Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica Magistrale

DEFINIZIONE DI UN SISTEMA DI RACCOMANDAZIONE BASATO SU RETI COMMERCIALI

Tesi di Laurea in Algoritmi Avanzati

Relatore:
Chiar.mo Prof.
Alan Albert Bertossi

Presentata da:
Marco Buzzoni

Relatore aziendale:
Dott.
Salvatore Agostino Romeo

Sessione 2
2012/2013

SOMMARIO

Nel contesto economico odierno i sistemi di raccomandazione rappresentano uno strumento utile al fine di aumentare le vendite con pubblicità e promozioni su misura per ciascun utente. Tali strumenti trovano numerose applicazioni nei siti di e-commerce, si pensi ad Amazon o a MovieLens. Esistono diverse tipologie di sistemi di raccomandazione, che si differenziano principalmente per il modo con cui sono prodotte le raccomandazioni per gli utenti. In questa tesi se ne vuole definire una nuova tipologia, che superi la restrizione del vincolo ad un sito a ad una società, fornendo agli utenti raccomandazioni di prodotti acquistabili in negozi reali e il più possibile accessibili, nel senso geografico del termine. Si e inoltre astratto il concetto di raccomandazione, passando da un insieme omogeneo di oggetti ad un insieme eterogeneo di entità ottenibili attraverso lo svolgimento di attività. Con queste premesse il sistema da definire dovrà raccomandare non più solo entità, ma entità e shop presso i quali sono disponibili per le persone.

Indice

Capitolo 1

Introduzione

In questa tesi viene presentato un algoritmo per un nuovo sistema di raccomandazione, che si colloca nel contesto delle reti commerciali. Per rete commerciale si intende un insieme di negozi collocati ad esempio in un centro commerciale o in una via. Un sistema di raccomandazione di questo tipo rappresenta un utile strumento da un lato per le persone, che riescono in questo modo a trovare prodotti interessanti nella propria zona senza dover necessariamente acquistare prodotti su internet, dall'altro per i commercianti, che dopo aver aderito al sistema avranno la possibilità di acquisire nuovi clienti senza campagne pubblicitarie particolarmente costose. Potranno semplicemente inserire nel sistema ciò che vogliono fornire ai propri potenziali clienti, e tra questi ultimi coloro che saranno interessati potranno recarsi presso il negozio. Questo tipo di raccomandazione è reso disponibile dagli smartphone, che consentono la geolocalizzazione dell'utente (previo consenso esplicito) e consentono di ricevere in tempo reale su quest'ultimo dispositivo le raccomandazioni su prodotti di proprio interesse nella propria zona. Essa può essere un quartiere, una singola rete commerciale ma anche una città, dipende dalla disponibilità dell'utente a spostarsi. Un altro obiettivo che si è voluto portare a termine con questo sistema di raccomandazione consiste nell'astrazione del concetto di raccomandazione. Esistono diversi sistemi di raccomandazione che si prestano a specifiche tipologie di raccomandazioni, MovieLens raccomanda film da guardare ai propri utenti, Amazon raccomanda l'acquisto di prodotti sul proprio sito, VERSIFI raccomanda musica e ne esistono altri. Nel sistema di raccomandazione di cui si discute in questa tesi il dominio delle raccomandazioni non è ristretto ad uno specifico settore: si

può raccomandare l'acquisto di un bene, l'ascolto di un brano, la partecipazione ad un evento organizzato da un negozio, o qualsiasi altra cosa sia stata decisa da un negoziante che ha aderito al sistema.

Nel capitolo 2 verrà definito con maggiore chiarezza il contesto di applicazione ed il modello di riferimento, oltre ad una illustrazione dello stato dell'arte per quanto concerne i sistemi di raccomandazione, che attualmente sono divisi in quattro categorie principali: *content-based*, *collaborative filtering*, *hybrid* e *social-semantic*. Nella prima categoria rientrano quei sistemi di raccomandazione che producono raccomandazioni a partire dalle precedenti preferenze di acquisto della persona. Le raccomandazioni devono essere diversificate, nel senso che quando una persona ha effettuato pochi acquisti avrà anche pochi interessi e il suo profilo non sarà ancora completamente definito, proprio per questo non sempre ha senso raccomandare oggetti strettamente appartenenti ai pochi interessi della persona. Ad esempio, in un sistema di raccomandazione di news non avrebbe senso raccomandare lo stesso articolo, magari scritto da due giornalisti diversi, alla stessa persona. I sistemi di tipo *collaborative filtering* si basano sulla similarità delle persone o degli oggetti per produrre raccomandazioni. In particolare, data una persona, vengono raccomandati oggetti acquistati da persone simili ad essa, ovvero che hanno una cronologia degli acquisti, dei rating o degli interessi simile ad essa. Quando il confronto viene fatto tra le persone, per poi raccomandare oggetti alla persona interessata, si parla di *user-based collaborative filtering*. In altri casi si verifica la similarità tra oggetti, dove si dice che due oggetti sono simili se sono stati acquistati o votati da almeno una persona in comune, per poi raccomandare un oggetto a persone che hanno acquistato oggetti simili ad esso ma che non ne sono in possesso: in questo caso si parla di *item-based collaborative filtering*. I sistemi di tipo *hybrid* combinano i metodi delle due tipologie precedenti per produrre raccomandazioni: possono ad esempio utilizzare un metodo *content-based* nelle previsioni iniziali per poi passare ad un metodo *collaborative filtering*, possono combinare linearmente le due tipologie di previsioni, oppure sfruttare le due tipologie di previsioni con altri metodi. Un sistema di tipo *social-semantic* è stato introdotto recentemente, nel 2012, e osserva persone ed oggetti contenuti nel dataset unendo le prime in un *social network*, ovvero un grafo dove i nodi sono le persone e gli archi sono una relazione tra esse. Nel caso di un sistema di raccomandazione possono essere ad esempio gli oggetti acquistati in

comune. La raccomandazione di un oggetto viene fatta eseguendo una visita nel *social network* e valutando le persone cui può essere proposto. Un altro criterio di distinzione tra i diversi sistemi di raccomandazione è derivato dalla funzione di raccomandazione. In un sistema *heuristic based* la funzione di raccomandazione è appunto una funzione euristica, che sfrutta il dataset nel suo complesso per produrre le raccomandazioni. In contrapposizione, in un sistema *model based* la funzione di raccomandazione sfrutta un modello matematico per poter ridurre la porzione di dataset da visitare.

Nel capitolo 3 viene definito il nuovo sistema di raccomandazione. In particolare, viene definito un grafo che riassume le relazioni tra *entità*, *persone* e *shop*, chiamato *r-grafo*, a partire dal quale viene calcolato il *social network* delle persone, chiamato anche *shop one-node projection graph*. A partire da questo grafo si crea un vettore *top-k degree* contenente i primi k nodi di grado più alto. A partire da ognuno di questi nodi, per ogni *entità*, si fa una visita *DFS* dello *shop one-node projection graph*, raccomandando l'*entità* fino a quando non si incontra un nodo a cui non è possibile raccomandare l'*entità* per assenza di *shop* che la forniscono, si incontra una *persona* che non soddisfa la similarità richiesta con l'*entità* oppure si raggiunge la lunghezza massima del cammino nel grafo. Il sistema di raccomandazione usa una funzione di similarità, che si basa sulla similarità semantica tra una *persona* ed una *entità*, per il cui funzionamento a quest'ultima devono essere associati uno o più alberi di tassonomia, mentre ad ogni *persona*, oltre alla cronologia delle *attività* svolte, deve essere associata una tassonomia che indica gli interessi relativi alle *attività* della propria cronologia ovvero i propri interessi. Per quanto riguarda le persone che ancora non hanno una loro cronologia di *attività*, le raccomandazioni vengono fatte in base al *trend* degli *shop* che esse conoscono. Questi ultimi vengono raccomandati in base alla cronologia di utilizzo, più precisamente in base ad un *forgetting factor* ed alla similarità con gli interessi della *persona*.

Nel capitolo 4 viene fatto un confronto sulla base di alcune metriche interessanti nel valutare un sistema di raccomandazione e vengono illustrati alcuni risultati peculiari di questo tipo di sistema di raccomandazione. La principale metrica di valutazione usata nel confronto è la f-measure, ottenuta combinando precision e recall. L'algoritmo *shop based* definito in questa tesi supera il *node based* [1] in tutti i confronti effettuati, tra i quali vi sono ad esempio il confronto della velocità di apprendimento, del comportamento

al variare del threshold imposto, del variare della configurazione della rete eccetera. Un risultato particolarmente notevole si ha nel confronto della velocità di apprendimento: mentre l'algoritmo *node based* migliora la qualità delle proprie raccomandazioni all'aumentare dei dati nel training set, come ci si aspetta di solito in questo tipo di valutazione, l'algoritmo *shop based* ha registrato valori nettamente superiori anche con training set molto piccolo, di 20-40 *entità* nella cronologia delle *attività* per ogni *persona*. Questo indica che vengono prodotte raccomandazioni particolarmente rilevanti anche per persone entrate da poco nel sistema.

Capitolo 2

Definizione del problema

I sistemi di raccomandazione trovano applicazione in diversi settori, ma il loro scopo è unico: aiutare le persone ad effettuare scelte basandosi su diversi aspetti. Data una *persona*, questi aspetti possono essere ad esempio la propria cronologia, ovvero gli acquisti già effettuati o i voti positivi già dati, o le preferenze di persone simili ad essa. In base a questi aspetti e a come sono prodotte le raccomandazioni i sistemi di raccomandazione si dividono in varie tipologie. Nelle successive sezioni viene introdotto il funzionamento alla base e le diverse tipologie. In questa tesi si vuole definire una nuova tipologia di sistema di raccomandazione, basato su reti commerciali, e specificare un possibile algoritmo.

2.1 Sistemi di raccomandazione

In questo capitolo si vuole definire dettagliatamente il contesto di applicazione dei sistemi di raccomandazione e i fattori che ne hanno motivato lo sviluppo, oltre a riassumerne lo stato dell'arte. Si definisce prima di tutto in cosa consiste un problema di raccomandazione[2]. Sia C l'insieme delle persone e sia S l'insieme degli oggetti che possono essere raccomandati. Lo spazio S può essere molto grande, fino a comprendere milioni di oggetti. Si consideri una funzione di utilità u che misura l'utilità dell'oggetto $s \in S$ per la persona $c \in C$. Nello specifico:

$$\forall c \in C, \quad s'_c = \underset{s \in S}{\operatorname{argmax}}\, u(c, s). \tag{2.1}$$

I rating degli oggetti possono essere definiti dalle persone oppure calcolati dal sistema, nel qual caso la funzione di utilità è una funzione di profitto. Una volta che sono stati calcolati i rating degli oggetti, la raccomandazione è fatta selezionando l'oggetto o gli oggetti che hanno prodotto il rating più alto. La funzione di utilità è quindi l'elemento più importante del sistema di raccomandazione, poiché determina la sua precisione.

Un sistema di raccomandazione è un utile strumento per produrre raccomandazioni di oggetti a partire dalla cronologia di utilizzo della persona che ne usufruisce. Come riportato in [2, 3] esistono già diverse applicazioni di questi strumenti, che costituiscono ancora oggi una importante area di ricerca, e spaziano dalla raccomandazione di beni di consumo nel caso di Amazon, alla raccomandazione di film nel caso di MovieLens alla raccomandazione di news o di articoli di ricerca. Principalmente un sistema di raccomandazione esegue tre operazioni: ottiene le preferenze delle persone dai dati in input, calcola le raccomandazioni e le presenta alle persone. I sistemi di raccomandazione possono essere raggruppati in tre categorie principali, in base all'approccio usato per ottenere le raccomandazioni, più una quarta che è nata in seguito alla crescente diffusione dei social network, e che viene enunciata per ultima:

- *Approcci content-based*: alla persona saranno raccomandati oggetti simili a quelli che ella ha preferito in passato;
- *Approcci collaborative*: alla persona saranno raccomandati oggetti che persone con preferenze simili alle sue hanno preferito in passato;
- *Approcci ibridi*: questa tipologia racchiude i sistemi di raccomandazione che combinano tecniche usate nelle due precedenti tipologie;
- *Approcci semantic-social*: si considera un insieme di persone ed uno di oggetti, dove le persone sono connesse da una rete sociale e sia queste ultime sia gli oggetti sono descritti da una tassonomia.

In base alla funzione utilizzata i sistemi di raccomandazione Content-Based e collaborative possono essere:

- *Heuristic-based*: solitamente prendono come input vari dati e calcolano le raccomandazioni sull'intero dataset delle persone usando metriche di similarità;

- *Model-based*: costruiscono un modello usando tecniche basate su training set e lo validano su dati di test set. Tale modello viene poi usato per calcolare lo score per gli elementi non ancora raccomandati alle persone. A differenza dei metodi heuristic-based viene dato solo un sottoinsieme di persone attive del dataset come input al modello, che produrrà da esso le previsioni.

Nel seguito verranno discusse in maniera approfondita queste diverse tipologie.

2.2 Raccomandazioni Content-Based

Questa tipologia di raccomandazioni, principalmente utilizzata per contenuti testuali, costituisce un'evoluzione degli approcci di information retrieval [4] con l'aggiunta di un profilo della persona che contiene informazioni sulle preferenze e sui bisogni della persona, recuperate in modo indiretto oppure diretto tramite questionari. Data la natura delle raccomandazioni solitamente un oggetto viene rappresentato come un insieme delle keyword più importanti, che sono calcolate con il metodo term frequency/inverse document frequency (TF-IDF) [5].

Sia N il numero di documenti raccomandabili alle persone e sia k_j una keyword che appare in n_i documenti. Sia $f_{i,j}$ il numero di volte che k_i appare nel documento d_j. Allora $TF(i,j)$, la funzione di term frequency della keyword k_i nel documento d_j, è definita come

$$TF_{i,j} = \frac{f(i,j)}{\max_z f(z,j)}, \tag{2.2}$$

dove il massimo è calcolato sulle frequenze $f_{i,j}$ di tutte le keyword k_i che appaiono nel documento d_j. La formula dell'inverse document frequency IDF della keyword k_i invece è definita come segue:

$$IDF_i = \log \frac{N}{n_i}. \tag{2.3}$$

Il peso TF-IDF per la keyword k_i nel documento d_j è definito come

$$w_{i,j} = TF_{i,j} * IDF_i \tag{2.4}$$

e il contenuto di un documento è definito come

$$Content(d_j) = w_{1,j}, ..., w_{k,j}. \tag{2.5}$$

Il motivo per cui si usa TF-IDF per calcolare i pesi delle parole è che le keyword che appaiono in troppi documenti diventano inutili nel determinare documenti rilevanti.

Quanto detto fino ad ora consente di calcolare il profilo dei documenti, ovvero lo strumento che viene utilizzato per determinare l'idoneità di un documento. Un altro elemento da tenere in considerazione è il profilo delle persone, che come già accennato contiene i loro interessi e bisogni e viene generato dalla cronologia delle loro attività. Sia $ContentBasedProfile(c)$ un vettore di pesi delle keyword di maggior interesse per la persona c. Per calcolare questi pesi si possono usare diversi metodi, uno dei quali è una sorta di media dei pesi dei vettori dei documenti facenti parte della cronologia della persona, oppure si può utilizzare un classificatore Bayesiano per calcolare la probabilità che un documento sia interessante. Per quanto riguarda i metodi heuristic-based, la funzione di utility $u(c, s)$ sopracitata è definita come:

$$u(c, s) = score(ContentBasedProfile(c), Content(s)), \tag{2.6}$$

e sapendo che i due argomenti della funzione di score sono vettori di pesi calcolati con TF-IDF si può calcolare la funzione di utility come similarità cosenica tra tali vettori [4]:

$$u(c, s) = cos(\vec{w}_c, \vec{w}_s) = \frac{\vec{w}_c \cdot \vec{w}_s}{\|\vec{w}_c\|_2 \times \|\vec{w}_s\|_2}. \tag{2.7}$$

In alternativa si possono utilizzare metodi di clustering [6]. Metodi model-based invece coinvolgono l'uso di classificatori Bayesiani [7], tecniche di clustering [8], tecniche di machine learning e reti neurali. Ad esempio un classificatore Bayesiano può essere usato per stimare la probabilità che la pagina p_j appartenga ad una certa classe C_i, nello specifico rilevante o irrilevante, dato un insieme di keywords $k_{1,j}, ..., k_{n,j}$ su quella pagina:

$$P(C_i|k_{1,j}\&...\&k_{n,j}) \tag{2.8}$$

Questo tipo di raccomandazione presenta diversi problemi. In primo luo-

go, è particolarmente indicata a contenuti testuali per via dell'esistenza di tecniche di information retrieval che consentono l'estrazione automatica degli elementi necessari al sistema di raccomandazione. Con altri dati, ad esempio video, sarebbe richiesto un approccio manuale, che sarebbe molto costoso in termini di risorse umane. Secondo, due elementi che presentano le stesse keyword sono indistinguibili. Un altro problema è la raccomandazione di oggetti che sono troppo simili ad oggetti già raccomandati ad una persona. Ad esempio una persona che ha appena letto una news su un particolare evento difficilmente vorrà rileggere la stessa news scritta da un altro giornalista: Daily Learner [8] filtra gli oggetti non solo se sono troppo diversi ma anche se sono troppo simili. In generale è meglio presentare alla persona un insieme eterogeneo di raccomandazioni. L'ultimo problema è che per una nuova persona, dato lo scarso numero di oggetti votati, avrà raccomandazioni poco efficaci.

2.3 Raccomandazioni Collaborative

Le raccomandazioni collaborative (o sistemi di raccomandazione collaborative filtering) prevedono l'utilità degli oggetti basandosi sui ratings di altre persone. In questo caso, l'utilità $u(c, s)$ dell'oggetto s per la persona c è basata sull'utilità $u(c_j, s)$ assegnata ad s da quelle persone $c_j \in C$ che sono *simili* alla persona c. Tipici di questo tipo di sistema di raccomandazione sono Amazon, oppure GroupLens [9], un sistema di raccomandazione di news Usenet che si affida ai gusti e alle conoscenze a priori delle persone per produrre le proprie raccomandazioni. In questo caso, alcuni algoritmi heuristic-based sono [10, 11]. In particolare [11] propone anche un sistema di raccomandazione basato sul Web mining e sulla tassonomia dei prodotti. In questo sistema, il web mining è utilizzato per popolare il database contenente le preferenze di consumo delle persone, mentre la tassonomia dei prodotti è utilizzata per velocizzare la ricerca dei vicini riducendo il database dei ratings. In [12] viene proposta una metodologia di raccomandazione basata su web mining, albero di decisione, association rule mining e tassonomia dei prodotti. Un altro approccio [13] consiste nell'applicazione della teoria dei grafi, mentre in [14] viene proposto l'uso di support vector machine.

In questi algoritmi il rating sconosciuto $r_{c,s}$ della persona c per l'oggetto s solitamente viene calcolato come un'aggregazione dei rating del sottoinsieme

delle N persone più simili a c, denotato con C', per lo stesso oggetto s:

$$r_{c,s} = \underset{c' \in C'}{\text{aggr}}\, r_{c',s}. \tag{2.9}$$

La funzione di aggregazione può essere una semplice media, ma in molti casi si utilizza la somma pesata:

$$r_{c,s} = k \sum_{c' \in C'} sin(c, c') \times r_{c',s} \tag{2.10}$$

dove k è un fattore normalizzante, solitamente $k = 1/\sum_{c' \in C'} |sim(c, c')|$ La somma pesata non tiene in considerazione il fatto che due persone possono usare *rating scale* differenti. Per risolvere questo problema si può usare la somma pesata aggiustata:

$$r_{c,s} = \bar{r}_c + k \sum_{c' \in C'} sin(c, c') \times (r_{c',s} - \bar{r}_{c'}) \tag{2.11}$$

dove il rating medio di una persona c, $\bar{r}_c$, è definito come

$$\bar{r}_c = (1/|S_c|) \sum_{s \in S_c} r_{c,s}, \tag{2.12}$$

dove $S_c = \{\, s \in S \mid r_{c,s} \neq \emptyset \,\}$. Sono stati utilizzati vari approcci per calcolare la similarità $sim(c, c')$ tra due persone: i più popolari sono la correlazione di Pearson [15] e la similarità cosenica[16], e si basano sui voti delle persone. Sia $S_{x,y}$ l'insieme di tutti gli oggetti covotati dalle persone x ed y, ovvero

$$S_{x,y} = \{\, s \in S \mid r_{x,s} \neq \emptyset \quad \& \quad r_{y,s} \neq \emptyset \,\} \tag{2.13}$$

Spesso, come anticipato, le funzioni di similarità utilizzate sono la correlazione di Pearson

$$sim(x, y) = \frac{\sum_{s \in S_{x,y}} (r_{x,s} - \bar{r}_x)(r_{y,s} - \bar{r}_y)}{\sqrt{\sum_{s \in S_{x,y}} (r_{x,s} - \bar{r}_x)^2 \sum_{s \in S_{x,y}} (r_{y,s} - \bar{r}_y)^2}} \tag{2.14}$$

oppure la similarità cosenica, dove i due persone x ed y sono trattati come vettori m-dimensionali, con $m = |S_{x,y}|$, e la similarità è data dal coseno

dell'angolo tra essi:

$$sim(x,y) = cos(\vec{x},\vec{y}) = \frac{\vec{x}\cdot\vec{y}}{\|\vec{x}\|_2 \times \|\vec{y}\|_2} = \frac{\sum_{s\in S_{x,y}} r_{x,s} r_{y,s}}{\sqrt{\sum_{s\in S_{x,y}} r_{x,s}^2}\sqrt{\sum_{s\in S_{x,y}} r_{y,s}^2}} \tag{2.15}$$

Solitamente si calcolano a periodi prefissati le similarità $sim(x,y)$ per tutte le persone e l'insieme $S_{x,y}$.

Quando le tecniche per calcolare la similarità sono applicate alle persone esse prendono il nome di *user-based collaborative filtering*. Sarwar et al. [17] hanno proposto di applicare tali tecniche agli oggetti: in questo caso si parla di *item-based collaborative filtering*. Questo secondo approccio, come riportato in [18, 17], permette di ottenere risultati di qualità superiore a quelli ottenuti usando il primo approccio e con tempi di calcolo migliori.

Alcuni algoritmi model-based sono: [19] che utilizza metodi di clustering per creare un algoritmo che consente di ottenere raccomandazioni in tempo $O(1)$, [20] che utilizza tecniche di analisi della semantica latente e molti altri. Essi usano collezioni di rating per apprendere un modello che è poi usato per generare previsioni. Ad esempio [16] propone un approccio probabilistico dove i rating sconosciuti sono calcolati come

$$r_{c,s} = E(r_{c,s}) = \sum_{i=1}^{n} i \times Pr(r_{c,s} = i | r_{c,s'}, s' \in S_c) \tag{2.16}$$

assumendo che i rating siano tra 0 ed n e l'espressione è la probabilità che la persona c dia un particolare voto all'oggetto s dati i precedenti voti della persona.

Anche per i sistemi di raccomandazione *collaborative filtering* esiste il problema del produrre raccomandazioni per nuove persone. Molte soluzioni a questo problema consistono nel usare l'approccio dei sistemi ibridi. Un problema analogo a questo, e che si risolve con la medesima strategia, è quello del raccomandare nuovi oggetti, che appena inseriti hanno ricevuto pochi voti e hanno di conseguenza pochi oggetti simili a loro da usare per produrre raccomandazioni. Un ulteriore problema che può presentarsi è la sparsità dei dati: in un sistema di raccomandazione i voti delle persone sono sempre un numero molto piccolo rispetto a tutti i rating che devono essere predetti. Questo porta a scarsità di persone simili e di conseguenza a scarse raccomandazioni [21]. Si cita infine un miglioramento della complessità usa-

to da Amazon per il calcolo della similarità tra oggetti nel proprio sistema di tipo *item-to-item collaborative filtering* [22]. Nel calcolo della similarità tra oggetti, soprattutto in grandi insiemi di oggetti come nel caso di Amazon, è la complessità $O(N^2M)$, dove M è il numero dei clienti ed N è il numero dei prodotti venduti, che comporta una inefficenza dell'algoritmo con conseguente spreco di CPU e memoria: questo perchè la matrice del grafo degli oggetti, analogamente al caso del grafo delle persone, presenta il problema della sparsità. Tale problema è stato risolto calcolando la similarità solo per oggetti effettivamente simili tra loro, ovvero che hanno in comune una o più persone. In questo caso particolare si ottiene una complessità inferiore,

Algorithm 2.1 Calcolo Efficiente Della Similarità tra Oggetti

```
for each item in items set I1 do
    for each customer C who purchased I1 do
        for each item I2 purchased by customer C do
                Record that a customer purchased Item I1 and I2
        end for
    end for
    for each customer C who purchased I1 do
            Compute the similarity between I1 and I2
    end for
end for
```

$O(NM)$. Lo pseudocodice di un algoritmo che usa questo sistema è riportato nell'Algoritmo 2.1.

Nel caso di un sistema user-based [22] per calcolare le raccomandazioni si potrebbe pensare ad una complessità $O(NM)$ perché, per ogni persona, si devono esaminare tutti i prodotti. Dato che nella maggior parte dei casi reali il vettore dei ratings delle persone è sparso, il tempo di calcolo spesso è vicino ad $N + M$, perché ogni persona tende ad essere esaminata $O(M)$ volte e non $O(MN)$ e solo un ristretto numero di persone ha votato o acquistato una percentuale significativa di oggetti in catalogo, richiedendo $O(N)$ tempo di calcolo. Un altro problema è il fattore di scaling: alcuni sistemi di raccomandazione possono contenere decine di milioni di persona. Amazon [22] tenta di risolvere il problema scartando le persone con pochi voti o acquisti, o scartando prodotti molto popolari o sconosciuti. Si può inoltre partizionare lo spazio in base alle categorie degli oggetti.

I metodi appena descritti hanno diversi effetti collaterali. In primo luogo

eliminando un certo numero di clienti, quelli selezionati saranno meno simili alla persona. In secondo luogo le persone che hanno votato solo gli oggetti molto popolari o sconosciuti, che sono stati rimossi dall'insieme degli oggetti, non riceveranno raccomandazioni. Terzo, il partizionamento dello spazio degli oggetti genera raccomandazioni ristrette ad una determinata categoria o partizione.

Fortunatamente un sistema di raccomandazione *collaborative filtering* fa pochi calcoli offline e i calcoli online scalano con la dimensione del dataset.

2.4 Raccomandazioni Ibride

Come è già stato detto in precedenza, gli approcci ibridi sono combinazioni di approcci *content-based* e *collaborative filtering*. In questo caso non si parla più di *heuristic-based* o di *model-based*, ma si categorizzano i sistemi in base al modo in cui vengono combinate le due precedenti categorie. Si hanno ad esempio l'introduzione di un previsore *content-based* in un approccio collaborativo per calcolare i rating addizionali [23], oppure la combinazione lineare dei rating dei due approcci sopracitati. In generale si può parlare delle seguenti classificazioni circa le combinazioni dei due approcci citati:

- implementazione dei due approcci separatamente per poi combinare le previsioni;
- incorporare alcune caratteristiche dei sistemi *content-based* in un approccio *collaborative filtering*;
- incorporare alcune caratteristiche dei sistemi *collaborative filtering* in un approccio *content-based*;
- costruire un modello unificato generale che incorpora entrambe le caratteristiche.

Di seguito si approfondiscono questi quattro approcci.

2.4.1 Combinazione delle Previsioni

Questo approccio consiste nell'implementare due diversi sistemi di raccomandazione, per poi procedere in uno dei due modi seguenti: si può usare una combinazione lineare dei rating in dei due sistemi di raccomandazione [24],

oppure si può scegliere il sistema da utilizzare in base a quale massimizza la metrica di qualità scelta. Come esempio del secondo metodo, il sistema DailyLearner [8] sceglie il sistema che può produrre raccomandazioni con il più alto livello di confidenza.

2.4.2 Aggiunta di Caratteristiche Content-Based all'Approccio Collaborativo

Alcuni sistemi, ad esempio quello descritto in [21], sono basati su tecniche collaborative mantenendo però un profilo *content-based* per ogni persona. Questo può consentire da un lato di risolvere alcuni problemi relativi alla sparsità dei ratings dell'approccio collaborativo, dato che non molte coppie di persone hanno un numero significativo di oggetti votati in comune. Inoltre con questo approccio un oggetto può essere raccomandato alla persona anche se non è stato votato bene da molte persone con profili simili.

2.4.3 Aggiunta di Caratteristiche Collaborativo all'Approccio Content-Based

In questa categoria l'approccio più comune è di applicare alcune riduzioni ad un gruppo di profili content-based. Un esempio è la semantica latente per creare una vista collaborativa di una collezione di profili delle persone rappresentati come vettori.

2.4.4 Sviluppo di un Modello Unificato di Raccomandazione

Un esempio di applicazione di questo approccio può essere l'uso di caratteristiche *content-based* e *collaborative filtering* (come gli anni o il sesso) in un singolo classificatore rule-based [25], in [26, 27] viene proposto un metodo probabilistico per combinare i due tipi di raccomandazione basato sull'analisi della semantica latente probabilistica [28].

Questo tipo di sistemi di raccomandazione può essere anche migliorato da tecniche knowedge-based come il ragionamento case-based per migliorare l'accuratezza delle raccomandazioni e mitigare alcune limitazioni quali ad esempio l'ingresso di nuove persone. Ad esempio, il sistema Entrée [29] usa la conoscenza del dominio dei ristoranti e del cibo per raccomandare ristoranti ai propri utenti. Per l'acquisizione della conoscenza del dominiio occorre che quest'ultima sia disponibile in un formato facilmente leggibile

da un calcolatore, ad esempio un'ontologia. Quest'ultimo approccio viene seguito in [30].

Inoltre, per velocizzare i tempi di computazione l'algoritmo parte dai primi K nodi del *user one-node projection graph*, passati ad esso in un vettore *top K degree*, ottenuti prendendo i primi K nodi di grado più alto.

2.5 Sistemi di Raccomandazione Semantic-Social

Questi sistemi di raccomandazione sono stati concepiti grazie al crescente diffondersi dei social network, in cui i nodi sono le persone e gli archi sono le interazioni sociali tra esse, ad esempio di amicizia. I social network hanno portato alla concezione di sistemi di raccomandazione in cui le persone sono in una rete sociale, e si hanno un insieme di persone ed uno di oggetti, entrambi descritti da una tassonimia. Dato un oggetto, si usa un algoritmo DFS per cercare nella rete sociale un certo insieme di utenti a cui raccomandare l'oggetto stesso esaminando il minor numero di nodi possibile. In [31] viene proposta una rappresentazione delle persone come vettore di punteggi per ogni elemento di una tassonomia che rappresenta diverse categorie di prodotti ed una misura di similarità semantica per un sistema di raccomandazione semantico. In [32] vengono proposti due algoritmi di raccomandazione su social network professionali. In [1] viene definito un sistema di raccomandazione di tipo semantic-social, che verrà analizzato in dettaglio nel seguito dato che contiene elementi rilevanti per il sistema di raccomandazione oggetto di questa tesi: in particolare verranno analizzate rispettivamente la sua componente semantica e quella sociale.

2.5.1 Componente Semantica

La componente semantica dell'algoritmo si basa su 3 elementi fondamentali:

- *preferenze della persona* raggruppate in un profilo della persona, per il quale può esservi una rappresentazione tassonomica;
- *tassonomia del dominio* che consiste in un insieme di entità organizzate in una struttura gerarchica *is-a* per descrivere gli oggetti del dominio. Si riporta un esempio nella Figura 2.1;

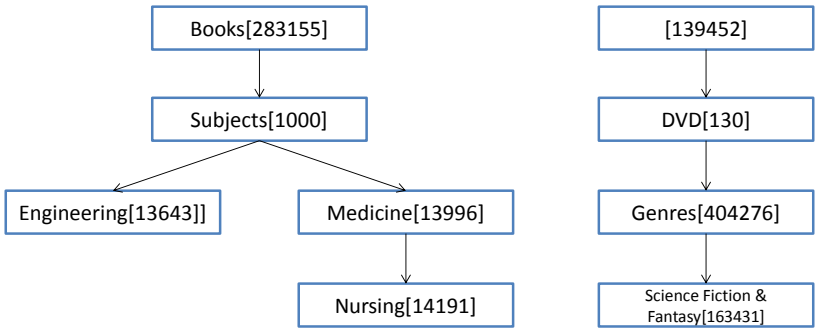
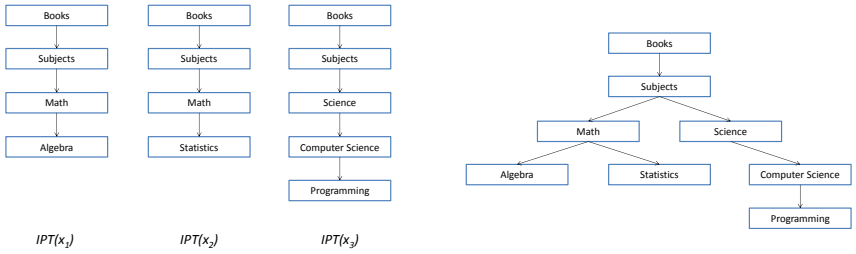


Figura 2.1: Frammento della tassonomia del sito Amazon.com. Gli indici tra parentesi quadre servono per identificare univocamente ogni categoria.

- *misura di similarità semantica* usata per calcolare la distanza tra i concetti dell'ontologia. Esistono 3 approcci derivanti dal fatto che una tassonomia può essere vista come un caso speciale di rete: *node-based* , *edge-based* e *ibridi* [33].

Il modello scelto usa una tassonomia del dominio per rappresentare la conoscenza di persone e oggetti ed una misura di similarità semantica ibrida per calcolare la similarità tra di essi. Più formalmente, dato un insieme di oggetti viene definito un *Semantic Taxonomy Tree* (STT) come un albero i cui nodi sono termini del dominio e gli archi rappresentano la gerarchia tra essi. Tale STT può essere rappresentato come una coppia $\{term, level\}$ dove $level$ indica il livello del termine nel STT.

Ogni oggetto viene associato ad una foglia di questo albero, in particolare viene definito un *Item Preference Tree* di un generico oggetto x, IPT(x), come l'insieme di coppie $\{term, level\}$ dalla radice alla foglia corrispondente all'interesse specifico dell'oggetto x nel STT. In Amazon ogni oggetto può avere uno o più IPT.



(a) *IPT dei singoli oggetti votati dall'utente.*

(b) *UPT dell'utente.*

Figura 2.2: Costruzione dell'UPT di un generico utente u a partre dagli IPT degli oggetti che ha votato: x_1, x_2 ed x_3.

Se per ogni utente u viene indicato l'insieme di oggetti $I(u)$ votati o comprati da egli, si può definire un *User Preference Subtree* dell'utente u (UPT(u)) come l'unione di tutti gli IPT degli oggetti comprati da u

$$UPT(u) = \cup_{x \in I(u)} IPT(x). \tag{2.17}$$

Esempio 2.5.1. *Si considerino i tre oggetti* x_1, x_2 *ed* x_3. *Siano* IPT(x_1)={(Books, 0), (Subjects, 1), (Math, 2), (Algebra, 3)}, IPT(x_2)={(Books, 0), (Subjects, 1), (Math, 2), (Statistics, 3)} *ed* IPT(x_1)={(Books, 0), (Subjects, 1), (Science, 2), (Computer Science, 3), (Programming, 4)}. *Si supponga che l'utente u abbia votato i tre suddetti oggetti, allora* UPTU=IPT(x_1) $\cup$ IPT(x_2) $\cup$ IPT(x_3) . *L'esempio è riportato per maggiore chiarezza nella Figura 2.2;*

La misura della similarità semantica utilizzata è quindi definita come segue. Siano P_1 e P_2 due insiemi di coppie { t,l } dove t è un termine ed l il livello nel STT, la similarità tra P_1 e P_2 è definita come segue.

$$\sigma(P_1, P_2) = \frac{1}{\mu}(\sum_{\{t,l\} \in P_1 \cap P_2} l) \tag{2.18}$$

dove

$$\mu = \min(\sum_{\{t,l\} \in P_1} l, \sum_{\{t,l\} \in P_2} l). \tag{2.19}$$

Sia u una persona ed x un oggetto, la similarità tra u ed x, denotata da $sim(u, x)$, è la similarità tra i loro preference tree associati:

$$sim(u, x) = \sigma(UPT(u), IPT(x)). \tag{2.20}$$

Esempio 2.5.2. *Si consideri la persona u discussa in precedenza, che ha votato gli oggetti* x_1, x_2 *ed* x_3. *Si consideri anche l'oggetto* x_4 *tale che* IPT(x_4)={(Books, 0), (Subjects, 1), (Literature & Fiction, 2), (Essays, 3), (General, 4)}. $sim(u, x_4) = \sigma(UPT(u), IPT(x_4))$. *In questo caso si ha che*

$$\mu = \min(\sum_{\{t,l\} \in UPT(u)} l, \sum_{\{t,l\} \in IPT(x_4)} l) = \min(18, 10) = 10. \tag{2.21}$$

mentre $\sum_{\{t,l\} \in UPT(u) \cap IPT(x_4)} l = 1$ *perchè solo le coppie* {Book, 0} *e* {Subject, 1} *appartengono all'intersezione dei due insiemi. Allora,* $sim(UPT(u), IPT(x_4)) = 0.1$. *Similmente, considerando l'oggetto* x_5 *tale che* IPT(x_5)={(Books, 0), (Subjects, 1), (Science, 2), (Computer Science, 3), (Theoretical Computer Science, 4)}, $sim(u, x_5) = 0.6$, *e come suggerisce anche il buon senso* x_5 *risulta molto più adeguato di* x_4 *ad essere raccomandato.*

La tassonomia descritta, ovvero di tipo *is-a*, può essere considerata da un punto di vista più teorico come un insieme C di categorie organizzate gerarchicamente in un albero [34]. Si può allora definire una funzione $p : C \to [0, 1]$ che per ogni $c \in C$ restituisce la probabilità di incontrare un'istanza del concetto c. Questo significa che p è monotona: se c_1 $IS-A$ c_2, $p(c_1) \leq p(c_2)$ e la probabilità della radice dell'albero di tassonomia è 1. Indicando il contenuto informativo di una categoria con $-log(p(c))$, si osserva che esso è più alto per le foglie, ovvero per le categorie più specifiche, mentre è nullo per la radice.

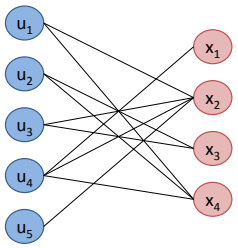
La similarità semantica tra due concetti può essere formulata sulla base del contenuto informativo che essi condividono. Maggiori informazioni condividono più i due concetti sono simili, ovvero

$$sim(c_1, c_2) = \max_{c \in S(c_1, c_2)} [-log(p(c))] \tag{2.22}$$

dove $s(c_1, c_2)$ è l'insieme di categorie che sussumono c_1 e c_2. Nel caso di ereditarietà multipla si deve considerare la similarità delle *entità* piuttosto che dei concetti, infatti può accadere che nel confrontare due *entità* esse

	x_1	x_2	x_3	x_4
u_1	-	5	-	3
u_2	-	-	3	4
u_3	-	2	4	-
u_4	5	5	-	1
u_5	-	3	-	-

(a) *Matrice rappresentativa dei rating degli utenti u_1, u_2, u_3, u_4 u_5.*



(b) *Grafo bipartito relativo alla matrice nella Figura (a).*

Figura 2.3: Esempio di grafo bipartito ottenuto dalla relativa matrice dei rating.

abbiano più di una categoria in comune che le identifica. Sia *concepts(e)* l'insieme di concetti nella tassonomia che rappresentano e e siano e_1 ed e_2 due *entità*, allora

$$sim(e_1, e_2) = \max_{c_1, c_2}[sim(c_1, c_2)] \tag{2.23}$$

dove c_1 varia in *concepts(e_1)* e c_2 varia in *concepts(e_2)*.

2.5.2 Componente Sociale

La componente sociale utilizza *collaboration social network* [35]. La rete sociale in questo caso è un grafo chiamato *user one-node projection* ed è ottenuta da *grafi bipartiti*: grafi in cui i nodi appartengono a due insiemi disgiunti V_1 e V_2 e gli archi non connettono nodi dello stesso insieme [36]. Più precisamente, nel caso del sistema di raccomandazione in oggetto, l'insieme dei nodi è formato dall'unione dei due insiemi disgiunti di persone ed oggetti. Un arco collega una persona u ad on oggetto x se u ha votato x.

Esempio 2.5.3. *Sia U l'insieme delle persone formato da u_1, u_2, u_3, u_4 u_5 e sia O l'insieme degli oggetti formato da x_1, x_2, x_3, x_4. I rating delle persone in questo esempio vanno da 1 a 5 e sono mostrati nella matrice nella Figura 2.3.*

Sia $I(u)$ l'insieme degli oggetti votati da u. Assumendo che per ogni oggetto $x \in I(u)$ u abbia dato un voto ad x tra 1 e 5, indicato con $\rho(u, x)$ (il voto può anche essere implicito, ovvero può essere stato ricavato dal sistema

in modo invisibile all'utente), si definisce *user preference* $lp(u_i)$ [1] l'insieme di oggetti definito come

$$lp(u_i) = \{ x \in I(u_i) \mid \rho(u, x) \geq 3 \} \tag{2.24}$$

Il peso degli dell'arco che collega due utenti u_1 ed u_2 tali che $Ip(u_1) \cap Ip(u_2) \neq \emptyset$ è definito come segue:

$$w(u_1, u_2) = |Ip(u_1) \cap Ip(u_2)| \tag{2.25}$$

Il grafo *user one-node projection* è ottenuto dal grafo bipartito come segue. L'insieme dei nodi corrisponde a quello delle persone. Date due persone u_1 ed u_2, un arco è tracciato tra essi se $w(u_1, u_2) > 3$. Questo controllo, come spiegato in [1], viene eseguito per assicurarsi di connettere tra loro solo persone che hanno votato oggetti in comune che reputano interessanti: in una scala di voti da 1 a 5 un voto di 1 o 2 significa che l'oggetto non è piaciuto a chi lo ha votato.

Esempio 2.5.4. *Dato l'esempio nella Figura 2.3, il corrispondente* one-node projection graph *si ottiene applicando il metodo appena descritto. Il grafo risultante è mostrato in 2.4.*

L'algoritmo definito in [1] applica le definizioni appena date, esplorando il grafo *user one-node projection* per restituire, dato un oggetto, l'insieme di persone alle quali esso va raccomandato. Il controllo che determina se proseguire o arrestare la visita può essere affidato alla sola funzione *sim* di similarità definita nell'equazione 2.4 nella variante *node based*, oppure può considerare anche il peso $w(u_1, u_2)$ definito nell'equazione 2.7 nella variante *node-edge based*.

Usare un grafo *one-node projection* è importante in quanto permette un calcolo delle raccomandazioni più efficiente rispetto al metodo classico di raccomandazione *collaborative filtering*. Dato un oggetto o si può infatti fare una visita depth first search (DFS) che permette di assegnare l'oggetto ai nodi che manifestano una similarità superiore ad un certo threshold θ. Per fare ciò si assegna ad ogni nodo del grafo *one-node projection* una label *visited* che deve essere inizializzata a false per tutti i nodi v. Se la similarità semantica $sim(v, x)$ tra v e l'oggetto x che si vuole raccomandare è superiore al thhreshold si aggiunge la persona corrispondente al nodo attuale alla

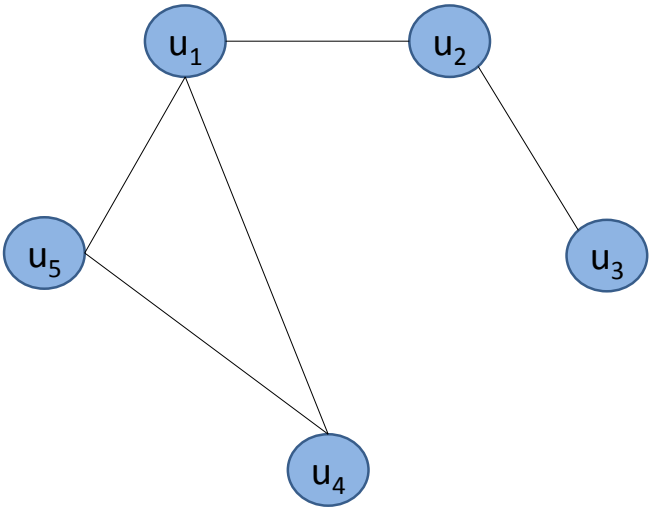


Figura 2.4: One-node projection relativo al grafo mostrato nella Figura 2.3 (b).

lista di persone alle quali raccomandare l'oggetto e si passa ad analizzare ricorsivamente i nodi connessi al nodo attuale non ancora visitati.

Il sistema che si vuole definire farà uso di una tassonomia per calcolare la similarità tra l'oggetto della raccomandazione e la persona della quale si vuole verificare la possibilità di raccomandare l'oggetto. Una tassonomia, come già accennato, è una classificazione gerarchica di concetti legati da una relazione e si può trovare un esempio in Figura 2.1.

Capitolo 3

Nuovo Approccio Teorico

Il sistema di raccomandazione oggetto di questa tesi ha il compito di raccomandare ad una *persona* un'*entità* che viene fornita da uno *shop* ad essa accessibile all'interno di una rete commerciale. Come è già stato anticipato l'approccio più simile che rappresenta lo stato dell'arte è il sistema di raccomandazione *semantic social* descritto nel capitolo precedente. Di seguito viene descritto il nuovo approccio utilizzato.

3.1 r-grafo

L'elemento alla base di questo sistema di raccomandazione è un grafo rappresentativo delle interazioni tra *persone*, *entità* e *shop*, definito in questa tesi, d'ora in avanti *r-grafo*. Questo grafo rappresenta lo stato del sistema di raccomandazione e permette di costruire il grafo *shop one node projection* necessario per la produzione delle raccomandazioni. In termini insiemistici, se l'*r-grafo* $R_g = (V, E)$, e se gli insiemi di *persone*, *shop* ed *entità* sono rispettivamente P, S ed E, allora $V = P \cup S \cup E$, mentre l'insieme degli archi è così definito:

- un arco collega una *entità* e ad uno *shop* s nel caso in cui s metta a disposizione e;

- un arco collega una *persona* p ad uno *shop* s nel caso in cui p abbia svolto almeno una *attività* relativa ad s;

Entità	Persona	Shop	Timestamp
1	2	1	1998-01-11
3	3	1	2000-01-11
2	3	1	2000-04-11
2	2	7	2000-01-11
0	2	15	2000-01-11
1	1	16	2000-03-11
0	0	18	2000-03-11

Tabella 3.1: Possibili attività facenti parte di un r-grafo

- un arco collega una *persona* p ad uno *shop* s nel caso in cui p conosca s, ovvero se s è raggiungibile da p perché è geograficamente abbastanza vicino;
- un arco collega una *persona* p ad una *entità* e nel caso in cui p abbia svolto una *attività* che la collegasse ad e.

Un'*attività* è una 4-pla formata dagli elementi seguenti: (1) l'*entità* oggetto dell'*attività*, (2) la *persona* che ha svolto l'*attività*, (3) lo *shop* presso il quale è stata svolta l'*attività* e (4) un timestamp relativo allo svolgimento dell'*attività*. Dopo quanto è stato detto si può pensare al *r-grafo* anche come ad un grafo formato dallo stesso insieme dei nodi ma da un insieme degli archi formato da iper-archi che collegano una *persona*, una *entità* ed uno *shop* come specificato dalle rispettive *attività*.

Esempio 3.1.1. *Si considerino le attività mostrate nella Tabella 3.1 e siano l'insieme degli* shop $S = \{ S_1, S_7, S_{15}, S_{16}, S_{18} \}$*, l'insieme delle persone* $P = \{ P_0, P_1, P_2, P_3 \}$ *e l'insieme delle* entità $E = \{ E_0, E_1, E_2, E_3 \}$*. L'*r-grafo *corrispondente è mostrato in Figura 3.1.*

3.2 Grafo Shop One-Node Projection

A partire dal *r-grafo* si può ottenere il corrispondente grafo *shop one-node projection* relativo alle persone. Prima di definire questo grafo occorre introdurre alcuni elementi.

Definizione 3.2.1. *Sia* p_i *una persona,* s_j *uno shop ed* e_k *un'entità, allora viene indicata con* $\xrightarrow[i,j,k]{}$ *l'attività che collega* p_i*,* s_j *e* e_k*.*

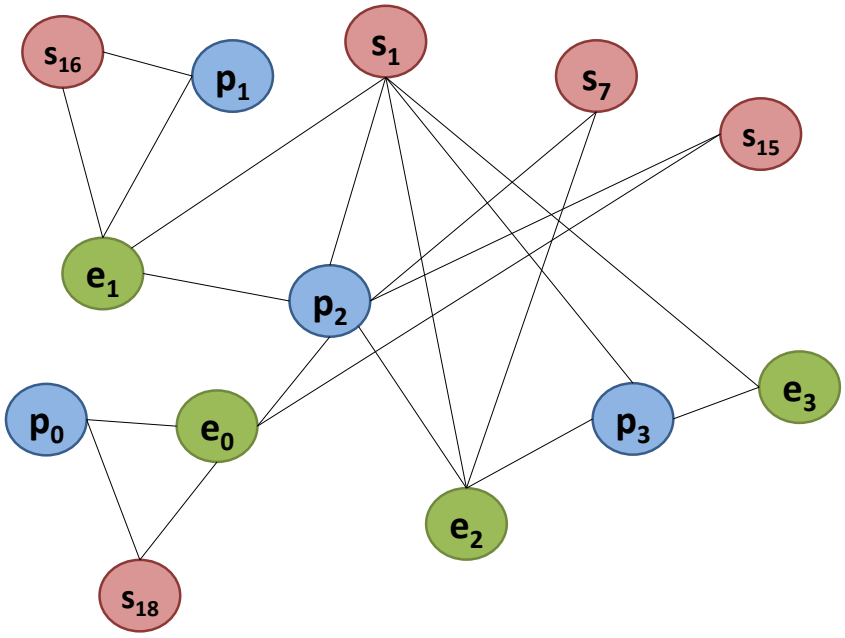


Figura 3.1: r-grafo relativo all'Esempio 3.1.1.

Definizione 3.2.2. *Sia p_i una persona, $act(p_i)$ è l'insieme di tutte le attività di p_i. Sia $a \in act(p_i)$, allora $ts(a)$ restituisce il timestamp di a, $shop(a)$ restituisce lo* shop *di a, $entity(a)$ restituisce l'*entità *di a e $person(a)$ restituisce la* persona *che ha svolto a.*

Definizione 3.2.3. *Sia p_i una* persona*, l'insieme di* shop *conosciuti da p_i, indicato con S_{p_i}, è l'insieme di tutti gli* shop *accessibili dalla* persona*. Tali* shop *si considerano accessibili se sono geograficamente vicini alla* persona *p_i, e il range di tolleranza può essere impostato come opzione mentre la distanza può essere calcolata basandosi sulla tecnologia GPS.*

Definizione 3.2.4. *Sia p_i una persona, l'insieme delle entità alle quali p_i è collegata per mezzo di un'attività, indicato con E_{p_i}, e rappresenta la cronologia delle entità legate a p_i. Quindi,*

$$E_{p_i} = \left\{ e_k \;\middle|\; \exists \xrightarrow[i,j,k]{} \right\} \tag{3.1}$$

Il grafo *shop one-node projection* $ONPG = V, A$ relativo alle persone è definito come segue: l'insieme dei vertici $V = P$, dato che si tratta della

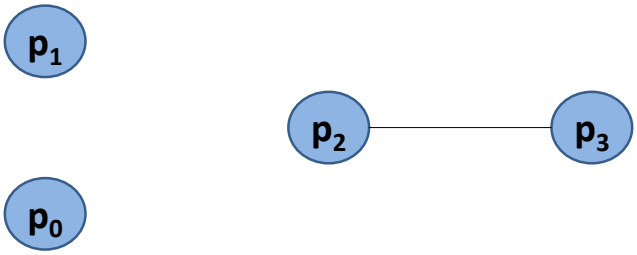


Figura 3.2: Costruzione del grafo ONPG relativo all'esempio 3.2.1.

proiezione sulle persone. L'insieme degli archi A invece è dato da archi che collegano coppie di persone che hanno almeno un numero ρ di shop in comune. Più formalmente,

$$A = \{\, i \to j \mid p_i, p_j \in P \wedge |E_i \cap E_j| \geq \rho \,\}, \tag{3.2}$$

dove ρ è un threshold assegnato come parametro. Anche se questo grafo viene costruito a partire da dati diversi ha la stessa struttura dello *one-node projection graph* definito in [1].

Esempio 3.2.1. *Si riporta in questo esempio il grafo* shop one-node projection *relativo al* r-grafo *nell'Esempio 3.1.1, da cui consegue che:* $S_{p_0} = \{\, s_{18} \,\}$, $S_{p_1} = \{\, s_{16} \,\}$, $S_{p_2} = \{\, s_7, s_{15} \,\}$, $S_{p_3} = \{\, s_1 \,\}$. *Applicando le definizioni date e considerando* $\rho = 1$ *si ottiene il grafo* one-node projection *della Figura 3.2.*

3.3 Metodo di Raccomandazione del Sistema

Una volta costruito il grafo *shop one-node projection*, la parte che manca al sistema è la raccomandazione vera e propria delle *entità* e degli *shop* alle persone. Il sistema di raccomandazione deve eseguire per ogni *entità* un controllo su quali persone possono essere interessate all'*entità* stessa ed in quali *shop* possono reperirla. Per fare ciò occorre definire una funzione di similarità: l'approccio scelto consiste nel valutare una similarità semantica tra la persona e l'*entità*, ovvero usando la Funzione 2.20.

Questo implica che ogni entità ha un proprio IPT, mentre l'UPT di una persona può essere ridefinito come segue, per essere coerenti con il nuovo approccio.

Definizione 3.3.1. *Sia $p_i \in P$, allora*

$$UPT(p_i) = \bigcup_{e \in E_{p_i}} IPT(e). \tag{3.3}$$

Sia l'IPT che l'UPT sono definiti in [1].

Per quanto riguarda invece la scelta degli *shop*, si parte dall'insieme di quelli reperibili dalla *persona* e si tengono solo i più adeguati e che forniscono l'*entità*: non ha senso raccomandarne una in ogni *shop* reperibile dalla *persona*, perché si renderebbe caotica la lista di raccomandazioni. Per ridurre la lista di *shop* si può procedere in diversi modi: si possono raccomandare i primi k in ordine di distanza in linea d'aria dalla *persona*, e ciò è fattibile usando la tecnologia GPS. Un altro approccio può essere la raccomandazione di *shop* che sono più "simili" alla persona, dopotutto si può costruire un albero di tassonomia semantico relativo allo *shop* analogamente alla costruzione di un UPT. L'approccio scelto consiste nell'utilizzare un *forgetting factor* [37] per dare ad ogni shop raggiungibile dalla *persona* una stima della sua importanza per quest'ultima basata sulla cronologia delle *attività* della *persona* stessa. In questo modo uno *shop* non più utilizzato da molto tempo andrà in fondo alla lista di quelli raccomandati oppure sarà rimosso in base a quanti *shop* è consentito raccomandare. Più precisamente, per ogni *persona* p_i, si deve associare ad ogni *shop* $s_j \in S_{p_i}$ un timestamp, indicato con $lu_i(s_j)$, per poter in seguito effettuare il calcolo del *forgetting factor*. Si è scelto in questa tesi di prendere il timestamp dell'ultima *attività* svolta da

p_i nello *shop* s_j, ovvero:

$$lu_i(s_j) = \max_{\{\ a \mid a \in act(p_i) \wedge shop(a) = s_j\ \}} (ts(a)) \tag{3.4}$$

dove $lu_i(s_j)$ è stato definito in questa tesi, dopo alcuni esperimenti, come il valore più adeguato da utilizzare nell'ordinamento degli *shop*.

Avendo associato il timestamp ad ogni *shop* è possibile calcolare il valore del *forgetting factor* ff associato, la cui formula è ricavata da [37]. Sia p_i una *persona* ed s_j uno *shop*,

$$ff(p_i, s_j) = \exp(\frac{-\ln(2) * time(p_i, s_j)}{hl_{p_i}}) \tag{3.5}$$

dove $time(p_i, s_j) = now - lu_i(s_j)$ e rappresenta i giorni dall'ultimo utilizzo dello shop s_j da parte della persona p_i e la data in cui viene calcolato $ff(p_i, s_j)$ e hl_{p_i} rappresenta la metà del ciclo di vita di una persona che utilizza il sistema. Dal momento che $ff(p_i, s_j)$ varia da 0 a 1 e che vale esattamente $1/2$ quando $time(p_i, s_j) = hl_{p_i}$, gli interessi verso i propri *shop* di una *persona* che usa il sistema da molto tempo caleranno più lentamente rispetto ad una *persona* che lo usa da poco tempo. Oltre il forgetting factor si usa la Funzione 2.20 per calcolare la similarità semantica tra la *persona* e lo *shop*, indicata con $ssim(p_i, s_j)$. Allora, il valore di similarità finale tra la *persona* p_i e lo *shop* s_j è definita in questa tesi come segue:

$$sw(p, s) = \alpha * ff(p_i, s_j) + (1 - \alpha) * ssim(p_i, s_j), \tag{3.6}$$

dove α è compreso tra 0 e 1. Usando anche la componente semantica $ssim(p_i, s_j)$ nel calcolo della similarità da un lato si tiene conto degli interessi della *persona* nella scelta dello *shop*, dall'altro si possono raccomandare anche nuovi *shop* alla *persona*, ovvero *shop* che non sono stati ancora visitati ma che possono essere di particolare interesse per la *persona* stessa.

3.3.1 Raccomandazione alle Nuove Persone

Un problema tipico di diversi sistemi di raccomandazione è il *cold start*, ovvero la raccomandazione di *entità* alle persone che sono nuove del sistema o che ancora non hanno effettuato un numero significativo di acquisti. Per cercare di risolvere tale problema si è sfruttato il fatto che ogni *persona* ha

accesso ad un ristretto sottoinsieme degli *shop* presenti nel sistema ed anche alle relative *entità*, di conseguenza si è sfruttato il loro trend all'interno dei vari *shop*. Per ognuno di essi, infatti, è possibile analizzare il trend delle *entità* con tempi accettabili. Per farlo sono stati usati due metodi, che consentono di ottenere risultati degni di nota.

Il primo metodo definito in questa tesi è una funzione euristica che considera le attività svolte in un determinato lasso di tempo. Sia s_j uno shop, e sia $act(s_j) = \{ a \mid \exists p_i.a \in act(p_i) \wedge shop(a) = s_j \}$, sia t la data in cui si vogliono ottenere i trend, Δt il range di tempo da considerare. Allora il trend di una *entità* e_k per lo *shop* s_j viene definito come

$$trend_{s_j}(e_k) = |\{ a \in act(s_j) \mid t - \Delta t \leq ts(a) \leq t \wedge entity(a) = e_k \}|. \quad (3.7)$$

Siano p_i una *persona* ed S_{p_i} l'insieme di *shop* conosciuti da p_i, allora si può definire il trend delle *entità* fornite dagli *shop* conosciuti da p_i come

$$trend_{p_i}(e_k) = \sum_{s_j \in S_{p_i}} trend_{s_j}(e_k). \quad (3.8)$$

Si noti che i trend delle *entità* sono collegati ad una *persona*, o più precisamente all'insieme di *shop* conosciuti da essa: in questo modo si ottengono dei trend specifici per una determinata *persona*.

Il secondo metodo consiste nell'applicazione del livellamento esponenziale, una tecnica già sfruttata nell'analisi delle serie storiche per rimuovere "rumore" dai dati osservati come, ad esempio, picchi stagionali nelle vendite [38]. Sia y_t il valore delle vendite osservato al tempo t, allora la serie osservata $y_1, y_2, ..., y_n$ può essere sostituita con la serie livellata $l_1, l_2, ..., l_n$ dove

$$l_n = \alpha \sum_{i=0}^{n-1} (1 - \alpha)^i y_{n-i} \quad (3.9)$$

ed α viene detta *costante di livellamento* e può variare tra 1.0 (nessun livellamento) e 0.0 (massimo livellamento) estremi esclusi.

Per calcolare y_t si è usato il primo metodo, ovvero $y_t = trend_{p_i}(e_k)$ prendendo per ogni y_t soltanto quelle attività che ricadevano tra t ed $t - 1$ per $1 \geq t \geq n$. Più formalmente, il trend di un'*entità* e_k per uno *shop* s_j è

stato modificato come segue:

$$trend_{s_j,t}(e_k) = |\{\, a \in act(s_j) \mid t-1 \leq ts(a) \leq t \,\}|, \tag{3.10}$$

dove per $t, t-1, ...$ si intendono le date durante le quali si è fatto il calcolo dei trend. Se ad esempio il calcolo è bimestrale e l'ultima volta è stato fatto il *02/09/13*, supponendo di voler tenere 6 osservazioni, avremo che t racchiude le attività svolte tra il *02/09/13* ed il *02/07/13*, che $t-1$ racchiude le attività svolte tra il *02/07/13* ed il *02/05/13* e così via. Allora $y_t = trend_{p_i,t}(e_k)$, dove

$$trend_{p_i,t}(e_k) = \sum_{s_j \in S_{p_i}} trend_{s_j,t}(e_k). \tag{3.11}$$

In questo modo si è ottenuta la successione $y_t, y_{t-1}, ..., 0$ relativa all'*entità* e_k per la *persona* p_i. Per ottenere l_t si può procedere usando una versione ricorsiva dell'espressione 3.9, ovvero

$$l_t = \alpha y_t + (1-\alpha) l_{t-1} \quad , t = 1, 2, ..., n \tag{3.12}$$

e il caso base, $l_0 = y_0$ [39]. Questo metodo ha portato risultati leggermente meno soddisfacenti del primo, ma comunque degni di nota.

3.3.2 Risultati di Raccomandazione

Dopo l'esecuzione della funzione di raccomandazione su una generica *persona* p si ottiene come output una lista di coppie $< e_i, < s_1, s_2, ..., s_k >>$ dove e_i è un'*entità* mentre $<< s_1, s_2, ... >$ è un elenco ordinato di *shop* tale che $sw(p, s_i) \geq sw(p, s_{i+1}) \quad 1 \leq i \leq k-1$. Questo risultato vale sia nel caso in cui la *persona* è nuova sia nel caso in cui è presente un profilo della *persona* e quindi p possiede una cronologia delle *attività*, solo che nel primo caso l'ordine degli *shop* deve essere basato su un fattore diverso dall'ultimo utilizzo, ad esempio la distanza.

3.4 Algoritmo del Sistema e Complessità

L'algoritmo descritto in questa sezione è stato chiamato *shop based*, mentre quello che rappresenta lo stato dell'arte e che verrà utilizzato per i confronti è il *node based* [1]. Per prima cosa deve essere mantenuto aggiornato l'r-

grafo, per via dei nuovi *shop*, delle nuove *persone*, delle nuove *entità* e delle nuove relazioni sorte con il passare del tempo. Inoltre, periodicamente è necessario ricalcolare il *forgetting factor* relativo agli *shop* per ogni *persona*, così come il grafo *shop one-node projection* relativo alle persone, che deve essere aggiornato per essere consistente con il nuovo *r-grafo*. Infine, dato il grafo *shop one-node projection*, si costruisce un vettore *top-k degree* [1], contenente i primi k nodi di grado più alto. Questo vettore consente di ridurre la complessità nel produrre raccomandazioni. Il calcolo del grafo *shop one-node projection* viene fatto confrontando le persone a coppie, e per ogni coppia si confrontano gli *shop* conosciuti da entrambe. Per farlo sono necessari due cicli di cui uno su tutte le n persone mentre l'altro è sulle persone di indice superiore a quella attualmente visitata, infatti nelle coppie l'ordine è ininfluente. Per un array di n persone $P = p_1, p_2, ..., p_n$ il numero di confronti C da fare è

$$\begin{aligned} C &= (n-1) * \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} j \\ &= (n-1) * \big((n-1) + (n-2) + ... + (1)\big) \\ &= (n-1) * \big(\frac{n-1}{2}\big) \\ &= O(n^2) \end{aligned} \tag{3.13}$$

Supponendo anche di avere m shop per ogni persona nel caso pessimo, con $m << n$, per ogni coppia di persone p_i e p_j vengono contati gli *shop* conosciuti da entrambi, ovvero $\left|S_{p_i} \cap S_{p_j}\right|$. occorre confrontare tutti gli elementi di S_{p_i} con tutti quelli di S_{p_j}, per un totale di m^2 confronti. La complessità totale per la costruzione del grafo *shop one-node projection* è quindi $O(n^2m^2)$ con $m << n$. Per vedere il codice si faccia riferimento all'Algoritmo 3.1. La procedura `BuildSONPG` prende in input un *r-grafo* e restituisce il relativo grafo *shop one-node projection*. S è l'insieme dei nodi corrispondenti agli *shop* presenti nell'*r-grafo*, che può essere ottenuto in tempo costante se viene creato online. P è l'insieme delle persone presenti nel grafo e può essere ottenuto in modo e tempo analogo ad S. Infine PS, per il quale valgono le stesse considerazioni dei due insiemi precedentemente definiti, è l'insieme degli archi che collegano una *persona* $p \in P$ ad uno *shop* $s \in S$ se p è cliente di s, ovvero se p ha svolto almeno un'*attività* relativa ad s. Il grafo *shop one-node projection* risultante conterrà un insieme di nodi corrispondente all'insieme

di persone del *r-grafo* ed un arco $\{p_i, p_k\}$ per ogni coppia di persone $\{p_i, p_k\}$ tali che esiste uno *shop* s e due archi p_i, s e p_k, s nel *r-grafo*.

Algorithm 3.1 Calcolo del Grafo Shop One-Node Projection

```
procedure BuildSONPG(r-grafo)   ▷ Builds shop one-node projection graph of r-grafo
    for each p_i ∈ P do
            Add node p_i to the shop one-node projection graph
    end for
    for each p_i ∈ P do
        for each p_k ∈ P such that {p_i, s} ∈ PS and {p_k, s} ∈ PS do
                Add edge {p_i, p_k} to the shop one-node projection graph
        end for
    end for
    return the shop one-node projection graph
end procedure
```

La costruzione del vettore *top-k degree* può richiedere $O(n)$ tempo, sempre con n persone nel grafo *shop one-node projection*, ma si può anche implementare nella costruzione del grafo stesso senza aumentarne la complessità. Per produrre le raccomandazioni l'algoritmo realizzato procede in due modi, in base alla cronologia di tutte le persone. Tale cronologia può essere ottenuta in tempo $O(n)$ se n è il numero complessivo di attività presenti nel *r-grafo*, ma si può creare in tempo costante nel caso le attività vengano aggiunte on-line, ovvero durante l'esecuzione del sistema di raccomandazione, quando effettivamente vengono svolte. La stessa considerazione vale per la costruzione degli UPT relativi alle persone, infatti nel momento in cui una *persona* p svolge un'*attività* relativa ad un'*entità* e basta aggiungere lo $IPT(e)$ allo $UPT(p)$.

Per calcolare il *forgetting factor* relativo ad una *persona* p_i si procede come segue. Per prima cosa ogni *shop* deve avere associato l'elenco di *attività* ad esso inerenti, e ciò è fattibile on-line, con complessità costante, oppure off-line, con complessità lineare nel numero di attività, infatti ognuna di esse è legata ad un solo *shop*, di conseguenza sarebbe necessario un primo ciclo su tutte le *attività* per assegnarle al rispettivo *shop* ed un secondo ciclo in cui per ogni *shop* si trova l'ultima *attività*. In pratica, se il numero di *attività* è n_a ed il numero di *shop* è n_s, usando il metodo off-line si avrebbero $O(n_a)$ passi di computazione per trovare lo *shop* di ogni *attività* ed $O(n_s * n_a)$ passi per calcolare l'ultima di ogni *shop*. Si potrebbe poi calcolare il *forgetting*

factor in tempo costante, semplicemente calcolandolo durante il reperimento dell'ultima *attività* per gli *shop* per una complessità totale di $O(n_a + n_a * n_s)$, ma essendo ogni *attività* legata ad un solo *shop* la complessità diventa $O(n_a)$. Se si sceglie il metodo off-line ad esempio si può fare l'assegnamento delle attività agli *shop* prima di calcolare il *forgetting factor*, e si tratta di una ristrutturazione di informazioni già presenti nel *r-grafo*. Il codice si può trovare nell'Algoritmo 3.2

Data una persona $p \in P$ sia S_p l'insieme degli *shop* della persona p, per raccomandare gli *shop* è stata definita una funzione che, data p, calcola il *forgetting factor* di s per p per ogni $s \in S_p$. La funzione prende in input una persona p, la data del calcolo, ovvero la data in cui viene chiamata, *now*, e un valore *halflife* che corrisponde alla metà del tempo tra *now* e la data di ingresso della persona p nel sistema. Gli ultimi due parametri sono espressi in giorni. La funzione `lastAccessed(p, s)` restituisce la data in giorni dell'ultima volta che p ha svolto un'attività relativa allo *shop* s. Tale funzione restituisce il risultato in tempo costante, infatti è possibile calcolare on-line il giorno di ultimo accesso ad s da parte di p semplicemente aggiornando una mappa relativa a p che per ogni *shop* contiene la data di ultimo utilizzo dello *shop*. La funzione restituisce la mappa, `userShopForgettingFactor` di *shop* e dei relativi *foretting factor* associati ad una *persona*. Il metodo `put(p,s,forgettingfactor)` della mappa consente di inserire una 3-pla nella mappa da restituire.

Algorithm 3.2 Calcolo del Forgetting Factor degli Shop per una Persona

```
procedure computeShopForgettingFactor(p, now, halflife)
    for each shop s ∈ S_p do
            dayOfDifference = now - lastAccessed(p, s)
            result = compute forgetting factor according to Equation 3.5
            userShopForgettingFactor.put(p, s, result)
    end for
end procedure
```

In aggiunta si calcola la similarità semantica tra la *persona* e lo *shop*. Sia i_p il numero di interessi della *persona*, a qualsiasi livello dell'albero di tassonomia, e sia analogamente i_s il numero di interessi dello *shop*, il calcolo richiede $O(i_p + i_s + i_s)$ che equivale a $O(i_p + i_s)$. L'Algoritmo 3.3 mostra il calcolo della similarità semantica tra una *persona* ed uno *shop*, ottenuta come già detto combinando linearmente il *forgetting factor* e la similarità seman-

tica, che può essere fatto in tempo costante durante l'ordinamento. Infine l'ordinamento degli *shop* in base alla similarità calcolata richiede $O(n \log n)$ passi di computazione.

Sia p una *persona* ed s uno *shop*, si hanno a disposizione *UPT(p)* ed $SPT(s)$, entrambi rappresentati come insiemi di coppie $\{term, level\}$ dove il termine indica la categoria della tassonomia ed il livello indica il livello all'interno dell'albero di tassonomia. Sia c una coppia, $fst(c)$ restituisce il termine ed $snd(c)$ restituisce il livello, entrambi in tempo costante. Allora, dalla formula per il calcolo della similarità semantica, occorre per prima cosa calcolare tutti gli elementi della formula e questo lo si fa calcolando la somma di tutte le coppie $c_i^1 \in UPT(p)$ e su tutte le coppie $c_i^2 \in SPT(s)$. Fatto questo occorre calcolare la somma tra tutti gli elementi in comune e questo è fattibile eseguendo un ciclo su tutte le coppie dell'insieme con cardinalità minore ed infine si calcola il risultato. La funzione accetta quindi in input una *persona* ed uno *shop*. La funzione `minS(`s_1`, `s_2`)`, dati due insiemi s_1 ed s_2, restituisce quello di cardinalità minore.

Algorithm 3.3 Calcolo della Similarità Semantica tra *Persone* e gli *Shop*

procedure COMPUTEPSSEMANTIICSIMILARITY(*person*, *shop*)
  sumUPTLevels = 0;
  for each $c_1 \in UPT(person)$ **do**
    sumUPTLevels = sumUPTLevels + $snd(c_1)$
  end for
  sumSPTLevels = 0;
  for each $c_2 \in SPT(shop)$ **do**
    sumSPTLevels = sumSPTLevels + $snd(c_2)$
  end for
  minimumSize = $min(sumUPTLevels, sumSPTLevels)$;
  smallerSet = $minS(UPT(person), SPT(shop)$
  largerSet = $minS(UPT(person), SPT(shop)$
  sumLevels = 0;
  for each $c_1 \in smallerSet$ **do**
    if *largerSet contains term fst(*c_1*)* **then**
      sumLevels = sumLevels + $snd(c_1)$
    end if
  end for
  return *sumLevels / minimumSize*
end procedure

Per il calcolo del *forgetting factor* relativo al primo metodo illustrato

	s_1	s_2	s_3	s_4
e_1	0	20	300	0
e_2	150	0	20	500
e_3	400	0	250	0
e_4	350	0	0	0
e_5	0	550	0	0
e_6	0	400	350	0

Tabella 3.2: Fruizione delle *entità* $e_1, e_2, e_3, e_4, e_5, e_6$ relativamente agli *shop* s_1, s_2, s_3, s_4 nel periodo considerato.

in precedenza, la complessità è $O(n_a)$ dove n_a è il numero di *attività* che rientrano nel periodo di tempo considerato, infatti usando quel metodo non si fa altro che calcolare la frequenza assoluta di fruizione delle *entità* in base alle *attività* presenti nel sistema. Dopo aver ottenuto per ogni *shop* il trend corrente, il calcolo dei trend delle *entità* per una generica persona p_i consiste nel sommare tra loro i trend della stessa *entità* nei negozi conosciuti da p_i, aggregando tutte le *entità* in una unica lista, che poi viene ordinata in base al trend delle stesse. Nel caso pessimo, ogni persona ha accesso a tutti gli shop del sistema, anche se nella pratica questa situazione non si potrà verificare, perchè vanificherebbe l'utilità stessa del sistema. Sapendo che n_e è il totale delle entità e che n_p è il totale delle persone, l'assegnamento delle n_e *entità* alle n_p persone nel caso pessimo richiede $O(n_e * n_p)$ tempo. Valutando il costo per una singola *persona* quindi la complessità è $O(n_a)$ nel caso pessimo. Infine, per ordinare la lista in base al trend delle *entità* contenute il costo del miglior algoritmo di ordinamento è $O(n_e \log n_e)$. L'Esempio 3.4.1 riporta il calcolo del trend e la generazione della raccomandazione per una *persona* priva di cronologia delle *attività*.

Esempio 3.4.1. *Si consideri la* persona p_i*, e si assuma che conosca gli* shop s_1, s_2, s_3, s_4 *e si considerino per semplicità le sole* entità $e_1, e_2, e_3, e_4, e_5, e_6$*. La tabella 3.2 mostra quantità di fruizioni delle* entità *per i suddetti* shop *in un periodo di 12 mesi. Aggregando tutti i trend in base alle* entità*, si ottengono i seguenti trend:* $\{e_1 = 320, e_2 = 670, e_3 = 650, e_4 = 350, e_5 = 550, e_6 = 750\}$*. Supponendo di voler raccomandare al massimo 4* entità*, dopo averle ordinate in base al rispettivo trend, alla persona* p_i *verrebbero raccomandate* $\{e_6, e_2, e_3, e_5\}$*.*

Usando il secondo metodo il calcolo dei trend per gli *shop* coinvolge il

livellamento esponenziale semplice e si deve tenere in considerazione la complessità di tale tecnica. Come nel primo caso occorre che per ogni *shop* si conosca il trend delle *entità* relative, allo stesso modo per una *persona* p_i è richiesta l'aggregazione in partizioni di tempo dei trend degli *shop* conosciuti da essa, per costruire la serie $y_t, y_{t-1}, ..., y_1$ dalla quale calcolare il livellamento esponenziale. Questa procedura richiede $O(n_a)$ passi dal momento che è simile all'aggregazione fatta nel caso precedente. A questo punto, prima di ordinare i risultati, è necessario calcolare la serie livellata $l_t, l_{t-1}, ..., l_1$ di cui ai fini dell'algoritmo che si stà definendo è necessario solo l_t. La relazione di ricorrenza T della funzione di livellamento esponenziale è definita come segue:

$$\begin{aligned} T(t) &= 1, \quad n = 1 \\ T(t) &= T(t-1) + 1, \quad n > 1 \end{aligned} \tag{3.14}$$

che ha complessità $O(t)$. Nel caso dell'algoritmo descritto in questa tesi, t è un numero molto piccolo, strettamente minore di 24, di conseguenza la complessità aggiunta è irrilevante. In generale la complessità è costante perché è un numero k di periodi di tempo considerati.

Esempio 3.4.2. *Si considerino l'*entità e_1 *fruita presso lo* shop s_1 *ed un biennio di monitoraggio delle fruizioni di* e_1 *presso il suddetto* shop *aggregate per bimestre. Si ottiene la lista ordinata* {50, 10, 30, 100, 80, 5, 90, 100, 120, 140, 110, 30}*, che significa che l'ultimo bimestre considerato* e_1 *è stata fruita 30 volte presso* s_1*, il penultimo è stata usufruita 110 ecc. Figura 3.3 permette di osservare come con il livellamento esponenziale il trend del prodotto non sia influenzato da variazioni periodiche o stagionali delle fruizioni.*

Per le persone aventi una cronologia delle *attività* si può sfruttare il grafo *shop one-node projection* dato che tali persone hanno uno *UPT* e quindi è possibile calcolare la similarità semantica tra una *persona* ed una *entità*. Per ogni *entità* presente nel sistema viene fatta una visita DFS del grafo *one-node projection* e per ogni *persona* visitata raccomanda l'*entità* solo se la similarità semantica tra la *persona* corrente e l'*entità* è superiore ad un certo threshold. La visita DFS viene eseguita k volte, partendo ogni volta da una *persona* diversa del vettore *top-k degree*. Con questo metodo si riesce a coprire un numero rilevante di nodi del grafo *one-node projection*. Lo

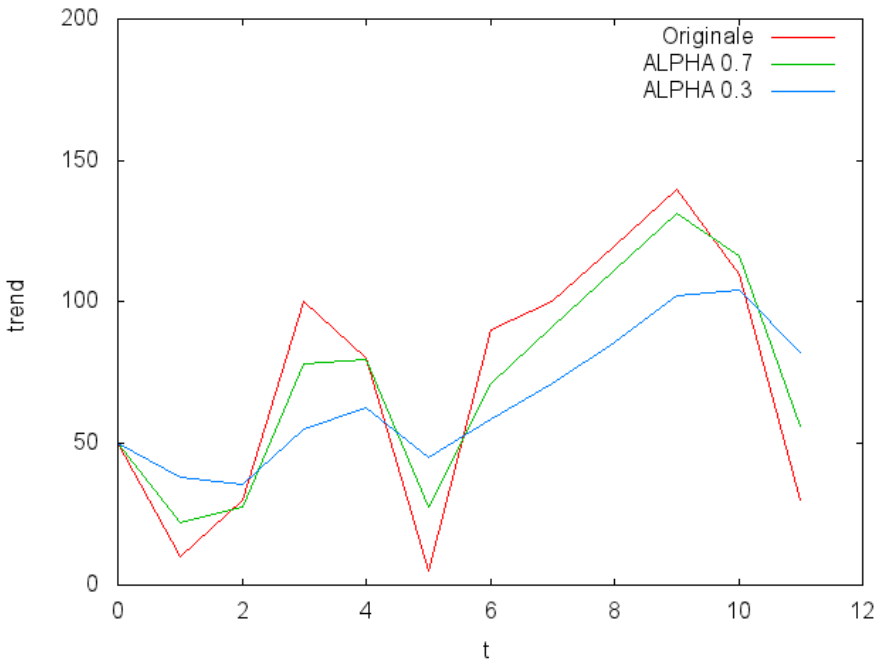


Figura 3.3: Livellamento esponenziale semplice applicato alla serie di dati dell'Esempio 3.4.2 con due diverse costanti di livellamento.

pseudocodice dell'algoritmo relativo a quanto appena detto è definito dagli Algoritmi 3.4 e 3.5.

La visita DFS ha una complessità $O(n)$, dove n è il numero di archi del grafo *shop one-node projection*.

Sia *Entities* l'insieme delle *entità* contenute nel sistema, per ogni *entità* $e \in$ *Entities*, a partire dai nodi contenuti nel vettore *top-k degree*, si esegue una visita DFS sul grafo *shop one-node projection* per cercare gli utenti a cui è possibile raccomandare e. Per prima cosa viene assegnata una label *visited* ad ogni *persona*, che permette di tenere traccia delle persone già visitate. Sia p_i la *persona* attualmente visitata, se a p_i è raccomandata l'*entità* e essa va memorizzata in una lista, `peopleList`, che contiene anche per ogni *persona* i relativi *shop* in cui è possibile effettuare l'*attività* legata ad e. La funzione interrompe la visita DFS corrente quando: (1) p_i non conosce nessuno *shop* in cui trovare l'*entità*, (2) la profondità raggiunta ha superato un certo limite ℓ, (3) non sono presenti vicini del nodo corrente da visitare oppure (4) la similarità semantica tra e e p è inferiore ad un certo threshold θ. Di seguito si riporta lo pseudo codice delle due procedure per l'inizializzazione delle

variabili e per la visita del grafo *shop one-node projection g*.

Per quanto riguarda le persone senza cronologia, viene usato il suddetto metodo che consente di calcolare il trend degli *shop*. Occorre aggiornarlo con una certa frequenza per avere il trend più recente nel momento in cui si eseguono le raccomandazioni. In questo caso, per ogni persona priva di cronologia delle *attività*, vengono calcolati i trend dei prodotti e vengono raccomandati i primi k prodotti con quello maggiore.

Algorithm 3.4 Initialize structures for recommendetions

procedure INITRECOMMENDATION(g, e, l, θ)
  for each node n in g **do**
    set *visited* to *false*
  end for
  create empty list *peopleList*.
  compute *top-k degree* ▷ can be computed during shop one-node projection graph creation
  for each node c in *top-k degree* vector **do**
    Recommend(c, e, θ, *peopleList*, l)
  end for
  return *peopleList*
end procedure

Algorithm 3.5 Inizializza le strutture per eseguire le raccomandazioni

procedure RECOMMEND(v, e, *peopleList* l, θ)
  if v is not visited **then**
    set v to *visited*
    def shopSorted ▷ crea una lista per gli shops che vendono questa entità
    if $SemanticSimilarity(v, e) > \theta$ **then**
      for each *shop* s in Shops(e) **do**
        if v knows s **then**
          similar = 0.5 * ff(shop) +
          0.5*SemanticSimilarity(v, s)
          add to shopSorted the pair *{s, similar}*
        end if
      end for
      sort shopSorted pairs by decreasing values
      add to *peopleList* the pair *v, {shopSorted}*
      if $l \geq 0$ **then**
        for each vertex v_i in Neighbors(v) **do**
          Recommend(v, e, *peopleList l-1*, θ)
        end for
      end if
    end if
  end if
  return *peopleList*
end procedure

Capitolo 4

Esperimenti e Confronti

In questo capitolo viene definito il dataset, per poi riportare i risultati ottenuti e i confronti con l'algoritmo *shop based*. Dal momento che è stato necessario modificare il dataset, nella sezione 4.1 viene definito La metrica maggiormente utilizzata è la f-measure, particolarmente indicata per il test di sistemi di raccomandazione e motori di ricerca. Si è osservato un superamento dell'algoritmo *node based* in tutti i test effettuati.

4.1 Dataset

Non avendo a disposizione un dataset tratto da reti commerciali reali, si è usato il dataset di Amazon modificato per contenere anche un insieme di *shop*.[1] In particolare, si è creata una lista di *shop* aventi una certa tassonomia, consistente con quella già presente nel dataset, per poi assegnare ad ogni *persona* un insieme di *shop* conosciuti in base a ciò che ha votato. Per ogni *entità* e votata da una *persona* p, se p contiene già nel proprio insieme di *shop* conosciuti uno che vende e si è generata un'*attività* che mette in relazione p, e, lo *shop* ed il *timestamp* e la si aggiungeva al dataset, scegliendo con probabilità molto bassa uno *shop* nuovo ed utilizzando quello dopo averlo aggiunto agli shop conosciuti da p. In questo modo si aggiunge un pò di casualità, e si è più fedeli a quanto avviene nella realtà, dove le persone non acquistano sempre un certo tipo di prodotti nello stesso negozio ma avolte cambiano, anche solo per provare a trovare maggior convenienza

[1] Amazon product co-purchasing network metadata, http://snap.stanford.edu/data/amazon-meta.html

o qualità del servizio. Se invece la *persona* p non conosce nessuno *shop* in grado di vendere e se ne è scelto uno a caso tra quelli disponibili e lo si è aggiunto all'elenco degli *shop* conosciuti da p. Ad ogni *entità* vengono anche aggiunti tutti gli *shop* presso i quali essa è disponibile. Di seguito si riporta un esempio di un dataset così generato, mostrato usando la sintassi XML:

```
<root>
  <shops>
    <shop id="0" name="SH000000">
      <taxonomies>
        <taxonomy>
          <category>Books[28375]</category>
          <category>Subjects[1400]</category>
          <category>Biographies \&amp;amp; Memoirs[24]
          </category>
          <category>General[2375]</category>
        </taxonomy>
        <taxonomy>
          <category>Books[28375]</category>
          <category>Subjects[1400]</category>
          <category>Religion \&amp;amp; Spirituality[5]
          </category>
        </taxonomy>
        ...
      </taxonomies>
    </shop>
    ...
  </shops>
  <entities>
    <entity id="0" name="EN000000">
      <taxonomies>
        <taxonomy>
          <category>Books[28375]</category>
          <category>Subjects[1400]</category>
          <category>Biographies \&amp;amp; Memoirs[24]</category>
          <category>Arts \&amp;amp; Literature[2777]</category>
          <category>Authors[2366]</category>
        </taxonomy>
        ...
      </taxonomies>
      <shops>
        <shop>1</shop>
        <shop>0</shop>
        <shop>30</shop>
```

```
      </shops>
    </entity>
    ...
  </entities>
  <people>
    <person id="0" name="NA000000" surname="SU000000"/>
    ...
  </people>
  <activities>
    <activity datetime="2000-01-11␣24:00:00" entity="2" id="0"
    person="2" shop="1" />
    <activity datetime="2000-01-10␣13:00:00" entity="3" id="1"
    person="3" shop="1" />
    ...
  </activities>
</root>
```

4.2 Framework di Valutazione e Metriche

Si è implementato l'algoritmo *node based* descritto in [1] ed è stata usata la relativa funzione di similarità semantica come funzione di similarità utente-oggetto. Questi due algoritmi non tengono conto degli shop, di conseguenza è stato utilizzato il dataset descritto nella sezione precedente ignorandoli completamente, e il risultato ottenuto è il dataset di Amazon usato anch'esso in [1]. Per testare entrambi gli algoritmi viene costruito il grafo *one-node projection* o *shop one-node projection*, dove un arco collega due persone se esse hanno votato almeno un oggetto in comune, e il peso degli archi è il numero di oggetti votati in comune, e viene poi calcolato il vettore *top-k degree* su tale grafo. Sono state scelte come metriche di valutazione quelle che valutano l'accuratezza delle previsioni. Nel caso dell'esecuzione off-line dell'algoritmo, che è il metodo utilizzato per valutare l'algoritmo *shop based* definito in questa tesi, le metriche di accuratezza sono influenzate dalla sparsità dei dati ed in particolar modo dalla lista di raccomandazioni generata. Il sistema di raccomandazione oggetto di questa tesi prevede un rating implicito, se un'*entità* e fa parte delle *entità* per la quale una certa *persona* p ha mostrato interesse, si assume che e sia di gradimento per la persona p: non è previsto un rating esplicito perché, dato il contesto in cui si colloca il sistema di raccomandazione, se una *persona* ha deciso di svolgere un'*attività* relativa

ad un'*entità*, significa che la persona è realmente interessata all'*entità*; si è inoltre voluto rendere il sistema il meno invasivo possibile per le persone. Per questo motivo nessuna *entità* viene rimossa dal training set, dal test set e dalla lista di raccomandazioni. In [40] viene suggerito di produrre una lista di raccomandazione priva di *entità* non votate, ma quel caso si applica dove il rating è esplicito e di conseguenza si potrebbero avere raccomandazioni di *entità* con o senza un rating. Scegliendo di eliminare quelle prive di rating si ha il problema noto di produrre risultati inaccurati. Anche la possibilità di produrre rating di default viene scartata, per lo stesso motivo, così come quella di considerare gli oggetti con un rating più alto. Le metriche più comuni per la valutazione di un sistema di information retrieval sono precision e recall, oltre a metriche derivate da queste ultime [40].

4.2.1 Precision, Recall e Metriche Derivate da Queste

Precision e recall sono calcolate suddividendo le *entità* in diversi insiemi. In particolare si devono classificare le *entità* in due dimensioni: quelle rilevanti oppure non rilevanti da un lato e quelle selezionate dal sistema di raccomandazione e quindi selezionate oppure non selezionate dall'altro. Si hanno quindi quattro insiemi di *entità* che sono riepilogati in Tabella 4.1. Le *entità* sono considerate rilevanti se sono state acquistate dalla *persona* che si sta considerando, altrimenti sono irrilevanti. In un sistema basato su rating esplicito bisogna per prima cosa convertire i rating in binario e successivamente assegnare l'*entità* all'insieme opportuno. Ad esempio, se fosse previsto un rating che va da 1 a 10 le *entità* con rating da 0 a 4 potrebbero essere considerate irrilevanti mentre quelle con rating da 5 a 10 sarebbero rilevanti. In generale il concetto di rilevanza è molto discusso, in un sistema di raccomandazione la rilevanza di un'*entità* è soggettiva pociché solo la *persona* destinataria della raccomandazione può determinare se l'*entità* è rilevante per se.

Precision rappresenta la probabilità che un *entità* selezionata sia rilevante, ed è definita nell'Equazione 4.1:

$$P = \frac{|SR|}{|S|} \tag{4.1}$$

dove $|S|$ è il numero delle *entità* selezionate ed $|SR|$ rappresenta il numero di *entità* rilevanti selezionate. Recall rappresenta la probabilità che un'*entità*

	Selezionati	Non selezionati
Rilevanti	*SR*	*SN*
Irrilevanti	*SI*	*NI*

Tabella 4.1: Classificazione delle *entità* per il calcolo di precision e recall

rilevante sia selezionata, ed è definita nell'Equazione 4.2

$$R = \frac{|SR|}{|R|} \tag{4.2}$$

dove $|R|$ è il numero degli elementi rilevanti. Entrambe le formule si trovano in [40] Calcolare quest'ultima misura in senso stretto non è fattibile perché ogni *persona* dovrebbe valutare tutte le *entità* e dire se sono rilevanti o irrilevanti. Nel caso di sistemi con tag esplicito, di conseguenza, si usano delle approssimazioni. Si è scelta una data che facesse da soglia e si è suddiviso il dataset in un training set, formato dall'insieme delle *attività* compiute prima di tale data, ed in un test set formato da tutte le altre attività. Per via della sparsità dei dati, il numero di oggetti rilevanti nel test set può essere una frazione del numero totale degli oggetti rilevanti presenti nel dataset, e questo influenza molto il risultato, e lo stesso vale per il calcolo di precision dato che è anch'esso influenzato dall'insieme di *entità* rilevanti. Nel caso di un sistema tag free il problema è simile perché può succedere che una *persona* abbia svolto un'*attività* legata ad una certa *entità* ma quest'ultima sia di grande importanza per la *persona* stessa, ad esempio perché non ha avuto tempo di usufruire di una raccomandazione o non ne ha usufruito per dimenticanza.

Precision e recall devono essere considerate insieme per valutare completamente le performance dell'algoritmo, e questo perché l'obiettivo è di raccomandare tutti gli oggetti rilevanti, di conseguenza anche recall deve essere considerata. Si è scelto di usare come metrica F_1, o f-measure, definita nell'Equazione 4.3 e in [40]:

$$F_1 = \frac{2PR}{P + R} \tag{4.3}$$

La figura 4.1 mostra un esempio del calcolo di precision e recall.

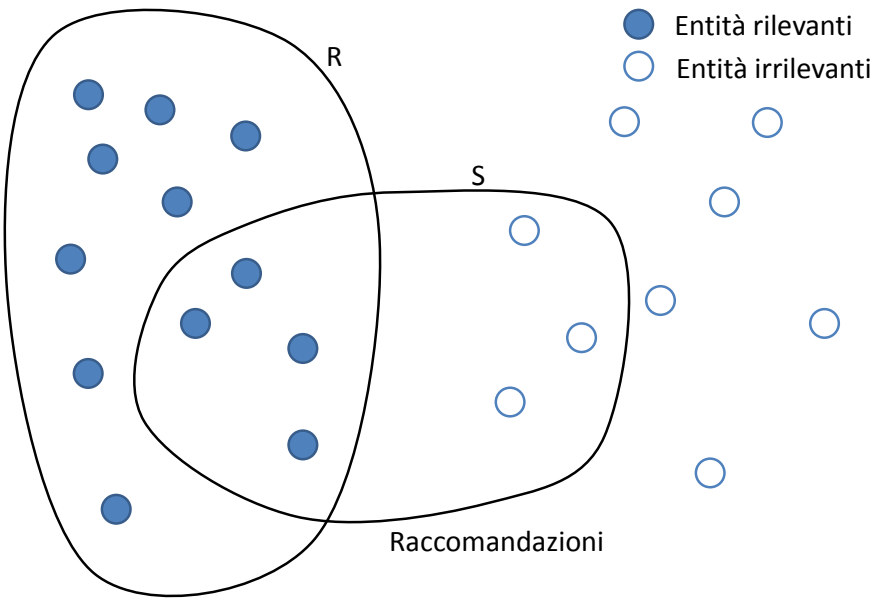


Figura 4.1: Dato l'insieme di *entità* raccomandate si hanno i risultati seguenti: $P = 4/7$, $R = 4/12$, $F_1 = 0.42$.

4.3 Risultati

Per il confronto è stato usato l'algoritmo *node-based* e sono state eseguite 50 query, ovvero si è usato un vettore *top-k degree* contenente 50 persone. Di queste, solo 24 avevano dei vicini mentre le restanti erano nuove e non avevano effettuato acquisti. Il numero di persone aventi vicini è uguale sia per l'algoritmo *node based* sia per l'algoritmo *shop based*. Questo perchP, se una *persona* non ha effettuato *attività*, nel primo algoritmo non potrà avere *entità* in comune con nessuno mentre nel secondo non potrà avere *shop* in comune con nessuno. Questo è un caso particolare, che aiuta nel confronto dei dati. In realtà il numero potrebbe essere diverso, ad esempio potrebbe succedere che una *persona* abbia effettuato diversi acquisti in uno *shop* che però non è conosciuto da nessun altro. Nel grafico riporato in Figura 4.2 sono confrontate le f-measure relative alle raccomandazioni fatte su 3000 *entità* e 31777 persone per entrambi gli algoritmi e su 37 *shop* per il secondo algoritmo. Per maggiore chiarezza nel confronto sono state riportate separatamente le f-measure medie di persone aventi vicini e prive degli stessi.

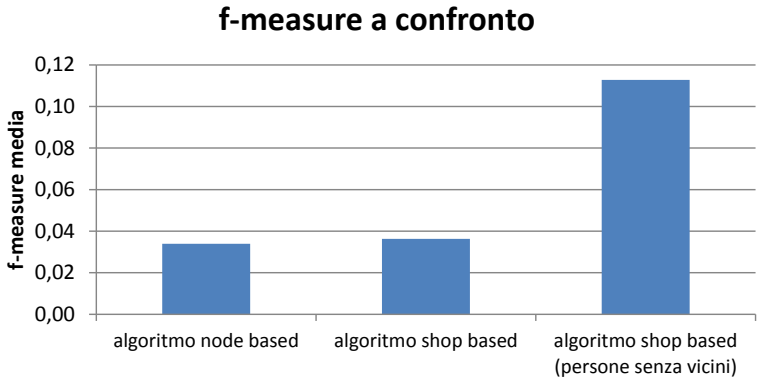


Figura 4.2: Nel grafico mostrato sono riportate le f-measure medie relative all'algoritmo *node based* su 24 persone aventi vicini, sullo stesso numero di persone con vicini per l'algoritmo *shop based* e su 26 persone prive di vicini dell'algoritmo *shop based*

.

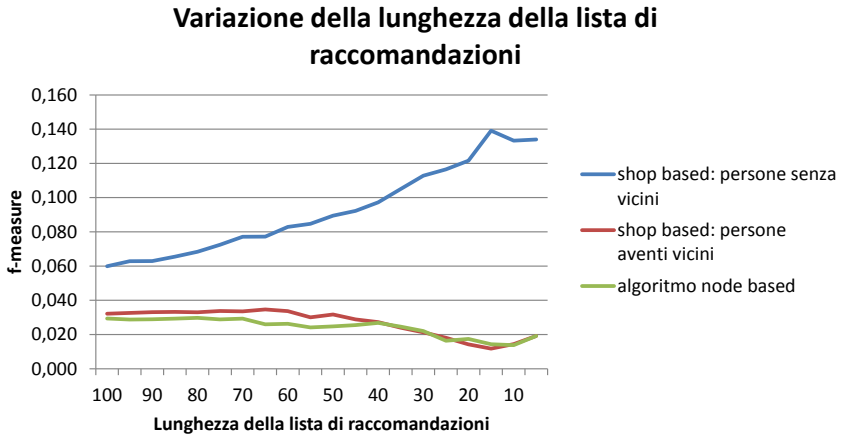


Figura 4.3: Nel grafico è riportato l'andamento della f-measure al variare della dimensione della lista di raccomandazioni, per l'algoritmo *node based* e per l'algoritmo *shop based*. Per il calcolo del trend è stata usata l'euristica che calcola la frequenza assoluta delle vendite

.

La f-measure viene fortemente influenzata dal numero di *entità* raccomandate. Maggiore è il numero di *entità* raccomandate ad una *persona*, maggiori sono le sue possibilità di ricevere raccomandazioni rilevanti. In questo caso però aumenta anche il numero di raccomandazioni non rilevanti. Sono stati confrontati i due algoritmi e sono state calcolate anche in questo caso due distinte f-measure per il nuovo algoritmo: una per le persone aventi vicini ed una per le persone che ne sono prive. Dai risultati mostrati nella Figura 4.3 si osserva che, per entrambi gli algoritmi, gli oggetti raccomandati a persone aventi vicini sono distribuiti in tutta la lista delle raccomandazioni. Questo dimostra che le persone non svolgono *attività* solo se sono relative ad *entità* aventi similarità semantica molto elevata, ma sono interessate anche ad altre *entità* appartenenti ad un diverso insieme di categorie. Per quanto riguarda le raccomandazioni fatte a *persone* prive di vicini, invece, si è osservato l'andamento opposto, e questo indica che le persone del campione analizzato sono interessate alle *entità* in maniera direttamente proporzionale al trend delle stesse.

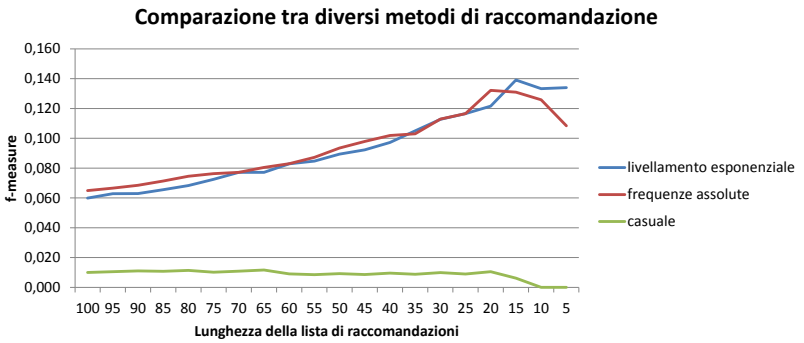


Figura 4.4: Nel grafico è riportato l'andamento della f-measure al variare della dimensione della lista di raccomandazioni. Sono confrontati tre metodi di scelta delle *entità* da raccomandare a persone prive di cronologia delle *attività*. Si è usata una costante di livellamento pari a 0.7 per il livellamento esponenziale semplice

.

Per calcolare le raccomandazioni per persone prive di vicini nella Figura 4.3 è stato usato il livellamento esponenziale semplice. Prendendo come data di riferimento quella usata per separare il training set dal test set, si è calcolata la serie delle *attività* degli ultimi 12 bimestri, usando come costante di livellamento 0.7. Oltre questa metrica si è tentato di utilizzare anche il calcolo dei trend usando la frequenza assoluta delle *attività*, che richiede meno calcoli del primo approccio utilizzato. Infine si è scelto casualmente quali *entità* raccomandare, come consigliato in [40], e il confronto tra questi tre approcci è riportato nella Figura 4.4. L'approccio di scelta casuale delle *entità* viene consigliato perché in diversi casi le persone sono interessate a diverse *entità* e non solo ad una loro categoria, ed essendo prive di cronologia delle *attività* può essere giusto produrre raccomandazioni basate su scelte casuali. Come ci si può aspettare però questo approccio è il meno efficiente, ottenendo una f-measure fortemente inferiore agli altri due approcci indipendentemente dalla lunghezza della lista di raccomandazioni.

Il threshold usato per nell'algoritmo DFS che produce la lista delle raccomandazioni determina quanto l'*entità* che si sta raccomandando deve essere simile alla *persona* attualmente visitata. Se il valore è troppo alto, ad ogni *persona* verranno raccomandate *entità* molto simili, dal punto di vista delle categorie di appartenenza, a quelle già acquisite. Questo comportamento crea il problema già visto per i sistemi di raccomandazione *content based*, ovvero la raccomandazione di troppe *entità* troppo simili tra loro Se il threshold è troppo basso si ottiene l'effetto opposto: qualsiasi *entità* può essere raccomandata. In quest'ultimo caso si hanno due effetti negativi: il primo è che la similarità semantica diventa irrilevante, se il threshold tende a 0; il secondo è che l'algoritmo si comporterà come nel caso pessimo, tendendo a raccomandare ogni *entità* ad ogni *persona* che è un comportamento estremamente inefficiente. Nella Figura 4.5 viene mostrato un confronto tra l'algoritmo *node based* e quello *shop based* al variare del threshold. La prima conclusione che si può trarre dai risultati è che è meglio raccomandare *entità* cercando di diversificare piuttosto che focalizzarsi sulle poche categorie già conosciute dalla cronologia delle *attività*, come si è già detto. Il picco di f-measure viene raggiunto da entrambi gli algoritmi quando il treshold è circa 0.55, e questo significa che vengono scartate solo *entità* appartenenti a categorie che non interessano alla *persona*. Ad esempio non si raccomanderanno articoli da pesca ad una persona che nella storia delle *attività* ha comprato solo libri di cucina, ma non si potranno neanche raccomandare esclusivamente libri di questa categoria. Non dovrebbe stupire il fatto che l'andamento di entrambi gli algoritmi confrontati è molto simile, poiché entrambi effettuano una visita DFS usando la similarità semantica come fattore discriminante.

Quando viene raccomandata un'*entità* il numero di persone per ogni *shop* è rilevante. Se per ipotesi si avesse un solo *shop*, tutti avrebbero usato solo quello e il grafo *shop one node projection* sarebbe un grafo completo. Se al contrario ogni *persona* conoscesse un solo *shop* e questo fosse diverso da tutti gli altri, il grafo sarebbe privo di archi e tra le persone non esisterebbero relazioni. Se l'algoritmo viene applicato al mondo reale, però, si ha un grafo mediamente connesso formato da diverse componenti connesse. Si pensi al caso di due città limitrofe, ad esempio Ferrara e Bologna, e si ignorino i paesi intermedi tra le due città. Se a Ferrara ci fossero 5 *shop* le persone residenti in quella città farebbero acquisti principalmente in quelli, con qualche eccezione e lo stesso vale per Bologna. Allora sarebbe inutile

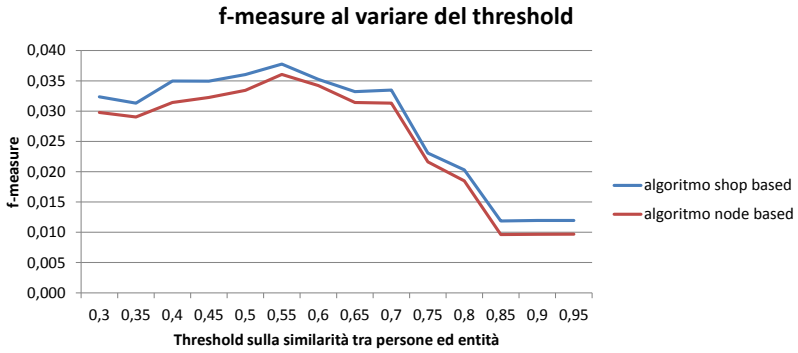


Figura 4.5: Nel grafico è riportato l'andamento della f-measure al variare del threshold applicato alla similarità tra *entità* e *persona* durante la visita DFS sono confrontati l'algoritmo *shop based* e il *node based*.

raccomandare una *entità* che si trova in uno *shop* di una città a persone che risiedono nell'altra, ma sarebbe sensato raccomandarla a persone che frequentano *shop* della stessa città. Questo è esattamente ciò che avviene, e per evitare di raccomandare *entità* a troppe persone si è introdotto un limite massimo alla lunghezza del cammino percorribile dall'*entità* nel grafo *shop one node projection*. In questo modo il numero di *persone* per *shop* influenza il numero di raccomandazioni di una *entità*. Un altro fattore che viene influenzato da tale numero è la f-measure, infatti minore è il numero di vicini di un nodo meno nodi saranno attraversati durante una raccomandazione e di conseguenza ogni *entità* sarà raccomandata ad un insieme minore di persone. Nella Figura 4.6 viene mostrato il valore di f-measure al variare della media del numero di *persone* per *shop*. Si osserva che il numero di persone non è particolarmente rilevante nella f-measure prodotta, va però notato che il numero medio è ottenuto aggiungendo degli *shop* e cercando di restringere la varianza, ma questo non implica un calo uniforme del numero di clienti di ogni singolo *shop*. Per come è strutturato l'algoritmo è impossibile fissare un numero esatto, e non avrebbe senso farlo perché nella realtà alcuni

Figura 4.6: Nel grafico è mostrato il valore della f-measure al variare del numero di persone per *shop*. I valori sono stati calcolati in modo da essere compatibili alla dimensione del dataset, più precisamente si hanno valori medi sull'asse x proporzionali al numero di shop

.

negozi possono avere poche centinaia di clienti mentre altri diverse migliaia. Il risultato mostrato può essere considerato buono perché indica che l'algoritmo riesce a produrre buone raccomandazioni in differenti contesti, da un ipotetico paese ad una città mediamente popolata.

Quando si parla di sistemi di raccomandazione, un aspetto molto importante è il tempo che impiegano per produrre raccomandazioni appropriate. Quando la quantità dei dati nel training set aumenta, la qualità delle raccomandazioni dovrebbe aumentare a sua volta. Diversi algoritmi possono raggiungere una qualità accettabile delle raccomandazioni a diverse velocità. Esistono tre classificazioni del tasso di apprendimento di un sistema di raccomandazione: il tasso di apprendimento complessivo, che è in funzione del numero di *entità* o di *persone*; il tasso di apprendimento per *entità*, che rappresenta la qualità delle sue previsioni in base al numero di attività disponibili per essa ed infine il tasso di apprendimento per *persona*, che rappresenta la qualità delle previsioni per una *persona* come funzione del numero di *attività* che essa ha svolto. La Figura 4.7 mostra un confronto tra

i due algoritmi *shop based* e *node based* relativi a quest'ultima tipologia di tasso di apprendimento. L'algoritmo *shop based* si comporta notevolmente meglio rispetto il *node based*, producendo raccomandazioni accettabili anche solo con 20 elementi del training set per ogni *persona*. Come misura della qualità delle raccomandazioni è stata usata la f-measure, ed è stato considerato un valore massimo di training set di 200, dato che entrambi gli algoritmi sembravano aver raggiunto risultati prossimi all'asintoto. Il motivo di una tale performance, confrontando questo risultato con i precedenti, può essere dovuto al fatto che uno *shop* possiede un sottoinsieme delle *entità* presenti nel sistema, di conseguenza le possibili raccomandazioni da fare ad una *persona* sono un sottoinsieme rispetto a quelle fattibili dall'algoritmo *node based*. Di queste raccomandazioni si sa, per come è definito il dataset, che sono presenti tutte le raccomandazioni di *entità* rilevanti. Da ciò si può concludere che l'insieme di *entità* non rilevanti presenti nell'algoritmo *shop based* è considerevolmente inferiore all'insieme di *entità* non rilevanti dell'algoritmo *node based* e statisticamente è più probabile produrre raccomandazioni rilevanti con il primo algoritmo piuttosto che con il secondo.

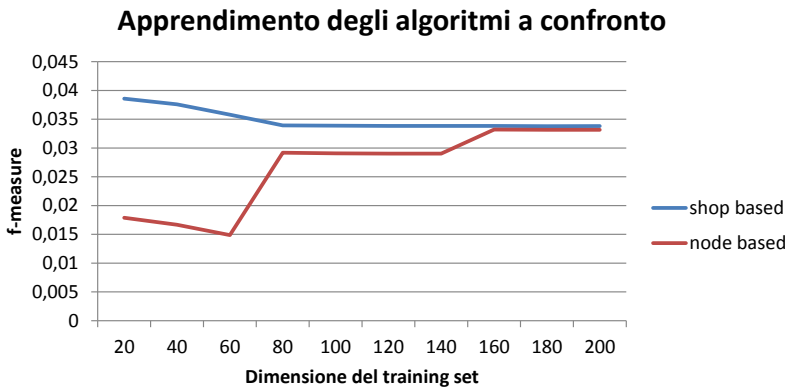


Figura 4.7: Analisi del tasso di apprendimento per *persona* medio degli algoritmi *shop based* e *node based*. Nell'asse x viene mostrata la dimensione del training set per ogni persona.

.

Appendice A

Codice dell'Algoritmo

```
package com.rollnext.algorithm;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Date;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import java.util.TreeMap;

import com.rollnext.classes.Activity;
import com.rollnext.classes.Entity;
import com.rollnext.classes.Person;
import com.rollnext.classes.Shop;

public class AlgorithmVersion3 {

  /**
   * Ratings for each user. One user has a hashmap of object
   * index and related rating.
   */
  //private HashMap<String, HashMap<Long, Long>> ratings;

  /**
   * List of people in the dataset
   */
```

```
private HashMap<Long, Person> peopleList;
/**
 * list of activities in the training set
 */
private HashMap<Long, Activity> activitiesList;
/**
 * list of activities in the dataset (training + test)
 */
private HashMap<Long, Activity> allActivitiesList;
/**
 * Contains people ratings. First is id of a people,
 * second is id of an object, third is rating.
 */
private HashMap<Long, HashMap<Long, Double>> ratingsPersone;

/**
 * computes with forgetting factor the age of each entry.
 */
private HashMap<Long, Float> entriesAge;

/**
 * shop list accessible by shop id
 */
HashMap<Long, Shop> shopList;

/**
 * Contains information about shops people have bougbt in.
 * First long is person, second is object and third is the shop.
 * similar to ratingPersone but with the shop instead of rating.
 */
private HashMap<Long, HashMap<Long, Long>> ratingsShopPersone;

/**
 * list of entities
 */
public static HashMap<Long, Entity> entitiesList;

/**
 * Item Preference Tree IPT of each item is a hashmap in which,
 * if x is an item, IPT(x) is a set of {term,level} pairs.
 * term is a category in the taxonomy tree, level is the level
 * of the term in the tree (starting from 0).
 * implemented like a hashmap of object ids to a hashmap
 * of terms and level.
```

```
 * item id in IPT starts from 0, in dataset starts from 1
 */
public static HashMap<Long, ArrayList<HashMap<Long, Long>>> IPT;


/**
 * User preference tree is computed like the union of IPTs of
 * each object purchased by the user.
 * It is implemented like a hashmap where keys are users
 * and values are related UPT.
 */
public static HashMap<Long, HashMap<Long, Long>> UPT;

/**
 * Per ogni persona contengono una hashmap di oggetti
 * e relativi voti.
 */
private HashMap<Long, HashMap<Long, Long>>
  personeOggettiVotiTraining;
private HashMap<String, HashMap<Long, Long>>
  personeOggettiVotiTest;

/**
 * grafo delle relazioni tra persone
 */
private HashMap<Long, HashMap<Long, Long>>
  oneNodeProjectionGraph;

/**
 * grafo delle relazioni tra persone
 */
private HashMap<Long, HashMap<Long, Long>>
  shopOneNodeProjectionGraph;

/**
 * list of entities. First Long is entity ID.
 */
private HashMap<Long, Entity> listaEntity;

/**
 * tiene traccia delle persone visitate per l'algoritmo
 * di visita del grafo
 */
private HashMap<Long, Boolean> personeVisitate;
```

```
/**
 * contains the number of pepople for each shop.
 * used for statistics
 */
private HashMap<Long, Long> peoplePerShop;

// ordina i nodi del grafo solo persone in base al loro grado,
// utile per costruire il top k vector
private TreeMap<Long, Long> OldopgInDegreeSorting =
  new TreeMap<Long, Long>(Collections.reverseOrder());
private ValueComparator2 vc2 =
  new ValueComparator2(OldopgInDegreeSorting);
private TreeMap<Long, Long> opgInDegreeSorting =
  new TreeMap<Long, Long>(vc2);

private TreeMap<Long, Long> OldopgInDegreeSortingShop =
  new TreeMap<Long, Long>(Collections.reverseOrder());
private ValueComparator2 vc2shop =
  new ValueComparator2(OldopgInDegreeSortingShop);
private TreeMap<Long, Long> opgInDegreeSortingShop =
  new TreeMap<Long, Long>(vc2shop);

public HashMap<Long, Long> getPeoplePerShop() {
  return peoplePerShop;
}

public HashMap<Long, HashMap<Long, Long>>
  getOneNodeProjectionGraph() {
  return oneNodeProjectionGraph;
}

public HashMap<Long, Entity> getEntitiesList() {
  return entitiesList;
}

public HashMap<Long, Person> getPeopleList() {
  return peopleList;
}

public HashMap<Long, HashMap<Long, Long>>
  getShopOneNodeProjectionGraph() {
  return shopOneNodeProjectionGraph;
}
```

```
/**
 * Build a ONPG similar to user-item case but with user-shop
 * bipartite graph. If user i and j have shops in common
 * there is an arc between i and j with a list of these shops.
 */
public void buildShopOneNodeProjectionGraph(long shopThreshold)
{
  shopOneNodeProjectionGraph =
    new HashMap<Long, HashMap<Long, Long>>();

  HashMap<Long, Person> personBackup =
    (HashMap<Long, Person>) peopleList.clone();

  // scan each pair of people
  for(Entry<Long, Person> personaI : peopleList.entrySet())   {
    // remove already defined person to avoid double comparison.
    // creates both edge in the inner for
    personBackup.remove(personaI.getKey());

    for(Entry<Long, Person> personaJ :
      personBackup.entrySet())     {

      if(personaI.getKey() != personaJ.getKey())  {

        // gets the set of shops of j
        HashSet<String> shopsOfJ =
          personaJ.getValue().getKnownShops();

        long shopInCommon  = 0;
        for(String shopOfJ : shopsOfJ)  {

          if (personaI.getValue().getKnownShops()
            .contains(shopOfJ) )
            shopInCommon ++;
        }

        // check shop "age"
        HashMap<Long, HashSet<Long>> aaa =
          personaI.getValue().getPerShopActivities();

        if(shopInCommon >= shopThreshold) {

          // makes sure to create the required data structures
```

```
            if(shopOneNodeProjectionGraph.get(personaI.getKey())
              == null)
              shopOneNodeProjectionGraph.put(personaI.getKey(),
                new HashMap<Long, Long>());
            if(shopOneNodeProjectionGraph.get(personaJ.getKey())
              == null)
              shopOneNodeProjectionGraph.put(personaJ.getKey(),
                new HashMap<Long, Long>());

            // creates an ark between i and j
            if( shopOneNodeProjectionGraph.get(personaI.getKey())
              .get(personaJ.getKey()) == null )
              shopOneNodeProjectionGraph.get(personaI.getKey())
                .put(personaJ.getKey(), shopInCommon);

            // creates an ark between j and i, because arcs
            // are bitirectional
            if( shopOneNodeProjectionGraph.get(personaJ
              .getKey()).get(personaI.getKey()) == null )
              shopOneNodeProjectionGraph.get(personaJ.getKey())
                .put(personaI.getKey(), shopInCommon);

          }
        }
      }
    }

    // costruisce un ordinamento in base al grado di ogni nodo
    for(Entry<Long, HashMap<Long, Long>> entry :
      shopOneNodeProjectionGraph.entrySet()) {
      opgInDegreeSortingShop.put((long) entry.getValue().keySet()
        .size(), entry.getKey());
    }
  }

  /**
   * Build one node projection graph. Every node v is a neighbor
   * of a user u if both have at least an item in common weight
   * of (u,v) is the number of common items
   */
  public void buildOneNodeProjectionGraph() {
    oneNodeProjectionGraph =
      new HashMap<Long, HashMap<Long, Long>>();
```

```
// makes a copy of peoplelist to avoid conflicts
HashMap<Long, Person> personBackup =
  (HashMap<Long, Person>) peopleList.clone();

// scan each pari of people to build arcs
for(Entry<Long, Person> personaI : peopleList.entrySet())   {
  // remove already defined person to avoid double comparison.
  // creates both edge in the inner for
  personBackup.remove(personaI.getKey());

  for(Entry<Long, Person> personaJ : personBackup.entrySet())
  {

    // makes sure to create every data structure
    if(oneNodeProjectionGraph.get(personaI.getKey()) == null)
      oneNodeProjectionGraph.put(personaI.getKey(),
        new HashMap<Long, Long>());
    if(oneNodeProjectionGraph.get(personaJ.getKey()) == null)
      oneNodeProjectionGraph.put(personaJ.getKey(),
        new HashMap<Long, Long>());

    if(personaI.getKey() != personaJ.getKey())  {

      boolean shopInCommon  = false;
      for(String shopOfJ :
        personaJ.getValue().getKnownShops())
        if (personaI.getValue().getKnownShops()
          .contains(shopOfJ) )
          shopInCommon = true;

      // build or updates bidirectional arc (i,j) if people
      // i and j has at lest one entity and one shop
      // in common
      if((personeOggettiVotiTraining.get(personaI.getKey())
        != null) &&
        (personeOggettiVotiTraining.get(personaJ.getKey())
         != null))
        for(Entry<Long, Long> entry :
        personeOggettiVotiTraining.get(personaI.getKey())
          .entrySet())  {
          Set<Long> oggettiDiJ =
            personeOggettiVotiTraining.get(personaJ.getKey())
              .keySet();
```

```
                if(oggettiDiJ.contains(entry.getKey()) &&
                  shopInCommon) {
                  // se non esiste l'arco tra i e j lo creo
                  if( oneNodeProjectionGraph.get(personaI.getKey())
                    .get(personaJ.getKey()) == null )
                    oneNodeProjectionGraph.get(personaI.getKey())
                      .put(personaJ.getKey(), (long)1);
                  else
                    oneNodeProjectionGraph.get(personaI.getKey())
                      .put(personaJ.getKey(), oneNodeProjectionGraph
                      .get(personaI.getKey())
                      .get(personaJ.getKey()) + 1);

                  if( oneNodeProjectionGraph.get(personaJ.getKey())
                    .get(personaI.getKey()) == null )
                    oneNodeProjectionGraph.get(personaJ.getKey())
                      .put(personaI.getKey(), (long)1);
                  else
                    oneNodeProjectionGraph.get(personaJ.getKey())
                      .put(personaI.getKey(), oneNodeProjectionGraph
                      .get(personaJ.getKey())
                      .get(personaI.getKey()) + 1);

                }
              }
            }
          }
        }

        // sort entries by degree
        for(Entry<Long, HashMap<Long, Long>> entry :
          oneNodeProjectionGraph.entrySet()) {
          opgInDegreeSorting.put(
            (long) entry.getValue().keySet().size(), entry.getKey());
        }

      }

      /**
       * Gets a list of people without neighborg in the ONPG.
       * Useful to apply the alternative algorithm
       * @return
       */
      public ArrayList<Long> getPeopleWithoutItemNeighborg() {
```

```
    ArrayList<Long> peopleWithoutItemNeighborg =
      new ArrayList<Long>();
    for(Entry<Long, HashMap<Long, Long>> node :
      oneNodeProjectionGraph.entrySet())  {
      if(node.getValue().size() == 0)
        peopleWithoutItemNeighborg.add(node.getKey());
    }

    return peopleWithoutItemNeighborg;
  }

  public ArrayList<Long> getPossibleNeighborg(Long person)  {
    ArrayList<Long> potentialNeighborg = new ArrayList<Long>();
    potentialNeighborg.addAll(shopOneNodeProjectionGraph
      .get(person).keySet() );
    return potentialNeighborg;
  }

  /**
   * Compute similarity between two people like the
   * user-item similarity.
   * @param user1
   * @param user2
   * @return
   */
  public double peopleSemanticSimilarity(long user1, long user2)
  {
    double result = 0.0;

    // get required elements
    HashMap<Long, Long> UPT_1 = UPT.get(user1);
    HashMap<Long, Long> UPT_2 = UPT.get(user2);

    // computes mu
    int sum1 = 0, sum2 = 0;
    for(Long l : UPT_1.values())
      sum1 += l;
    for(Long l : UPT_2.values())
      sum2 += l;

    int mu = Math.min(sum1, sum2);

    // intersection is computed this way:
    // for all term in UPT of u it is in the intersection
```

```
    // if is contained in IPT of item x
    int sumIntersection = 0;
    for(Entry<Long, Long> entry : UPT_1.entrySet())
      if(UPT_2.containsKey(entry.getKey()))
        sumIntersection += entry.getValue();

    result = sumIntersection / (double)mu;

    return result;
  }

  /**
   * Compute similarity between two people like the
   * user-item similarity.
   * @param user1
   * @param user2
   * @return
   */
  public double shopPersonSemanticSimilarity(long shop,
    long person) {
    double result = 0.0;

    // get required elements
    Shop tmpShop = shopList.get(shop);

    ArrayList<ArrayList<String>> SPT =
      tmpShop.getTaxonomiesList();
    HashMap<Long, Long> UPT_2 = UPT.get(person);

    HashMap<Long, Long> SPT_shop = new HashMap<Long, Long>();

    for(ArrayList<String> tax : SPT)
      for(long i = 0; i< tax.size(); i++) {
        int firstIndex = tax.get((int) i).lastIndexOf('[')+1;
        int lastIndex = tax.get((int) i).lastIndexOf(']');
        SPT_shop.put(Long.parseLong(tax.get((int) i)
          .substring(firstIndex, lastIndex)), i);
      }

    // creates a general IPT, a tree not al list
    HashMap<Long, Long> globalIPTx = new HashMap<Long, Long>();

    // computes mu
    int sum1 = 0, sum2 = 0;
```

```
    for(Long l : SPT_shop.values())
      sum1 += l;
    for(Long l : UPT_2.values())
      sum2 += l;

    int mu = Math.min(sum1, sum2);

    // intersection is computed this way:
    // for all term in UPT of u it is in the intersection
    // if is contained in IPT of item x
    int sumIntersection = 0;
    for(Entry<Long, Long> entry : SPT_shop.entrySet())
      if(UPT_2.containsKey(entry.getKey()))
        sumIntersection += entry.getValue();

    result = sumIntersection / (double)mu;

    return result;
  }

  /**
   * Reset visited vertices, for the main algorithm in paper
   * 2012 Graph Searching Algorithms For
   * Semantic-Social Recommendetion.pdf
   */
  public void resetVisitedVertices()  {
    personeVisitate.clear();
    for(Long persona : peopleList.keySet()) {
      personeVisitate.put(persona, false);
    }
  }

  /**
   * Compute the Top-N-vector of people sorted by degree of the
   * node in the one node projection graph.
   * @param k size of the Top-N-vector
   * @return
   */
  public TreeMap<Long, Long> computeTopKDegreeVector(int k) {
    ValueComparator2 vc2 =
      new ValueComparator2(opgInDegreeSorting);
    TreeMap<Long, Long> tmp = new TreeMap<Long, Long>(vc2);
    int i=0;
    for(Entry<Long, Long> es : opgInDegreeSorting.entrySet()) {
```

```
      if (i==k)
        break;
      tmp.put(es.getKey(), es.getValue());
      i++;
    }

    return tmp;
  }

  /**
   * Compute the Top-N-vector of people sorted by degree
   * of the node in the one node projection graph.
   * @param k size of the Top-N-vector
   * @return
   */
  public TreeMap<Long, Long> computeTopKDegreeVectorShop(int k) {
    ValueComparator2 vc2 =
      new ValueComparator2(opgInDegreeSortingShop);
    TreeMap<Long, Long> tmp = new TreeMap<Long, Long>(vc2);
    int i=0;
    for(Entry<Long, Long> es :
      opgInDegreeSortingShop.entrySet())  {

      if (i==k)
        break;
      tmp.put(es.getKey(), es.getValue());
      i++;
    }

    return tmp;
  }

  /**
   * Computes node-based visit of the passed node,
   * a DFS search, in which if a node is similar to the
   * given item it is added to the userList.
   * @param v the start node (usally taken from the Top N vector
   * @param x the id of the item to search for buyers
   * @param theta the threshold in range [0:1].
   * @param userList the list of users to recommend item x in
   * a list of possible shops
   * @throws Exception
   */
```

```
public void nodeBasedVisit(Long v, long x,
  double theta, HashMap<Long, Double> userList) {

  // if v.label == unvisited
  if(!personeVisitate.get(v)) {
    // v.label = visited
    personeVisitate.put(v, true);
    //if sim(v, x) > THETA
    double ssimilarity = semanticSimilarity(v, x);
    if(ssimilarity > theta) {
      // if the entity is reachable in a shop by person
      boolean shopPossible = false;

      userList.put(v, ssimilarity);

      /*
      for(Long cShop :
        entitiesList.get( (long) x ).getShops() ) {
        if(peopleList.get( v ).getKnownShops()
          .contains(cShop.toString()) ) {
          shopPossible = true;
          if(userList.get(v) == null) {
            // add v to the current value of
            // userList create the structure for containing
            // shops
            userList.put(v, new ArrayList<Long>());

          }
          // add this shop (that user knows) to
          // recommended shops
          userList.get(v).add(cShop);

        }
      }*/


      // for all e = (v, v') and (v', v) in E
      // do the node based visit recursively
      for (Long v1 : oneNodeProjectionGraph.get(v).keySet())  {
        nodeBasedVisit(v1, x, theta, userList);
      }
    }
  }
}
```

```
/**
 * Similar to nodeBasedVisitShop but for statistics
 * acquisition with different sizes.
 * given item it is added to the userList.
 * @param v the start node (usally taken from the Top N vector
 * @param x the id of the item to search for buyers
 * @param theta the threshold in range [0:1].
 * @param userList the list of users to recommend item x
 * in a list of possible shops
 * @throws Exception
 */
public void nodeBasedVisitShopForStatistics(long v, long x,
  double theta, HashMap<Long, Double> userList, long hopMax) {
  // if v.label == unvisited
  if(!personeVisitate.get(v)) {
    // v.label = visited
    personeVisitate.put(v, true);
    //if sim(v, x) > THETA
    double similarity = semanticSimilarity(v, x);
    if(similarity > theta) {
      // if the entity is reachable in a shop by person
      boolean shopPossible = false;

      for(long cShop :
        entitiesList.get( (long) x ).getShops() ) {
        if(peopleList.get( v ).getKnownShops()
          .contains(String.valueOf(cShop)) ) {
          shopPossible = true;
          if(userList.get(v) == null) {
            // add v to the current value of userList create
            // the structure for containing shops
            userList.put(v, similarity);
          }
        }
      }

      // check if maximum distance is reached
      if(hopMax >= 0) {
        // for all e = (v, v') and (v', v) in E do the node
        // based visit recursively ONLY if they are without
        // neighborgs
        for (long v1 :
          shopOneNodeProjectionGraph.get(v).keySet()) {
```

```
            nodeBasedVisitShopForStatistics(
              v1, x, theta, userList, (hopMax - 1));
          }
        }
      }
    }
  }

  /**
   * Computes node-based visit of the passed node,
   * a DFS search, in which if a node is similar to the
   * given item it is added to the userList.
   * @param v the start node (usally taken from the Top N vector
   * @param x the id of the item to search for buyers
   * @param theta the threshold in range [0:1].
   * @param userList the list of users to recommend
   * item x in a list of possible shops
   * @throws Exception
   */
  public void nodeBasedVisitShop(long v, long x, double theta,
    Map<Long, ArrayList<Long>> userList, long hopMax) {

    // if v.label == unvisited
    if(!personeVisitate.get(v)) {
      // v.label = visited
      personeVisitate.put(v, true);
      //if sim(v, x) > THETA
      if(semanticSimilarity(v, x) > theta)  {
        // if the entity is reachable in a shop by person
        boolean shopPossible = false;

        HashMap<Long, Float> shopSortedHash =
          new HashMap<Long, Float>();
        ValueComparator vc2 =
          new ValueComparator(shopSortedHash);
        TreeMap<Long, Float> shopSorted =
          new TreeMap<Long, Float>(vc2);

        for(long cShop :
          entitiesList.get( (long) x ).getShops() ) {
          if(peopleList.get( v ).getKnownShops()
            .contains(String.valueOf(cShop)) )  {
            shopPossible = true;
            if(userList.get(v) == null) {
```

```
            // add v to the current value of userList create
            // the structure for containing shops
            userList.put(v, new ArrayList<Long>());
            //shopSortedHash = new TreeMap<Long, Float>();
          }
          // add this shop (that user knows) to
          // recommended shops
          if(peopleList.get(v).getKnownShops()
            .contains(String.valueOf(cShop))) {
            //userList.get(v).add(cShop);
            float simShop = (float) 0.0;
            if( peopleList.get(v).getPerShopForgettingFactor()
              .containsKey(cShop) )
              simShop = (float) (
                0.5*peopleList.get(v)
                  .getPerShopForgettingFactor().get(cShop)
                + 0.5 * shopPersonSemanticSimilarity(cShop, v));
            else
              simShop = (float) (
                0.5 * shopPersonSemanticSimilarity(cShop, v));
            shopSortedHash.put(cShop, simShop);
          }
        }
      }

      shopSorted.putAll(shopSortedHash);
      for(long cShop : shopSorted.keySet())
        userList.get(v).add(cShop);

      // check if maximum distance is reached
      if(hopMax >= 0) {
        // for all e = (v, v') and (v', v) in E do the
        // node based visit recursively ONLY if they are
        // without neighborgs
        for (long v1 :
          shopOneNodeProjectionGraph.get(v).keySet()) {
          nodeBasedVisitShop(v1, x,
            theta, userList, (hopMax - 1));
        }
      }
    }
  }
}
```

```
/**
 * Computes semantic similarity measure of user u
 * and item x like in definition 3 of paper
 * "Graph Searching Algorithms For Semantic-Social
 * Recommendetion".
 * Quando ci sono diversi IPT per un dato oggetto
 * considera quello che produce similarita massima, perche:
 * siano A e B due oggeti.
 * Sapendo che A ha 2 IPT che producono similarita 0.8 e
 * 0.1 e che B ha 2 IPT che producono 0.5 e 0.5,
 * l'oggetto con similarita piu alta dovrebbe essere A con 0.8.
 * Se invece di prendere il massimo si prendesse la somma
 * delle similarita si favorirebbero oggetti con molti IPT,
 * oltre a far vincere oggetti che hanno pochi IPT ma
 * abbastanza alti come nell'esempio, in cui vincerebbe B.
 * "Graph Searching Algorithms For Semantic-Social
 * Recommendetion"
 * @param u string id of the user
 * @param x item id in range [0:n-1] where n is the
 * size of the dataset
 * @return
 */
public double semanticSimilarity(long u, long x)  {
  double result = 0.0;

  // get required elements
  HashMap<Long, Long> UPTu = UPT.get(u);
  ArrayList<HashMap<Long, Long>> IPTx = IPT.get(x);

  // creates a general IPT, a tree not al list
  HashMap<Long, Long> globalIPTx = new HashMap<Long, Long>();

  try {
    for(HashMap<Long, Long> currentIPTx : IPTx) {
      for(Entry<Long, Long> entry : currentIPTx.entrySet())
        globalIPTx.put(entry.getKey(), entry.getValue());
    }
  } catch(NullPointerException e) {
    System.err.println(e.getMessage());
    System.out.println("x " + x);
    System.out.println("u " + u);
    System.out.println("IPTx " + IPTx);
  }
```

```
    // computes mu
    int sumu = 0, sumx = 0;
    for(Long l : UPTu.values())
      sumu += l;
    for(Long l : globalIPTx.values())
      sumx += l;

    int mu = Math.min(sumu, sumx);

    // intersection is computed this way:
    // for all term in UPT of u it is in the intersection if
    // is contained in IPT of item x
    int sumIntersection = 0;
    for(Entry<Long, Long> entry : UPTu.entrySet())
      if(globalIPTx.containsKey(entry.getKey()))
        sumIntersection += entry.getValue();

    result = sumIntersection / (double)mu;

    return result;
  }

  /**
   * Computes node-based visit of the user
   * one node projection graph contained in otherAlg,
   * and returns the list of the users for which
   * this object must be recommended to.
   * @param otherAlg instance of GSASSR
   * containing a one node projection graph
   * @param x the object to recommend
   * @param n the limit of users top n vector
   * of users with highest centrality degree
   * @param theta the threshold limit,
   * to avoid a full visit of the graph
   * @return the users list to recommend
   * object x in a list of possible shops
   */
  public TreeMap<Long, Double> nodeBasedVisitAlgorithm(
    AlgorithmVersion3 otherAlg, long x, int n, double theta) {

    // 1 - resets the "visited" label of each node
    otherAlg.resetVisitedVertices();

    // 2 -  create the empty user list (when this algorithm runs
```

```
    //     with an Item as input this list contains
    //     users to recommends the item to.
    HashMap<Long, Double> usersList =
      new HashMap<Long, Double>();
    semanticSimilarityComparator ssc =
      new semanticSimilarityComparator(usersList);
    TreeMap<Long, Double> usersSortedList =
      new TreeMap<Long, Double>(ssc);

    // 3 -  compute the degree centrality of each vertex
    //            in the one node projection graph
    //     and creates the top-n-vector with
    //     first n nodes with highest degree
    TreeMap<Long, Long> topKVector =
      otherAlg.computeTopKDegreeVector(n);

    // 4 -  for each vertex v in topKVector comutes
    //     node-based visit
    otherAlg.resetVisitedVertices();
    for(Long userName : topKVector.values())  {
      otherAlg.nodeBasedVisit(
        userName, x, theta, usersList);
    }

    // sort the map
    usersSortedList.putAll(usersList);

    return usersSortedList;
  }

  /**
   * Similar to shopNodeBasedVisitAlgorithm but
   * used for statistics generation
   * @param otherAlg instance of GSASSR containing
   * a one node projection graph
   * @param x the object to recommend
   * @param n the limit of users top n vector of
   * users with highest centrality degree
   * @param theta the threshold limit, to avoid
   * a full visit of the graph
   *
   * @param hopMax
   * @return the users list to recommend object
   * x in a list of possible shops
```

```
 */
public TreeMap<Long, Double>
  shopNodeBasedVisitAlgorithmForStatistics(
    AlgorithmVersion3 otherAlg, long x,
    int n, double theta, long hopMax) {

  // 1 - resets the "visited" label of each node
  otherAlg.resetVisitedVertices();

  // 2 -  create the empty user list (when this
  //    algorithm runs with an Item as input this list contains
  //    users to recommends the item to.
  //Map<Long, ArrayList<Long>> usersList =
    new HashMap<Long, ArrayList<Long>>();
  HashMap<Long, Double> usersList =
    new HashMap<Long, Double>();
  semanticSimilarityComparator ssc =
    new semanticSimilarityComparator(usersList);
  TreeMap<Long, Double> usersSortedList =
    new TreeMap<Long, Double>(ssc);

  // 3 -  compute the degree centrality of each
  //    vertex in the one node projection graph
  //    and creates the top-n-vector with first n
  //    nodes with highest degree
  TreeMap<Long, Long> topKVector =
    otherAlg.computeTopKDegreeVectorShop(n);

  // 4 -  for each vertex v in topKVector comutes
  //    node-based visit
  otherAlg.resetVisitedVertices();
  for(long userName : topKVector.values())  {
    //if(shopOneNodeProjectionGraph.get(userName).size() == 0)
    otherAlg.nodeBasedVisitShopForStatistics(
      userName, x, theta, usersList, hopMax);
  }

  // sort the map
  usersSortedList.putAll(usersList);

  return usersSortedList;
}

/**
```

```
 * Computes node-based visit of the shop one node projection
 * graph to avoid problems like users without recommendations
 * becouse they have no nodes in the onpg
 * @param otherAlg instance of GSASSR containing a one node
 * projection graph
 * @param x the object to recommend
 * @param n the limit of users top n vector of users with
 * highest centrality degree
 * @param theta the threshold limit, to avoid a full
 * visit of the graph
 *
 * @param hopMax
 * @return the users list to recommend object x in a list of
 * possible shops
 */
public Map<Long, ArrayList<Long>> shopNodeBasedVisitAlgorithm(
  AlgorithmVersion3 otherAlg, long x, int n, double theta,
  long hopMax) {

  // 1 - resets the "visited" label of each node
  otherAlg.resetVisitedVertices();

  // 2 -  create the empty user list (when this algorithm runs
  //     with an Item as input this list contains
  //     users to recommends the item to.
  Map<Long, ArrayList<Long>> usersList =
    new HashMap<Long, ArrayList<Long>>();

  // 3 -  compute the degree centrality of each vertex in
  //     the one node projection graph
  //     and creates the top-n-vector with first n nodes
  //     with highest degree
  TreeMap<Long, Long> topKVector =
    otherAlg.computeTopKDegreeVectorShop(n);

  // 4 -  for each vertex v in topKVector comutes node-based
  //     visit
  otherAlg.resetVisitedVertices();
  for(long userName : topKVector.values()) {
    otherAlg.nodeBasedVisitShop(userName, x, theta,
      usersList, hopMax);
  }

  return usersList;
```

```
}

/**
 * Add activities to each shop.
 */
public void updateShopList() {
  // add all related activities to each shop
  for(Activity activity : activitiesList.values()) {
    shopList.get(activity.getShop())
      .addActivity(activity.getEntity(), activity);
  }
}

/**
 * Computes shop trends for each shop using
 * exponential leveling.
 * @param endDate
 */
public void computeShopTrendsWithExponentialLeveling(
  Date endDate) {
  // compute trends for each shop
  for(Long shopId : shopList.keySet()) {
    Shop tmpShop = shopList.get(shopId);
    tmpShop.computeShopTrendsWithExponentialLeveling(endDate);
  }
}

/**
 * Computes shop trends for each shop using
 * absolute frequencies.
 * @param endDate
 */
public void computeShopTrendsWithAbsoluteFrequencies(
  Date endDate) {
  // compute trends for each shop
  for(Long shopId : shopList.keySet()) {
    Shop tmpShop = shopList.get(shopId);
    tmpShop.computeEntitiesTrend(endDate);
  }
}

/**
 * Computes the activity weights of the people.
 * @param nowDate
```

```
 * @param halfLife
 */
public void computeActivityWeights(Date nowDate, int halfLife)
{
  for(long i = 0; i < (long)peopleList.size(); i++)
    this.peopleList.get(i).computeActivityWeights(
      nowDate, halfLife);
}

/**
 * Comput forgetting factor for each shop of each person
 * @param now
 * @param halfLife
 */
public void computePeopleShopForgettingFactor(
  Date now, int halfLife) {
  // comput forgetting factor for each shop of each person
  for(Person p : this.peopleList.values())  {
    p.computeShopForgettingFactor(now, halfLife);
  }

  return;
}

/**
 * Makes first initializations of the algorithm.
 * This is the 3rd version and contains both state of
 * the art algorithm and shop based one
 * @param activitiesList
 * @param allActivitiesList
 * @param entitiesList
 * @param IPT
 * @param UPT
 * @param personeOggettiVotiTraining
 * @param personeOggettiVotiTest
 * @param peopleList
 * @param shopList
 */
public AlgorithmVersion3(
  HashMap<Long, Activity> activitiesList,
  HashMap<Long, Activity> allActivitiesList,
  HashMap<Long, Entity> entitiesList,
  HashMap<Long, ArrayList<HashMap<Long, Long>>> IPT,
  HashMap<Long, HashMap<Long, Long>> UPT,
```

```
    HashMap<Long, HashMap<Long, Long>> personeOggettiVotiTraining,
    HashMap<String, HashMap<Long, Long>> personeOggettiVotiTest,
    HashMap<Long, Person> peopleList,
    HashMap<Long, Shop> shopList) {

    // initialization
    this.IPT = IPT;
    this.UPT = UPT;
    this.activitiesList = activitiesList;
    this.allActivitiesList = allActivitiesList;
    this.personeOggettiVotiTraining = personeOggettiVotiTraining;
    this.personeOggettiVotiTest = personeOggettiVotiTest;
    this.peopleList = peopleList;
    this.entitiesList = entitiesList;
    this.shopList = shopList;

    personeVisitate = new HashMap<Long, Boolean>();
    entriesAge = new HashMap<Long, Float>();
    peoplePerShop = new HashMap<Long, Long>();

    // istanzia la matrice del grafo bipartito
    ratingsPersone = new HashMap<Long, HashMap<Long, Double>>();
    ratingsShopPersone =
      new HashMap<Long, HashMap<Long, Long>>();

    // add the rating matrix
    for(Activity activity : activitiesList.values())  {
      this.peopleList.get(activity.getPerson())
        .addActivity(activity);
      this.peopleList.get(activity.getPerson())
        .addShop( String.valueOf( activity.getShop() ));
      this.peopleList.get(activity.getPerson())
        .addPerShopActivity(activity.getShop(), activity
        .getIdActivity(), activity.getDateTime());

      if(!peoplePerShop.containsKey( activity.getShop() ))
        peoplePerShop.put( activity.getShop() , (long) 1);
      else
        peoplePerShop.put(activity.getShop(),
          peoplePerShop.get(activity.getShop()) + 1);
    }

    // comutes average people per shop for statistics
    double sum = 0.0;
```

```
    for (long numPeople: peoplePerShop.values())
      sum += numPeople;

    sum = sum / (double)peoplePerShop.size();

    System.out.println("People per shop (AVG: " + sum);

    for(Activity activity : allActivitiesList.values()) {
      this.peopleList.get(activity.getPerson())
        .addKnownShop( String.valueOf( activity.getShop() ));
    }

    return;
  }

  class ValueComparator2
    implements Comparator<Long>, Serializable {

    /**
     *
     */
    private static final long serialVersionUID
      = 8882531331739831768L;

    Map<Long, Long> base;
    public ValueComparator2(Map<Long, Long> base) {
      this.base = base;
    }

    // Note: this comparator imposes orderings that
    // are inconsistent with equals.
    public int compare(Long a, Long b) {
      if (a >= b) {
        return -1;
      } else {
        return 1;
      } // returning 0 would merge keys
    }
  }

  public class ValueComparator
    implements Comparator<Long> {

    Map<Long, Float> base;
```

```
  public ValueComparator(Map<Long, Float> base) {
    this.base = base;
  }

  public int compare(Long o1, Long o2) {
    // TODO Auto-generated method stub
    if (base.get(o1) >= base.get(o2)) {
      return -1;
    } else {
      return 1;
    } // returning 0 would merge keys
  }

}

/**
 * Comparator for product trends.
 * @author Marco
 *
 */
class ProductTrendsComparator
  implements Comparator<Long>, Serializable {

  /**
   * for serialization used sometimes in testing
   */
  private static final long serialVersionUID
    = 8882531331739831768L;

  Map<Long, Float> base;
  public ProductTrendsComparator(Map<Long, Float> base) {
    this.base = base;
  }

  // nuovo
  public int compare(Long o1, Long o2) {

    if(o1 == null)
      return -1;
    else if(o2 == null)
      return 1;

    if(base.get(o1) == null)
      return -1;
```

```
      else if(base.get(o2) == null)
        return 1;

      if (base.get(o1) >= base.get( o2 )) {
        return -1;
      } else {
        return 1;
      } // returning 0 would merge keys
    }
  }

  /**
   * Comparator used to sort entities by semantic similarity
   * @author Marco
   *
   */
  class semanticSimilarityComparator implements Comparator<Long> {

    Map<Long, Double> base;
    public semanticSimilarityComparator(Map<Long, Double> base) {
      this.base = base;
    }

    // this comparator imposes orderings that are
    // inconsistent with equals.
    public int compare(Long a, Long b) {
      if (base.get(a) >= base.get(b)) {
        return -1;
      } else {
        return 1;
      }
    }
  }

}
```

Appendice B

Codice della Classe Activity

```
package com.rollnext.classes;

import java.io.Serializable;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;

import com.rollnext.exceptions.ShopUnavailableException;

/**
 * A person join an entity by creating an activity.
 * An activity is the relation between people and
 * entities. It contains informations like date of
 * the activity creation and the shop in which the
 * user joined the entity.
 * @author Marco
 *
 */
public class Activity implements Serializable{


  /**
   *
   */
  private static final long serialVersionUID
    = -3135589627683229142L;

  /**
   * Creates a new activity. If the shop is not in the entity's
```

```
     * shop list throws an exception
     * @param person
     * @param entity
     * @param shop
     * @param dateTime
     * @param idActivity
     * @param description
     * @throws ShopUnavailableException not yet implemented
     */
    public Activity(long person, long entity, long shop,
        Date dateTime, long id, String description) {
      super();
      this.person = person;
      // TODO gestire l'eccezione
      this.entity = entity;
      this.shop = shop;
      this.dateTime = dateTime;
      this.idActivity = id;
      this.description = description;

      interests = new ArrayList<String>();

    }

    private long person;
    private long entity;
    private long shop;
    private Date dateTime;
    private long idActivity;
    private String description;
    private String interest;

    private Double rate;
    // list of most specific interests of the entity related to
    // this activity
    private ArrayList<String> interests;

    /**
     * Add an interests to the list of interests about this
     * activity. Activity's interests should be the most
     * specific ones of the entity this activity is related to
     * @param interest
     */
    public void addInterest(String interest)  {
```

```
    interests.add(interest);
  }

  public ArrayList<String> getInterests() {
    return interests;
  }



  /**
   * Get the description of this activity.
   * @return
   */
  public String getDescription() {
    return description;
  }

  /**
   * Set the description of this activity.
   * @param description
   */
  public void setDescription(String description) {
    this.description = description;
  }

  /**
   * Get the person idActivity of who made this activity.
   * @return the person ID
   */
  public long getPerson() {
    return person;
  }
  /**
   * Set the person ID of who made this activity.
   * It should not be edited after the first
   * assignment.
   * @param person
   */
  public void setPerson(long person) {
    this.person = person;
  }

  /**
   * Get the idActivity of the entity related to this activity.
```

```
     * @return the entity idActivity
     */
    public long getEntity() {
      return entity;
    }

    /**
     * Set the idActivity of the entity related to this activity.
     * It should not be edited after the first
     * assignment.
     * @param entity
     */
    public void setEntity(long entity) {
      this.entity = entity;
    }

    /**
     * Get the shop idActivity related to this activity.
     * @return the shop idActivity
     */
    public long getShop() {
      return shop;
    }

    /**
     * Set the idActivity of the shop related to this activity.
     * It should not be edited after the first
     * assignment. It should be one of the shops
     * available in the entity's shop list
     * @param shop
     */
    public void setShop(long shop) {
      this.shop = shop;
      // TODO aggiungere la segnalazione dell'eccezione.
    }

    /**
     * Get the timestamp of this activity
     * @return the activity's timestamp
     */
    public Date getDateTime() {
      return dateTime;
    }
```

```
  /**
   * Set the timestamp of the activity creation date.
   * @param timestamp
   */
  public void setDateTime(Date dateTime) {
    this.dateTime = dateTime;
  }

  /**
   * Get the activity idActivity
   * @return
   */
  public long getIdActivity() {
    return idActivity;
  }

  public String getInterest() {
    return interest;
  }

  public void setInterest(String interest) {
    this.interest = interest;
  }

  public String toString()  {
    return "[idActivity: "+idActivity+", person: "+person+
    ", entity: "+entity+", date: "+dateTime+", interests: "+
    interests+"]";
  }

}
```

Appendice C

Codice della Classe Entity

```
package com.rollnext.classes;

import java.io.Serializable;
import java.util.ArrayList;

import com.rollnext.exceptions.IncorrectOntologyException;

public class Entity implements Serializable {

  /**
   *
   */
  private static final long serialVersionUID
    = -1740796627848929225L;
  private long id;
  private String name;
  private ArrayList<Long> shops;
  private String taxonomy;

  /**
   * a list of taxonomies, in which each taxonomy is a
   * list of interests in which each interest
   * is at index corresponding to the level in the taxonomy tree
   */
  private ArrayList<ArrayList<String>> taxonomiesList;

  public Entity(long id, String name, ArrayList<Long> shops,
      String taxonomy) {
    super();
    this.id = id;
```

```
    this.name = name;
    this.shops = shops;
    this.taxonomy = taxonomy;
    // TODO gestire la scorretta ontologia
    taxonomiesList = new ArrayList<ArrayList<String>>();
  }


  public void setId(long id) {
    this.id = id;
  }


  /**
   * One object may have multiple IPT (see amazon dataset)
   * so each IPT needs to be added individually
   * @param IPT
   */
  public void addTaxonomy(ArrayList<String> IPT)  {
    taxonomiesList.add(IPT);
  }

  public ArrayList<ArrayList<String>> getTaxonomiesList() {
    return taxonomiesList;
  }


  public void setTaxonomiesList(
      ArrayList<ArrayList<String>> taxonomiesList) {
    this.taxonomiesList = taxonomiesList;
  }


  /**
   * get the entity id
   * @return the entity id
   */
  public long getId() {
    return id;
  }

  /**
   * get the entity name
   * @return the entity name
```

```
 */
public String getName() {
  return name;
}

/**
 * set the entity name
 * @param name
 */
public void setName(String name) {
  this.name = name;
}

/**
 * get the entiy shops list. Only the id of the shops
 * are reported
 * @return
 */
public ArrayList<Long> getShops() {
  return shops;
}

/**
 * set the entity shops.
 * @param name
 */
public void setShops(ArrayList<Long> shops) {
  this.shops = shops;
}

/**
 * get the entity taxonomy. The taxonomy is the id of the
 * category the entity belongs to,
 * For example {Thing, Book, Horror book} may be a taxonomy
 * for a horror book, and the taxonomy returned by this method
 * probably will be something similar
 * to #horror_book.
 * @return the taxonomy (list) of the entity.
 */
public String getTaxonomy() {
  return taxonomy;
}

/**
```

```
    * set the entity taxonomy. The taxonomy must exist in the
    * ontology tree, otherwise a
    * IncorrectTaxonomyException is thrown.
    * @param taxonomy
    */
   public void setTaxonomy(String taxonomy) {
     this.taxonomy = taxonomy;
   }

   public String toString()  {
     return "[id: "+id+", name: "+name+", shops: "+shops+
     ", taxonomy: "+taxonomy+", taxonomiesList"+
     taxonomiesList+"]";
   }
}
```

Appendice D

Codice della Classe Person

```
package com.rollnext.classes;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;

public class Person implements Serializable {

  /**
   *
   */
  private static final long serialVersionUID
    = 499287523953675642 1L;

  private long id;
  private String Name;
  private String Surname;

  public void setId(long id) {
    this.id = id;
  }

  /**
   * contains for each shop a set of activities id
   */
  private HashMap<Long, HashSet<Long>> perShopActivities;
  /**
```

```
 * contains for each shop last used date: the maximum date
 * of activities for this shop
 */
private HashMap<Long, Date> perShopLastUsedDate;

/**
 * contains for each shop last used date: the maximum date
 * of activities for this shop
 */
private HashMap<Long, Double> perShopForgettingFactor;

/**
 * Interests and weights for each one of this person.
 */
private HashMap<Long, Double> interests;
/**
 * a list of taxonomies, in which each taxonomy is a list
 * of interests in which each interest
 * is at index corresponding to the level in the
 * taxonomy tree (USED FOR VERSION 2
 */
private ArrayList<ArrayList<String>> taxonomiesList;
/**
 * Activities of the person. each activity is associated
 * to a weight.
 */
private HashMap<Activity, Double> activityList;
/**
 * The set of entities this user owns. Used for faster
 * controls in user similarity calculation
 */
private Set<Long>entities;
/**
 * The set of shop in which the user do activities
 */
private HashSet<String> usedShops;
/**
 * shops in the user's area
 */
private HashSet<String> knownShops;

/**
 * Each person has a list of neighbors. Neighbors are people
 * related to
```

```
 * at least one entity this person is related to.
 * This means the two persons have at least one interest
 * in common.
 */
private ArrayList<Long> neighbors;
private HashMap<Long, Double> neighborSimilarity;

/**
 * Create a new person with essential parameters.
 * @param id
 * @param name
 * @param surname
 */
public Person(long id, String name, String surname) {
  super();
  this.id = id;
  Name = name;
  Surname = surname;

  interests = new HashMap<Long, Double>();
  activityList = new HashMap<Activity, Double> ();
  neighbors = new ArrayList<Long>();

  neighborSimilarity = new HashMap<Long, Double>();
  entities = new HashSet<Long>();
  usedShops = new HashSet<String>();
  knownShops = new HashSet<String>();

  taxonomiesList = new ArrayList<ArrayList<String>>();

  perShopActivities = new HashMap<Long, HashSet<Long>>();
  perShopLastUsedDate = new HashMap<Long, Date>();

  perShopForgettingFactor = new HashMap<Long, Double>();
}

public HashMap<Long, Date> getPerShopLastUsedDate() {
  return perShopLastUsedDate;
}

public HashMap<Long, HashSet<Long>> getPerShopActivities() {
  return perShopActivities;
}
```

```
/**
 * add activity to given shop. Set last used date for shop,
 * to compute shop
 * forgetting factor.
 * @param shop
 * @param activity
 * @param dateOfActivity
 */
public void addPerShopActivity(long shop, long activity,
    Date dateOfActivity) {
  // makes sure the shop has an empty list of activities
  if(perShopActivities.get(shop) == null)
    perShopActivities.put(shop, new HashSet<Long>());

  // add the activity to current shop
  perShopActivities.get(shop).add(activity);

  // makes sure the shop has at least a date, the actual
  // maximum one
  if(perShopLastUsedDate.get(shop) == null)
    perShopLastUsedDate.put(shop, dateOfActivity);

  // or compares actual dates
  Date oldDate = perShopLastUsedDate.get(shop);
  if(oldDate.after(dateOfActivity))
    perShopLastUsedDate.put(shop, oldDate);
  else
    perShopLastUsedDate.put(shop, dateOfActivity);

  return;
}

/**
 * Compute forgetting factor for each shop in range (0,1].
 * 0 means real old shop, 1 means a shop used at now date
 * @param now
 * @param halfLife
 */
public void computeShopForgettingFactor(Date now, int halfLife) {

  for (long shop : perShopLastUsedDate.keySet()) {
    // gets difference in days between the dates
    long dateDifference = now.getTime() - perShopLastUsedDate
      .get(shop).getTime();
```

```
    dateDifference = dateDifference / (1000 * 60 * 60 * 24);

    // compute weight of the date based on the given values
    double result = Math.exp( -1*( ( Math.log(2) *
      dateDifference) / halfLife ) );
    perShopForgettingFactor.put(shop, result);
  }

}

/**
 * One object may have multiple IPT (see amazon dataset)
 * so each IPT needs to be added individually
 * @param IPT
 */
public void addTaxonomy(ArrayList<String> IPT)  {
  taxonomiesList.add(IPT);
}

public ArrayList<ArrayList<String>> getTaxonomiesList() {
  return taxonomiesList;
}

public void setTaxonomiesList(ArrayList<ArrayList<String>>
    taxonomiesList) {
  this.taxonomiesList = taxonomiesList;
}

public long getId() {
  return id;
}

public String getName() {
  return Name;
}

public void setName(String name) {
  Name = name;
}

public String getSurname() {
  return Surname;
}
```

```
public void setSurname(String surname) {
  Surname = surname;
}

public HashMap<Long, Double> getInterests() {
  return interests;
}

public void setInterests(HashMap<Long, Double> interests) {
  this.interests = interests;
}

public HashMap<Activity, Double> getActivityList() {
  return activityList;
}

public ArrayList<Long> getNeighbors() {
  return neighbors;
}

public Set<Long> getEntities() {
  return entities;
}

public void setEntities(Set<Long> entities) {
  this.entities = entities;
}

public HashSet<String> getUsedShops() {
  return usedShops;
}

public HashSet<String> getKnownShops() {
  return knownShops;
}

public void setKnownShops(HashSet<String> knownShops) {
  this.knownShops = knownShops;
}

/**
 * Add a shop to user's shop list
 * @param shop
 */
```

```
public void addShop(String shop)  {
  usedShops.add(shop);
}

/**
 * Add a shop to user's known shop list
 * @param shop
 */
public void addKnownShop(String shop) {
  knownShops.add(shop);
}



public HashMap<Long, Double> getPerShopForgettingFactor() {
  return perShopForgettingFactor;
}

/**
 * Add an activity to the activity list of the user
 * @param activity
 */
public void addActivity(Activity activity)  {
  activityList.put(activity, (double) 0);
  entities.add(activity.getEntity());

}

/**
 * Update the list of interests of the person
 * If it is first time this interests is update it adds
 * this interest with weight 0
 * If the person already has this interest with
 * weight n the interest is updated to weight n+1
 * @param interest the interest to be updated
 */
/*
public void updateInterests() {
  long currentWeight = (interests.get(interest) != null) ?
      (interests.get(interest)+1) : 1;
  interests.put(interest, currentWeight);
}*/

/**
```

```
 * Compute the time-weights of each activity of this person
 * with provided informations
 * @param now the date to be used for maximum weight
 * (should be current date)
 * @param halfLife the value used to control the forgetting
 * factor of each activity.
 * Lower halflife means activities are forgotten rapidly.
 * If difference between now and
 * an activity date is equals to halflife,
 * computed weight will be 0.5
 */
public void computeActivityWeights(Date now, int halfLife) {
  for(Activity activity : activityList.keySet()) {
    // gets difference in days between the dates
    long dateDifference = now.getTime() - activity
      .getDateTime().getTime();
    dateDifference = dateDifference / (1000 * 60 * 60 * 24);

    // compute weight of the date based on the given values
    double result = Math.exp( -1*( ( Math.log(2) *
      dateDifference) / halfLife ) );
    activityList.put(activity, result);

    // add the activity weight to interests weight or insert
    // it the first time
    for(String interest : activity.getInterests()) {
      double iWeight = (interests.get(interest) != null) ?
        interests.get(interest) : 0.0;
      interests.put(Long
        .parseLong(interest), iWeight + result);
    }


  }
  return;
}

/**
 * Add the neighbor only if it is not already in the
 * neighbors list
 * This check makes easier the addition of neighbors
 * returned by a query.
 * @param neighbor
 */
```

```
  public void addNeighbor(Long neighbor){

    if(neighbor != this.id)
      if(! neighbors.contains(neighbor))  {
        neighbors.add(neighbor);
      }
  }

  /**
   * Compute 1-norm of the interest weights vector. O(n)
   */
  public void normalizeInterests()  {
    double sum = 0.0;
    for(double curr : interests.values())
      sum += curr;
    for(Long i : interests.keySet())
      interests.put(i, interests.get(i)/sum);
  }

  public void addSimilarity(long idNeighbor, double similarity) {
    neighborSimilarity.put(idNeighbor, similarity);
  }

  @Override public boolean equals (Object other)  {
    // check for self comparison
    if ( this == other ) return true;

    if(other instanceof Person) {
      // thu person ar the same if they have same idPerson
      Person p = (Person) other;
      if (this.id == p.getId())
        return true;
      else
        return false;
    }
    else
      return false;
  }

  public String toString()  {
    return "[id: "+id+", name: "+Name+", surname: "+Surname+"]";
  }

}
```

Appendice E

Codice della Classe Shop

```
package com.rollnext.classes;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Comparator;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
import java.util.TreeMap;

import com.rollnext.algorithm.AlgorithmVersion3;


public class Shop implements Serializable{

  /**
   *
   */
  private static final long serialVersionUID
    = -237765088861435149L;

  /**
   * Interests of this shop. Interests are the most specific
   * of each entity of the shop
   */
  private HashMap<String, Double> interests;

  /**
```

```
 * a list of taxonomies, in which each taxonomy is a list
 * of interests in which each interest
 * is at index corresponding to the level in the taxonomy tree
 */
private ArrayList<ArrayList<String>> taxonomiesList;

private ArrayList<Long> entities;



public ArrayList<Long> getEntities() {
  return entities;
}

/**
 * get the shop id
 * @return
 */
public long getId() {
  return id;
}

/**
 * set the shop id. It should not be modified after
 * first assignment
 * @param id
 */
public void setId(long id) {
  this.id = id;
}

/**
 * get the shop name
 * @return
 */
public String getName() {
  return name;
}

public HashMap<String, Double> getInterests() {
  return interests;
}

public void setInterests(HashMap<String, Double> interests) {
```

```
    this.interests = interests;
}

/**
 * One object may have multiple IPT (see amazon dataset)
 * so each IPT needs to be added individually
 * @param IPT
 */
public void addTaxonomy(ArrayList<String> IPT)  {
    taxonomiesList.add(IPT);
}

public ArrayList<ArrayList<String>> getTaxonomiesList() {
    return taxonomiesList;
}

public void setTaxonomiesList(
        ArrayList<ArrayList<String>> taxonomiesList) {
    this.taxonomiesList = taxonomiesList;
}

/**
 * set the shop name
 * @param name
 */
public void setName(String name) {
    this.name = name;
}

public HashMap<Long, ArrayList<Activity>>
        getActivitiesByEntity() {
    return activitiesByEntity;
}

/**
 * Adds a new activity to specified entity's activity list.
 * The activities are used to calculate entity's trend
 * for this shop.
 * @param entityId
 * @param activity
 */
public void addActivity(Long entityId, Activity activity) {
    // if the list does not exists it creates a new one
    if(activitiesByEntity.get(entityId) == null)
```

```
      activitiesByEntity.put(entityId,
        new ArrayList<Activity>());

    // then adds the activity to entityId's activity list
    activitiesByEntity.get(entityId).add(activity);
  }

  /**
   * Computes entities' trend for this shop. The trend of
   * an entity is the number
   * of activities related to the entity in the month ending
   * in computationDate
   * @param computationDate
   * @return
   */
  public TreeMap<Float, Long> computeEntitiesTrend(
      Date computationDate) {
    HashMap<Float, Long> itemsByTrend
      = new HashMap<Float, Long>();
    FloatComparator fc = new FloatComparator(itemsByTrend);
    TreeMap<Float,Long> sortedItemsByTrend
      = new TreeMap<Float,Long>(fc);

    HashMap<Long, Float> hmIBT = new HashMap<Long, Float>();
    ItemsTrendComparator itc = new ItemsTrendComparator(hmIBT);
    TreeMap<Long,Float> itemByTrend
      = new TreeMap<Long,Float>(itc);

    // set minimum date to 1 months ago
    Calendar c = Calendar.getInstance();
    c.setTime(computationDate);
    c.add(Calendar.MONTH, -24);
    Date minimumDate = c.getTime();

    // for each entity computes th me number of sold item
    // in  the specified time interval
    for(Long entity : activitiesByEntity.keySet()) {
      for(Activity activity : activitiesByEntity.get(entity)) {
        Date actDate = activity.getDateTime();
        if(actDate.after(minimumDate) &&
            actDate.before(computationDate)) {
          // avoid to add duplicate entities
          if(!sortedItemsByTrend.containsValue(entity))
            sortedItemsByTrend.put( (float) activitiesByEntity
```

```
            .get(entity).size(), entity);
        if(!hmIBT.containsKey(entity))
          hmIBT.put( (long) entity,
            (float) activitiesByEntity.get(entity).size());
      }

      // update trend of interests for the shop from the
      // shop born time to computation date
      ArrayList<ArrayList<String>> listTaxonomies
        = AlgorithmVersion3.entitiesList.get(activity
          .getEntity() ).getTaxonomiesList();
      ArrayList<String> iList = new ArrayList<String>();

      for(ArrayList<String> tList : listTaxonomies)
        iList.add(tList.get(tList.size() - 1));

      for ( String interest : iList ) {
        double tValue = (double) (interests.get(interest)
          != null ? interests.get(interest) : 0.0);
        interests.put(interest, (double) (tValue + 1.0));
      }
    }
  }

  //sortedItemsByTrend.putAll(itemsByTrend);

  itemByTrend.putAll(hmIBT);

  this.sortedItemsByTrend = sortedItemsByTrend;
  this.itemByTrend = itemByTrend;
  return sortedItemsByTrend;
}


/**
 * Computes shop trends with exponential levelling
 * (Hunter, 1986)
 * @param computationDate
 */
public void computeShopTrendsWithExponentialLeveling(
  Date computationDate) {

  // activities by entity for tyme
  // mappa oggetto x (tempo t x vendite)
```

```
HashMap<Long, HashMap<Integer, Float>> ArrayhmIBT
  = new HashMap<Long, HashMap<Integer, Float>>();

// array of items sorted by trend
HashMap<Long, Float> hmIBT = new HashMap<Long, Float>();
ItemsTrendComparator itc = new ItemsTrendComparator(hmIBT);
TreeMap<Long,Float> itemByTrend
  = new TreeMap<Long,Float>(itc);

// set an array of date interval: t, t-1, t-2, ...
Date[] times = new Date[13];
times[12] = computationDate;
Calendar c = Calendar.getInstance();
c.setTime(computationDate);
for(int i=11; i>=0; i--) {
  c.add(Calendar.MONTH, -2);
  times[i] = c.getTime();
  //System.out.println(i + "\t" + times[i]);
}

// for each entity computes th me number of sold item in
// the specified time interval
for(long entity : activitiesByEntity.keySet()) {
  if(ArrayhmIBT.get(entity) == null)
    ArrayhmIBT.put(entity, new HashMap<Integer, Float>());

  for(Activity activity : activitiesByEntity.get(entity)) {
    Date actDate = activity.getDateTime();

    for(int i=1; i<=12; i++) {
      // ensures to avoid NullPointerException
      if( ! ArrayhmIBT.get(entity).containsKey(i-1))
        ArrayhmIBT.get(entity).put( i-1, (float) 0.0);
      if(actDate.before(times[i]) &&
        actDate.after(times[i-1])) {

        ArrayhmIBT.get(entity).put( i-1, ArrayhmIBT
          .get(entity).get(i-1) + 1);
      }
    }
  }
}

// calcolo la costante di livellamento su tutti gli oggetti
```

```
    for(Entry<Long, HashMap<Integer, Float>> entry :
      ArrayhmIBT.entrySet())  {
      hmIBT.put(entry.getKey(),
          l(11, (float) 0.7, entry.getValue()));
    }

    itemByTrend.putAll(hmIBT);

    this.itemByTrend = itemByTrend;
    return;
  }

  /**
   * Calcola il livellamento secondo Roberts.
   * uno per ogni prodotto.
   * @param t indica il tempo: 0 ë il caso base
   * @param alpha costante di livellamento
   * @param y contiene t x valore
   */
  private float l(int t, float alpha,
    HashMap<Integer, Float> y)  {
    // caso base
    // restituisce il valore iniziale per convenzione
    if(t == 0)  {
       return y.get(0);
    }
    // caso ricorsivo
    // alpha * l(t) + (1-alpha) * l(t-1)
    return (alpha * y.get(t)) + ((1-alpha) * l(t-1, alpha, y));
  }

  public TreeMap<Long, Float> getItemByTrend() {
    return itemByTrend;
  }

  /**
   * create a new shop with provided parameters.
   * @param id
   * @param name
   */
  public Shop(long id, String name) {
    super();
    this.id = id;
    this.name = name;
```

```
    activitiesByEntity =
      new HashMap<Long, ArrayList<Activity>>();
    interests = new HashMap<String, Double>();

    taxonomiesList = new ArrayList<ArrayList<String>>();
  }

  public String toString() {
    return "[id: "+id+", name: "+name+", interests: "+
      interests+"]";
  }

  public TreeMap<Float, Long> getSortedItemsByTrend() {
    return sortedItemsByTrend;
  }

  public void setSortedItemsByTrend(TreeMap<Float, Long>
    sortedItemsByTrend) {
    this.sortedItemsByTrend = sortedItemsByTrend;
  }

  private long id;
  private String name;
  /**
   * This treemap contains a set of entity ids sorted by
   * entity trend
   */
  public TreeMap<Float,Long> sortedItemsByTrend;
  public TreeMap<Long,Float> itemByTrend;
  /**
   * Contains a set of activities related to this shop grouped
   * by entity
   */
  private HashMap<Long, ArrayList<Activity>> activitiesByEntity;
}

class FloatComparator implements Comparator<Float> {

  Map<Float, Long> base;
  public FloatComparator(Map<Float, Long> base) {
    this.base = base;
  }
```

```
  // Note: this comparator imposes orderings that are
  // inconsistent with equals.
  public int compare(Float a, Float b) {
    if (a >= b) {
      return -1;
    } else {
      return 1;
    }
  }
}

class ItemsTrendComparator implements Comparator<Long> {

  Map<Long, Float> base;
  public ItemsTrendComparator(Map<Long, Float> base) {
    this.base = base;
  }

  // Note: this comparator imposes orderings that are
  // inconsistent with equals.
  public int compare(Long a, Long b) {
    if (base.get(a) >= base.get(b)) {
      return -1;
    } else {
      return 1;
    } // returning 0 would merge keys
  }
}
```

Bibliografia

[1] D. Sulieman, M. Malek, H. Kadima, and D. Laurent, "Graph searching algorithms for semantic-social recommendation," in *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, pp. 733–738, IEEE Computer Society, 2012.

[2] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, no. 6, pp. 734–749, 2005.

[3] K. Wei, J. Huang, and S. Fu, "A survey of e-commerce recommender systems," in *Service Systems and Service Management, 2007 International Conference on*, pp. 1–5, IEEE, 2007.

[4] R. Baeza-Yates, B. Ribeiro-Neto, *et al.*, *Modern information retrieval*, vol. 463. ACM press New York, 1999.

[5] G. Salton, "Automatic text processing. 1989," 1989.

[6] B. Xu, M. Zhang, Z. Pan, and H. Yang, "Content-based recommendation in e-commerce," in *Computational Science and Its Applications–ICCSA 2005*, pp. 946–955, Springer, 2005.

[7] R. J. Mooney and L. Roy, "Content-based book recommending using learning for text categorization," in *Proceedings of the fifth ACM conference on Digital libraries*, DL '00, (New York, NY, USA), pp. 195–204, ACM, 2000.

[8] D. Billsus and M. J. Pazzani, "User modeling for adaptive news access," *User modeling and user-adapted interaction*, vol. 10, no. 2-3, pp. 147–180, 2000.

[9] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl, "Grouplens: applying collaborative filtering to usenet news," *Communications of the ACM*, vol. 40, no. 3, pp. 77–87, 1997.

[10] B. K. Mohan, B. J. Keller, and N. Ramakrishnan, "Scouts, promoters, and connectors: the roles of ratings in nearest neighbor collaborative filtering," in *Proceedings of the 7th ACM conference on Electronic commerce*, pp. 250–259, ACM, 2006.

[11] Y. H. Cho and J. K. Kim, "Application of web usage mining and product taxonomy to collaborative recommendations in e-commerce," *Expert systems with Applications*, vol. 26, no. 2, pp. 233–246, 2004.

[12] Y. H. Cho, J. K. Kim, and S. H. Kim, "A personalized recommender system based on web usage mining and decision tree induction," *Expert Systems with Applications*, vol. 23, no. 3, pp. 329–342, 2002.

[13] C. C. Aggarwal, J. L. Wolf, K.-L. Wu, and P. S. Yu, "Horting hatches an egg: A new graph-theoretic approach to collaborative filtering," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 201–212, ACM, 1999.

[14] S.-H. Min and I. Han, "Recommender systems using support vector machines," in *Web Engineering*, pp. 387–393, Springer, 2005.

[15] U. Shardanand and P. Maes, "Social information filtering: algorithms for automating "word of mouth"," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 210–217, ACM Press/Addison-Wesley Publishing Co., 1995.

[16] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pp. 43–52, Morgan Kaufmann Publishers Inc., 1998.

[17] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th international conference on World Wide Web*, pp. 285–295, ACM, 2001.

[18] M. Deshpande and G. Karypis, "Item-based top-n recommendation algorithms," *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 143–177, 2004.

[19] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins, "Eigentaste: A constant time collaborative filtering algorithm," *Information Retrieval*, vol. 4, no. 2, pp. 133–151, 2001.

[20] T. Hofmann, "Latent semantic models for collaborative filtering," *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 89–115, 2004.

[21] M. Balabanović and Y. Shoham, "Fab: content-based, collaborative recommendation," *Communications of the ACM*, vol. 40, no. 3, pp. 66–72, 1997.

[22] G. Linden, B. Smith, and J. York, "Amazon. com recommendations: Item-to-item collaborative filtering," *Internet Computing, IEEE*, vol. 7, no. 1, pp. 76–80, 2003.

[23] P. Melville, R. J. Mooney, and R. Nagarajan, "Content-boosted collaborative filtering for improved recommendations," in *AAAI/IAAI*, pp. 187–192, 2002.

[24] M. Claypool, A. Gokhale, T. Miranda, P. Murnikov, D. Netes, and M. Sartin, "Combining content-based and collaborative filters in an online newspaper," in *Proceedings of ACM SIGIR workshop on recommender systems*, vol. 60, Citeseer, 1999.

[25] C. Basu, H. Hirsh, W. Cohen, *et al.*, "Recommendation as classification: Using social and content-based information in recommendation," in *AAAI/IAAI*, pp. 714–720, 1998.

[26] A. Popescul, D. M. Pennock, and S. Lawrence, "Probabilistic models for unified collaborative and content-based recommendation in sparse-data environments," in *Proceedings of the Seventeenth conference on*

Uncertainty in artificial intelligence, pp. 437–444, Morgan Kaufmann Publishers Inc., 2001.

[27] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock, "Methods and metrics for cold-start recommendations," in *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 253–260, ACM, 2002.

[28] T. Hofmann, "Probabilistic latent semantic analysis," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pp. 289–296, Morgan Kaufmann Publishers Inc., 1999.

[29] S. Trewin, "Knowledge-based recommender systems," *Encyclopedia of library and information science*, vol. 69, no. Supplement 32, p. 69, 2000.

[30] S. E. Middleton, N. R. Shadbolt, and D. C. De Roure, "Ontological user profiling in recommender systems," *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 54–88, 2004.

[31] C.-N. Ziegler, G. Lausen, and L. Schmidt-Thieme, "Taxonomy-driven computation of product recommendations," in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pp. 406–415, ACM, 2004.

[32] M. Malek and D. Sulieman, "Exhaustive and guided algorithms for recommendation in a professional social network," in *7th conference on Application of Social Network Analysis (ASNA), Zurich*, 2010.

[33] J. J. Jiang and D. W. Conrath, "Semantic similarity based on corpus statistics and lexical taxonomy," 1997.

[34] P. Resnik, "Using information content to evaluate semantic similarity in a taxonomy," *arXiv preprint cmp-lg/9511007*, 1995.

[35] J. J. Ramasco, "Social inertia and diversity in collaboration networks," *The European Physical Journal Special Topics*, vol. 143, no. 1, pp. 47–50, 2007.

[36] T. Zhou, J. Ren, M. Medo, and Y.-C. Zhang, "Bipartite network projection and personal recommendation," *Physical Review E*, vol. 76, no. 4, p. 046115, 2007.

[37] N. Zheng and Q. Li, "A recommender system based on tag and time information for social tagging systems," *Expert Systems with Applications*, vol. 38, no. 4, pp. 4575–4587, 2011.

[38] J. M. Lucas and M. S. Saccucci, "Exponentially weighted moving average control schemes: properties and enhancements," *Technometrics*, vol. 32, no. 1, pp. 1–12, 1990.

[39] S. Roberts, "Control chart tests based on geometric moving averages," *Technometrics*, vol. 1, no. 3, pp. 239–250, 1959.

[40] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl, "Evaluating collaborative filtering recommender systems," *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, pp. 5–53, 2004.