliorare questo tipo di applicazioni è quindi sia quella di velocizzare la ricerca in quanto l'utente necessita di informazioni in tempo reale, sia quella di riuscire a rispondere con una lista ordinata di documenti che siano quanto più possibile vicini alle esigenze dell'utilizzatore del servizio.

Per ottenere un tale risultato è necessaria un'indicizzazione estesa dei contenuti del Web ed una funzione di ordinamento dei risultati (*ranking function*) che tenga conto dell'attività degli utenti e delle informazioni contenute nei documenti.

Sfruttare l'attività passata degli utenti del motore di ricerca, è un metodo molto efficace per trovare possibili relazioni tra utenti e contenuti ed è resa possibile grazie alle informazioni contenute nel *Query Log*, nel quale sono archiviate tutte le interazioni con il sistema.

Una delle tecniche che recentemente si è integrata con la ricerca nel Web al fine di aiutare l'utente è l'utilizzo di sistemi di suggerimento (*Recommender System*) che

Figura 1.1: Recommender System del motore di ricerca Yahoo!

propongono informazioni potenzialmente interessati che possano migliorare e velocizzare l'attività dell'utente.

Sistemi di questo tipo sono stati utilizzati sia per il suggerimento di URL che di query, per aiutare l'utente ad interagire con i motori di ricerca, o per le esigenze più disparate, dal suggerimento di musica e video [1], al suggerimento di news e libri[2], ma anche per suggerire immagini, articoli e quant'altro.

Inizialmente i Recommender System hanno avuto enorme successo in applicazioni tipo l'*e-commerce* con lo scopo di proporre articoli commerciali interessanti all'acquirente e migliorare quindi le probabilità di acquisto.

Questo lavoro di tesi si concentra sui Query Recommender System (di cui un esempio viene riportato in figura 1.1), cioè dei particolari sistemi di raccomandazione, specifici per motori di ricerca, che suggeriscono all'utente delle possibili query semanticamente correlate alle richieste che ha già sottomesso.

In particolare verrà affrontato il problema della valutazione della qualità degli oggetti suggeriti. Per il caso particolare dei *Query Recommender System*, infatti, la letteratura è priva di confronti tra algoritmi di tipologie diverse ed inoltre spesso i test di qualità non vengono effettuati in maniera automatizzata, ma le valutazioni vengono svolte manualmente dagli stessi autori (*user-study*).

Le motivazioni alla base della lacuna di valutazione sono da ricercare nella difficoltà di capire le volontà dell'utente, che spesso si esprime utilizzando query brevi e spesso con termini ambigui. Inoltre, molto di frequente, l'utente ha una scarsa conoscenza di ciò che cerca, ed utilizza quindi un linguaggio vago ed approssimativo.

[1] Alcuni esempi sono Last.fm e iLike per la musica e www.youtube.com per i video
[2] come ad esempio Reddit e Daily Me

1.2 Obiettivi

Come accennato nel paragrafo precedente, uno dei problemi degli algoritmi di *Query Recommendation* è la valutazione dell'efficacia.

I sistemi di valutazione utilizzati in letteratura, come lo sfruttamento della similarità sintattica, sono in genere poco significativi o non sono automatizzati rendendo le valutazioni piuttosto soggettive e non utilizzabili con grandi quantità di dati.

Nonostante questo, nel corso degli anni, sono state presentate diverse tecniche per il suggerimento di query che sfruttano diverse tra le possibili informazioni a disposizione, ma non è ancora chiaro, quale di queste può essere considerata migliore.

Ci proponiamo quindi di analizzare e implementare alcune delle soluzioni proposte in letteratura per poi ideare un sistema di valutazione automatizzato, capace di valutare i diversi algoritmi con metriche diverse, con il quale poter ottenere dei risultati oggettivi sull'efficacia di tali algoritmi.

Proponiamo, inoltre, alcuni algoritmi di suggerimento basati su un modello incrementale, come variazione a quelli proposti ed analizzati in letteratura e ne testeremo le prestazioni, in termini di efficacia, comparandoli con quelli che non fanno uso di tale modello.

Gli obiettivi principali di questa tesi possono essere quindi riassunti in tre punti:

- Definizione di una metodologia per la valutazione degli algoritmi di recommendation;
- Progettazione e implementazione di un *framework* per facilitare la realizzazione e la successiva valutazione dei Query Recommender System;
- Progettazione implementazione e valutazione di Query Recommender System incrementali.

Capitolo 2

Recommender System

2.1 Definizione

L'idea di Recommender System nasce nella prima metà degli anni '90 quando viene introdotto il concetto di *Collaborative Filtering* in contrapposizione al tradizionale *Content-based Filtering*. L'intento era quello di utilizzare i giudizi degli utenti in un sistema di *filtering* e sfruttare i *newsgroup* per condividere le informazioni tramite mail [19].

Il termine più generale di *Recommender System* viene coniato successivamente [32] per indicare che tali sistemi non si basano su una esplicita collaborazione di utenti e che gli oggetti (*item*) suggeriti vengono aggiunti alle informazioni filtrate.

Il concetto di Recommender System si è evoluto nel tempo e una definizione più attuale è la seguente:

Definizione 1. Recommender System è l'area di ricerca che si propone di risolvere il problema del *Recommendation Problem* che può essere formalizzato come segue [1]:

Sia C l'insieme degli utenti e S l'insieme degli *item* che possono essere suggeriti. Sia u una funzione di utilità che misura quanto è interessante un *item* s per l'utente c, ad esempio, $u : C * S \rightarrow R$, dove R è un insieme totalmente ordinato, allora per ogni utente $c \in C$ vogliamo scegliere alcuni *item* $s' \in S$ che massimizzano la funzione di utilità:

$$\forall c \in C, s'_c = argmax_{s \in S} u(c, s)$$

Il problema fondamentale dei recommender system è che la funzione di utilità u non è in genere definita in tutto il dominio $C * S$ ma solo in un sottoinsieme. Questo significa

che tale funzione deve essere estrapolata in tutto lo spazio $C * S$. Nei recommender system la funzione u è generalmente rappresentata dai giudizi degli utenti. Il sistema quindi deve essere in grado di stimare quanto un oggetto può interessare ad un particolare utente. Tale informazione può essere estratta con:

1. specifiche euristiche che definiscono la funzione u;
2. costruire la funzione di utilità per ottimizzare alcuni criteri di performance.

Una volta definita la funzione di utilità, i valori che essa assume, possono essere valutati con differenti metodi messi a disposizione sia dal *machine learning* che dall'*approximation theory* o, utilizzare delle euristiche.

2.2 Tipi di Recommender System

I sistemi di suggerimento sono in genere classificati in base all'approccio che viene utilizzato per generare i suggerimenti:

Content-based recommendation: All'utente vengono proposti *item* che risultano essere simili a quelli utilizzati da lui in passato;

Collaborative recommendation: All'utente vengono proposti *item* che sono stati contrassegnati come buoni da utenti simili;

Hybrid recommendation: Questo metodo combina i due precedenti.

2.2.1 Content-based recommendation

In questo tipo di sistemi, la funzione di utilità $u(c, s)$ rispetto all'*item* s per l'utente c è stimata in base alla funzione stessa calcolata su un altro *item* ($s_i \in S$) che è simile ad s.

L'approccio content-based utilizzato nei Recommender system, ha le sue radici nei sistemi di *information retrieval* [31] e *information filtering* [8], infatti, molti dei sistemi che suggeriscono oggetti, si basano sull'informazione testuale contenuta in essi. Il miglioramento rispetto ai tradizionali metodi sono dati dall'utilizzo di informazioni relative al profilo degli utenti, tra le quali il comportamento e i gusti.

Le informazioni che il sistema utilizza possono essere ottenute sia esplicitamente, chiedendo all'utente di compilare questionari, oppure implicitamente, analizzando il comportamento.

Questo tipo di sistema è particolarmente indicato per suggerire *item* dove il contenuto è solitamente composto da parole chiave. Una delle più importati misure utilizzate in questo contesto è la *TF-IDF* descritta nell'appendice A.1.

Il limite di questa tecnica sono le quantità di informazioni in genere associate ad un *item*, che possono essere troppo poche qualora i suggerimenti non siano composti da testo come ad esempio immagini. Un altro problema è quello di suggerire *item* a nuovi utenti; infatti il sistema non è in grado di capire le preferenze dell'utente ed è poco probabile che gli vengano offerti buoni suggerimenti.

2.2.2 Collaborative recommendation

Diversamente dai sistemi basati sul contesto, i sistemi collaborativi, suggeriscono basandosi sulle scelte fatte da altri utenti. Più formalmente l'utilità di un *item* s ($u(c, s)$) è stimata basandosi sul valore che assume $u(c_j, s)$ dove c_j è un altro utente in C simile a c.

Questo tipo di algoritmi possono essere classificati in due sottoclassi: *Memory-Based* e *Model-Based*.

Gli algoritmi *Memory Based*, sono essenzialmente euristiche che predicono valutazioni in base all'intera collezione degli *item* precedentemente utilizzati. Il valore delle valutazioni $r_{c,s}$ per l'utente c e *item* s viene computato utilizzando le valutazioni degli altri utenti per l'*item* s.

$$r_{c,s} = aggr_{c' \in C} r_{c',s}$$

dove C denota il set degli N utenti che sono i più simili a c e che hanno valutato l'*item* s. La funzione $aggr$, nei casi più semplici, può essere una semplice media o una media pesata. La funzione di similarità tra utenti, invece, è essenzialmente una misura di distanza basata sulle preferenze dell'utente. Gli approcci più comuni per il calcolo di tale funzione sono correlazione (si veda la voce *Indice di correlazione di Pearson* in appendice A.7), similarità basata sul coseno (Appendice A.2) o differenza quadratica media [34].

In particolare è da notare che le misure di distanza impiegate sono le stesse utilizzate nell'approccio content-based, quello che cambia è l'informazione, qui si utilizzano le preferenze dell'utente, mentre nel content-based si utilizzano informazioni sui termini.

In contrapposizione al modello Memory-Based, il modello Model-Based, si propone di utilizzare la collezione delle preferenze degli utenti per costruire un modello che viene

poi utilizzato per predire le preferenze. Un esempio di questo tipo di approccio consiste nell'utilizzo di un algoritmo probabilistico [9] in cui la probabilità viene calcolata tramite un cluster o con reti Bayesiane.

Nel primo metodo, gli utenti vengono raggruppati in classi. Una volta trovata la classe di un utente, i suggerimenti dipenderanno dalla sua appartenenza ad uno specifico cluster. Nel metodo con reti Bayesiane invece ogni utente è un nodo in una rete Bayesiana, dove gli stati per ogni nodo, corrispondono al fatto che l'utente abbia o meno espresso la sua preferenza su determinati *item*.

Questa tipologia di sistemi è in genere affetta da alcuni problemi:

- proporre suggerimenti a nuovi utenti;
- proporre come suggerimenti nuovi item;
- la sparsità del modello.

2.2.3 Hybrid recommendation

Il sistema ibrido non è altro che la combinazione dei due metodi precedenti. I suggerimenti possono essere combinati in maniera diversa, sia utilizzando i due metodi in maniera indipendente e quindi aggregare i suggerimenti utilizzando una specifica funzione, sia aggiungendo ad uno dei due modelli, caratteristiche dell'altro. Un approccio di questo tipo viene utilizzato da Pazzani [12] che utilizza i dati ottenuti esplicitamente dagli utenti tramite una pagina web, per suggerire dei ristoranti.

Il metodo utilizza un *filtering* collaborativo basandosi sulle informazioni contenute nelle descrizioni dei locali (*Collaboration via content*) per cui gli utenti hanno sottomesso un giudizio. Nell'articolo viene inoltre fatta una comparazione dei risultati ottenuti con il metodo ibrido rispetto ai metodi presentati precedentemente (figura 2.1). Le valutazioni sono state effettuate dagli utenti stessi.

2.3 Utilizzare i Recommender System per migliorare la ricerca nel Web

I motori di ricerca memorizzano nei propri server, una grande quantità di informazioni. Tra queste ci sono tutte le query sottomesse dai vari utenti nel corso del tempo. Questi dati sono molto utili per capire il comportamento degli utenti e permettono di inferire

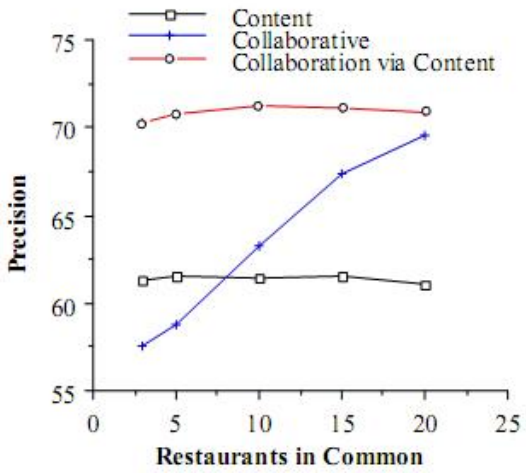

Figura 2.1: La precisione dei tre metodi di apprendimento rispetto alla sparsità dei dati [12].

conoscenza per suggerire in futuro informazioni potenzialmente interessanti. Infatti, se assumiamo che ogni operazione effettuata dall'utente sia significativa, allora le query inviate ai motori di ricerca sono una fonte, quasi infinita, di conoscenza implicita. Le informazioni in genere disponibili nel query log sono:

clientId: un codice identificativo dell'utente;

query: la query sottomessa;

timestamp: la data in cui viene eseguita la ricerca;

click-through: l'URL cliccato dall'utente dopo aver sottomesso la query;

scrolling behaviour: il comportamento dell'utente nella pagina dei risultati;

time on page: il tempo che l'utente rimane nella pagina dei risultati.

Il query log quindi può essere sfruttato per dare suggerimenti agli utenti, utilizzando alcune o tutte le informazioni disponibili o impiegare i dati a disposizione per ottenerne degli altri, come ad esempio servirsi degli URL per ottenere le informazioni contenute nei documenti cliccati o usufruire delle query per ottenere la lista dei risultati del motore di ricerca.

Tra i vari sistemi di raccomandazione, tratteremo in questa tesi quelli di Query Recommendation. Descriviamo quindi i due sistemi principali.

2.3.1 Query expansion

Query expansion è una tecnica di suggerimento che genera una nuova query aggiungendo termini alla query originale, termini che possono essere trovati sia in query precedenti che nei documenti relativi agli URL cliccati. Gli approcci al query expansion sono molteplici e possono essere classificati in diverse tipologie [29]:

1. la similarità tra termini viene calcolata basandosi sulla loro co-occorrenza. I termini vengono classificati settando un valore di soglia per la similarità e l'espansione della query viene effettuata aggiungendo tutti i termini contenuti nelle classi di appartenenza;

2. un altro metodo prevede l'utilizzo della classificazione tra documenti. Una volta classificati i documenti con un apposito algoritmo, i termini meno frequenti nelle classi dei documenti sono classificati come simili e assegnati alla stessa classe. I termini della query vengono quindi estesi con i termini appartenenti alle relative classi;

3. un metodo alternativo è quello di utilizzare il contesto sintattico. I termini vengono relazionati utilizzando le regole del linguaggio utilizzato. In questo ambito vengono sfruttate le regole grammaticali e il vocabolario della lingua. Ad ogni termine viene quindi associata una lista di altri possibili vocaboli che vengono poi usati per l'espansione;

4. un ultimo sistema prevede l'utilizzo delle informazioni attinenti alla ricerca. Queste informazioni sono utilizzate per costruire un'unica struttura implementativa come un dizionario o un *minimum spamming tree*. La bontà del metodo dipende in gran parte dalla pertinenza delle informazioni utilizzate per la costruzione della struttura. Per migliorare questa tecnica vengono in genere utilizzate metodologie per l'analisi dell'occorrenza di termini in relazione alla struttura dei documenti ricercati. Queste tecniche possono suddividersi in analisi globali o analisi locali a seconda se si utilizzano tutti i documenti o sono una parte di essi. É stato inoltre mostrato [39] che l'analisi locale è più efficiente della globale.

In aggiunta a questi sistemi automatici di espansione, sono stati sviluppati anche metodi semi-automatici che propongono all'utente diversi possibili termini per poter estendere la query e la scelta finale viene effettuata dall'utente.

2.3.2 Query suggestion

La tecnica del*Query suggestion* permette di sfruttare le informazioni delle precedenti query inviate degli utenti attraverso analisi statistica o con metodi di *data mining* del query log. A differenza del query expansion vengono suggerite solo query precedentemente sottomesse dagli utenti, e non ne vengono create di nuove.

Il processo può essere diviso in tre parti: in una prima fase i dati contenuti nel query log vengono preprocessati per identificare utenti, sessioni, pagine visitate o altro, in una seconda fase vengono utilizzati metodi specifici del *Data Mining*, come *clustering*, classificazione, *sequential pattern discovery*, per inferire conoscenza e in una terza fase, le informazioni ricavate vengono utilizzate per suggerire query all'utente.

Un primo approccio al problema è stato quello di associare tra loro query sintatticamente simili [40]. Il problema di tale approccio è che spesso le query utilizzano termini che possono essere ambigui, ad esempio la query "leopard" potrebbe indicare il felino, ma anche il famoso sistema operativo di casa Apple.

Un altro sistema è quello di utilizzare i risultati del motore di ricerca [30], e calcolare la similarità fra query usando le differenze nell'ordine dei documenti ritornati.

Queste prime tecniche non tengono però conto delle informazioni implicite che l'utente invia al motore di ricerca. Successivamente, Wen *et al.* [36] propone di clusterizzare le query per il suggerimento di URL, utilizzando quattro diverse nozioni di similarità fra query basate su:

1. Termini e frasi contenute nelle query;
2. *String Matching* tra i termini;
3. URL cliccati in comune;
4. distanza dei documenti relativi agli URL cliccati.

Gli algoritmi più significativi di questa categoria sono illustrati nel capitolo 3.

Capitolo 3

Algoritmi per Query Recommendation

Gli algoritmi per Query Recommendation utilizzano in genere informazioni estratte dal query log. In genere non tutti gli algoritmi sfruttano le stesse informazioni. Uno schema che descrive le diverse possibilità per scoprire relazioni tra query è realizzato in figura 3.1. Ogni metodo ha vantaggi e svantaggi che verranno descritti nel dettaglio per ogni

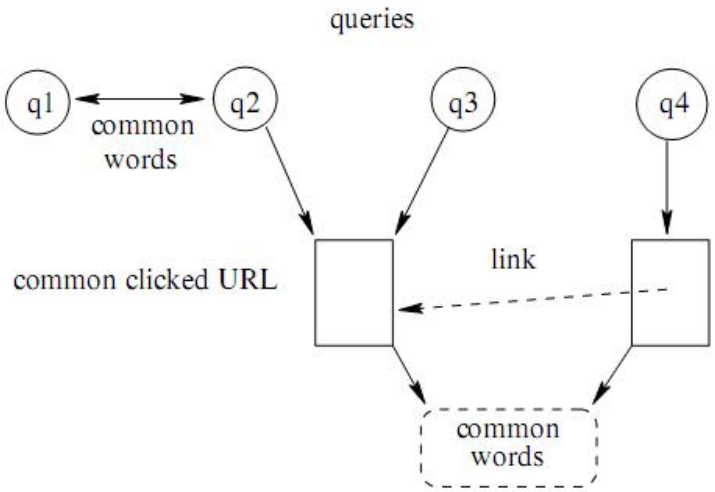

Figura 3.1: Differenti relazioni tra le query [3].

algoritmo.

3.1 Algoritmi basati sul mining delle sessioni

Gli algoritmi di questa tipologia, si propongono di capire le esigenze dell'utente analizzando il suo comportamento passato. Il problema di questo approccio deriva dalla difficoltà intrinseca di capire con certezza quali sono le sessioni utente. Spesso vengono utilizzate informazioni come indirizzo IP e timestamp, con il problema di scovare utenti che utilizzano *proxy* o *NAT*, oppure è possibile l'utilizzo di *cookie*, sempre con il problema che l'utente non abbia attivato la possibilità di riceverli o che li abbia cancellati nel tempo.

Un esempio di algoritmo basato su regole associative funziona in due fasi [17]:

1. *analisi del query log ed estrazione delle sessioni;*

2. *estrazione delle regole associative [24] e identificazione di query relazionate.*

Viene definita *sessione utente*, l'insieme di tutte le query inviate da un utente in un preciso intervallo di tempo t (ad esempio $t = 10$ minuti [17]).

Le sessioni vengono rappresentate tramite vettori. Sia $I = \{I_1...I_m\}$ il set delle query e siano T le sessioni. Una sessione $t \in T$ sarà un vettore dove $t[k]$ è uguale a 1 se la sessione t contiene la query k ($k \in [1..m]$) altrimenti sarà uguale a 0.

Una transazione t soddisfa X(sottoinsieme di I) se ogni query appartenente a X sta in t. Creiamo quindi le regole associative come $X \Rightarrow Y$ dove $X \subset I$ e $Y \subset I$ e $X \cap Y \neq 0$.

Per ogni regola associativa definiamo:

confidenza di $X \Rightarrow Y$ se nel $c\%$ delle transazioni che contengono X contengono anche Y;

supporto di $X \Rightarrow Y$ se nell'$s\%$ delle transazioni sono presenti $X \cup Y$.

Viene fissata quindi una soglia *MinSup* e vengono generate tutte le regole che la rispettano. Nel nostro caso siamo inoltre interessati ai soli casi in cui X e Y sono *singleton* cioè sono composti da un solo elemento.

Abbiamo quindi per ogni query una serie di regole associative ($\{I_k\} \Rightarrow I_1, \{I_k\} \Rightarrow I_2...\{I_k\} \Rightarrow I_n$) ordinate rispetto alla confidenza. Questo in genere è già sufficiente per dare dei suggerimenti.

3.2 Algoritmi basati su click-through

Il click-through è un'informazione molto significativa, in quanto deriva da scelte fatte dall'utente. Essendo un approccio *content-ignorant*[1], permette di trovare associazioni tra query anche se i documenti cliccati non contengono informazioni testuali. Più in generale un approccio di questo tipo permette:

1. di salvare pochi dati (non ci servono le pagine puntate dagli URL);
2. può essere utilizzato in alcuni casi in cui il *content-aware* non dà buoni risultati:
   (a) pagine senza testo (ad esempio basate su tecnologia flash);
   (b) pagine ad accesso limitato;
   (c) pagine con contenuto dinamico.

Uno svantaggio è quello che non possiamo sapere nulla sulle query sottomesse, per le quali non viene cliccato nessun documento.

Un primo approccio tra gli algoritmi che sfruttano click-through risale al 2000 [7]. L'idea è quella di costruire un grafo bipartito.

Un grafo bipartito è un grafo $G = (V, E)$ dove $V = V_1 \bigcup V_2$ e dove ogni arco in E connette un nodo in V_1 e uno in V_2 e dove non ci sono archi tra nodi dello stesse insieme. I nodi appartenenti ad un insieme corrispondono alle query e i nodi dell'altro corrispondono agli URL. Il grafo viene costruito connettendo una query ad un URL

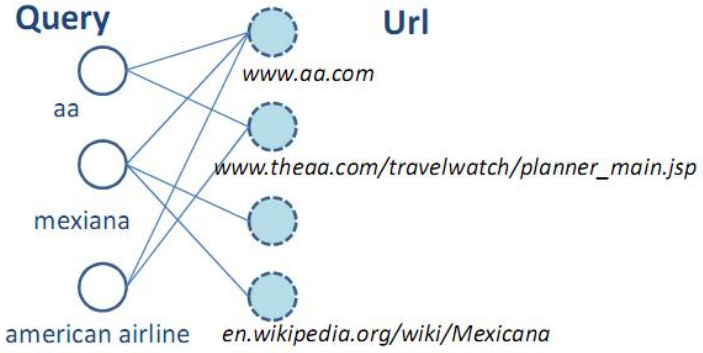

Figura 3.2: Esempio di un grafo bipartito Query-Url [26].

[1] per *content-ignorant* si intende un approccio che non utilizza le informazioni relative al contesto

qualora per la query, inviata al motore di ricerca, venga poi cliccato il particolare URL.

Una volta inserite le query nel modello, queste sono mappate nel grafo come nodi, la similarità può essere quindi computata, calcolando l'intersezione tra i nodi vicini.

La funzione di similarità utilizzata è la seguente:

$$\sigma(x,y) \stackrel{def}{=} \begin{cases} \frac{N(x)\cap N(y)}{N(x)\cup N(y)} & \text{se } |N(x)\cup N(y)| > 0 \\ 0, & \text{altrimenti} \end{cases}$$

dove $N(x)$ sono i vicini dell' nodo x La similarità tra nodi appartiene all'intervallo [0...1]. Per trovare gruppi di query viene utilizzato un *clustering* agglomerativo [20] tra tutti i nodi associati alle query. Il vantaggio di questa struttura è di riuscire ad utilizzare lo stesso metodo, per clusterizzare anche gli URL. La procedura di clustering è descritta di seguito:

1. *calcoliamo la similarità fra tutti i nodi delle query ed uniamo i 2 con la similarità più alta;*

2. *facciamo lo stesso con gli URL;*

3. *ricomincio dal punto 1 finché non ho finito:*

$$\max \sigma(q_1, q_2) = 0 \ \ e \ \ \max \sigma(u_1, u_2) = 0$$

dove q_1 e q_2 sono query del modello e u_1, u_2 sono URL presenti nel modello.

L'algoritmo si propone di clusterizzare finché il grafo risultante consiste di sole componenti connesse con una sola query e un solo URL. La condizione è abbastanza rigida e sono possibili alcune alternative tra cui l'utilizzo di una semplice ricerca in profondità.

Il vantaggio di clusterizzare query ed URL alternativamente risiede nel fatto che è possibile scoprire relazioni che inizialmente non erano visibili, come nell'esempio in figura 3.3.

Un altro algoritmo presentato nel 2007 [5] utilizza ancora un grafo bipartito Query-URL. Il grafo viene definito basandosi sulla nozione 3, vista nel paragrafo 2.3.2.

Questa volta ogni query viene vista come un punto in uno spazio vettoriale, dove ogni dimensione è un URL. Ad ogni componente del vettore diversa da zero, viene assegnato un peso, pari alla frequenza con cui il dato URL è stato cliccato. Due nodi (query)

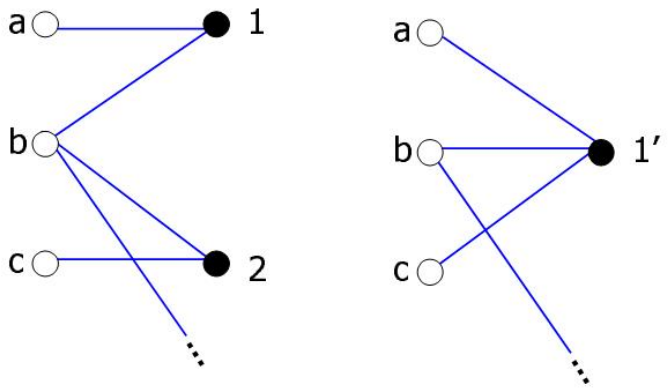

Figura 3.3: A sinistra: I vertici a e c hanno similarità $\sigma(a, c) = 0$. A destra: Dopo aver riunito due URL, i nodi a e c risultano essere quasi simili [7].

sono connessi *se e solo se* condividono almeno un URL u. Otteniamo così un grafo non orientato, dove ogni arco viene pesato con una *Cosine Similarity* calcolata sui vettori relativi alle query che connette. Quindi se $e = \{q, q'\}$ e lo spazio degli URL ha D dimensioni, il peso è dato da:

$$W(e) = \frac{\overline{q} \cdot \overline{q}\prime}{\| \overline{q} \| \| \overline{q}\prime \|} = \frac{\sum_{i \subseteq D} q(i) \cdot q\prime(i)}{\sqrt{\sum_{i \subseteq D} q(i)^2} \sqrt{\sum_{i \subseteq D} q\prime(i)^2}}.$$

Il grafo può essere migliorato utilizzando più tipi di arco:

Identical Cover $UC_{q_1} = UC_{q_2}$ L'arco non è orientato. Implica che i due nodi sono equivalenti;

Strict Complete Cover $UC_{q_1} \subset UC_{q_2}$ Arco orientato da q1 a q2;

Partial Cover $UC_{q_1} \cup UC_{q_2} \neq 0$ e non sono soddisfatte le precedenti.

Uno dei problemi di questo algoritmo è la sparsità del modello, le query che ne fanno parte infatti sono tutte quelle che sono state cliccate almeno una volta. Per poter diminuire la dimensione, l'autore utilizza un filtro sia sui nodi che sugli archi, eliminando le query con pochi click, e gli archi con il peso più basso. In questo modo si ha anche l'effetto di diminuire il rumore dei dati.

Un altro possibile miglioramento è dato dal riconoscimento degli URL che contengono informazioni relative a diversi argomenti (*Multitopical URL*). Questa tipologia di URL porta spesso ad avere relazioni molto deboli in quanto l'URL utilizzato è condiviso da query poco correlate semanticamente. Per risolvere questo problema viene utilizzata un'euristica, in cui vengono eliminati gli URL che sono più spesso implicati in relazioni deboli.

Un successivo algoritmo [26] utilizza ancora un modello a grafo bipartito tra query e URL, ma questa volta, il grafo viene attraversato utilizzando un *random walk* partendo dal nodo per il quale deve essere effettuato il suggerimento, per poi percorrere un arco con probabilità proporzionale al suo peso:

$$p_{ij} = \frac{w(i,j)}{d_i}$$

dove $w(i,j)$ è il peso dell'arco tra i e j mentre

$$d_i = \sum_{j \in V_2} w(i,j).$$

Ovviamente se si è interessati ai vertici di una sola delle due parti del grafo bipartito, la probabilità deve essere calcolata come:

$$p_i = \sum_{k \in V_2} \frac{w(i,k)}{d_i} \frac{w(k,j)}{d_k}.$$

Definiamo ora il concetto di *Hitting Time*:

sia A un sottoinsieme di V, denotiamo con X_t la posizione del *random walk* al tempo t. Definiamo *Hitting Time* T^A il momento in cui il cammino tocca per la prima volta un nodo appartenente ad A quindi: $T^A = min\{t : X_t \in A, t \geq 0\}$. Definiamo inoltre h_i^A il valore ottenuto da T^A considerando che si parte dal nodo i. Calcoliamo quindi l'*Hitting Time* dal seguente sistema lineare che ha unica soluzione:

$$\begin{cases} h_i^A = 0 & \text{se } i \in A \\ h_i^A = \sum_{j \notin A} p_{ij} h_j^A + 1 & \text{se } i \notin A \end{cases}$$

Quindi dato il grafo Query-URL (ma anche un altro tipo di grafo) si procede come segue:

1. *Sia Q_T la query sottomessa dall'utente;*

2. *Poniamo* $A = \{Q_T\}$ *e computiamo* $h^A(i)$ *per tutte le altre query* Q_i *;*

3. *Restituiamo le prime k.*

Questo metodo è computazionalmente oneroso sia per la dimensione del grafo, sia per la computazione del sistema lineare.

L'algoritmo è stato quindi modificato per ottenere migliori complessità computazionali. Il risultato è il seguente:

1. *Data una query, si costruisce un sottografo tramite una ricerca in profondità (DFS). La DFS termina quando vengono raggiunti un numero massimo di query (valore predefinito);*

2. *Si implementa un* random walk *nel sottografo calcolando* p_{ij} *tra le query (non vogliamo mischiare URL e Query):*

$$p_{ij} = \sum_{k \in V_2} \frac{w(i,k)}{d_i} \frac{w(i,k)}{d_k}$$

3. *Per ogni query eccetto la prima si itera la seguente formula*

$$h_i(t+1) = \sum_{j \neq s} p_{ij} h_j(t) + 1$$

per un certo numero m di volte , iniziando da $h_i(0) = 0$

4. *Sia* h_i^* *il valore finale di* $h_i(t)$*, ritorniamo le k query con* h_i^* *più alto.*

Recentemente è stato anche proposto un algoritmo basato sul grafo Query-URL che differisce dai precedenti perché utilizza delle relazioni asimmetriche [22] basate su un sistema di allocazione dinamica delle risorse.

A differenza del metodo asimmetrico visto precedentemente [4] che estrae relazioni asimmetriche solo se l'insieme degli URL cliccati per una query copre l'insieme degli URL cliccati per un'altra query.

3.3 Algoritmi basati sul contenuto dei documenti cliccati

Questo genere di algoritmi utilizza le informazioni presenti nei documenti cliccati, è un approccio utilizzabile solo quando i documenti contengono informazioni testuali, ma è necessaria una grande quantità di informazioni da salvare.

Una possibile implementazione [4], si propone di clusterizzare le query in base al contenuto dei documenti cliccati sfruttando una versione modificata della *TF-IDF*. Inoltre provvede ad un sistema di *ranking* basato su un criterio di rilevanza di una query rispetto al cluster di appartenenza. L'algoritmo considera solo le query che appaiono nel query log e ogni query inviata corrisponde ad una diversa *Query session*. La nozione di *Query session* è quindi piuttosto semplice:

$$QS := (\text{Query}, (\text{ClickedURL})^*)$$

L'algoritmo prevede una fase offline, di preprocessing e una online:

1. *Partizionamento tramite clustering delle query tramite il testo delle pagine corrispondenti agli URL cliccati (In fase di preprocessing);*

2. *Data una query, prima cerchiamo il cluster, poi computiamo la rilevanza rispetto agli altri suggerimenti;*

3. *Si ritornano le query in ordine.*

Il ranking viene calcolato secondo le seguenti notazioni:

Similarità della query Viene utilizzata una funzione di similarità

Supporto della query Viene utilizzata una misura di quanto è rilevante la query nel cluster. Il supporto viene calcolato come la frazione tra i documenti ritornati dalla query e i documenti cliccati. Tale valore viene stimato dal query log.

La computazione della similarità avviene costruendo un vettore dei pesi dei termini per ogni query, dove il vocabolario sono i differenti termini presenti negli URL cliccati, escludendo una lista di *stopwords*.

Data una query q, e un URL u, sia *Pop(q,u)* la popolarità di u rispetto a q. Sia inoltre *Tf(t,u)* il numero di occorrenze del termine di t nell'URL u. Rappresentiamo quindi q

con un vettore dove *q[i]* è l'i-esima componente del vettore associato all'i-esimo termine del vocabolario:

$$q[i] = \sum_{URL_u} \frac{\mathrm{Pop}(q,u) * \mathrm{Tf}(t_i,u)}{max_t \mathrm{Tf}(t,u)}$$

La funzione è la classica *TF-IDF* dove viene utilizzata la frequenza di click di del documento al posto della inverse document frequency. I cluster vengono computati utilizzando chiamate successive a *k-means* (Appendice A.3). Le diverse chiamate vengono utilizzate per scegliere il valore di k. L'utilizzo di k-means rispetto ad altri algoritmi di clustering viene scelto per la minore complessità.

Un altro algoritmo più recente intende diminuire le grande quantità di informazioni necessarie utilizzando solo i link presenti nei documenti [2]. Il metodo basa la sua validità su uno studio di Eiron e McCurley [13] nel quale viene sottolineata la similarità tra il testo dei link e le query.

In particolare viene definito *anchor text* il testo cliccabile che viene visualizzato in una pagina HTML. Ad esempio se abbiamo il seguente tag: ⟨*a href="index.html"*⟩ *foo* ⟨ */a*⟩ allora l'anchor text è *foo*.

Altre informazioni che possono essere ricavate sono l'indirizzo puntato, *anchor text document*, il testo che si trova vicino al link, *anchor text summary*.

Ci sono però molte difficoltà nell'utilizzo di questo metodo, per le query più popolari ad esempio abbiamo a disposizione molti link, molti dei quali non sono interessanti, perché possono essere generati in maniera automatica o sono link poco significativi per altri motivi. Quindi è necessario un sistema che possa ordinare gli *anchor text* per qualità. Tuttavia è possibile sfruttare un classico approccio al *document retieval problem*. Indicizzare quindi ogni singolo anchor text come se fosse un documento e utilizzare *TF-IDF* per il *ranking*.

3.4 Approcci Misti

Per cercare di aumentare l'efficacia e il numero di suggerimenti, si è cercato in alcuni casi di combinare le proprietà di più algoritmi.

In un articolo del 2001 Sarwar *et al.* [33] propongono un sistema basato su una tecnica originale rispetto alle Memory-Based e Model-Based viste in precedenza. Gli autori si propongono infatti di utilizzare un algoritmo basato su un *Collaborative Filtering Item-*

Based, cioè si propone prima di trovare relazioni tra gli *item* e poi tra gli utenti, basandosi sul fatto che le relazione tra item sono più statiche rispetto a quelle tra gli utenti.

Le informazioni tipiche che abbiamo in un sistema di questo tipo sono: una lista di utenti $U = \{u_1, u_2, ..., u_m\}$ e una lista di item $I = i_1, i_2, ...i_n$. Ogni utente ha associata una lista di item I_{u_i} che sono stati espressi come preferenza dall'utente. L'idea è quella di suggerire all'utente oggetti che gli interessano in base alle sue preferenze e a quelle dei suoi "simili".

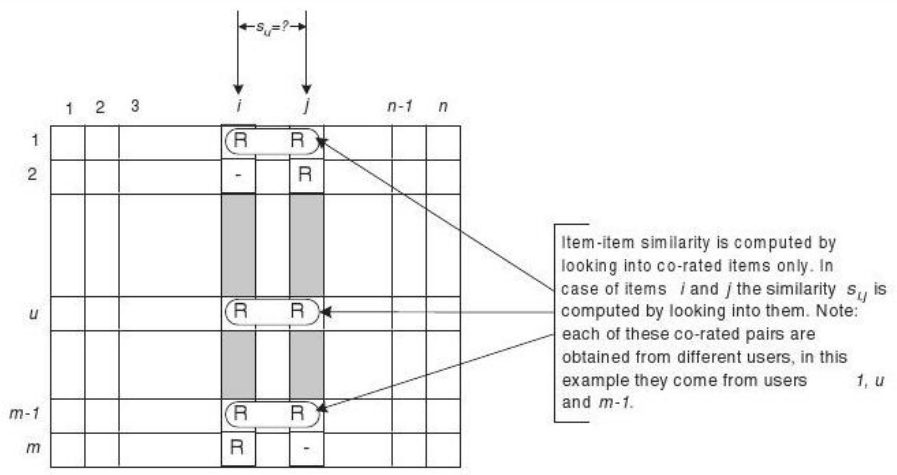

Figura 3.4: Computazione della similarità e isolamento degli *co-rated items* [33].

La similarità fra item può essere computata in diverse maniere:

Cosine-based similarity: due item sono visti come vettori (figura 3.4) e viene utilizzata una *cosine similarity* tra i due vettori, $sim(i, j)$ è data da:

$$sim(i,j) = cos(\overrightarrow{i}, \overrightarrow{j}) = \frac{\overrightarrow{i} \cdot \overrightarrow{j}}{\| \overrightarrow{i} \|_2 * \| \overrightarrow{j} \|_2}$$

Correlation-based similarity: la similarità viene computata utilizzando la correlazione di Pearson-r. Vengono tenuti in considerazione i casi in cui gli *item* sono presenti per lo stesso utente come in figura 3.4. Sia U l'insieme degli utenti che

hanno espresso una preferenza sia per i che per j, allora la correlazione è:

$$sim(i,j) = \frac{\sum_{u \in U}(R_{u,i} - \overline{R}_i)(R_{u,j} - \overline{R}_j)}{\sqrt{\sum_{u \in U}(R_{u,i} - \overline{R}_i)^2}\sqrt{\sum_{u \in U}(R_{u,j} - \overline{R}_j)^2}}$$

Adjusted Cosine Similarity: è una cosine similarity che tiene conto dei diversi utenti:

$$sim(i,j) = \frac{\sum_{u \in U}(R_{u,i} - \overline{R}_u)(R_{u,j} - \overline{R}_u)}{\sqrt{\sum_{u \in U}(R_{u,i} - \overline{R}_u)^2}\sqrt{\sum_{u \in U}(R_{u,j} - \overline{R}_u)^2}}$$

dove $\overline{R}_u$ è al media delle preferenze dell'u-esimo utente.

Per la computazione delle previsioni vengono proposte due tecniche:

Weighted Sum La computazione dell'*item* i per l'utente u prevede la somma delle preferenze dell'utente per gli *item* simili ad i. Ogni preferenza è pesata in base alla propria similarità con i : $s_{i,j}$ Formalmente:

$$P_{u,i} = \frac{\sum_{allsimilaritems,N}(s_{i,N} * R_{u,N})}{\sum_{allsimilaritems,N}(\|s_{i,N}\|)}$$

Regressione Questo metodo è simile al precedente, ma invece di utilizzare direttamente gli *item* per cui l'utente ha espresso una preferenza, utilizza un'approssimazione basata su regressione. Quindi si utilizza la stessa funzione di *weighted sum*, ma al posto di utilizzare direttamente i valori $R_{u,N}$ viene utilizzato $R'_{u,N}$ definito come:

$$\overline{R'_N} = \alpha\overline{R_i} + \beta + \epsilon$$

dove R_i e R_N sono i rispettivamente il vettore dell'*item* i e il vettore degli oggetti simili, α e β sono determinati dai vettori delle preferenze e ϵ è l'errore della regressione.

Questo metodo, risulta comunque essere particolarmente buono soprattutto nell'ambito del commercio elettronico (*e-commerce*) dove gli *item* sono piuttosto statici e le sessioni utente sono estraibili con maggiore precisione, in quanto spesso gli utenti sono iscritti.

Un'altra tecnica si propone invece di utilizzare una parte *content-based* e una *content-ignorant* [41] basata sulle sessioni utente.

Le due parti dell'algoritmo possono essere schematizzate come:

1. *analisi dell'opera di raffinamento degli utenti;*
2. *tradizionale metodo basato sul contenuto.*

Per tenere traccia dell'opera di raffinamento degli utenti si realizzano dei grafi dove i nodi rappresentano le query. Tali nodi sono collegati tra loro tramite archi pesati con un valore detto *dumping factor*, $d \in [0, 1]$, tale arco esiste se i nodi sono relativi a query consecutive nella stessa sessione.

Per le query non consecutive tale valore viene calcolato moltiplicando il *dumping factor* degli archi che le separano. Per esempio in figura 3.5, la similarità tra la *query1* e *query3* è $\sigma(1,3) = d * d = d^2$ Una volta calcolate le similarità delle query per ogni sessione, vengono poi aggiornate in base ai valori delle diverse sessioni, sommando tra loro i valori ottenuti come in figura 3.5.

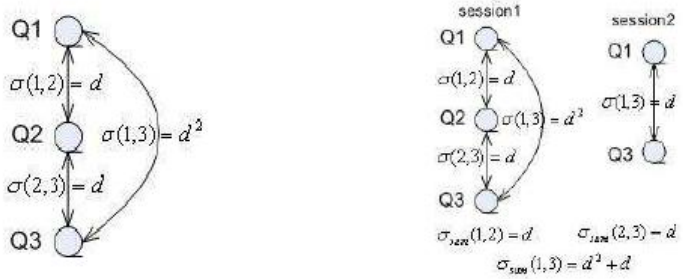

Figura 3.5: Calcolo similarità dalle sessioni [41].

Per quanto riguarda la parte content-based, viene utilizzata la *cosine similarity*:

$$\sigma(p,q) = \frac{\sum_{i=1}^{k}(cw_i(p)) * (cw_i(q))}{\sqrt{\sum_{i=1}^{m} w_i^2(p)} * \sqrt{\sum_{i=1}^{n} w_i^2(q)}}$$

dove $cw_i(p)$ e $cw_i(q)$ sono i pesi dell'i-esima *keyword comune* rispettivamente in p e in q e $w_i(p)$ e $w_i(q)$ sono i pesi dell'i-esima *keyword* rispettivamente di p e q.

Il peso dei termini della query sono calcolati utilizzando la *SF-IDF* al posto della *TF-IDF*, dove *SF* (Search Frequency) è la frequenza di ricerca. La ragione per la quale viene utilizzata questa variante è che data la sparsità dei termini nelle query, la frequenza sarebbe risultata troppo bassa per produrre differenze significative.

Usando i due metodi otteniamo due differenti cluster di query. Quindi vengono utilizzati dei parametri α e β nel seguente modo:

$$similarity = \alpha * sim_{query\_consecutive} + \beta * sim_{contenuto}$$

dove i due parametri α e β sono pesati per dare più o meno importanza ad uno dei due metodi. Nella pubblicazione originale, i due valori sono stati presi uguali.

Un algoritmo completamente *content-ignorant* [23], appartenente a questa classe, si basa sia sul click-through che sulle sessioni utente, sfruttando due diversi grafi: un Query-URL e un User-Query.

L'algoritmo si ripropone di apprendere:

1. "Query latent feature space"

2. Come suggerire query rilevanti

Per quanto riguarda il primo problema, si usa un metodo basato sulla fattorizzazione delle matrici. Quindi viene costruito un grafo delle query basato sul *latent feature space*.

Per il secondo problema si utilizza una nuovo modello di propagazione delle similarità che oltre ai suggerimenti, ne calcola la rilevanza.

Prima di calcolare i suggerimenti è necessario preprocessare i dati di click-through. Quindi trasformiamo i due grafi in matrici R^{m*n} dove m sono gli utenti e n sono le query e S^{n*p} dove n sono le query e p gli URL.

L'idea è di fattorizzare la matrice R in U e Q, dove U è una matrice $d * m$ e dove ogni colonna diventa il vettore d-dimensionale che descrive l'utente. Mentre Q è $d * n$ e ogni colonna è un vettore d-dimensionale che descrive la query.

Viene definita quindi la funzione di ottimizzazione come:

$$H(R,U,Q) = min_{U,Q} \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{n} I_{ij}^{R} (r_{ij} - U_i^T Q_j)^2 + \frac{\alpha_u}{2} ||U||_F^2 + \frac{\alpha_u}{2} ||Q||_F^2$$

dove α_u e α_q sono numeri interi positivi piccoli, $|| \cdot ||_F^2$ è la norma di Frobenius e $I_i j^R$ è la funzione che risulta uguale a 1 se l'utente i ha inviato la query j altrimenti è uguale a 0.

Lo scopo dell'ottimizzazione è di approssimare r_{ij} come $U_i^T Q_j$, il prodotto di due vettori. Inoltre per poter diminuire il rumore, viene normalizzato r_{ij} con il massimo

valore della riga i della matrice *user-query (R)* che viene chiamato r^*_{ij}. Anche il valore $U_i^T Q_j$ viene normalizzato con la funzione $g(x) = \frac{1}{1+exp(-x)}$

La funzione di ottimizzazione diventa quindi:

$$H(R,U,Q) = min_{U,Q} \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{n} I_{ij}^R (r^*_{ij} - g(U_i^T Q_j))^2 + \frac{\alpha_u}{2} ||U||_F^2 + \frac{\alpha_u}{2} ||Q||_F^2$$

Lo stesso viene fatto considerando la matrice query-URL e si costruiscono due matrici L e Q.

Infine per apprendere il "latent query feature space", di entrambi i grafi contemporaneamente, vengono fuse le due equazioni precedenti e tenendo condivisa la matrice Q:

$$\begin{aligned} &H(S,R,U,Q,L) = \\ &\frac{1}{2} \sum_{j=1}^{n} \sum_{k=1}^{p} I_{jk}^S (s^*_{jk} - g(Q_j^T L_k))^2 + \frac{\alpha_r}{2} \sum_{i=1}^{m} \sum_{j=1}^{n} I_{ij}^R (r^*_{ij} - g(U_i^T Q_j))^2 + \\ &+ \frac{\alpha_u}{2} ||U||_F^2 + \frac{\alpha_q}{2} ||Q||_F^2 + \frac{\alpha_l}{2} ||L||_F^2 \end{aligned} \tag{3.1}$$

dove α_r è un numero intero positivo piccolo. Un minimo locale di tale funzione può essere calcolato con il metodo del gradiente.

Per il calcolo della similarità viene utilizzato sistema di propagazione, basato sull'equazione differenziale del fenomeno fisico della propagazione del calore:

$$\begin{cases} \frac{\partial f(x,t)}{\partial t} - \Delta f(x,t) = 0, \\ f(x,0) = f_0(x) \end{cases}$$

dove $f(x,t)$ è la temperatura della locazione x al tempo t con la distribuzione iniziale $f_o(x)$ al tempo zero e Δf è l'operatore *Laplace-Beltrami* per la funzione f [21].

Come prima cosa si costruisce il grafo delle similarità tra query, basato sul *query feature space* calcolato in precedenza. Consideriamo quindi un grafo pesato orientato $G = (V, E, W)$ dove V sono i vertici: $V = \{q_1, q_2, ..., q_n\}$ ed E sono gli archi:

$$E = \{(q_i, q_j) | \text{c'è un arco da } q_i \text{ a } q_j \text{ e } q_j \text{ è nell'insieme dei } k \text{ nodi più}$$

$$\text{vicini a } q_i\}$$

Dopo aver applicato il modello di diffusione proposto, i suggerimenti possono essere dati seguendo tre passi:

1. *data una query q, selezioniamo n query che hanno almeno una parola in comune con q.* $S = \{\hat{q}_1, \hat{q}_2, ...\hat{q}_n\}$ *quindi utilizziamo l'equazione (3.3) per calcolare la similarità tra q e ogni query appartenente a S;*

2. *usiamo la (3.2) per calcolare la propagazione della similarità e calcolare il vettore* $f(1)$*;*

3. *ordiniamo* $f(1)$ *in maniera decrescente e suggeriamo le prime* N *query.*

$$f(1) = e^{\alpha R} f(0), R = \gamma H + (1 - \gamma) g 1^T \tag{3.2}$$

$$f_{\hat{q}_i}(0) = \frac{|W(q) \cap W(\hat{q}_i)|}{|W(q) \cup W(\hat{q}_i)|} \tag{3.3}$$

Più semplice è l'algoritmo pensato da Ji-Rong Wen *et al.* [37]. Gli autori si propongono di clusterizzare le query basandosi su due criteri:

Contenuto della Query Se le query contengono gli stessi termini o termini simili, indica che sono richieste informazioni simili. Per ovviare al problema delle poche informazioni dovute a query in genere corte, viene utilizzato il secondo metodo;

Click-Through se per due query sono stati cliccati gli stessi documenti, le query sono simili.

Per clusterizzare le query secondo i due metodi precedenti, viene proposto l'utilizzo di DBSCAN (Appendice A.4) o la sua versione incrementale [14], in quanto non richiedono input da parametro come ad esempio il K-Means e nello stesso tempo permette di considerare come rumore le query più isolate.

La funzione di similarità basata sui termini in comune è definita come:

$$similarity_{keyword}(p, q) = KN(p, q)/Max(kn(p), kn(q))$$

dove $kn(\cdot)$ è il numero di termini nella query e $KN(p, q)$ sono i termini in comune tra le query. Allo stesso modo sarebbe possibile utilizzare delle frasi al posto dei singoli

termini, perché la frase è più significativa, ma è necessario un sistema di identificazione delle frasi.

Per la funzione di similarità basata sul click-through vengono utilizzati i documenti cliccati in comune:

$$similarity_{document} = RD(p,q)/Max(rd(p), rd(q))$$

dove $rd(\cdot)$ sono il numero di documenti cliccati per una query e $RD(p,q)$ sono il numero di documenti cliccati in comune tra p e q.

Come metodo alternativo per la similarità basata sul click dell'utente viene proposto un metodo basato sulla gerarchia dei documenti cliccati. Il metodo è però molto oneroso nel caso del web mentre l'articolo si propone di suggerire query nel motore di ricerca di Encarta[2] dove i documenti appartengono a delle categorie semantiche.

Per poter avere una misura di similarità complessiva, vengono utilizzati due parametri α e β:

$$similarity_{comp} = \alpha * similarity_{keyword} + \beta * similarity_{document}$$

La similarità sintattica fra query è stata inoltre utilizzata anche in combinazione con l'estrazione delle sessioni [27]. Viene costruito un grafo: *Term Connection Graph*, utilizzando una misura di similarità basata sui termini. Ogni query è rappresentata con un insieme di termini: $Q = W_q = W_1W_2...W_n$ dove $W_i, i = 1, 2, 3..n$ sono i differenti termini nella query Q. Per ogni query, vengono cercate tutte le query Q_c tali che $W_q \subseteq W_{q_c}$ cioè tutte le query formate aggiungendo termini a Q. Quindi nel grafo viene creato un nuovo arco che connette Q con tutte le Q_c. Viceversa, vengono cercate tutte le query Q_l tali che $W_{q_l} \supseteq W_q$. Tali query vengono quindi connesse a Q nel grafo. Ogni arco del grafo indica quindi una specializzazione o una generalizzazione della query. La misura di similarità utilizzata è la seguente:

$$T_s = \begin{cases} (\frac{1}{2^D}) & \text{se } W_{q_a} \subseteq W_{q_b} \vee W_{q_b} \subseteq W_{q_a} \\ 0, & \text{altrimenti} \end{cases}$$

dove D è la distanza fra i termini di due query.

Viene inoltre costruito un grafo delle query sottomesse durante la stessa sessione chiamato *"Session Relationship Graph"*. Le query appartenenti ad una sessione ven-

[2]Encarta è l'enciclopedia multimediale di Microsoft.

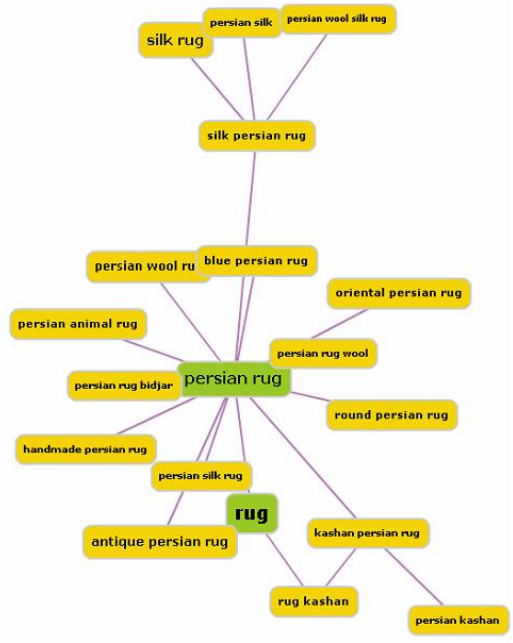

Figura 3.6: Term Connection Graph [27].

gono connesse secondo l'ordine cronologico. Quindi se un utente sottomette le query $Q_1Q_2Q_3Q_4$ allora nel grafo viene connessa Q_1 con Q_2, Q_2 con Q_3 e Q_3 con Q_4 . Ovviamente gli archi sono orientati dalla query precedente verso la successiva. Il peso di ogni arco viene calcolato in base al supporto delle connessioni. In particolare, l'autore propone la seguente euristica:

$$
\begin{aligned}
S_s &= 0.9 \quad se\ N > 10000 \\
&= 0.8 \quad se\ 10000 \geq N > 6000 \\
&= 0.7 \quad se\ 6000 \geq N > 1000 \\
&= 0.6 \quad se\ 1000 \geq N > 200 \\
&= 0.5 \quad se\ 200 \geq N > 50 \\
&= 0.4 \quad se\ 50 \geq N > 20 \\
&= 0.3 \quad se\ 20 \geq N > 6 \\
&= 0.2 \quad se\ 6 \geq N > 4 \\
&= 0.1 \quad se\ 4 \geq N \geq 3 \\
&= 0 \quad altrimenti
\end{aligned}
$$

Dove N è il numero delle sessioni in cui appaiono le relazioni. Ai due precedenti grafi si aggiunge una terza struttura dati per mappare le query al contesto. Si utilizza una misura di similarità (cosine similatity) tra documenti sfruttando le descrizioni delle pagine (snippet).

Le tre strutture dati sono poi utilizzate per produrre un unico risultato:

$$C_s = \alpha T_s + \beta S_s + \gamma K_s$$

dove $\alpha + \beta + \gamma = 1$.

In altri articoli presenti in letteratura sono poi presenti articoli sullo studio del *post-browsing* [11] [38] degli utenti che però hanno bisogno di informazioni supplementari che non sono in genere disponibili in questo ambito di ricerca.

Capitolo 4

Implementazione del framework

Come visto nel capitolo precedente gli algoritmi di Query Recommendation sono molto eterogenei tra loro nella struttura. Spesso le architetture che descrivono tali sistemi sono uniche e non facilmente accomunabili con altri sistemi dello stesso tipo.

Uno degli obiettivi che ci proponiamo in questa tesi, è l'implementazione di un *framework* software che consenta l'esecuzione di diversi tipi di Query Recommender System per poterli valutare in maniera automatizzata.

La realizzazione del *framework* ha richiesto inizialmente l'identificazione di una struttura comune insita negli algoritmi da dover testare, per poter ottenere un prodotto quanto più generico possibile in modo da poter essere esteso con facilità.

Le funzioni che vogliamo siano implementate nel software sono: la possibilità di simulare l'esecuzione di diversi algoritmi e successivamente valutare gli output con una serie di metriche, tralasciando per ora gli aspetti relativi alla complessità computazionale.

4.1 Struttura generale degli algoritmi

Gli algoritmi di Query Recommendation, come i Recommender System in genere, si basano sull'estrazione della conoscenza da un insieme di informazioni iniziali che in questo caso sono presenti nel *query log*.

L'estrapolazione delle informazioni può essere effettuata con diverse modalità, utilizzando le diverse tecniche prese in prestito, per esempio, dal Data Mining. Tali informazioni sono poi organizzate in un particolare modello e utilizzate per i suggerimenti.

I vari algoritmi quindi differiscono principalmente per i particolari dati che sfruttano,

per la particolare tecnica di mining che utilizzano, e ovviamente per il modello che viene generato e che in genere è basato su grafi, alberi o cluster.

Il generico algoritmo per Query Recommendation come si vede dalla figura 4.1, è composto da due componenti principali. Una prima componente, detta *online*, si occupa di tutte quelle operazioni che vengono eseguite in seguito ad una richiesta da parte dell'utente. Questa componente deve essere quindi abbastanza veloce da produrre i risultati in tempo utile per il fruitore del servizio, proprio per questo motivo la parte più onerosa in termini di complessità viene effettuata dall'altra componente, detta *offline*, che viene eseguita periodicamente e che si occupa del preprocessing dei dati e della generazione del modello.

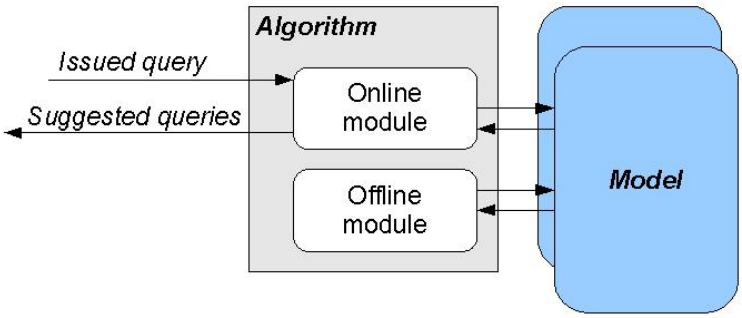

Figura 4.1: Struttura generale di un algoritmo per Query Recommendation.

Mentre la componente *online* è necessaria per il funzionamento del sistema, un sottoinsieme particolare di questi algoritmi, detti *Online*, operano senza una componente *offline*.

Per poter ottenere una versione Online dell'algoritmo, le operazione in genere svolte dalla componente offline, devono essere eseguite in seguito alla sottomissione delle query. Ovviamente la componente online necessita di alcune caratteristiche molto restrittive che hanno reso tali sistemi poco comuni. Il limite più grande è che il tempo per aggiornare il modello e generare i suggerimenti deve mantenersi al di sotto di una soglia temporale oltre la quale l'utente non si riterrà soddisfatto (nell'ordine di decimi di secondo per ogni suggerimento). Inoltre il modello deve poter essere aggiornato man mano che arrivano nuove query per l'impossibilità di ricalcolarlo completamente ogni volta. È necessario quindi l'utilizzo di una versione incrementale, che ne permetta l'aggiornamento con un

costo computazionale ridotto.

Degli algoritmi presenti in letteratura, la quasi totalità non propone architetture online utilizzando sistemi Offline che ricalcolano completamente il modello ad intervalli di tempo regolari.

Per questo motivo non ci sono studi che evidenziano i vantaggi e gli svantaggi dell'una o dell'altra architettura, ed è quello che ci proponiamo di fare in questa tesi.

4.2 Struttura del Framework

Il *framework* è stato ideato per eseguire e quindi valutare il maggior numero possibile di algoritmi di Query Recommendation. Le due operazioni principali sono state divise in due moduli separati:

Il Simulatore: utilizza i dati del query log in ordine cronologico per poter simulare la sottomissione delle query nel motore di ricerca e produrre i suggerimenti in base all'algoritmo utilizzato;

Il Valutatore: utilizza i dati prodotti dal simulatore e attraverso l'utilizzo di alcune metriche, ritorna un valore che indica la qualità del suggerimento.

4.2.1 Il simulatore

Questa componente è composta dai vari algoritmi di Query Recommendation e permette di eseguirli su un qualsiasi query log utilizzando un insieme di interfacce. Le componenti principali, del nostro *framework*, come mostrato in figura 4.2 sono:

Reader si occupa di estrarre le informazioni dal query log, per poi inviarle all'*Engine*;

Engine si occupa di inizializzare il particolare algoritmo da utilizzare e di inviargli le informazioni provenienti dal *Reader*. Si occupa anche di inviare al *Writer*, i suggerimenti calcolati dall'algoritmo;

Algorithm contiene le implementazioni dei diversi algoritmi;

Model contiene i modelli generati e utilizzati dai diversi algoritmi implementati;

Writer si occupa di scrivere il file con le informazioni relativa ai suggerimenti.

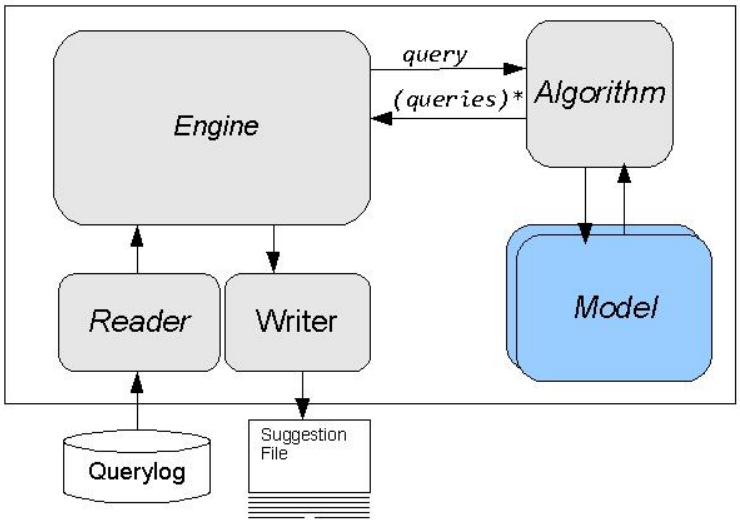

Figura 4.2: Struttura della componente per l'esecuzione degli Algoritmi.

Il simulatore utilizza i vari componenti tramite diverse interfacce che permettono l'utilizzazione di diversi *Reader*, algoritmi e modelli. Inoltre è stato necessario anche l'utilizzo di un formato standard per i dati in modo da permettere l'utilizzo dei diversi *dataset* in maniera del tutto trasparente per gli algoritmi. Il dato *GenericType*, mostrato nel Listato 4.1 contiene quindi tutte le informazioni di cui un Recommender System potrebbe aver bisogno. In particolare:

clientId identifica l'utente. Codificato con un *long*[1];

timestamp identifica la data in cui una query è stata sottomessa, anche questo è un *long* in quanto la data è espressa in *epoch*;

query è la stringa che identifica la query sottomessa;

clickedUrl è l'URL del link cliccato anche questo di tipo stringa;

rank è di tipo int[2] e indica la posizione dell'URL rispetto alla lista dei risultati ritornati dal motore di ricerca.

[1] il framework è stato sviluppato con il linguaggio Java per cui long è un tipo primitivo che contiene interi per un massimo di 8 byte

[2] è un intero a 4 byte

Tabella 4.1: Esempio di file dei suggerimenti.

ClientId	Timestamp	Query	Recommendation(2)	
369347	1145192874000	weather.com	weather channel	weatherchannel.com
369454	1147668169000	yahoo.com	yahoo	yahoo mail
...	...	...	...	...
...	...	...	...	...

```
public class GenericType
{
        public long clientId;
        public long timestamp;
        public String query;
        public String clickedUrl;
        public int rank;
}
```

Listato 4.1: Formato dati nel Framework

Il simulatore, durante l'esecuzione, produce un file di output contenente le informazioni sui suggerimenti generati. Per poter utilizzare efficacemente le metriche tuttavia sono necessarie altre informazioni che sono stata aggiunte al file ottenendo una struttura come quella in Tabella 4.1.

Come si può vedere, oltre ai suggerimenti, sono presenti l'identificatore dell'utente che ha sottomesso la query e la data di sottomissione.

4.2.2 Il valutatore

Questa componente prende in ingresso il file dei suggerimenti visti precedentemente e utilizza una delle metriche disponibili per computare la valutazione. La struttura generale è descritta in figura 4.3.

Le componenti principali che realizzano il valutatore sono:

Reader si occupa di estrarre le informazioni dal file dei suggerimenti, per poi inviarle all'*Evaluator*;

Evaluator invia le informazioni provenienti dal *Reader* alla particolare metrica e si occupa anche di inviare al *Writer*, le valutazioni ottenute come risposta dal *Collector*;

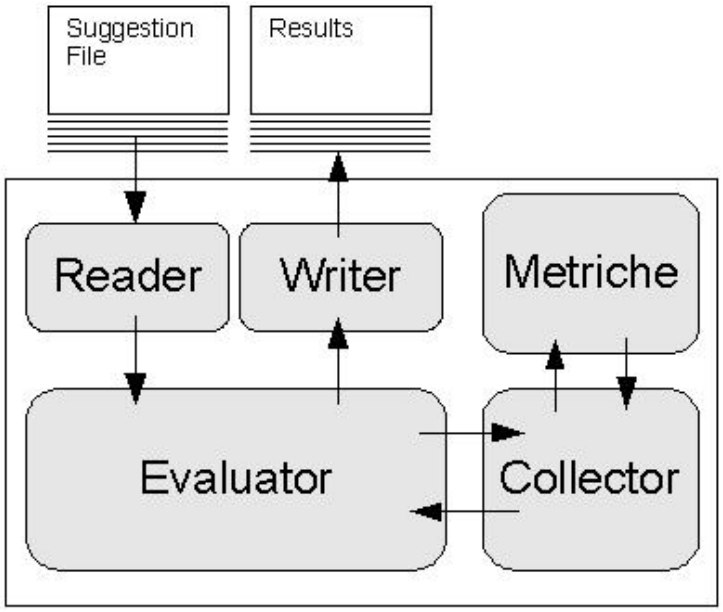

Figura 4.3: Struttura della componente per la valutazione dei suggerimenti.

Metric contiene le implementazioni delle diverse metriche;

Collector contiene le classi che si occupano di aggregare le valutazioni;

Writer si occupa di scrivere il file con le informazioni relativa alle valutazioni.

Anche in questo caso, come nel simulatore, il componente è stato progettato per essere esteso in un secondo momento con una qualsiasi metrica, utilizzando l'interfaccia *MetricInterface* (Listato 4.2). Per implementare una nuova metrica è sufficiente estendere l'interfaccia e implementare quindi il *evaluate()* per ritornare la valutazione.

```
public interface RecommenderMetricInterface
{
        public double evaluate(Suggestion s);
}
```

Listato 4.2: MetricInterface

Anche il modo di aggregare i dati può essere personalizzato utilizzando l'interfaccia *CollectorInterface* (listato 4.3). In questo caso è necessario implementare due metodi,

il primo *evaluate(s)* per chiamare la particolare metrica e salvare i risultati, il secondo *flush* per ritornare i valori finali in base alla politica scelta.

```
public interface CollectorInterface
{

        double evaluate(Suggestion sugg);

        double flush();

}
```

Listato 4.3: CollectorInterface

Capitolo 5

Implementazione degli Algoritmi e delle Metriche

In questo capitolo analizziamo gli algoritmi che sono stati implementati e le metriche che sono state scelte per la valutazione.

Uno degli scopi che ci siamo proposti è di capire se sfruttare alcune informazioni piuttosto che altre possa portare a risultati più o meno significativi per gli utenti. Vogliamo utilizzare le diverse informazioni che vengono evidenziate nel capitolo 3, in particolare:

Click-through è un informazione implicita disponibile nel query log. È ritenuta significativa perché è l'utente stesso che seleziona il link in base alle proprie preferenze;

Sessioni utente sono composte da tutte le query che l'utente ha sottomesso nel corso del tempo, quindi si sfrutta l'evoluzione delle query appartenenti alla sessione, per capire se esiste un filo conduttore tra la query di partenza e quella in cui l'utente è arrivato;

Contenuto dei documenti cliccati in questo caso si sfrutta il click-through per inferire una relazione tra la query e un insieme di documenti. Vengono quindi utilizzate le informazioni contenute nei documenti, come ad esempio termini o frasi.

La scelta degli algoritmi da implementare deriva dall'esigenza di avere a disposizione suggerimenti ottenuti con i diversi metodi appena descritti.

Inoltre, volendo valutare i suggerimenti in maniera automatica, vogliamo poter definire delle metriche che abbaino validità semantica, evitando l'approccio comune che è quello di ricorrere all'*user-study*.

5.1 Algoritmi implementati

Gli algoritmi implementati sono stati scelti tra quelli descritti nel capitolo 3, in modo da soddisfare le nostre esigenze.

In più, oltre ai convenzionali sistemi di raccomandazione, sono state sviluppate alcune alternative *Online* per cercare di valutare i pregi e i difetti.

Gli algoritmi da cui siamo partiti sono essenzialmente tre e sono i seguenti:

Cover Graph utilizza i dati sul click-through per popolare un grafo delle similarità;

Association Rules si basa sull'estrazione delle sessioni e utilizza delle regole associative per scoprire dei *pattern* frequenti;

Term Vector utilizza le informazioni nelle pagine web per associare ogni query ad un vettore di termini. Le query simili vengono trovate tramite clustering.

5.1.1 Cover Graph

L' algoritmo è stato descritto in precedenza nella sezione 3.2. Abbiamo visto che questo algoritmo sfrutta le informazioni sul click-through, cioè genera un modello basandosi soltanto sugli URL cliccati dagli utenti. Per questo motivo l'algoritmo è completamente *content-ignorant* e le query simili vengono ricavate tramite il calcolo di una similarità tra i vettori degli URL cliccati.

Nella nostra implementazione, ad ogni query sono associate essenzialmente due array. Il primo contiene tutti gli URL cliccati per la specifica query, mentre l'altro contiene tutte le query simili calcolate. Ovviamente il secondo array è ordinato rispetto alla similarità.

Per migliorare la velocità di esecuzione ogni query viene mappata con i dati ad essa associati tramite un'HashMap.

L'algoritmo nella sua versione originale prevede come primo passo l'estrazione dei dati presenti nel query log. Essendo un algoritmo di tipo *offline*, il calcolo delle similarità viene effettuato dopo aver aggiunto nel modello tutte le informazioni estratte. Una vola calcolate le similarità con la procedura di *update* desritta del Listato 1, l'algoritmo suggerirà sulla base dei dati presenti nel modello.

Il modello utilizzato e descritto in precedenza viene ricalcolato completamente ogni qual volta è necessario un nuovo *update*.

Algoritmo 1 Update per del modello in CoverGraph.

```
for i = 1 to numeroQuery do
  n1 ← norma(query[i].clickedUrl)
  for j = i + 1 to numeroQuery do
    n2 ← norma(query[j].clickedUrl)
    prod ← 0
    for k = 0 to numeroUrl do
      prod ← prod + query[i].clickedUrl[k] * query[j].clickedUrl[k]
    end for
    cosSim ← prod/(n1 * n2)
    if i ≥ SIM − THRESHOLD then
      Aggiungi Query Simile
    end if
  end for
end for
```

Questo procedimento non risulta troppo oneroso in termini di complessità, se si mantiene la lista degli URL cliccati ordinata, infatti, i pesi degli archi possono essere computati in tempo lineare nel caso peggiore, e in media è veloce perché molte query condividono pochi URL.

Più formalmente: sia M il massimo numero di URL condivise da due query. Allora il grafo può essere computato in tempo $max\{ME, nlogn\}$ dove E sono il numero di archi nel grafo e n è il numero di nodi. In media $M = O(1)$.

Nell'implementazione è stata inoltre inserita un euristica che permette il riconoscimento dei *Multitopical URLs*.

Questa procedura classifica gli URL in base al numero di volte che compaiono in archi deboli cioè con peso al di sotto di una certa soglia. Gli URL con il punteggio più alto vengono quindi rimossi.

Nell'implementazione, la procedura nell' Algoritmo 1 viene effettuata due volte. La prima iterazione per rimuovere i *Multitopical URL* mentre nella seconda vengono ricomputate le similarità fra query.

Versione Online

La versione online dell'algoritmo funziona sostanzialmente nello stesso modo, ma con alcune precauzioni per rendere il modello più maneggevole e per migliorare la velocità di risposta.

Come prima cosa sono state implementate delle politiche di *caching* per mantenere in memoria solo alcune delle informazioni, questo permette di mantenere tutto il modello in memoria principale. Lavorare su un sottoinsieme dei dati permette anche di avere un tempo di risposta migliorato. Per quanto riguarda le query la politica utilizzata è *LRU* cioè teniamo soltanto le query più recenti con l'idea che sono più utili rispetto alle meno recenti. Anche per gli URL è stata aggiunta una cache di topo *LRU* con lo stesso intento di quella utilizzata per le query.

Per velocizzare la fase di calcolo delle similarità inoltre, la procedura che calcola la *cosine similarity* tra le query viene eseguita una sola volta invece che due e l'euristica per eliminare i *multitopical URL* non viene eseguita.

Per poter diminuire il numero di confronti inoltre è stata aggiunta un'HashMap che mappa gli URL cliccati con le relative query sottomesse, in questo modo risulta più facile trovare tutte le query che condividono almeno un URL, per poi computare le similarità solo con tali query.

Il modello in questo caso è stato inoltre semplificato. Ogni qual volta una query viene sottomessa, ed un link viene cliccato, la similarità fra la query sottomessa e tutte quelle precedentemente trovate come simili, devono essere ricomputate, ed è anche necessario trovare eventuali nuove query che condividono l'ultimo URL cliccato. Ne deriva che, mantenere in memoria la lista delle query simili non porta nessuno vantaggio, se non lo svantaggio di modificare ogni volta tali strutture. Per questo motivo si è scelto di non memorizzare queste informazioni, ma semplicemente ritornarle una volta computate.

5.1.2 Association Rules

L'algoritmo implementato è lo stesso descritto nella sezione 3.1, l'idea è di estrarre le sessioni per cercare tutte quelle coppie di query che sono presenti in più sessioni.

L'algoritmo utilizza una definizione di sessione basata sull'utente e sul tempo, infatti si identifica la sessione come l'insieme delle query sottoposte da un utente nell'intervallo temporale stabilito da una soglia massima.

La ricerca dei pattern frequenti è implementata utilizzando una versione modificata di Apriori (Appendice A.5), per permettere una veloce ricerca delle regole associative in cui gli *itemset* (insiemi di item) sono costituiti da un solo elemento[1].

Il modello utilizzato consiste in due HashMap, una associa ad ogni sessione le query

[1]Se $X \Rightarrow Y$ è la regola associativa, ed X e Y sono gli *itemset*, vogliamo che $|X| = 1$ e $|Y| = 1$

che vi appartengono, in ordine cronologico, l'altra associa ad ogni query le sessioni in cui è presente, in questo modo è sufficiente contare le sessioni in comune tra due query e verificare che siano nell'ordine giusto per capire se le query sono simili, il ranking viene effettuato tramite il supporto.

L'algoritmo risulta essere molto più rapido rispetto al *CoverGraph* visto in precedenza, e la realizzazione di una versione *online* è stata un'operazione sostanzialmente semplice, infatti differisce, dalla versione originale vista nell'articolo, solo per l'introduzione di una cache LRU per le sessioni. Per le query invece non è stato necessario aggiungere cache perché sono limitate dal numero delle sessioni presenti nel modello.

5.1.3 TermVector

Il terzo algoritmo che è stato implementato si basa sul contenuto dei documenti cliccati dall'utente ed è stato descritto nella sezione 3.3.

Rispetto agli altri algoritmi, visti precedentemente, questo è un algoritmo basato sul contesto. La quantità di informazioni in più da gestire richiede un maggiore consumo di memoria, infatti, per ogni query, deve mantenere la lista degli URL cliccati e la lista dei termini incontrati nei documenti.

Anche in questo caso l'algoritmo nella sua versione originale è di tipo *Offline* ed utilizza un algoritmo di clustering (*k-means*), per trovare query simili.

In questo caso, per la realizzazione della versione Online, sono stati riscontrati alcuni problemi. In primo luogo, ogni algoritmo Online dovrebbe poter aggiornare il modello velocemente, mentre l'utilizzo del contenuto dei documenti richiede tempo per essere estrapolato e successivamente utilizzato. Nel documento associato al link, infatti, ci sono in genere delle pagine web, dove è necessario rimuovere alcune informazioni:

- *tag html*;
- codici relativi a script (ad esempio *javascript*);
- eventuali codici relativi ai fogli di stile (*css*);
- rimozione delle *stopword*.

Successivamente le informazioni rimanenti devono essere inserite nel modello e utilizzate per il calcolo della *TF-IDF*.

Un problema successivo sarebbe stato quello della clusterizzazione. Il k-means, nella sua versione originale, non permette un aggiornamento del modello in modalità incrementale. Quindi l'algoritmo avrebbe dovuto utilizzare una versione incrementale del k-means [25] o una versione incrementale di un altro algoritmo di clustering [10, 14] riducendo ancora le prestazioni.

Le caratteristiche dell'algoritmo, la grande quantità di dati e l'utilizzo di k-means, non hanno quindi permesso di implementarne una versione Online.

L'algoritmo, nella versione implementata (*Offline*), procede come i precedenti, estraendo dal query log tutte le informazioni sulle specifiche query e per ogni URL cliccato, viene presa la pagina, ripulita dai vari tag, rimosse le stopword e i termini rimasti vengono associati all'URL nel modello dell'algoritmo. Una volta conclusa l'estrazione dei dati, questi verranno processati computando inizialmente una TF-IDF (Appendice A.1) per assegnare un peso ai termini associati alle query, quindi viene utilizzato k-means per la clusterizzazione rispetto ai vettori appena calcolati.

5.2 Metriche utilizzate

Per poter valutare l'accuratezza dei recommender system, per ogni oggetto suggerito, dobbiamo stimare quanto è vicino alle reali preferenze dell'utente. Una prima definizione di accuratezza [18] può essere:

$$accuratezza = \frac{numero\ di\ suggerimenti\ significativi}{numero\ di\ suggerimenti}$$

cioè vogliamo sapere quanti tra i suggerimenti ottenuti sono effettivamente interessanti.

Un'altra definizione può invece basarsi sull'assunzione che il numero di suggerimenti significativi è equivalente a dire quanto è vicino il suggerimento alle preferenze dell'utente.

Sia quindi $P(u, i)$ la previsione del recommender system per l'utente u e l'*item* i e sia $p(u, i)$ la funzione che descrive le reali preferenze dell'utente u. La funzione p può essere esplicitamente espressa dall'utente o implicitamente calcolata utilizzando, ad esempio, alcune informazioni sul passato dell'utente. Le due funzioni $p(u, i)$ e $P(u, i)$ sono in genere binarie cioè ad ogni oggetto restituiscono 1 o 0 a seconda se l'oggetto è ritenuto significativo o meno.

L'accuratezza può essere riformulata come:

$$accuratezza = \frac{\sum_{(\forall u,i/r(u,i)=1)} 1 - |p(u,i) - P(u,i)|}{R}$$

In questo caso p e P sono considerate funzioni binarie. In più $r(u,i)$ è 1 se il suggerimento i viene realmente presentato all'utente u, altrimenti è 0. Inoltre $R = \sum_{u,i} r(u,i)$ è il numero totale di suggerimenti dati agli utenti.

Un altro sistema molto utilizzato nei Recommender System è il *mean absolute error* (MAE), la metrica misura il valore assoluto della deviazione media tra gli oggetti suggeriti ($P(u,i)$) e quelli preferiti dall'utente ($p(u,i)$). Sia N il numero osservazioni disponibili allora il MAE può essere calcolato come:

$$MAE = \frac{\sum_{u,i} |p(u,i) - P(u,i)|}{N}$$

Un'altra tipologia di misurazioni si basa, invece, su alcune operazioni caratteristiche dell'*Information Retrieval*, in particolare per i sistemi di suggerimento può essere utile calcolare *precision* e *recall*.

	rilevanti	non rilevanti
suggeriti	a	b
non suggeriti	c	d

Tabella 5.1: Matrice di confusione. Gli elementi diagonali a e d contano le decisione corrette mentre c e b sono i casi non corretti.

Per il calcolo si utilizza una matrice di confusione come quella in Tabella 5.2, che riflette le quattro possibilità di ogni decisione estratta. La bontà del suggerimento può essere quindi valutata in maniera binaria assegnando 1 ad un buon suggerimento e 0 altrimenti.

$$precision = \frac{a}{a+b} \tag{5.1}$$

$$recall = \frac{a}{a+c} \tag{5.2}$$

Il significato delle due misure : *precision* (5.1) e *recall* (5.2) è abbastanza intuitivo. *Recall* misura la capacità del recommender system di predire il maggior numero di oggetti

significativi. D'altra parte, *precision*, misura la capacità di estrarre solo suggerimenti rilevanti.

Un'alternativa alle precedenti misure può essere data dalla *Receiver Operating Characteristic* (*ROC*) [16], che viene calcolata utilizzando il *recall* in rapporto al *fallout*, dove per fallout si intende quanti tra i suggerimenti non rilevanti vengono suggeriti (Equazione 5.2).

$$fallout = \frac{b}{b+d} \tag{5.3}$$

Altre tecniche di misurazione molto utilizzate sono derivate da *precision/recall* e sono le misure F che cercano di condensare in un unica misurazione sia *precision* che *recall*. La forma generale è data da F_β (si veda l'equazione 5.4) mentre la più diffusa è la $F1$ (5.5) dove $\beta = \frac{1}{2}$ e non è altro che la media armonica tra *precision* e *recall*.

$$F_\beta = \frac{precision \cdot recall}{(1-\beta) \cdot precision + \beta \cdot recall} \tag{5.4}$$

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{5.5}$$

Da queste definizioni, possiamo quindi implementare le metriche utilizzando le diverse informazioni a nostra disposizione. Alcune delle metriche implementate sono state trovate in letteratura:

- TermBased
- EditDistance
- ResultBased

altre invece sono stare definite in questa tesi:

- Omega[2]
- LinkOmega
- LinkOverlap
- QueryOverlap

[2]la Omega era già stata utilizzata in un ambito diverso dai Query Recommender System [6], la versione implementata differisce per alcune caratteristiche che la rendono utilizzabile in questo contesto.

Uno dei metodi più semplici per capire l'utilità di un suggerimento è quello di andare a verificare la similarità sintattica tra la query originale e i suggerimenti. Questa tecnica, nella realtà non sempre corrisponde ad una similarità semantica a causa di omonimi o polisemia.

Altri sistemi invece prevedono il confronto tra i risultati ottenuti sottomettendo le query al motore di ricerca [15]. La lista dei risultati può essere utilizzata operando una semplice intersezione tra i primi risultati oppure è possibile utilizzare metodi più sofisticati come la correlazione rispetto alla posizione dei risultati. Anche questo metodo però, potrebbe essere poco significativo, infatti una query potrebbe essere un buon suggerimento, ma la lista dei risultati potrebbe avere link completamente differenti. Supponiamo ad esempio la query 'motori di ricerca', potremmo avere i seguenti buoni suggerimenti: 'google', 'ask' e 'yahoo' che pur essendo attinenti alla ricerca, e sicuramente utili per gli utenti, provocano, se inseriti in qualunque motore di ricerca, risultati del tutto differenti.

Oltre alla realizzazione delle metriche appena descritte, quindi ne sono state implementate anche altre più originali, che basano la propria valutazione sull'attività degli utenti, sfruttando le informazioni disponibili nel query log.

Le metriche implementate, cercano quindi di analizzare i risultati ottenuti in vari aspetti, per cercare di capire quali tra i suggerimenti riportati sono più significativi per l'utente.

Le metriche implementate per i test sono descritte in dettaglio nei paragrafi seguenti.

5.2.1 TermBased

Questa metrica si basa sui termini che compongono le query. Quindi utilizza solo informazioni sintattiche. L'idea è contare quanti termini contenuti nella query iniziale, sono presenti nelle query suggerite.

Un frammento del codice è schematizzato in Algoritmo 2. In pratica per ogni suggerimento vengono calcolati il numero di termini in comune, diviso il numero di termini che compongono la query iniziale. La somma di questi valori viene divisa per il numero di suggerimenti dati per ogni query.

Il valore ritornato dalla funzione è compreso tra 0 e 1 ed è una media della similarità sintattica tra i vari suggerimenti e la query. Se definiamo con q l'insieme dei termini della query e s_i l'insieme dei termini per il suggerimento i allora abbiamo:

Algoritmo 2 Pseudocodice per la metrica TermBased

$queryTerms \leftarrow split(Suggestion.query)$
for $j = 0$ to $numSuggestion$ **do**
  $suggTerms \leftarrow split(Suggestion.suggestion[j])$
  $num \leftarrow commonTerms(queryTerms, suggTerms)$
  $qValue \leftarrow qValue + \frac{num}{queryTerms.size}$
end for
$value \leftarrow value + \frac{qValue}{numSuggestion}$

$$TermBased(q, s_1, ..., s_m) = \frac{\sum_{i=1}^{m} \frac{q \cap s_i}{|q|}}{m}$$

5.2.2 EditDistance

È un'altra metrica basata sulla sintassi delle query. Questa volta, invece di controllare i termini viene calcolata la distanza tramite l'algoritmo di Levenshtein (Appendice A.6).

La normalizzazione avviene in maniera molto simile alla metrica *TermBased* vista precedentemente. Viene calcolata la distanza tra ogni suggerimento e la query per poi fare una media aritmetica. La formula utilizzata per calcolare la distanza tra due query è descritta nell'equazione sottostante, dove LD è la distanza di Levenshtein normalizzata a 1, dividendo per la lunghezza della stringa più lunga; tale valore viene sottratto a 1 per ottenere una misura di similarità. A questo punto la somma di tutti valori ottenuti per la stessa query vengono divisi per il numero di suggerimenti, per ottenere la media.

$$EditDistance(q_T, q_1, ...q_m) = \frac{\sum_{i=1}^{m} 1 - LD(q_T, q_i)}{m}$$

5.2.3 Results Metric

Questa metrica cerca di capire quali sono le relazioni semantiche tra le query utilizzando i risultati ritornati dal motore di ricerca. Per ogni query viene fatta una richiesta al motore di ricerca [3] richiedendo i primi dieci risultati.

Quindi viene fatto un'intersezione tra i risultati ottenuti per la query di cui si vogliono dei suggerimenti e i risultati ottenuti per ognuno dei suggerimenti. L'idea è che due query sono tanto più simili quanto più condividono i link dei risultati. Il numero di link in comune tra la query e il suggerimento viene diviso per il numero di link che si controllano,

[3] per questo test ho utilizzato i risultati di Yahoo! Web Serch Engine

nel nostro caso dieci, quindi analogamente ai casi precedenti viene fatta una media tra le valutazioni effettuate su tutti i suggerimenti ottenuti per la stessa query:

$$ResultsMetric(q_T, q_1, ...q_m) = \frac{\sum_{i=1}^{m} \frac{results(q) \cap results(q_i)}{10}}{m}$$

5.2.4 Omega Metric

La Metrica *Omega* a differenza delle metriche appena descritte si propone di valutare le relazioni fra query utilizzando il comportamento dell'utente, che viene prelevato utilizzando le sessioni.

Ogni sessione dell'utente viene divisa in due metà. La prima metà verrà valutata, mentre la seconda viene utilizzata per la valutazione. Il valore della Omega viene calcolato computando l'intersezione tra le query suggerite nella prima metà della sessione con le query sottomesse dall'utente nella seconda metà come mostrato in Figura 5.1.

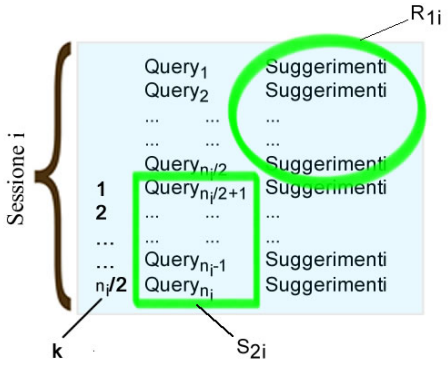

Figura 5.1: Rappresentazione della metrica Omega.

Più formalmente:

$$\Omega = \sum_{k=1}^{n_i/2} < o_k \in S_{2i} \cap R_{1i} > \cdot \frac{f(k)}{K}$$

dove < expr > rappresenta una funzione booleana il cui risultato è 1 se *expr* vale *true*, 0 altrimenti. S_{2i} è la seconda metà della sessione, R_{1i} sono i suggerimenti della prima metà della sessione e $f(x)$ è una funzione di normalizzazione che permette di definire un peso diverso se gli elementi in comune sono più vicini alla fine della sessione. Il valore K è definito invece come $\sum_{k=1}^{n_i/2} f(k)$.

L'equazione ci dà quindi un valore normalizzato ad uno per ogni sessione, infatti se ogni elemento suggerito è presente nella seconda parte della sessione, allora sarà valida l'Equazione 5.6.

$$\sum_{k=1}^{n_i/2} < o_k \in S_{2i} \cap R_{1i} > \cdot \frac{f(k)}{K} = \sum_{k=1}^{n_i/2} \frac{f(k)}{K} = \frac{\sum_{k=1}^{n_i/2} f(k)}{\sum_{k=1}^{n_i/2} f(k)} = 1 \tag{5.6}$$

L'idea alla base della metrica è che se il suggerimento proposto nella prima metà della sessione, è stato successivamente sottomesso dall'utente, allora l'algoritmo di suggerimento è riuscito a predire la volontà dell'utente e quindi la valutazione è positiva. La funzione $f(\cdot)$, come accennato in precedenza può essere utilizzata per aumentare la valutazione di quei suggerimenti che corrispondono alle query inviate dall'utente a fine sessione, utilizzando ad esempio $f(k) = k$. L'idea è che l'utente nel corso della ricerca, raffina le richieste, rendendo più precise quelle più vicine alla fine della sessione.

In questo caso la metrica utilizza il file dei suggerimenti ordinato per sessione e *timestamp*, quindi è stata anche implementata in una versione che permette l'utilizzo di un file ordinato per *timestamp* (si veda *QueryOverlap* nel paragrafo 5.2.5), in modo da poter calcolare la differenza di prestazioni il relazione al tempo.

5.2.5 QueryOverlap

Questa metrica è una versione modificata della *Omega* per poter valutare i risultati del suggerimento in ordine cronologico, con il vantaggio di poter verificare l'andamento delle valutazioni con il modificarsi del modello nel tempo.

La differenza fondamentale è che la valutazione non è relativa alla sessione, ma viene ritornato un valore per ogni insieme di suggerimenti dati per una query, come si può vedere in Figura 5.2.

Anche qui vengono valutati solo i suggerimenti relativi alla prima metà di ogni sessione, ma il valore viene calcolato effettuando l'intersezione con tutte le query che sono state sottoposte in seguito alla query che si sta analizzando. Sia j l'indice della query corrente che si sta analizzando, ed R_{ji} i suggerimenti dati per la query j della sessione i. Definiamo la *QueryOverlap* come:

$$QueryOverlap = \sum_{k=1}^{n_i - j} < o_k \in S_{2i} \cap R_{ji} > \cdot \frac{f(k)}{K}$$

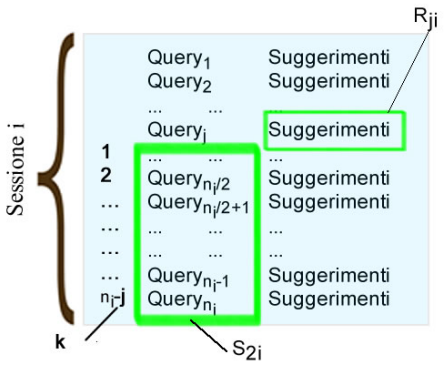

Figura 5.2: Rappresentazione della metrica QueryOverlap.

Anche in questo caso le valutazioni sono normalizzate a uno. In modo analogo a quanto visto nell'equazione 5.6, ricordando che in questo caso abbiamo $K = \sum_{k=1}^{n_i-j} f(k)$.

5.2.6 LinkOmega

Il funzionamento di questa metrica è basato sul fatto che se un utente clicca un URL dopo la sottomissione della query, allora ritiene tale URL significativo per l'argomento a cui è interessato.

L'implementazione di questa metrica è molto simile a quella vista per Omega, con la sola differenza che invece di utilizzare direttamente le query, si utilizzano i link che vengono cliccati dall'utente.

Il limite della Omega infatti, potrebbe essere quello che si suggeriscono query utili ma che differiscono da quelle cercate nella seconda parte della sessione dell'utente a causa di errori ortografici o a causa dell'utilizzo di sinonimi.

Per evitare questi problemi, per ogni query suggerita, vengono cercati nel query log gli URL che gli utenti hanno cliccato dopo la sottomissione. Con questa lista di URL viene effettuata l'intersezione con gli URL che l'utente ha visitato nella seconda metà della sessione (Figura 5.2.7).

Un ulteriore fatto positivo è che oltre ai pregi appena descritti, questa metrica tiene conto del fatto che per alcune query sottomesse dall'utente non vengono cliccati URL, con il significato che la query non ha prodotto risultati interessanti.

Anche questa metrica, come la *Omega* utilizza un file ordinato per sessioni ed è stata

quindi implementata anche una versione modificata, descritta in 5.2.7, che ne permette l'utilizzo con un file ordinato per *timestamp*.

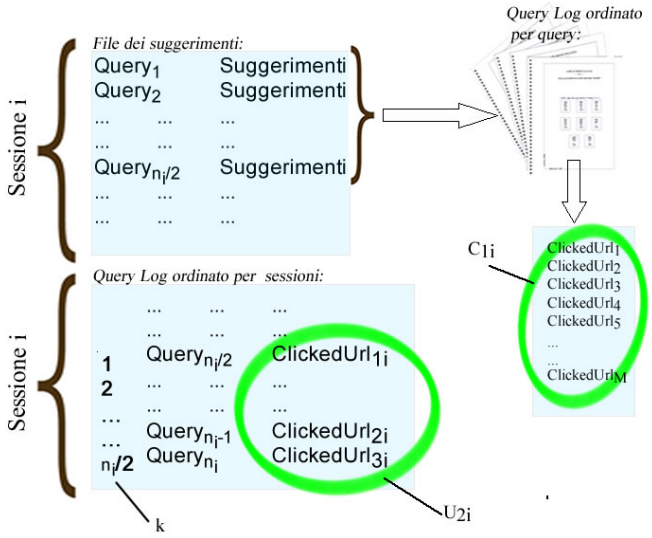

Figura 5.3: Rappresentazione della metrica LinkOmega.

L'equazione che descrive il funzionamento di questa metrica è sostanzialmente la stessa vista per la Omega dove: U_{2i} sono link cliccati nella seconda metà della sessione utente ed C_{1i} i link cliccati da altri utenti per le query suggerite. Allora l'equazione è la seguente:

$$LinkOmega = \sum_{k=1}^{n_i/2} < o_k \in U_{2i} \cap C_{1i} > \cdot \frac{f(k)}{K}$$

L'equazione ci dà quindi un valore normalizzato ad uno per ogni sessione, come visto nell'Equazione 5.6.

Ancora una volta utilizziamo la funzione $f(\cdot)$ per dare più importanza ai link cliccati alla fine delle sessioni.

5.2.7 LinkOverlap

Come accaduto per la Omega, anche per la LinkOmega è stata realizzata una versione della metrica capace di utilizzare un query log ordinato per timestamp. Le modifiche apportate rispetto alla versione originale sono le stesse viste nel paragrafo 5.2.5.

In uno schema del funzionamento in Figura 5.4, si possono notare le differenze con lo schema in Figura .

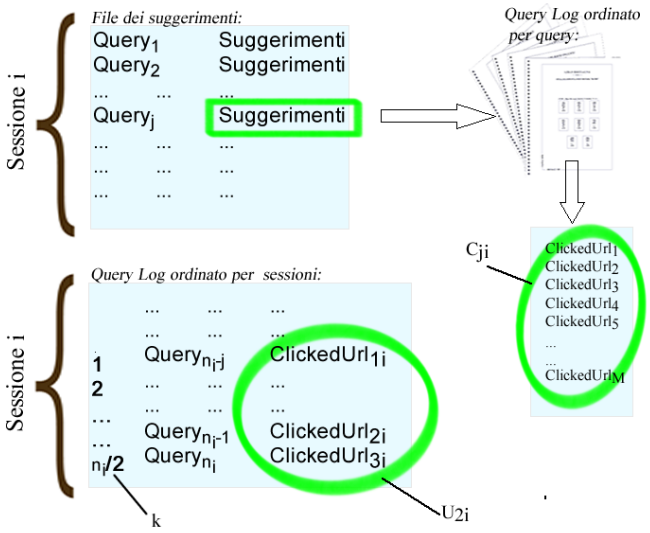

Figura 5.4: Rappresentazione della metrica LinkOverlap.

In particolare:

- valutazioni per ogni lista di suggerimenti associati ad una query invece che valutazione della sessione

- valutazione calcolata su tutte le query che seguono quella per cui si sta valutando invece di valutare utilizzando solo la seconda metà della sessione.

5.2.8 Percentage

Non è una vera e propria metrica, perché non ritorna nessun valore che abbia a che fare con le relazioni semantiche tra le query. Questa classe ritorna semplicemente una percentuale che ci dice quante volte vengono dati suggerimenti.

Accade infatti che per alcune query particolari, in particolari stati dell'algoritmo, non è possibile dare dei suggerimenti. Con questa classe sappiamo precisamente quanti sono questi casi.

5.3 Il Collector

Tutte queste metriche ritornano un valore per ogni riga del file dei suggerimenti, o per ogni sessione come nel caso delle metriche Omega.

Spesso però è utile, per ottenere dei dati più chiari, poter aggregare alcune informazioni, per semplificare la schematizzazione.

A questo scopo sono state implementate alcune classi che collezionano i dati. Alcune di queste classi sono descritte di seguito:

ZeroCollector in questa classe i valori della metrica vengono ritornati esattamente come sono, il risultato è come se il collector non ci fosse;

AllCollector in questo caso, la classe somma tutti i valori della metrica e mantiene il numero di valutazioni ritornare dalla metrica. Nel momento in cui viene chiamato il metodo flush, al termine della valutazione, viene ritornato il valore medio di tutte le valutazioni;

TimeCollector anche qui l'idea è fare una media dei valori. La differenza è che la media viene ritornata ogni qual volta è stato superato un periodo di tempo limite tra i *timestamp* delle query;

LogTimeCollector come TimeCollector, ma il lasso temporale viene raddoppiato ogni volta che viene ritornato un valore. in modo da ottenere valutazioni più frequenti all'inizio e più rade alla fine;

OmegaCollector questa è una classe speciale, ideata per collezionare i dati proveniente da metriche come la Omega che ritornano una valutazione per ogni sessione. Dopo aver sommato tutti i valori ritornati li divide per il numero di sessioni incontrate;

PercentageCollector anche questa classe è stata ideata per essere utilizzata con la metrica Percentage. L'idea è quella di ritornare una percentuale dei valori, quindi conta il numero di valori ritornati e li divide per il numero di valori totali che sono stati richiesti. Viene ritornato il valore ottenuto moltiplicato per cento.

Capitolo 6

Risultati sperimentali

Una volta completata l'implementazione del framework, delle metriche e degli algoritmi, ci siamo proposti di effettuare alcune valutazioni per capire quanto siano affidabili le metriche, valutare l'utilità e la flessibilità del framework nonchè cercare di confrontare gli algoritmi implementati.

6.1 Dataset

Per lo svolgimento dei test è stato utilizzato il Query Log di *AOL* [28] che contiene le richieste fatte al motore di ricerca in un periodo di tre mesi compresi tra il primo Marzo 2006 e il 31 Maggio dello stesso anno. Per un totale di 36.389.567 linee di dati, con 10.154.742 query distinte e 657.426 diversi client ID.

Le informazioni contenute sono:

AnonID un numero ID anonimo che rappresenta l'user;

Query la query inviata dall'utente, modificata in carattere minuscolo e con la maggior parte della punteggiatura rimossa;

QueryTime - il momento in cui la query è stata sottomessa;

ItemRank - se l'utente clicca su un risultato, rappresenta la posizione del risultato cliccato;

ClickURL - se l'utente clicca su un risultato, il dominio dell'URL del risultato cliccato.

Ogni linea nel log rappresenta due tipi di eventi:

1. La query che è stata sottomessa dall'utente senza nessun click;
2. Le informazioni sul click through relativi ad una specifica query sottomessa dall'utente.

6.2 Svolgimento dei Test

Prima di poter eseguire gli algoritmi sui dati sono state svolte alcune operazioni per poter rendere più affidabili i risultati e le metriche.

Per prima cosa il query log è stato riordinato per *timestamp* crescente, cioè le query sottomesse sono accessibili in ordine cronologico, in modo che il simulatore potesse processare i dati nello stesso ordine con cui questi sono giunti al motore di ricerca ed ottenere così dei risultati più realistici.

Da alcuni test iniziali inoltre è stato notato che il query log in genere contiene molte query semanticamente equivalenti, ma sintatticamente differiscono per alcuni caratteri, che rendono alcuni risultati poco significativi. Ad esempio la presenza di query come: yahoo.com yahoo www.yahoo.com che pur indicando la stessa cosa sono mappate negli algoritmi come query differenti. Per aggirare questo problema che porta ad avere come suggerimenti query equivalenti, è stata utilizzata nei test un'espressione regolare (*regex*) che permettesse di rimuovere alcune sotto-stringhe presenti in molte query, ma che non erano significative.

Tale *regex* utilizzata in questo ambito è la seguente:

$$[.](com|net|biz|us|gov|co|cc|uk|org)|http[s]* : //|www[.]|[]+$$

che ovviamente può essere migliorata aggiungendo altri casi comuni, anche se per la quantità di varianti possibili, è praticamente impossibile definire una funzione che li copra tutti.

6.3 Risultati

Un primo test che è stato effettuato comprende l'analisi dei risultati dei vari algoritmi nelle versioni originali rispetto all'aumento della dimensione del training set.

Gli algoritmi vengono quindi allenati con una quantità di dati sempre maggiore, ci aspettiamo quindi dalle valutazioni che anche l'afficacia dei suggerimenti sia crescente.

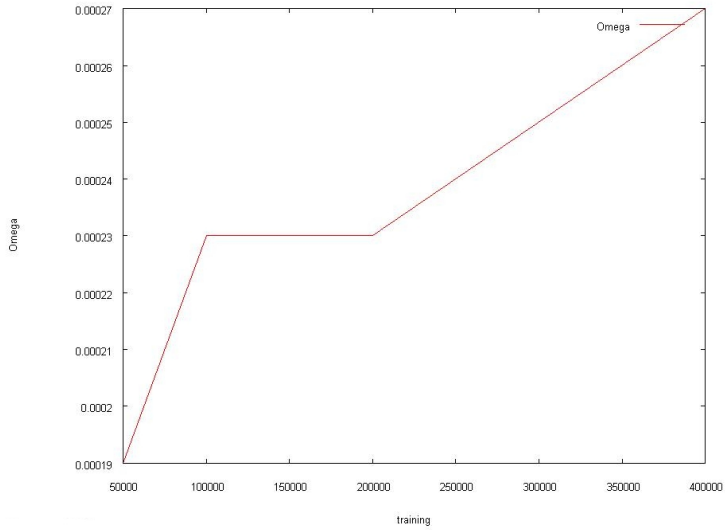

Figura 6.1: Valore medio della Omega utilizzando CoverGraph con training set di dimensioni differenti.

Nella figura 6.1 è stata utilizzata la metrica Omega per valutare l'algoritmo CoverGraph in versione Offline. Nell'asse delle ascisse abbiamo le dimensioni del training set utilizzato. Nella figura 6.1 abbiamo effettuato la stessa valutazione con la metrica LinkOmega. Entrambe le valutazioni sono state ottenute operando una media tra tutti i valori trovati. Come si nota, entrambe le curve sono monotone crescenti, come era lecito aspettarsi. Questa misurazione ci da una prima prova della validità delle due metriche.

Un'ulteriore prova della validità delle metriche è stata ottenuta valutando gli algoritmi Online eseguiti con cache di dimensioni diverse.

Nei grafici 6.3 e 6.4 dove le ascisse sono le dimensioni delle cache utilizzate, espresse in numero di elementi che contengono, e le ordinate esprimono i valori delle metriche, si nota che anche qui un aumento delle informazioni utilizzate corrisponde ad un aumento di efficacia per gli algoritmi interessati.

Altri test basati sulle stesse informazioni, sono stati svolti utilizzando invece le metriche basate sulla sintassi. In questo caso i risultati ottenuti sono meno precisi. Lo stesso test è stato effettuato per l'algoritmo Association Rules che ha portato a risultati simili riportati nelle figure 6.5 e 6.6

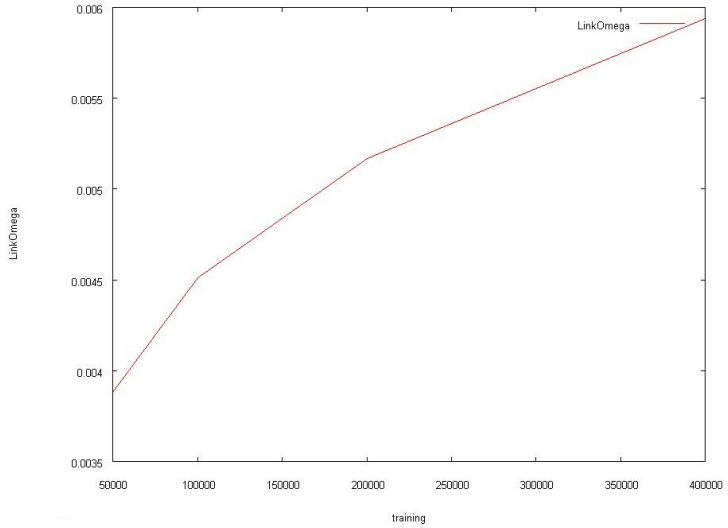

Figura 6.2: Valore medio della LinkOmega utilizzando CoverGraph con training set di dimensioni differenti.

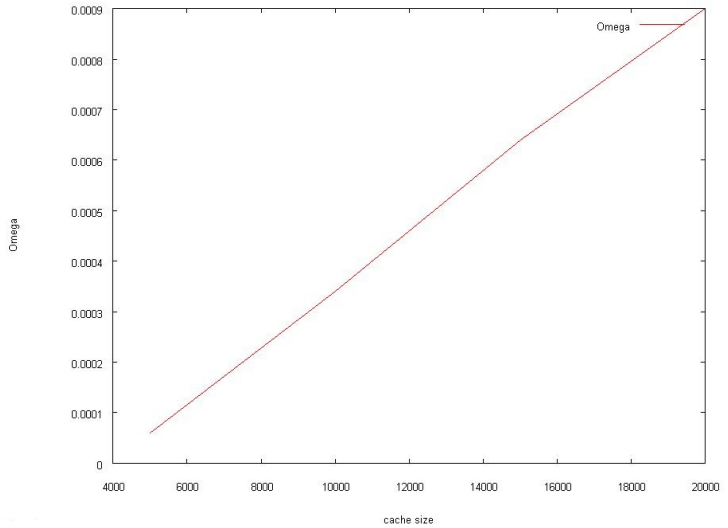

Figura 6.3: Valore medio della Omega utilizzando AssociationRules in versione Online con cache di dimensioni differenti.

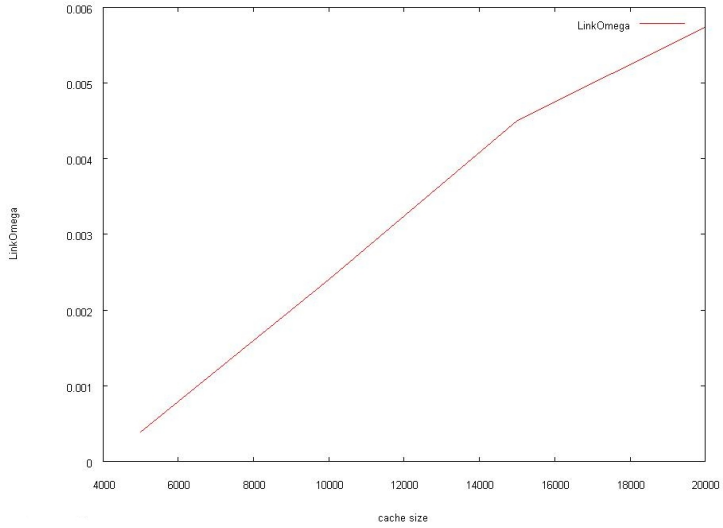

Figura 6.4: Valore medio della LinkOmega utilizzando AssociationRules in versione Online con cache di dimensioni differenti.

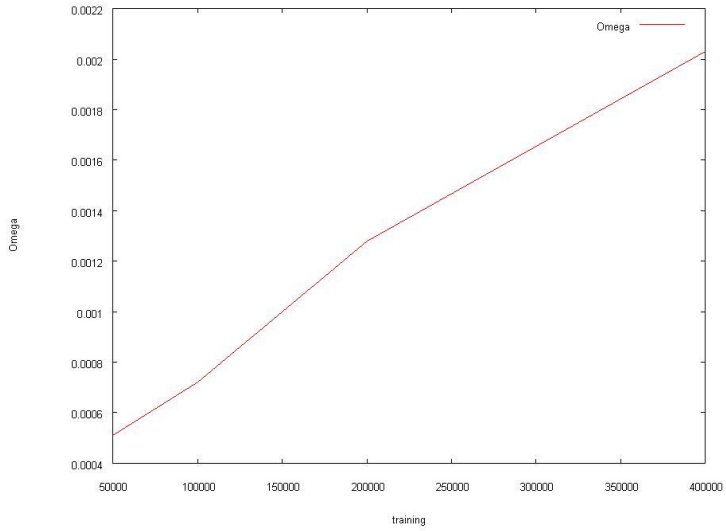

Figura 6.5: Valore medio della Omega utilizzando AssociationRules con training set di dimensioni differenti.

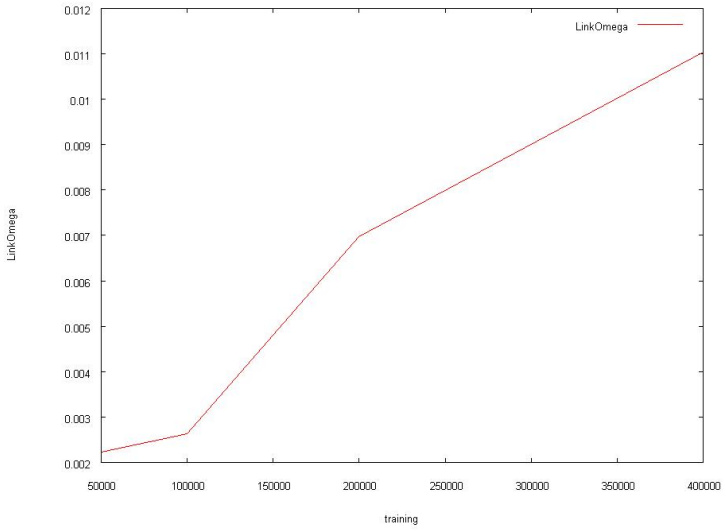

Figura 6.6: Valore medio della LinkOmega utilizzando AssociationRules con training set di dimensioni differenti.

I test effettuati sugli stessi suggerimenti con le valutazioni basate sulla sintassi (grafico 6.8) hanno dato valori meno precisi, tanto che per le valutazione dei suggerimenti generati con il training set più grande diminuiscono.

Anche la valutazione dell'algoritmo in versione Online (6.7) con metriche basate sulla sintassi rimane quasi costante all'aumentare della dimensione della cache.

Nei successivi test svolti, abbiamo messo a confronto tra loro le valutazioni dei differenti algoritmi. Come primo test sono state confrontate le curve ottenute valutando il comportamento degli algoritmi offline con l'aumento del training set.

Nel grafico 6.9, si può notare che la curva relativa all'algoritmo basato su regole associative cresce molto più velocemente di quella relativa all'algoritmo basato su click-throught. Questo comportamento è probabilmente dovuto al fatto che la quantità di regole associative che vengono trovate utilizzando un training set piccolo è molto esiguo, e questo numero aumenta molto velocemente. Il lato negativo è che l'algoritmo basato su click-through riesce in media a suggerire un numero maggiore di volte come si evince dal grafico 6.10.

Nelle Figure 6.10 e 6.9 è stato messo a confronto anche l'algoritmo basato sul contesto (TermVector). Nonostante l'algoritmo utilizzi una grande quantità di informazioni, le valutazioni non sono migliori rispetto agli altri algoritmi. C'è da notare, comunque, che

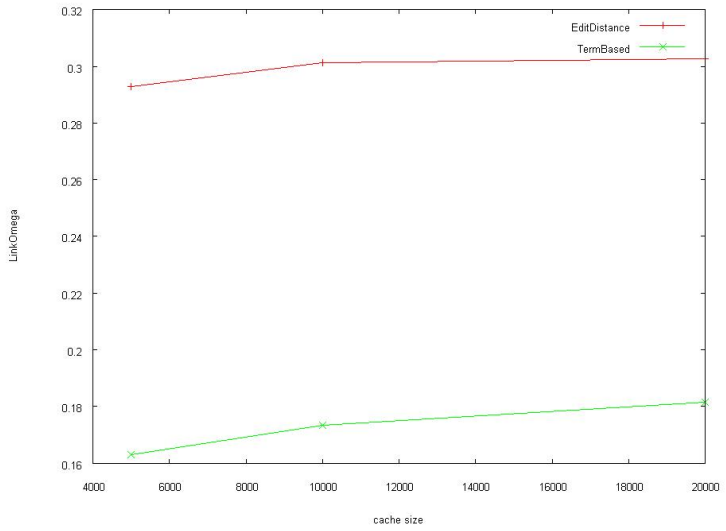

Figura 6.7: Valore medio calcolato con metriche basate sulla sintassi utilizzando CoverGraph in versione Online con cache set di dimensioni differenti.

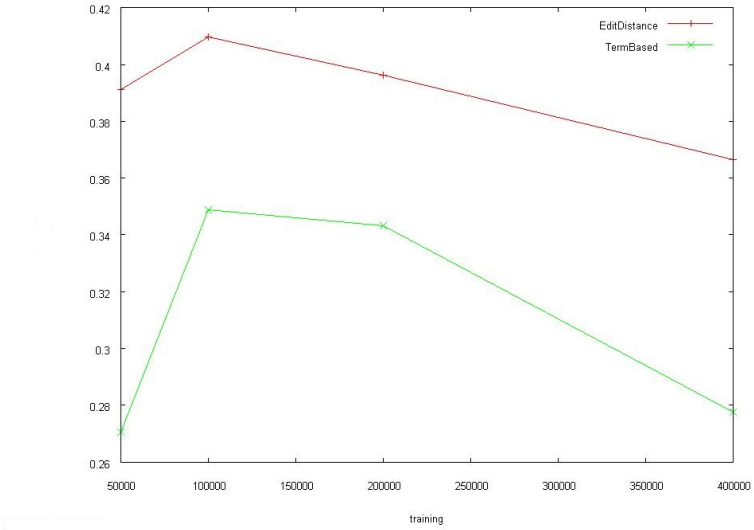

Figura 6.8: Valore medio calcolato con metriche basate sulla sintassi utilizzando CoverGraph con training set di dimensioni differenti.

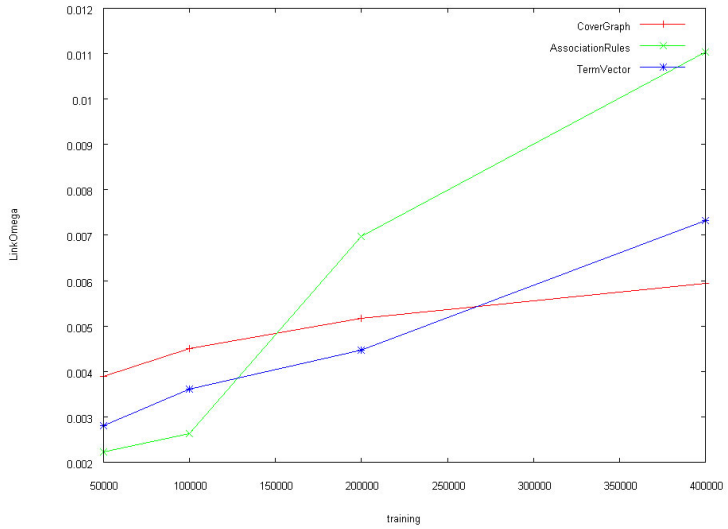

Figura 6.9: Confronto tra gli algoritmi AssociationRules, CoverGraph e TermVector.

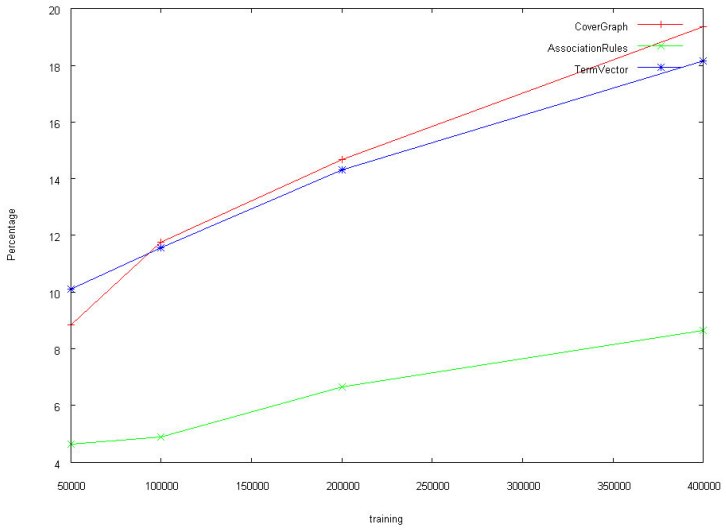

Figura 6.10: Percentuale dei suggerienti dati su suggerimenti richiesti.

sono stati utilizzati i documenti attualmente disponibili nel Web che sono quindi più recenti rispetto al query log. Questo potrebbe aver portato ad una perdita di precisione dovuta alla probabile modifica di alcune pagine, nonché la possibilità che alcuni domini utilizzati all'epoca vengano ora utilizzati per scopi completamente diversi.

Un test successivo è servito per valutare le differenze tra i classici sistemi Offline e i sistemi Online da noi proposti. Per il primo test sono stati confrontati i risultati ottenuti utilizzando la metrica LinkOverlap con il collector Time. L'utilizzo di tale collector ha permesso di valutare il comportamento dei sistemi di suggerimento con l'avanzare del tempo. I risultati ottenuti riescono a stimare in tal modo quanto influisce l'invecchiamento del modello nelle prestazioni degli algoritmi Offline.

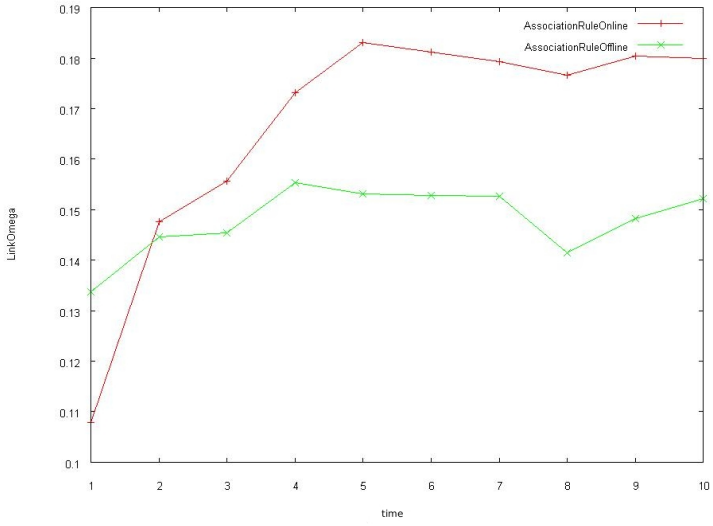

Figura 6.11: Differenze tra versione Online e Offline per l'algoritmo AssociationRules.

Il grafico mette in relazione l'algoritmo basato su regole associative in versione Offline con la versione Online. Per la valutazione, è stato diviso il periodo in cui sono stati dati i suggerimenti in dieci intervalli, alla scadenza di ogni intervallo temporale sono state calcolate le medie delle valutazioni ottenute dalla metrica.

Dal grafico si nota che, mentre le valutazioni ottenute per la versione Offline rimangono pressochè costanti nel tempo, l'algoritmo Online, produce suggerimenti migliori man mano che il modello viene popolato con informazioni più recenti.

Anche per l'algoritmo CoverGraph (Figura 6.12)la versione Online presenta qualche miglioramento anche se l'andamento delle due curve è molto simile. Un altro vantaggio

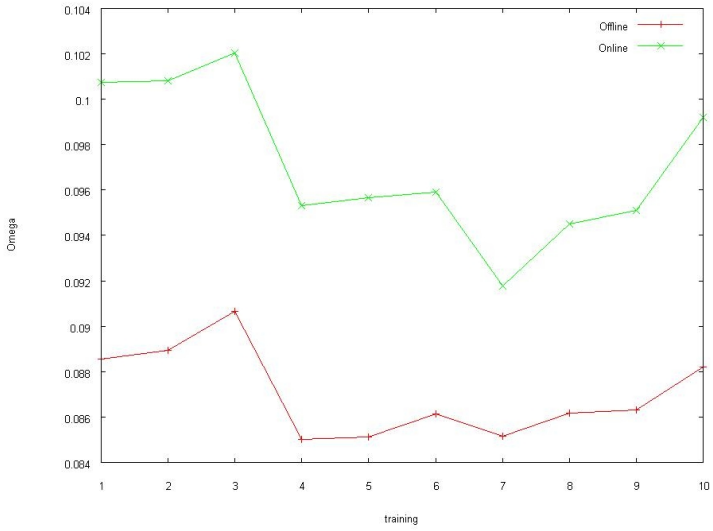

Figura 6.12: Differenze tra versione Online e Offline per l'algoritmo CoverGraph.

portato da questa versione Online è che il numero di suggerimenti dati rispetto a quelli non dati aumenta del 2% questo significa che considerando il numero di suggerimenti richiesti, vengono dati circa 60mila suggerimenti in più.

Capitolo 7

Conclusioni e Sviluppi Futuri

In questa tesi si è svolta in una prima fase, una ricerca dello stato dell'arte dei sistemi di raccomandazione, per il suggerimento di query, in motori di ricerca Web. Nel Capitolo 3, viene presentata una panoramica generale sugli algoritmi sviluppati fino ad ora in questo settore di ricerca cercando di evidenziare le differenze e le analogie che li caratterizzano.

Le problematiche legate a tali software sono la valutazione dell'efficacia cioè capire quanto sono attinenti i suggerimenti in relazione a ciò che l'utente sta cercando.

Ci siamo quindi posti come obiettivo quello la valutazione di Query Recommender System svolgibile in modo automatizzato, permettendoci di evitare l'approccio comune allo *user-study*.

Ci siamo inoltre proposti di progettare ed implementare alcune varianti Online per verificarne pregi e difetti, rispetto alle architetture Offline che in genere vengono proposte.

A tale scopo sono stati scelti alcuni tra gli algoritmi dello stato dell'arte in modo da avere a disposizione suggerimenti generati da sistemi basati su informazioni diverse. Tali algoritmi sono stati implementati sia nelle versioni Offline (non incrementali) trovate in letteratura, che nelle versioni Online (incrementali), ideate nell'ambito della tesi.

Tali algoritmi sono stati quindi integrati in un *framework*, realizzato appositamente in questo lavoro di tesi. Tale software, oltre che per simulare l'esecuzione di Query Recommender System, è stato progettato anche per la valutazione dei suggerimenti generati dagli algoritmi stessi.

Grazie alla flessibilità e alla adattabilità dimostrata, è stato possibile eseguire i diversi tipi di algoritmi, basandoci sui dati contenuti nel query log di *AOL* (America Online).

In seguito siamo stati in grado di valutare i suggerimenti utilizzando alcune metriche, tra le quali alcune ideate appositamente per questo tipo di problema.

La valutazioni hanno evidenziato come primo risultato una migliore affidabilità delle metriche ideate per sfruttare l'attività passata degli utenti, rispetto alle altre implementate, a sottolineare il fatto che la storia degli utenti è fondamentale per la previsione delle query.

Anche le valutazioni degli algoritmi Online hanno dato buoni risultati, sia qualitativamente che quantitativamente, indicando un minor degrado del modello nel tempo.

Lo studio svolto in questa tesi, oltre a permetterci di intuire le potenzialità dei sistemi Online, rende disponibile un sistema software estendibile con altri tipi di algoritmi e nuove metriche, tale sistema rende possibile un più veloce sviluppo di sistemi per Query Recommendation.

Una eventuale futura estensione di tale sistema con gli algoritmi descritti nel capitolo 3, porterà un immediato contributo a completare l'analisi in questo settore di ricerca.

Appendice A

Definizioni

A.1 TF-IDF

Questa sigla è l'acromimo di *term frequency-inverse document frequency* ed indica una funzione di peso che misura quanto è importante un termine per un documento. L'importanza aumenta in proporzione al numero di volte che la parola appare nel corpo del documento. Questa funzione è spesso utilizzata in *Information Retrieval* e nel *Text Mining*, e viene utilizzata nelle sue varianti nei motori di ricerca per ordinare i documenti da restituire in base al valore che assume su ognuno di essi.

Più formalmente, sia $f_{i,j}$ il numero di volte che il termine k_i appare nel documento d_j, allora definiamo TF come:

$$TF_{i,j} = \frac{f_{i,j}}{\max_x f_{z,j}}$$

dove il massimo è calcolato sulle frequenze di tutti i termini k_z che compaiono nel documento d_j

Definiamo quindi IDF come:

$$IDF_i = \log \frac{N}{n_i}$$

dove N è il numero totale di documenti e n_i è il numero di volte che il termine compare nei documenti.

La misura *TF-IDF* è calcolata come:

$$w_{i,j} = TF_{i,j} \cdot IDF_i$$

Un altro valore di questa misura, per un documento, ci dice che ci sono molte occorrenze del termine rispetto alle occorrenze totali in tutti i documenti. La misura tende quindi a filtrare i termini più comuni.

A.2 Cosine similarity

Indica una misura di similarità tra due vettori di dimensione n, calcolando il coseno dell'angolo.

Dati due vettori, A e B, la cosine similarity è calcolata utilizzando il prodotto scalare dei vettori diviso il prodotto delle norme:

$$CosineSimilarity = \cos(\Theta) = \frac{A \cdot B}{\| A \| \| B \|}$$

Nel ambito del *text matching*, i vettori A e B sono di solito i vettori *TF* (Appendice A.1).

A.3 K-Means

K-Means è un algoritmo di clustering che a partire da un parametro k e da un insieme di oggetti, divide gli oggetti in k gruppi, partizionandoli in base ai loro attributi. L'algoritmo si propone di minimizzare la distanza media di ogni oggetto al proprio centroide calcolato inizialmente nello spazio degli oggetti [1].

L'algoritmo segue una procedura iterativa che può essere riassunta come segue:

1. si scelgono k centroidi nello spazio degli oggetti che si vogliono clusterizzare;
2. ogni oggetto viene assegnato ad un centroide utilizzando una qualsiasi funzione di distanza;
3. una volta che tutti gli oggetti sono stati assegnati, si calcolano dei nuovi centroidi come media degli oggetti ai quali gli sono stati assegnati;

[1] nella versione più semplice i centroidi iniziali sono calcolati casualmente.

4. si ripetono i due passi precedenti finché i centroidi ricalcolati sono uguali ai precedenti.

La somma delle distanze medie degli oggetti dai propri centroidi diminuisce necessariamente ad ogni iterazione e l'algoritmo termina sempre.

A causa della scelta casuale dei centroidi iniziali però, non è detto che l'algoritmo trovi sempre un minimo globale, ma potrebbe terminare su un minimo locale. Per questo motivo, spesso, l'algoritmo viene ripetuto più volte per poi scegliere la configurazione migliore.

Alcune varianti invece utilizzano metodi più complessi per determinare i k centroidi, metodi che vengono comunque scelti in base alle esigenze.

Nonostante i limiti dell'algoritmo, tra i quali anche il problema di dover specificare il numero di cluster all'inizio della computazione, il k-means risulta essere molto più veloce di altri algoritmi di clustering.

A.4 DBSCAN

Il DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*) è un metodo di clustering basato sulla densità, quindi raggruppa regioni di punti con densità sufficientemente alta.

Per ogni oggetto, saranno trovati i vicini che ricadono in un raggio dato come parametro in ingresso; se il numero di tali vicini è superiore ad un fattore di soglia (anch'esso fornito in input all'algoritmo), allora tali punti faranno parte del medesimo cluster di quello dell'oggetto che si sta osservando (in questo caso il nostro punto sarà denominato *core point*).

Al termine dell'algoritmo avremo dei punti appartenenti a cluster e punti lasciati liberi; questi ultimi saranno *rumore*.

L'algoritmo si basa sulla suddivisione dei punti in:

Punti Core se hanno più di un numero prefissato di punti in un loro intorno di raggio prefissato;

Punti Bordo (o Frontiera) se sono nell'intorno di un punto core, ma non hanno sufficienti vicini per essere anch'essi punti core;

Punti Rumore tutti quei punti che non sono nè core nè bordo.

A.5 Apriori

L'Apriori è un algoritmo per l'apprendimento di regole associative. L'Apriori opera su database contenente transazioni.

Dato un insieme di *itemset* (insime di oggetti), l'algoritmo cerca di trovare sottoinsiemi di essi che siano presenti nell'insieme delle transazioni almeno S volte, dove S è il supporto definito dall'utente.

L'Apriori utilizza un approccio breadth-first per la generazione dei candidati e sfrutta la proprietà di anti-monotonia del supporto, per operare un *pruning*[2] sulle regole da generare.

$$\forall X, Y : (X \subseteq Y) \longrightarrow s(X) \geq s(Y)$$

L'algoritmo inizia controllando il supporto dei singoli elementi, elimina dal modello tutti quelli con supporto non sufficiente e genera tutti le coppie possibili con gli elementi rimasti. Quindi si procede di nuovo al calcolo delle similarità.

Più precisamente:

1. Sia $k = 1$
2. Si generano gli *itemset* frequenti di lunghezza 1
3. Ad ogni passo:
   (a) Si generano i candidati di lunghezza $k+1$ a partire dagli *itemset* frequenti di lunghezza k.
   (b) Si eliminano i candidati che contengono *itemset* di lunghezza k non frequenti
   (c) Calcolare il supporto dei candidati rimasti.
   (d) Eliminare i candidati non frequenti
4. Terminazione quando non si trovano nuovi *itemset* non frequenti.

A.6 Levenshtein Distance

La distanza di *Levenshtein*, o *edit distance*, è una misura per la differenza fra due stringhe e serve a determinare quanto due stringhe sono simili.

[2] per pruning si intende l'esclusione di alcuni rami dell'albero che descrive la generazione di regole associative

La distanza di Levenshtein tra due stringhe A e B è il numero minimo di modifiche elementari che consentono di trasformare la A nella B. Per modifica elementare si intende:

1. la cancellazione di un carattere;
2. la sostituzione di un carattere con un altro;
3. l'inserimento di un carattere.

Un algoritmo usato comunemente per calcolare la distanza di Levenshtein richiede l'uso di una matrice di $(n+1)\times(m+1)$, dove n e m rappresentano le lunghezze delle due stringhe. Lo pseudocodice in Algoritmo 3 rappresenta la funzione *LevenshteinDistance* che prende come argomenti due stringhe *str1* e *str2* di lunghezza *lenStr1* e *lenStr2* e ne calcola la distanza di Levenshtein.

Algoritmo 3 Levenshtain Distance

```
for i1 := 0 to lenStr1 do
    d[i1, 0] ← i1
end for
for i2 := 0 to lenStr2 do
    d[0, i2] ← i2
end for
for i1 := 1 to lenStr1 do
    for i2 := 1 to lenStr2 do
        if str1[i1] = str2[i2] then
            cost ← 0
        else
            cost ← 1
        end if
        d[i1, i2] ← minimum(d[i1 − 1, i2] + 1, d[i1, i2 − 1] + 1, d[i1 − 1, i2 − 1] + cost)
    end for
end for
```

A.7 Indice di correlazione di Pearson

Il coefficiente di correlazione (lineare) di Pearson (detto anche di *Bravais-Pearson*) tra due variabili aleatorie o due variabili statistiche X e Y è definito come la loro covarianza

divisa per il prodotto delle deviazioni standard delle due variabili:

$$\rho_{xy} = \frac{\sigma_{xy}}{\sigma_x \cdot \sigma_y}$$

dove ρ_{xy}, è la covarianza tra X e Y, mentre σ_x e σ_y sono le due deviazioni standard.

Il coefficiente assume valori compresi tra -1 e $+1$.

Bibliografia

[1] Gediminas Adomavicius and Er Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17:734–749, 2005.

[2] Reiner Kraft And. Mining anchor text for query refinement, 2004.

[3] Ricardo Baeza-Yates. Graphs from search engine queries. In *Theory and Practice of Computer Science (SOFSEM)*, volume 4362 of *LNCS*, pages 1–8, Harrachov, Czech Republic, January 2007. Springer.

[4] Ricardo Baeza-Yates, Carlos Hurtado, and Marcelo Mendoza. *Query Recommendation Using Query Logs in Search Engines*, volume 3268/2004 of *Lecture Notes in Computer Science*, pages 588–596. Springer Berlin / Heidelberg, November 2004.

[5] Ricardo Baeza-Yates and Alessandro Tiberi. Extracting semantic relations from query logs. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 76–85, New York, NY, USA, 2007. ACM.

[6] Ranieri Baraglia and Fabrizio Silvestri. An online recommender system for large web sites. In *WI '04: Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 199–205, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Doug Beeferman and Adam Berger. Agglomerative clustering of a search engine query log. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 407–416, New York, NY, USA, 2000. ACM.

[8] Nicholas J. Belkin and W. Bruce Croft. Information filtering and information retrieval: two sides of the same coin? *Commun. ACM*, 35(12):29–38, 1992.

[9] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. pages 43–52.

[10] Moses Charikar, Chandra Chekuri, and Tomás Feder Rajeev Motwani. Incremental clustering and dynamic information retrieval, 1997.

[11] Silviu Cucerzan and Ryen W. White. Query suggestion based on user landing pages. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 875–876, New York, NY, USA, 2007. ACM.

[12] Michael Pazzani Department and Michael J. Pazzani. A framework for collaborative, content-based and demographic filtering. *Artificial Intelligence Review*, 13:393–408, 1999.

[13] Nadav Eiron and Kevin S. McCurley. Analysis of anchor text for web search. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 459–460, New York, NY, USA, 2003. ACM.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 323–333, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[15] Barry Smyth Evelyn Balfe. A comparative analysis of query similarity metrics for community-based web search. In *Case-Based Reasoning Research and Development*, pages 63–77. Springer Berlin / Heidelberg, 2005.

[16] Tom Fawcett. An introduction to roc analysis. *Pattern Recogn. Lett.*, 27(8):861–874, 2006.

[17] Bruno Fonseca Federal, Bruno M. Fonseca, and Edleno S. De Moura. Using association rules to discover search engines related queries. In *In LA-WEB*, pages 66–71, 2003.

[18] Elena Gaudioso Felix H. del Olmo, Eduardo H.Martin. A common framework and metric for recommender systems:a proposal. 2007.

[19] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35:61–70, 1992.

[20] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[21] John Lafferty and Guy Lebanon. Diffusion kernels on statistical manifolds. *J. Mach. Learn. Res.*, 6:129–163, 2005.

[22] Zhiyuan Liu and Maosong Sun. Asymmetrical query recommendation method based on bipartite network resource allocation. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 1049–1050, New York, NY, USA, 2008. ACM.

[23] Hao Ma, Haixuan Yang, Irwin King, and Michael R. Lyu. Learning latent semantic relations from clickthrough data for query suggestion. In *CIKM '08: Proceeding of the 17th ACM conference on Information and knowledge management*, pages 709–718, New York, NY, USA, 2008. ACM.

[24] Le Gruenwald Margaret H. Dunham, Yongqiao Xiao. A survey of association rules. 2001.

[25] Ricard Marxer, Piotr Holonowicz, and Hendrik Purwins. Dynamical hierarchical self-organization of harmonic, motivic, and pitch categories. In *Music, Brain and Cognition. Part 2: Models of Sound and Cognition, held at NIPS)*. Vancouver, Canada, 2007. (to appear).

[26] Qiaozhu Mei, Dengyong Zhou, and Kenneth Church. Query suggestion using hitting time. In *CIKM '08: Proceeding of the 17th ACM conference on Information and knowledge management*, pages 469–478, New York, NY, USA, 2008. ACM.

[27] Nish Parikh and Neel Sundaresan. Inferring semantic query relations from collective user behavior. In *CIKM '08: Proceeding of the 17th ACM conference on Information and knowledge management*, pages 349–358, New York, NY, USA, 2008. ACM.

[28] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 1, New York, NY, USA, 2006. ACM.

[29] Yonggang Qiu and Hans-Peter Frei. Concept based query expansion. In *SIGIR '93: Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 160–169, New York, NY, USA, 1993. ACM.

[30] Vijay V. Raghavan and Hayri Sever. On the reuse of past optimal queries. In *in Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 344–350. ACM Press, 1995.

[31] B. Ribeiro-Neto R.Baeza-Yates. Modern information retrieval, 2004.

[32] Paul Resnick and Hal R. Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, March 1997.

[33] Badrul Sarwar, George Karypis, Joseph Konstan, and John Reidl. Item-based collaborative filtering recommendation algorithms. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 285–295, New York, NY, USA, 2001. ACM.

[34] Upendra Shardanand and Patti Maes. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1, pages 210–217, 1995.

[35] Danny Sullivan. Searches per day. `http://searchenginewatch.com/2156461`.

[36] Ji-Rong Wen, Jian-Yun Nie, and Hong-Jiang Zhang. Clustering user queries of a search engine. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 162–168, New York, NY, USA, 2001. ACM.

[37] Ji-Rong Wen, Jian-Yun Nie, and Hong-Jiang Zhang. Clustering user queries of a search engine. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 162–168, New York, NY, USA, 2001. ACM.

[38] Ryen W. White and Dan Morris. Investigating the querying and browsing behavior of advanced search engine users. In *SIGIR '07: Proceedings of the 30th annual*

international ACM SIGIR conference on Research and development in information retrieval, pages 255–262, New York, NY, USA, 2007. ACM.

[39] Jinxi Xu and W. Bruce Croft. Query expansion using local and global document analysis. In *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 4–11, New York, NY, USA, 1996. ACM.

[40] Osmar R. Zaïane and Alexander Strilets. Finding similar queries to satisfy searches based on query traces. In *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 207–216, London, UK, 2002. Springer-Verlag.

[41] Zhiyong Zhang and Olfa Nasraoui. Mining search engine query logs for query recommendation. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 1039–1040, New York, NY, USA, 2006. ACM.