Prefazione

Il "Cisco Visual Index" prevede che nel 2016 il traffico video costi tuirà circa l'86% del traffico Internet globale. Il progetto *Massive Adaptive Internet VIdeo STreaming using the clOud* (MAIVISTO) ha l'obiettivo di superare lo stato dell'arte internazionale nel cam po della distribuzione di video su Internet in streaming adattivo utilizzando infrastrutture di Cloud Computing. Si studieranno tec niche di stream switching DASH compliant, tecniche di orchestra zione intra cloud e inter cloud mediante Control Plane e tecniche di ranking e recommendation dei contenuti video.

L'obiettivo complessivo del progetto è la realizzazione del proto tipo di una piattaforma altamente scalabile per la distribuzione di contenuti audiovisivi dotata di sistemi automatici per l'adattamen to del video alla capacità della connessione Internet (non garantita e impredicibile), alle caratteristiche dei molteplici dispositivi di ac cesso nonché alle preferenze indicate in modo esplicito dall'utente o automaticamente determinate dal sistema.

Si svilupperà un Control Plane in grado di orchestrare le risorse di calcolo, di storage e di comunicazione distribuite su una o più cloud per gestire in modo economico ed elastico la distribuzione di video. La ricerca dei contenuti sarà effettuabile sulla base di infor

mazioni semantiche (es. tag) associate agli stessi dal distributore e/o dal produttore o dagli utenti. Infine, si svilupperà un sistema di recommendation dei video basato sia sulle abitudini di visione degli utenti, sia sul rank assegnato nel tempo ai contenuti (likes), sia sulla descrizione del contenuto.

Il lavoro di questa tesi si concentra proprio su quest'ultimo aspetto, in particolare si andrà a descrivere un esperimento di va lutazione(offline) della componente di raccomandazione del sistema MAIVISTO. L'obiettivo è quello di confrontare le performance del sistema rispetto ad altri algoritmi descritti in letteratura.

Il documento è così organizzato:

- **Capitolo 1**: Introduzione generale sui recommender system, panoramica delle tecniche di raccomandazione più utilizzate e una spiegazione del problema del cold start.
- **Capitolo 2**: Descrizione della componente di raccomanda zione Seed Recommender, modelli utilizzati per l'implemen tazione e l'algoritmo di raccomandazione.
- **Capitolo 3**: Motivazioni della scelta di RIval come strumen to per l'implementazione dell'esperimento.
- **Capitolo 4**: Descrizione formale delle componenti dell'espe rimento.
- **Capitolo 5**: Descrizione implementativa dell'esperimento.

- **Capitolo 6**: Risultati.

Indice

Capitolo 1

Recommender System

1.1 Introduzione

I Recommender System sono sistemi che prendendo in input un insieme enorme di oggetti e una descrizione dei bisogni di un utente e restituiscono un insieme abbastanza piccolo di oggetti che soddisfino le necessità di quell'utente.

Il loro scopo è quindi quello di aiutare ad effettuare una scelta in un ambiente in cui il range di possibilità è enorme.

Negli ultimi anni la mole di informazioni presenti in rete è aumentata esponenzialmente; un utente che vuole effettuare una ricerca di un qualsiasi contenuto viene soverchiato dalla quantità di possibilità che gli vengono poste davanti (information overloading), questo porta ad una difficoltà nel processo decisionale da parte dell'utente che non riesce ad esaminare una per una tutte le scelte possibili.

Nasce quindi il bisogno di filtrare le informazioni in qualche maniera.

I Recommender rispondono a questa necessità scegliendo tra l'insieme di possibilità, i contenuti che sono *utili* ed *interessanti* per

uno specifico utente utilizzando informazioni generali sulle necessità di quell'utente.

I Recommender vengono utilizzati in una varietà di campi di versi, praticamente in ogni contesto in cui si avverta la necessità di fornire un consiglio; ad esempio siti di e commerce, motori di ricerca, siti di streaming video o musicale...

I Recommender forniscono vantaggi sia a chi li utilizza che a chi li mette a disposizione:

- L'utente (spesso cliente) riesce a trovare gli elementi del sito che gli interessano e che meglio soddisfano le sue necessità in poco tempo e con poco sforzo. Spesso scopre nuovi contenu ti che non conosceva e che non avrebbe scoperto autonoma mente, divertendosi a esaminare i contenuti raccomandati dal sistema;

- Il fornitore (spesso venditore) fornisce un servizio automatiz zato ma allo stesso tempo personalizzato per ogni utente che incrementa la fiducia e la fedeltà del cliente. E' dimostrato che un sistema di raccomandazione incrementa il tasso di ven dita (Amazon ad esempio ha dichiarato che una percentuale tra il 30 e il 70 percento delle vendite totali è dovuto alla qua lità delle raccomandazioni che offre). Si possono utilizzare le raccomandazioni anche per promuovere un item in particolare

o come metodo di persuasione all'aquisto. Si riescono anche a raccogliere molte informazioni sui clienti.

In breve, il lavoro di un Recommender è quello di fare un match tra utente e item, tentando di quantificare il grado di apprezzamento dell'utente per quell'item assegnandogli un qualche punteggio.

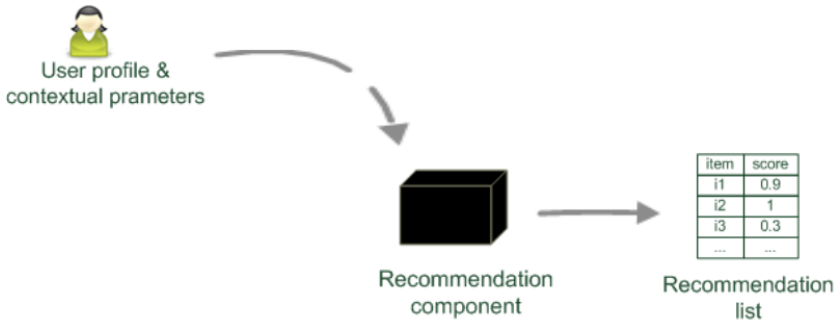
Nel campo dei Recommender System le informazioni in possesso di un sito sugli utenti vengono organizzate in una matrice user item, ogni cella rappresenta il grado di apprezzamento dell'utente rispetto a quell'item. Nella figura sottostante è riportato un esempio di una matrice del genere, in questo caso il grado di apprezzamento è espresso in notazione binaria «mi piace/non mi piace».

Movie ratings

	Amy	Jef	Mike	Chris	Ken
The Piano	-	-	+		+
Pulp Fiction	-	+	+	-	+
Clueless	+		-	+	-
Cliffhanger	-	-	+	-	+
Fargo	-	+	+	-	?

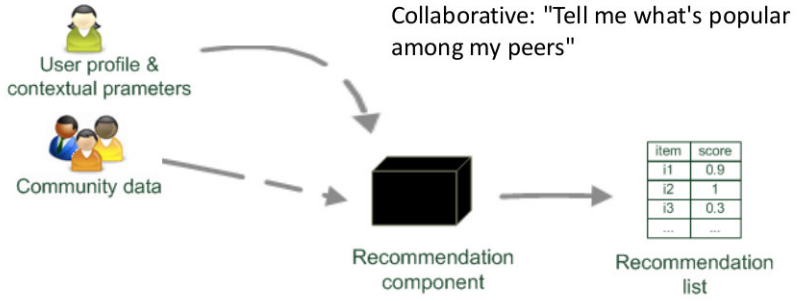
Matrici di questo tipo sono per loro natura molto sparse, ogni utente infatti esprime preferenze per un sottoinsieme molto limitato degli item presenti nel sistema. Il compito di un Recommender è

cercare di riempire le caselle vuote effettuando predizioni su item che non sono stati ancora votati.



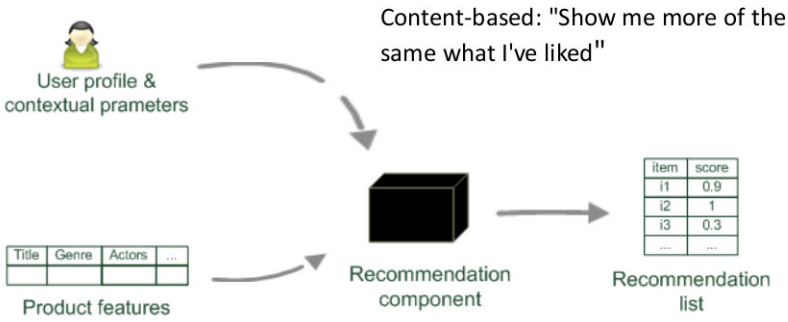
In base alle informazioni utilizzate per calcolare gli score si classificano diversi tipi di Recommender.

Un paradigma molto popolare è il *Collaborative Filtering*, ba sato su un database delle preferenze espresse da ogni utente, che viene utilizzato per trovare utenti con interessi simili (*neighborhood*) e mostrare all'utente oggetti interessanti che non ha ancora visto, ma che sono stati apprezzati da utenti con gusti affini.

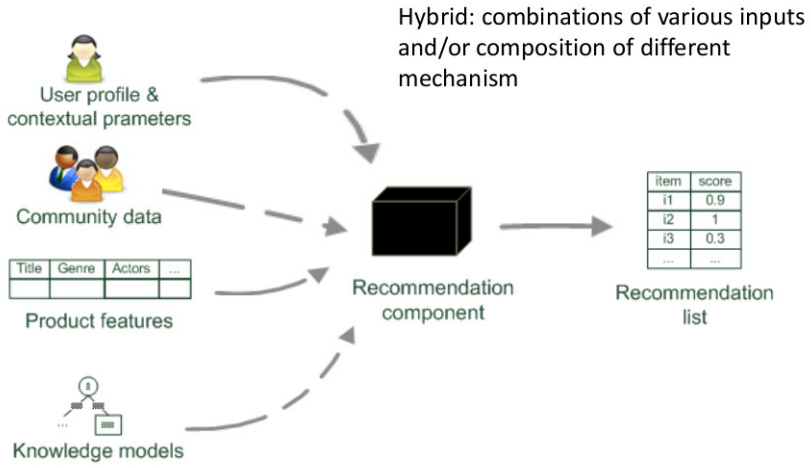


Un'altro paradigma molto utilizzato è il *Content Based*, in cui l'approccio è completamente differente. Si utilizzano infatti in formazioni sul contenuto dell'item per ricercare item simili e pro

porre all'utente una lista di item affini a quelli che ha già votato positivamente.



Vi sono infine sistemi più complessi che utilizzano un mix dei due paradigmi.



Nei prossimi paragrafi verranno discusse più approfonditamente queste tecniche.

1.2 Collaborative Filtering

Il *Collaborative Filtering* è un algoritmo per generare predizioni molto popolare. L'assunzione fondamentale dietro questo paradig

ma è che molto probabilmente un utente apprezzerà item che sono stati votati positivamente da utenti con gusti simili ai suoi, ma che non ha ancora visto. Vengono utilizzate quindi le informazioni sul le preferenze degli utenti per calcolare un indice di similarità tra user o tra item, il quale viene utilizzato in seguito per calcolare la predizione. Le preferenze possono essere su diverse scale (es. da 1 a 5 stelle, oppure binarie «like/dislike»), oltre che implicite (un acquisto ad esempio è un indice di apprezzamento molto forte) o esplicite.

In pratica un algoritmo di tipo *Collaborative* necessita di:

- Una lista di preferenze espresse dagli utenti;
- Un utente attivo per il quale vengono generate le raccoman dazioni;
- Una metrica per misurare le similarità tra utenti;
- Un metodo per selezionare il vicinato (*neighborhood);*
- Un metodo per generare gli score da attribuire agli item non ancora votati dall'utente;

Gli algoritmi di questo tipo si dividono in due tipi, i **memory-based** e i **model-based**. I primi utilizzano la matrice user item in modo diretto per fare le predizioni, e non scalano molto bene, an che perchè molti scenari di uso pratico hanno milioni di item e user,

e quindi la matrice è troppo grande per essere utilizzata in modo dinamico. I secondi invece lavorano offline sulla matrice, effettuan do un processo di model learning e generando quindi i modelli che sono poi utilizzati in tempo reale per effettuare le raccomandazioni. I modelli vengono aggiornati periodicamente.

1.2.1 Baseline Predictors

Prima di discutere gli algoritmi di tipo collaborative, vediamo al cuni metodi per calcolare le baseline[1], che sono predizioni non personalizzate molto utili per comparare gli algoritmi veri e propri, ma soprattutto per normalizzare la matrice, rendendo l'applicazione degli algoritmi che discuteremo in seguito più semplice.

Denotiamo la predizione della baseline per l'user u rispetto l'item i con $b_{u,i}$.

La baseline più semplice è la media di tutti i rating presenti nel sistema: $b_{u,i} = \mu$. Questa può essere migliorata utilizzando invece della media di tutti i rating, la media delle preferenze espresse solo da quell'utente o delle preferenze espresse solo su quell'item: $b_{u,i} = \overline{r_u}$ oppure $b_{u,i} = \overline{r_i}$.

In generale un predittore di baseline può essere espresso come $b_{u,i} = \mu + b_u + b_i$, dove b_u e b_i sono rispettivamente la user e la item baseline. Entrambe possono essere definite utilizzando l'offset medio come segue:

$$b_u = \frac{1}{|I_u|} \sum_{i \epsilon I_u} (r_{u,i} - \mu)$$

dove I_uè l'insieme degli item che sono stati votati dall'utente *u;*

$$b_i = \frac{1}{|U_i|} \sum_{u \epsilon U_i} (r_{u,i} - b_u - \mu)$$

dove U_i è l'insieme degli utenti che hanno votato l'item *i.*

Queste formule possono essere regolarizzate ulteriormente in corporando un «damping term»[2]:

$$b_u = \frac{1}{|I_u| + \beta_u} \sum_{i \epsilon I_u} (r_{u,i} - \mu)_{i \epsilon I_u}$$

$$b_i = \frac{1}{|U_i| + \beta_i} \sum_{u \epsilon U_i} (r_{u,i} - b_u - \mu)$$

Questi termini servono a rendere la baseline più vicina alla me dia globale se l'utente o l'item ha poche preferenze, Simon Funk [2] spiega che un buon damping term è 25.

Le baseline sono molto utili in quanto sottraendole dalla matri ce dei rating si può ottenere una matrice normalizzata che lascia all'algoritmo solo il compito di catturare le interazioni tra item e

user. Inoltre i valori non conosciuti della matrice diventano 0 invece che sconosciuti, rendendo i calcoli più semplici.

1.2.2 User-User Collaborative Filtering

L'angoritmo *user user collaborative filtering* detto anche *k NN col laborative filtering*, è un algoritmo di tipo memory based, perchè è basato sul calcolo di un sottoinsieme di utenti simili all'utente at tivo (*neighborhood*) dalla quale calcolare le predizioni, e quindi ha bisogno di leggere continuamente dalla matrice dei rating.

C'è bisogno quindi di un modo per calcolare le similiratità tra utenti, cioè si deve definire una funzione $s : U \times U \rightarrow \mathfrak{N}$ [1]che associa ad ogni coppia di utenti un valore reale che rappresenta il grado di similarità tra i due utenti.

Vi sono varie definizioni possibili di questa funzione, vediamo le due più comuni:

- **Coefficiente di correlazione di Pearson:**

$$s(u,v) = \frac{\sum_{i \in I_u \bigcap I_v}(r_{u,i} - \overline{r_u})(r_{v,i} - \overline{r_v})}{\sqrt{\sum_{i \in I_u \bigcap I_v}(r_{u,i} - \overline{r_u})^2}\sqrt{\sum_{i \in I_u \bigcap I_v}(r_{v,i} - \overline{r_v})^2}}$$

  Questa misura prende in considerazione aspetti del compor tamento dell'utente utilizzando la media personale dei rating di ogni utente. Si riesce a normalizzare le differenti predispo sizioni degli utenti a votare in modo eccessivamente alto o

eccessivamente basso gli item. Non è molto efficace nel caso in cui i rating in comune tra i due utenti siano pochi;

- **Similarità del coseno:**

$$s(u,v) = \frac{\sum_i r_{u,i} r_{v,i}}{\sqrt{\sum_i r_{u,i}^2}\sqrt{\sum_i r_{v,i}^2}}$$

In questo approccio l'utente è visto come un vettore in uno spazio $|I|$ dimensionale le cui coordinate sono le preferenze che ha espresso, viene calcolata la «vicinanza», in questo spa zio, dei due vettori utente utilizzando l'angolo fra i due. I rating sconosciuti sono posti uguali a zero. Se i due uten ti hanno votato lo stesso set di item allora la similarità del coseno è equivalente al coefficiente di Pearson.

Tramite la funzione s si calcola il vicinato $N \subseteq U$ dell'utente attivo. A questo punto le predizioni vengono calcolate utilizzando in genere la media pesata dei rating degli utenti in N sull'oggetto i:

$$p_{u,i} = \overline{r_u} + \frac{\sum_{u' \in N} s(u,u')(r_{u',i} - \overline{r_{u'}})}{\sum_{u' \in N} |s(u,u')|}$$

A questo punto, l'unico fattore da decidere è la cardinalità dell'insime N, cioè decidere quanti «vicini» considerare.

In alcuni sistemi il vicinato è scelto includendo tutti gli utenti tranne quello attivo, ma è stato discusso in [3] come limitare l'in

sieme dei vicini possa portare ad un miglioramento nell'accuratezza delle predizioni in quanto gli utenti con un valore di similarità molto basso introducono più rumore che altro, nel calcolo della predizione. Per scegliere quanti utenti prendere in considerazione si può defini re a priori una cardinalità per N, oppure definire un limite inferiore al coefficiente di similarità per gli utenti da inserire nell'insieme.

1.2.3 Item-item Collaborative Filtering

L'idea principale [4]dietro questo tipo di algoritmo è di analizzare la rappresentazione matriciale per identificare relazioni tra item che verranno poi utilizzate per effettuare le predizioni per una data coppia user item.

L'assioma è che un utente sarà interessato ad item simili a quelli che ha già aquistato, o comunque apprezzato in passato, e non vorrà vedere item simili a quelli per i quali ha espresso un giudizio negativo.

Questa metodologia non necessita il calcolo di un vicinato per ogni predizione e quindi tende a effettuare raccomandazioni in ma niera molto più veloce. L'algoritmo è perciò di tipo *model based*, in quanto le similarità tra item possono essere calcolate offline e salvate per il successivo utilizzo in real time.

A differenza dell'approccio *user based*, in questo caso l'algorit mo cerca fra gli item che l'utente attivo ha già votato e seleziona

i k item più simili all'item attivo. Per fare ciò si calcola un coefficiente di similarità s come nel caso precedente, questa volta però il coefficiente si riferisce ad una coppia item item piuttosto che ad una coppia user item.

Il primo passo è quindi quello di calcolare la similarità tra due item, ci sono diverse metriche che possono essere utilizzate, ne vediamo due molto popolari:

- **Coefficiente di correlazione di Pearson:**

$$s(i,j) = \frac{\sum_{u \in U_i \bigcap U_j}(r_{u,i} - \overline{r_i})(r_{u,j} - \overline{r_j})}{\sqrt{\sum_{u \in U_i \bigcap U_j}(r_{u,i} - \overline{r_i})^2}\sqrt{\sum_{u \in U_i \bigcap U_j}(r_{u,j} - \overline{r_j})^2}}$$

  Stesso coefficiente usato prima, in questo caso però bisogna trovare gli utenti che hanno co votato i due item e in base a questi calcolare quanto i voti siano simili.

- **Similarità del coseno**:

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

  Che non è altro che la stessa misura che abbiamo considerato prima per la similarità tra utenti, in questo caso però sono gli item a essere rappresentati come vettori in uno spazio $|U|$ dimensionale; si considera la distanza del coseno tra due vettori per computare la similarità. Il simbolo $\cdot$ sta per pro

dotto vettoriale. Un'altra versione di questa metrica prende in considerazione le differenze tra le scale di rating che diversi utenti utilizzano (ad esempio un utente potrebbe votare con il massimo qualsiasi item gli interessi, mentre un'altro potrebbe attenersi a votazioni più basse anche per item che gli piacciono molto):

$$s(i,j) = \frac{\sum_{u \in U}(r_{u,i} - \overline{r_i})(r_{u,j} - \overline{r_j})}{\sqrt{\sum_{u \in U}(r_{u,i} - \overline{r_i})^2}\sqrt{\sum_{u \in U}(r_{u,j} - \overline{r_j})^2}}$$

Il passo successivo è il calcolo della predizione vera e propria per la generazione della lista di raccomandazione. Anche in questo caso si utilizza la media pesata, ma N non è il vicinato, bensì la lista di item più simili all'item i:

$$p_{u,i} = \overline{r_u} + \frac{\sum_{i' \in N} s(i,i')(r_{u,i} - \overline{r_u})}{\sum_{i' \in N} |s(i,i')|}$$

I siti di e commerce più grandi, operano su una scala che stressa in modo significativo le implementazioni di algoritmi *neighborhood based*, in particolare il calcolo delle similarità per la generazione del vicinato è il collo di bottiglia delle performance che rende l'uso di ta li algoritmi inpraticabile in un contesto real time. Con l'approccio appena descritto si riesce a isolare il processo di generazione del l'insieme N da quello del calcolo delle predizioni. In questo modo piuttosto che calcolare la similarità al momento si cerca la similarità

in un modello precomputato riducendo la complessità dell'algoritmo a $\mathcal{O}(n^2)$ dove n è il numero di item.

1.3 Content-based Filtering

Un sistema[5] di tipo *content based* non fa altro che analizzare un insieme di documenti o descrittori di item già votati dall'utente per costruire un modello degli interessi di quell'utente. Il processo di raccomandazione consiste nel trovare corrispondenze tra questo modello o profilo che si è costruito, con oggetti che l'utente deve ancora vedere.

Questa strategia quindi, necessita di tecniche per rappresentare gli item, produrre il profilo utente e compararli.

Il processo di raccomandazione è eseguito in tre passi:

- **Analisi del contenuto**: Molto spesso le informazioni ri guardanti un item sono disponibili in forma non strutturata, ad esempio una descrizione testuale. In questa fase avviene una trasformazione del testo non strutturato in un modello adeguato alle successive fasi del processo.

- **Profile learner**: A questo punto le informazioni vengono generalizzate, in modo da riuscire a creare un profilo utente. In genere per questo scopo si utilizzano tecniche di machine learning per inferire gli interessi dell'utente basandosi su item che ha apprezzato o che ha disprezzato.

- **Filtering component**: Infine si confronta il profilo utente con gli item da raccomandare per generare la lista di racco mandazione utilizzando metriche di similarità.

1.3.1 Rappresentazione degli Item

Gli item possono essere rappresentati da un insieme di *feature* o *attributi* o *proprietà*. In molti sistemi di raccomandazione *content based* gli item sono una serie di parole chiave estratte da varie fonti, ad esempio la descrizione di un prodotto. Questa rappresentazione permetti di applicare gli algoritmi di Machine Learning per creare un modello degli interessi dell'utente.

Una metodologia molto popolare è quella del **Vector Space Model (VSM)** con pesatura **TF-IDF**.

Si tratta di una rappresentazione spaziale di un documento di testo. Ogni documento infatti viene rappresentato come un vettore in uno spazio n dimensionale in cui ad ogni dimensione corrisponde un termine del vocabolario complessivo del set di documenti presi in considerazione.

Formalmente, ogni documento è rappresentato come un vettore di pesi dei termini, cioè il grado di importanza di quel termine in quel documento.

Sia $D = \{d_1, d_2, ..., d_N\}$ un insieme di documenti, possiamo ricavare l'insieme $T = \{t_1, t_2, ..., t_n\}$ dei termini che compaiono

nell'insieme D utilizzando tecniche dell'information retrieval come la tokenizzazione, la rimozione delle stop word e lo stemming. Ogni documento è rappresentato da $d_j = \{w_{1j}, w_{2j}, ..., d_{nj}\}$, dove w_{ij} è il peso del termine t_i nel documento d_j.

Il calcolo del peso dei termini è effettuato tramite la **TF-IDF (Term Frequency-Inverse Document Frequency)**:

$$TF - IDF(t_k, d_j) = TF(t_k, d_j) \cdot log\frac{N}{n_k}$$

dove

$$TF(t_k, d_j) = \frac{f_{k,j}}{max_z f_{z,j}}$$

dove $f_{z,j}$ è la frequenza con cui compaiono i termini del documento nel documento stesso.

Queste equazioni sono di solito normalizzate per ottenere il peso finale del termine:

$$w_{k,j} = \frac{TF - IDF(t_k, d_j)}{\sqrt{\sum_{s=1}^{|T|} TF - IDF(t_s, d_j)^2}}$$

La similarità tra due item infine, è ancora una volta calcolata con la similarità del coseno nel seguente modo:

$$sim(d_i, d_j) = \frac{\sum_k w_{k,i} \cdot w_{k,j}}{\sqrt{\sum_k w_{k,i}^2} \cdot \sqrt{\sum_k w_{k,j}^2}}$$

1.3.2 Modellazione del profilo utente

In questa fase vengono utilizzati metodi di machine learning per costruire un categorizzatore di testi binario: il set di categorie è $C = \{c_+, c_-\}$, dove c_+ è la classe di item positivi per l'utente e c_-è la classe di item negativi per l'utente.

Andiamo ad analizzare due importanti algoritmi di machine learning utilizzati nel contesto dei recommender system.

1.3.2.1 Metodi probabilistici e Naïve Bayes

Naïve Bayes è un approccio probabilistico che appartiene alla classe dei classificatori Bayesiani. Si tratta di generare un modello proba bilistico sulla base dei dati osservati. Il modello stima la probabilità a posteriori $P(c|d)$ che un documento d appartenga alla classe c. Questa stima si basa sulla probabilità a priori $P(c)$ di osservare un documento nella classe c, la probabilità $P(d|c)$ di osservare un documento d data la classe c e la probabilità $P(d)$ di osservare l'istanza d.

Con queste probabilità è possibile applicare il Teorema di Bayes nel seguente modo:

$$P(c|d) = \frac{P(c)P(d|c)}{P(d)}$$

Per classificare il documento d si sceglie la classe con la più alta probabilità:

$$c = argmax_{c,j} \frac{P(c_j)P(d|c_j)}{P(d)}$$

La parte problematica in questo tipo di approccio è il calcolo di $P(d|c)$ poichè difficilmente si riesce ad incontrare lo stesso documento più di una volta nel training set, quindi la quantità di informazioni spesso non è sufficiente a costruire un modello probabilistico abbastanza efficace. Per sopperire a questo problema si semplifica il modello andando a considerare indipendenti le probabilità delle parole all'interno del documento data la classe. In questo modo si può stimare la probabilità parola per parola invece che dell'intero documento. Un modello che incorpora questa assunzione e fornisce un metodo di stima della probabilità è il modello eventi *multinomial*, che tratta un documento come un vettore di valori nel vocabolario in cui ogni elemento rappresenta la presenza o meno del termine nel documento. Questo vettore viene poi utilizzato per stimare la probabilità nel seguente modo:

$$P(c_j|d_j) = P(c_j) \prod_{w \epsilon V_{d_i}} P(t_k|c_j)^{N_{(d_i,t_k)}}$$

dove $N_{(d_i,t_k)}$ è definito come il numero di volte il termine $t_k$appare nel documento d_i.

1.3.2.2 Relevance Feedback e Algoritmo di Rocchio

La *Relevance Feedback* è una tecnica dell'Information Retrieval in cui l'utente restituisce al sistema un feedback sulla rilevanza delle informazioni ottenute. L'*algoritmo di Rocchio[6]* è l'adattamento di questa tecnica al contesto della text categorization.

Questo algoritmo rappresenta i documenti come vettori in modo tale che documenti simili abbiano rappresentazioni vettoriali simili. Ogni elemento di questo vettore è tipicamente una parola del do cumento e il peso di ogni elemento è calcolato usando la TF IDF. L'apprendimento è realizzato combinando questi vettori in un vet tore prototipo per ogni classe da classificare. Per classificare un documento quindi viene calcolata la similarità tra la rappresenta zione vettoriale del documento ed il vettore prototipo delle classi, il documento verrà assegnato alla classe più vicina nello spazio dei vettori.

Più formalmente il metodo di Rocchio costruisce un classifica

tore $\overrightarrow{c_i} =< \omega_{1i}, ..., \omega_{|T|i} >$ per la categoria c_i tramite la formula:

$$\omega_{ki} = \beta \cdot \sum_{\{d_j \epsilon POS_i\}} \frac{\omega_{kj}}{|POS_i|} - \gamma \cdot \sum_{\{d_j \epsilon NEG_i\}} \frac{\omega_{kj}}{|NEG_i|}$$

dove ω_{kj} è il peso TF IDF del termine k nel documento j, POS_i e NEG_i sono gli esempi positivi e negativi del training set per la classe c_j e β e γ sono parametri per gestire il peso nella metrica degli esempi positivi e negativi.

1.4 Sistemi ibridi

Sia gli approcci collaborative che content based hanno diversi svan taggi.

Gli algoritmi[4] di tipo *Collaborative* soffrono di due principali punti deboli:

- **Sparsità**: In un contesto pratico, un recommender per un sito di e commerce lavora con un insieme di item enorme; un utente medio sicuramente esprimerà preferenze per una percentuale bassissima di tutti gli item presenti nel sistema e quindi l'algoritmo potrebbe non essere in grado di genera re una lista di raccomandazione per un determinato utente poichè non è detto che riesca a trovare un vicinato.

- **Scalabilità**: Le computazioni necessarie a far girare un al goritmo user based crescono in maniera esponenziale sia con

il numero di utenti che con il numero di item, in un contesto con milioni di utenti e item, un recommender soffre di seri problemi di scalabilità.

Gli algoritmi[5] di tipo *Content based* invece soffrono:

- **Limitazioni nell'analisi del contenuto**: cioè il limite nel le feature da poter associare ad un item, molte volte infatti vi è la necessita di conoscere il dominio contestuale dell'i tem (ad esempio se l'item è un film bisogna trovare feature come il regista o il cast). Qualsiasi algoritmo content based infatti è inservibile se non vi sono abbastanza informazioni per discriminare gli item che l'utente apprezza da quelli che non vuole vedere, e anche così vi sono molti attributi che non vengono considerati ma che sono molto influenzanti nel parere dell'utente.

- **Over-specialization**: Detta anche *serendipity* è l'attitudine del recommender a produrre raccomandazioni sempre uguali, con un limitato coefficiente di *novelty.*

- **Cold-start user**: Un utente appena entrato nel sistema non ha abbastanza informazioni perchè il recommender possa creare il modello del suo profilo e quindi le raccomandazioni sono inaffidabili.

Quindi entrambi gli approcci hanno le loro limitazioni. Si è pensato allora di creare recommender che integrino entrambe le tecniche in modo da affrontare le diverse situazioni in modi diversi. Questi sistemi vengono chiamati *hybrid system*, intendendo sistemi che applicano strategie di ibridazione.

Si può pensare di utilizzare algoritmi di tipo *content based* e *collaborative* separatamente e poi combinare i risultati, oppure integrare i risultati del content base per misurare la similarità tra item o user in un algoritmo collaborative.

1.5 Il problema del «cold-start»

Il problema del «cold start», come accennato prima, si riscontra ogni volta che un utente entra nel sistema e non ha nessuna preferenza da cui generare la lista di raccomandazione. Anche nel caso di un nuovo item il problema esiste, anche se solo nei sistemi di tipo collaborative. Infatti un sistema di tipo content based non avrà difficoltà a estrarre le feature di un nuovo item e raccomandarlo quando opportuno. Nel caso del collaborative invece il nuovo item non rientrerà mai in una lista di raccomandazione poichè non è stato sicuramente votato da nessuno e non rientrerà mai in un vicinato.

Il problema del cold start user invece affligge entrambi i paradigmi, poichè anche nel caso del content based il fatto di non

avere l'insieme degli item votati dall'utente porta all'impossibilità di calcolare le similarità, e quindi di generare una lista di raccomandazione.

Il problema si estende anche ad utenti con pochi voti in quanto non è possibile costruire un profilo dettagliato e di conseguenza le raccomandazioni non sono affidabili.

Sono state proposte diverse soluzioni in letteratura[8, 9, 7], ma tutte hanno l'obbiettivo di carpire le preferenze del nuovo utente in modo da riuscire a fornire raccomandazioni personalizzate fin da subito. Possiamo distingure due gruppi molto generali:

- **Metodi impliciti**: si osserva il comportamento dell'utente, e da azioni che compie durante la navigazione si prendono le informazioni che potrebbero essere utili al sistema. (per esempio si possono utilizzare la cronologia, le ricerche effettuate o feedback da altri siti).

- **Metodi espliciti**: l'utente fornisce direttamente le informazioni tramite un questionario o lo si costringe a votare alcuni item predefiniti al momento della registrazione al sistema.

Un metodo esplicito è l'*Active Learning* descritto in [9] con la quale si selezionano un insieme di item da far votare all'utente nella prima fase di accesso al sistema. La scelta di questo insieme è la componente critica di questa tecnica, è necessario infatti individuare il più piccolo insieme di elementi che siano altamente informativi.

Chiedere all'utente di votare degli elementi ha un costo, l'utente infatti dovrà compiere uno sforzo cognitivo in una fase di utilizzo del sistema che non gli interessa. L'utente medio infatti non ama ri spondere a molte domanda o fornire troppi feedback espliciti, anche se si aspetta performance molto alte non appena inizia ad utilizzare il sistema. Per questo motivo bisogna cercare di ottenere il maggior numero di informazioni utili con il minor sforzo possibile da parte dell'utente.

Ci sono infine due diversi metodi per generare l'insieme di item da presentare in fase di registrazione al nuovo utente:

- **Metodi adattivi**: in cui gli item vengono selezionati in real time durante l'inserimento del feedback da parte dell'utente, in pratica dopo ogni feedback dato l'insieme viene ricalcolato tenendo presente la preferenza appena ricevuta;

- **Metodi non adattivi**: in cui lo stesso insieme viene mostra to a tutti i nuovi utenti. Il vantaggio di questo approccio è che l'insieme viene calcolato una volta sola, magari cercando quegli item che sono più informativi.

Capitolo 2

Seed Recommender

L'algoritmo oggetto della valutazione di questa tesi viene denominato SeedRecommender, si tratta di un approccio Item based che utilizza varie matrici di similarità tra item per generare le predizioni.

2.1 Introduzione

Come detto nella prefazione, l'algoritmo considerato si colloca nella costruzione della componente di raccomandazione del progetto MAIVISTO.

La componente deve essere in grado di raccomandare item in una varietà di situazioni. In particolare il problema del cold start, sia per quanto riguarda gli utenti che gli item, non deve essere un ostacolo nella generazione delle predizioni.

Come visto nel capitolo precedente, gli approcci di tipo collaborative soffrono parecchio il cold start, in quanto devono considerare un vicinato calcolato basandosi su preferenze già contenute nel sistema.

Si è pensato perciò di realizzare il sistema utilizzando una solu zione ibrida che comprenda sia approcci collaborative che content based. In questo modo si possono sfruttare le peculiarità dei due diversi approcci in situazioni in cui rendono meglio, e allo stesso tempo sopperire ai punti deboli di questi paradigmi.

Ad esempio nel caso in cui il sistema contenga poche preferenze, si può utilizzare la parte content based per suggerire item simili ba sandosi solo del contenuto; oppure nel caso in cui le descrizioni degli item siano povere o assenti, l'approccio collaborative può restituire risultati migliori.

In poche parole si è cercato di rendere il sistema quanto più ver satile possibile in una fase di sviluppo preliminare in cui il contesto in cui lavorerà il recommender non è ancora chiaro.

Il problema del cold start è quindi risolto utilizzando i contenuti degli item come dato per calcolare le similarità, però nel caso in cui l'utente non abbia fornito nessun rating, il problema persiste in quanto non si hanno item da cui partire per generare la lista di raccomandazione. La soluzione proposta consiste in un **seed item-set standard**, che consiste in un insieme di item standard da utilizzare come trigger per il calcolo delle similarità.

Il seed item set standard si compone di almeno quattro item:

- L'item più popolare, cioè l'item con il maggior numero di rating e/o visualizzazioni;

- L'item più popolare rispetto ad un periodo di tempo, cioè quell'item che è risultato più popolare nella scorsa settimana o mese;
- L'item inserito per ultimo nella piattaforma;
- L'item che ha ricevuto un rating positivo più di recente nel sistema;

Questo insieme ovviamente è stato scelto con criteri euristici e può essere quindi soggetto a cambiamenti tramite l'utilizzo di altri criteri per la scelta degli item.

Si possono anche aggiungere i cosidetti seed esterni, cioè preferenze che l'utente ha espresso al di fuori della piattaforma, ad esempio ricerche sul web o "mi piace" da Facebook o Twitter.

Il seed item set garantisce la possibilità di generare raccomandazioni anche quando l'utente è completamente anonimo al sistema.

La fase successiva è il calcolo delle similarità tra item. A questo scopo vengono utilizzate diverse matrici di similarità tra item a cui vengono assegnati pesi diversi nel calcolo finale. L'utilizzo di più matrici è necessario per andare a considerare diverse misure di similarità, una per ogni paradigma diverso che il sistema ibrido incorpora.

Infine il recommender vero e proprio utilizza due algoritmi diversi per distinguere il caso del cold start user da quello del non

cold start user.

I successivi paragrafi spiegano in dettaglio come sono calcolate le matrici e lo pseudocodice dell'algoritmo.

2.2 Matrici di similarità

Possiamo pensare alle diverse matrici di similarità come diversi li velli di descrizione di un item. Infatti ognuna di queste matrici calcola una similarità diversa basandosi su diverse informazioni del l'item e quindi riferendosi a caratteristiche diverse dell'item, come i metadati, oppure la descrizione, o ancora la co occorrenza tra user per quell'item.

In questo modo il concetto di similarità è esteso a più livelli, a ognuno dei quali si può assegnare un peso diverso, compreso tra 0 e 1, in base alla caratteristica particolare che si crede sia più significativa nel calcolo della similarità in quel contesto.

Nell'implementazione che si valuterà in questa tesi, le matrici considerate sono tre, basate su: similarità del coseno, co occorrenza tra item e textual similarity sulle descrizioni degli item.

2.2.1 Similarità del coseno

Si considera la matrice user item in cui le celle sono il rating di ogni utente per ogni item. Ogni item i è visto come un vettore che consiste nella colonna i della matrice, quindi composto dai rating

di tutti gli utenti per quell'item. La similarità tra due item i e j è il coseno dell'angolo dei due vettori che caratterizzano gli item. La formula è la seguente:

$$sim_{i,j} = \frac{\sum_{k=1}^{n}(r_{k,i} \cdot r_{k,j})}{\sqrt{\sum_{k=1}^{n} r_{k,i}^2}\sqrt{\sum_{k=1}^{n} r_{k,j}^2}}$$

Si crea in questo modo una matrice simmetrica item item dove nella cella $r_{i,j}$ vi sarà la similarità del coseno tra gli item i e j, la simmetria è data dal fatto che $sim_{i,j} = sim_{j,i}$. Questi valori sono compresi nel range $[-1, 1]$ dove 1 significa massima similarità e 1 nessuna similarità.

2.2.2 Co-occorrenza

La co occorrenza tra due item i e j non è altro che il numero di utenti che hanno votato sia i che j. La matrice item item costruita in questo modo contiene tutti i valori di co occorrenza per tutte le coppie di item, questi valori vengono normalizzati in un range $[0, 1]$.

2.2.3 Textual Similarity

La matrice del content è costruita tramite un modello computa zionale che implementa il concetto di *Distributional Semantics*, la *Random Indexing.*

Per *Distributional Semantics* si intende quell'area di ricerca che si occupa di calcolare la similarità semantica tra item linguistici (parole, frasi o interi testi) basandosi sulle proprietà distributive degli stessi in un dataset linguistico molto ampio. L'idea di fondo è la *Distributional Hypothesis* del linguista J.R. Firth, secondo la quale, parole che occorrono spesso negli stessi contesti hanno molto probabilmente lo stesso significato. Ciò suggerisce di utilizzare le occorrenze di apparizione in un contesto di un termine per definire uno spazio vettoriale in cui definire i termini stessi e calcolarne la vicinanza. In questo modo si può ottenere un modello delle parole che viene chiamato *Semantic Space.*

In base a come si definisce il contesto si diversificano diversi approcci alla *Distributional Semantics.* Modelli che prendono in considerazione una sola definizione di contesto sono detti model li semplici, mentre quelli che ne utilizzano più di una alla volta sono detti strutturati. In particolare modelli che utilizzano item linguistici come contesto sono denominati *Word Space Model.*

Un WSM può essere definito come una quadrupla $< A, B, S, M >$[10] dove:

- B è l'insieme delle basi che determinano la dimensionalità del lo spazio semantico e come ogni dimensione va interpretata, spesso è un insieme di parole, ma può anche trattarsi di frasi o interi testi;

- A è la funzione che trasforma la co occorrenza tra parole e basi in modo tale da poter rappresentare ogni termine come un vettore del tipo $[A(b1,t), A(b2,t), ..., A(bD,t)]$, un esempio può essere la funzione identità;

- S è la similarità tra vettori che definisce la distanza tra i termini nello spazio semantico;

- M è una funzione che trasforma uno spazio semantico in un altro, molto spesso utilizzata per ridurre la dimensionalità dello spazio.

Il punto centrale è quindi il calcolo di una matrice che contiene tutti i vettori che rappresentano i termini del corpus.

Ogni tipologia di WSM definisce queste quattro componenti in modo diverso.

La costruzione dello spazio semantico può essere problematica, infatti la matrice delle co occorrenze termine contesto può essere molto grande e quindi difficile da calcolare.

Il *Random Indexing* è una possibile soluzione a questo proble ma. Si tratta infatti di una tecnica di costruzione incrementale dello spazio semantico in cui i vettori che rappresentano i termini sono accumulati incrementalmente. Alla base del *Random Inde xing* vi è l'assunzione che vettori altamente dimensionali sono molto probabilmente ortogonali tra loro.

Data una matrice $n \times m$ A e una matrice composta da k vettori random R di dimensione $m \times k$, si definisce una nuova matrice B di dimensione $n \times k$ tale che:

$$B^{n,k} = A^{n,m} \cdot R^{m,k}$$

con $k \ll m$.

B ha la proprietà di conservare la distanza tra i punti dello spazio di origine A.

Per creare il nuovo spazio B, considerando un contesto w co stituito da un insieme di termini, il *Random Indexing* esegue due passi:

- Assegna un vettore contesto ad ogni termine, questo vettore è sparso, altamente dimensionale e ternario;
- Il vettore semantico di un termine è calcolato sommando i vettori contesto dei termini contenuti nella finestra di termini w, che co occorrono con il termine.

SeedRecommender utilizza il *Random Indexing* per creare lo spazio semantico e crea un vettore semantico per ogni testo sommando i vettori semantici dei termini che compongono il testo. La similarità tra due testi, e quindi tra due item, è calcolata tramite la similarità del coseno.

2.3 Algoritmo

L'algoritmo utilizzato è diverso in base all profile size dell'utente in considerazione.

Utenti con un numero di rating compreso tra zero e diciannove sono considerati in situazione di cold start, tutti gli altri sono in situazione di non cold start.

Entrambi gli algoritmi prendono in input:

- Un identificatore univoco dell'utente,
- Il numero di item da suggerire;
- Un insieme opzionale di seed esterni;
- Un booleano che indica se usare o meno l'insieme standard dei seed che abbiamo discusso prima.

E restituiscono una lista ordinata di item da suggerire all'utente.

2.3.1 Algoritmo per utenti in situazione di cold start

Nel caso in cui l'utente sia in situazione di cold start, l'algoritmo calcola un insieme iniziale di item da utilizzare come punto di partenza per trovare un vicinato nelle varie matrici di similarità.

Chiamiamo questo insieme *SEEDS*, inizialmente vuoto. Come prima cosa si controlla se l'insieme dei seed esterni è vuoto, in caso negativo si aggiungono tutti a *SEEDS* e gli si assegna uno score.

Dopodichè si legge il booleano per capire se bisogna considerare anche i seed standard. In caso positivo si estraggono i seed standard dal catalogo e si aggiungono a *SEEDS*, anche per questi si calcola uno score basato sulla media dei rating dell'utente.

Infine se l'utente ha espresso almeno una preferenza, si conside rano tutti gli item votati positivamente dall'utente e si aggiungono a *SEEDS* con uno score pari al voto espresso.

Per item votato positivamente si intende un item che ha uno score maggiore o uguale alla media voti dell'utente.

Per ognuna di queste operazioni si eliminano gli item aggiunti a *SEEDS* dalla lista di item raccomandabili.

Lo pseudocodice di questa fase è il seguente:

```
rec list coldStartAlgorithm (user, n, seedItemSet, activateStandardSeed)
        rec list = {}
        SEEDS = {}
        userHistory = rating dell utente user
        REC = tutti gli item (catalogo) registrati nella piattaforma


        if seedItemSet non è vuoto then
                for all seed in seedItemSet do
                        SeedScore = assegna un punteggio a seed
                end for
        SEEDS = SEEDS + seedItemSet
        REC = REC / seedItemSet
        end if

        if activateStandardSeed = true then
                standardSeed = ritrova i seed standard nel catalogo
                for all seed in standardSeed do
                        if seed non è in seedItemSet and seed non è in userHistory then
                                SeedScore = assegna un punteggio a seed
                                rec list = < seed, SeedScore , null >
                        end if
                end for
                SEEDS = SEEDS + standardSeed
                REC = REC / seedItemSet
        end if
```

```
if userHistory non è vuota then
        POS = ritrova i voti positivi dell utente user
        SEEDS = SEEDS + POS
        REC = REC / userHistory
end if
```

La fase successiva consiste nel riempire la lista di raccomanda zione con n item.

Per fare ciò l'algoritmo seleziona per ogni item in *SEEDS* l'i tem più simile da ogni matrice di similarità. A questo item viene assegnato un punteggio pari al prodotto tra lo score del seed e la similarità tra il seed e l'item.

Questo processo viene iterato finchè la lista di raccomandazione non contiene gli n item necessari.

Ad ogni iterazione k viene selezionato il k esimo item più simile dalle matrici.

```
k = 1

while rec list non contiene n item distinti do
        for all seed in SEEDS do
                SeedScore = score di seed in SEEDS (rating positivo o score assegnato)
                for all mat in similarityMatrices do
                        item = k-esimo item più simile a seed nella matrice mat
                        if REC contiene item then
                                sim seed,item = similarità tra gli item seed e item in mat
                                score item = score seed * sim tra seed e item
                                rec list = < item, score item , mat >
                                REC = REC / item
                        end if
                end for
        end for
k = k+1
end while

return rank(rec list, n)
```

Dopo questo passo, potrebbe accadere che un item sia stato raccomandato da più di una matrice.

In questo caso le probabilità che quell'item sia rilevante è molto alta, quindi nella fase di ordinamento degli item nella lista questo ti po di item hanno una posizione migliore. Per fare ciò l'ordinamento si basa sul numero di matrici che hanno raccomandato l'item come primo criterio e sullo score come secondo criterio.

Lo score finale degli item raccomandati è calcoltato in questo modo:

$$\frac{\sum_{i \in rec\ \ list} score_i * w_{m_i}}{\sum_{i \in rec\ \ list} w_{m_i}}$$

dove w_{m_i} è il peso della matrice che ha raccomandato l'item. Per i seed esterni e standard la w è uguale a uno.

2.3.2 Algoritmo per utenti in situazione di non cold start

Nel caso di utenti con un numero di rating superiore a venti, si utilizza un algoritmo diverso.

In questa situazione si considera comunque un insieme *SEEDS* utilizzato per calcolare il vicinato. Questa volta però l'insieme dei seed standard non è considerato, l'insieme è infatti formato dagli item votati dall'utente (sia positivamente che negativamente) più l'eventuale insieme di seed esterni.

Dopodichè si considera l'insieme di tutti gli item raccomandabili presenti nel catalogo, e per ognuno di questi si calcola un vicinato di dimensione V. A questo punto si considera l'intersezione del vicinato con l'insieme *SEEDS*, che chiameremo *NEIGHS*, e per ognuno degli oggetti presenti in questo insieme si calcola uno score come il prodotto della similarità tra l'item considerato, l'elemento di *NEIGHS* e il peso della matrice da cui si è preso. Alla fine lo score dell'item sarà dato dalla media pesata dei singoli score degli elementi di *NEIGHS* in base ai pesi delle matrici:

$$score_{item} = \sum_{neigh \in NEIGHS_m} \frac{sim_{item,neigh} * score_{neigh} * w_m}{\sum_{neigh \in NEIGHS_m} w_m}$$

L'item viene quindi inserito nella lista di raccomandazione con il relativo score.

Infine la lista di raccomandazione viene ordinata in base allo

score e vengono restituiti i primi n elementi.

```
rec list noColdStartAlgorithm (user, n, seedItemSet)

rec list = {}
SEEDS = {}
userHistory = rating dell utente user
REC = tutti gli item (catalogo) registrati nella piattaforma


if seedItemSet non è vuoto then
        for all seed in seedItemSet do
                score seed = assegna un punteggio a seed
        end for

        SEEDS = SEEDS + seedItemSet
end if

SEEDS = SEEDS + userHistory
REC = REC / SEEDS

for all item in REC do
        itemWeight = 0
        itemScore = 0

        for all mat in similarityMatrices do
                VIC = seleziona V vicini di item dalla matrice mat
                NEIGHS = VIC intersezione SEEDS

                for all neigh in NEIGHS do
                        itemScore = itemScore + (Sim(item,neigh) * neighScore * matWeight )
                        itemWeight = itemWeight + matWeight
                end for
        end for

itemScore = itemScore / itemWeight
rec list += < item, itemScore >

end for

return rec list
```

2.4 Implementazione dell'algoritmo e Lenskit

L'algoritmo è implementato tramite Lenskit[1], un tool open source scritto in Java che consente di costruire algoritmi di raccomanda zione in maniera modulare.

Lenskit [11]è stato creato con lo scopo di supportare la ricer ca riproducibile sui recommender system e fornire una piattafor ma robusta ma flessibile, per la sperimentazione con tecniche di raccomandazione differenti.

Lenskit fornisce vari strumenti chiave:

- **API** comuni a diversi compiti del recommender, come re commend e predict, in modo che i ricercatori possano imple mentare applicazioni ed esperimenti in modo da non doversi preoccupare dell'algoritmo.

- **Implementazioni di algoritmi standard**, rendendo facile incorporare tecniche di raccomandazione presenti in lettera tura in progetti o ricerche.

- **Un toolkit di valutazione**, per misurare la performance di un recommender su dataset comuni.

- **Codice di supporto**, per permettere l'implementazione di nuovi algoritmi con il minimo sforzo.

[1] http://lenskit.org/

Un recommender scritto in Lenskit comprende un insieme di in terfacce che forniscono raccomandazioni, predizioni e altri servizi utilizzando uno o più algoritmi collegati ad una sorgente dati.

Lenskit divide il processo di raccomandazioni in diverse par ti utilizzando delle interfacce che vengono implementate in base all'algoritmo. Nella maggior parte degli algoritmi le componenti lavorano in questo modo:

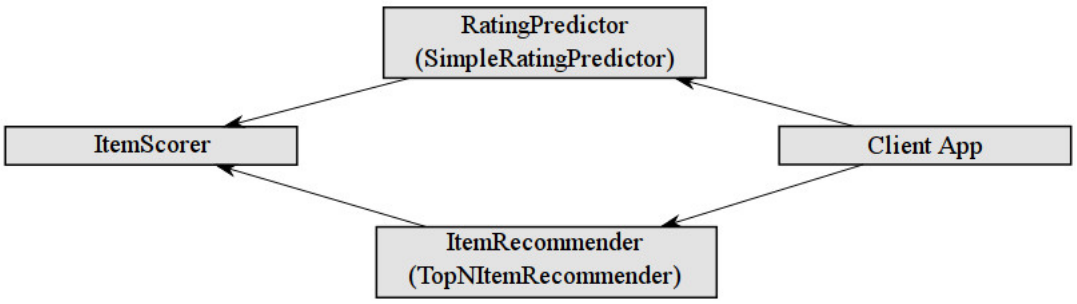


Figura 2.1: Componenti principali di Lenskit

- **ItemScorer**: è il cuore della maggior parte degli algoritmi scritti in Lenskit. Assegna uno score agli item in modo per sonalizzato rispetto ad un utente. Questo score può essere qualsiasi cosa: predizioni, probabilitò, TF IDF o similarità del coseno tra item e profilo utente. L'unica restrizione è che un punteggio più alto deve significare uno score migliore.

- **RatingPredictor**: condivide la stessa interfaccia dell'Item Scorer. Anche in questo caso l'output è una predizione, con

la differenza che lo score sarà mappato nella scala che si vuo le implementare. In pratica Lenskit divide il calcolo del le predizioni in due fasi: l'ItemScorer calcola lo score e il RatinPredictor lo mappa nel range di rating scelto.

- **ItemRecommender**: restituisce la lista di item da racco mandare. Un semplice ItemRecommender ad esempio re stituisce gli n item con gli score forniti dall'ItemScorer più alti.

La configurazione di un recommender si ottiene selezionando una implementazione per ognuna delle componenti. In fase di run le componenti sono istanziate tramite la **dependency injection**, in particolare Lenskit utilizza **Grapth**, il quale si occupa di generare un albero delle dipendenze per l'istanziazione delle classi da cui dipendono le componenti.

Lenskit permette di definire solo quelle caratteristiche che si discostano dalla implementazione classica, se non si specifica un particolare dettaglio implementativo, questo viene lasciato al valore di default(es. dimensione del vicinato).

L'algoritmo SeedRecommender in particolare, utilizza una im plementazione di ItemScorer fornita da Lenskit, denominata **UserMeanItemScorer**, per il calcolo degli score da assegnare ai seed.

UserMeanItemScorer dipende da un'altra classe, **ItemMeanItemScorer**, che produce un baseline score per un item uguale alla

somma tra la media voti globale μ e la differenza media tra il voto di un utente e la media globale:

$$b'_{u,i} = \mu + \bar{\mu}_i$$

con

$$\overline{\mu_i} = \frac{\sum_{u \in U_i}(r_{u,i} - \mu)}{|U_i| + \gamma}$$

A questo punto la UserMeaItemScorer calcola la differenza me dia $\hat{\mu}_u$ tra il voto dell'utente per ogni item e lo score della baseline per quell'item. Lo score di ogni item è la somma tra lo score baseline e la media utente calcolata:

$$b_{u,i} = b'_{u,i} + \mu_u$$

con

$$\hat{\mu}_u = \frac{\sum_{i \in I_u}(\mu_i - b'_{u,i})}{|I_u| + \gamma}$$

L'algoritmo vero e proprio invece, si trova nell'implementazione della classe ItemRecommender, denominata **SeedRecommender**, in particolare nella definizione del metodo recommend.

Riporto la configurazione delle componenti utilizzate:

```
LenskitConfiguration config = new LenskitConfiguration();

config.set(MeanDamping.class).to(5.0);
config.set(NeighborhoodSize.class).to(20);


config.bind(EventDAO.class).to(data.getEventDAO());



config.bind(ItemScorer.class).to(UserMeanItemScorer.class);

config.bind(UserMeanBaseline.class, ItemScorer.class)
        .to(ItemMeanRatingItemScorer.class);


config.bind(CoOccurrenceModel.class, ItemItemModel.class)
        .to(CoOccurrenceMatrixModel.class);
config.bind(CosineSimilarityModel.class, ItemItemModel.class)
        .to(ItemItemModel.class);
config.bind(VectorSimilarity.class)
        .to(CosineVectorSimilarity.class);
config.bind(ItemContentSimilarityModel.class, ItemItemModel.class)
        .to(ItemContentMatrixModel.class);

config.bind(ItemRecommender.class).to(SeedRecommender.class);
```

Capitolo 3

RiVal

RiVal è un toolkit open source implementato interamente in Java che permette di valutare un algoritmo di raccomandazione in un ambiente altamente controllato.

E' in grado di fornire in modo semplice risultati comparabili attraverso differenti framework di raccomandazione quali Apache Mahout, Lenskit e MyMediaLite.

Essendo un tool open source RiVal permette una analisi traspa rente del processo di valutazione, rendendo così la sperimentazione effettuata in questa tesi facilmente riproducibile e analizzabile. Il punto centrale di RiVal è proprio la riproducibilità dei risultati ottenuti, elemento molto importante per l'avanzamento della ri cerca in un campo data intensive come quello della valutazione di recommender system.

3.1 Problemi nella ricerca sui recommender system

La letteratura disponibile nel campo dei recommender system ha avuto una crescita esponenziale negli ultimi anni, ciò è dovuto a

una crescente popolarità dei suddetti sistemi grazie a colossi come Amazon o Netflix.

La mole di dati a disposizione ha portato ad una considere vole difficoltà nello studio di risultati, soprattutto nel confronto dell'accuratezza di algoritmi diversi.

Lo scoglio maggiore si incontra quando si tenta di riprodurre e eventualmente estendere risultati provenienti dalla letteratura, ciò è dovuto a diversi fattori[12]:

- Miglioramenti degli algoritmi esistenti in letteratura vengono spesso espressi in formule matematiche invece che in codice funzionante, quindi spesso quando si tenta di reimplementarli ci sono dettagli e casi limite che non vengono discussi nella pubblicazione e pertanto rendono impossibile la replicazione;

- Lo stato dell'arte nel campo dei recommender system si è sviluppata incrementalmente e in maniera decentralizzata, quindi la letteratura originale riguardante un certo algoritmo potrebbe non tenere in considerazione modifiche importanti avvenute in seguito;

- La valutazione di recommender non è gestita consistentemen te tra pubblicazioni diverse, anche con la stessa struttura di base ci sono diverse metriche e strategie di valutazione che sono state usate, queste informazioni vengono spesso omesse

portando alla difficoltà citata prima di confrontare risultati provenienti da lavori di ricerca diversi.

I ricercatori sprecano tempo reimplementando algoritmi già cono sciuti e la nuova implementazione potrebbe mancare di alcuni detta gli dei successivi miglioramenti attuati su quell'algoritmo rendendo i risultati non confrontabili.

Con alcune eccezioni (ad esempio dati sensibili di una azien da), miglioramenti su un algoritmo descritti in una pubblicazio ne dovrebbero essere accompagnati da codice funzionante in un framework standard, includento i dettagli della valutazione per permettere la riproduzione dei risultati.

Una pubblicazione ad hoc per la crescita della ricerca dovrebbe prendere in considerazione due aspetti principali:

- Pubblicare il codice sorgente delle modifiche apportate all'al goritmo in considerazione, pronto a girare in un ambiente di valutazione standard, in modo tale da favorire la comprensi bilità e la riusabilità dei miglioramenti effettuati. In questo modo si è sicuri di non tralasciare nessun dettaglio implemen tativo che potrebbe portare a una impossibilità di utilizzare i risultati in ricerche successive;

- Pubblicare i data set, i metodi di valutazione e il codice di analisi (ad esempio script per la generazione di grafici o sommari).

3.2 Le fasi del processo di valutazione

Dal paragrafo precedente si evince che è necessario un protocollo standard per condurre un qualsiasi esperimento di valutazione di un recommender.

Il lavoro di Alan Said e Alejandro Bellogín[13] propone un pro tocollo molto interessante, basato su quattro fasi principali ben distinte che devono essere necessariamente discusse in dettaglio in qualsiasi pubblicazione che descriva un processo di valutazione:

- **Data splitting:** Partizioni di test e training diverse pos sono avere un impatto non indifferente sulla performance di un recommender, e vi sono vari metodi di data splitting che possono essere presi in considerazione. Approcci basati su una timeline ad esempio, prendono in considerazione l'ordi ne temporale in cui sono avvenute le interazioni nel dataset, prendendo un punto nel tempo come split point per la divi sioni in training data e test data. Oltre a questa metodologia vi sono almeno tre metodi diversi per dividere gli item: si può designare un dato numero di item per ogni utente, per tutti gli utenti indifferentemente, oppure decidere una percentuale di tutte le interazioni utilizzando la cross validation (dividendo in folds per item o per user). Il protocollo di partizionamento deve essere chiaramente specificato oltre che ragionevolmente

adatto al contesto di valutazione.

- **Recommendation:** Il recommender che deve poi andare a effettuare il test sulle partizioni appena create deve essere chiaramente specificato a livello di implementazione, bisogne rebbe utilizzare inoltre altri algoritmi ben noti in letteratu ra (ad esempio user based collaborative filtering, item based CF e matrix factorization), in modo da avere un confronto facilmente visualizzabile.

- **Candidate items generation:** Questa fase si occupa di scegliere l'insieme di item che si vanno a considerare come target della valutazione effettuata dal recommender, e che indicheremo con L_u dove u rappresenta un utente. Vi sono tre approcci proposti: prendere come L_u l'insieme di item presenti nel test set per quell'utente, prendere tutti gli item nel sistema tranne quelli votati dall'utente nel training set, oppure prendere un insieme di item altamente rilevanti per l'utente a cui aggiungere N item presi in maniera casuale.

- **Performance measurement:** L'ultima fase consiste nel misurare la performance dei punteggi ottenuti. Molto po polari sono le metriche basate sul ranking, come precision, recall, nDCG, che hanno l'obbiettivo di catturare la qualità di una particolare predizione.

3.3 Struttura di RiVal

RiVal è un toolkit completamente open source scritto in Java che si occupa solamente della fase di data splitting e valutazione. Non contiene infatti una componente per la generazione di raccomanda zioni, che viene affidata a tool esterni come Lenskit, Apache Mahout e MyMediaLite. RiVal comprende sei moduli distinti:

1. **rival-core** in cui vi sono le strutture dati e gli oggetti co muni utilizzati in tutte le altre componenti, in particolare alcune classi per manipolare i dati come DataModel, in gra do di immagazzinare item, user, preferenze e timestamp. Si tratta di un modello standard molto semplice e facilmente estendibile. Contiene anche diverse classi che implementano semplici versioni di parser per leggere le preferenze da un file di testo, e una interfaccia Parser che tutte le diverse versioni implementano.

2. **rival-split** che contiene classi che consentono diverse strate gie di data splitting oltre che a implementazioni di parser per specifici dataset liberamente scaricabili, come MovieLens e LastFm. Il tutto è basato su una interfaccia chiamata Splitter che definisce un unico metodo che prende in input un dataset e restituisce gli split. Sono presenti in questo modulo tre imple mentazioni dell'interfaccia Splitter: CrossValidationSplitter,

TemporalSplitter e RandomSplitter. La prima divide il data set utilizzando la tecnica cross validation generando insiemi di training e test garantendo che ogni interazione presente nel dataset appaia una sola volta per ogni test split. La seconda prende in considerazione i timestamp associati a ogni inte razione ponendo interazioni più vecchie nel training set. La terza infine divide in due il dataset in modo randomico e offre la possibilità di scegliere la percentuale di utenti da porre nel training set;

3. **rival-recommend** dove risiedono classi di supporto per l'u tilizzo del recommender stesso attraverso una classe astratta principale AbstractRunner che viene estesa da due classi: Ma houtRecommenderRunner e LenskitRecommenderRunner. La prima implementa un runner per il tool Apache Mahout for nendo la possibilità di configurare il recommender da file. La seconda si occupa della configurazione e del lancio di un re commender sviluppato su Lenskit fornendo anche una clas se wrapper per la componente DAO utile per collegare il DataModel di RiVal alla sorgente dati di Lenskit;

4. **rival-evaluate** contiene metriche e strategie per la valuta zione. Contiene classi che valutano i file dei risultati prodotti dal recommender rispetto ai file di test. Questa componente consente il calcolo di misure basate sull'errore quali la RMSE

(Root Mean Square Error) e misure basate sul ranking quali la Precision, la Recall, la MAP (Mean Average Precision) e il NDCG (Normalized Discounted Cumulative Gain). Vi sono inoltre classi per la configurazione di una strategia di valutazione che definiscono come le metriche citate prima debbano essere calcolate scegliendo l'insieme degli item considerati (altamente)rilevanti. Da una classe astratta AbstractRunner si estendono cinque classi: AllItems, RelPlusN, TestItems, TrainItem e UserTest. La prima utilizza come candidati tutti gli item, la seconda utilizza un insieme di item altamente rilevanti unito a un numero N di item non rilevanti (utilizzato in [14]), la terza utilizza solo item provienti dal test set, la quarta solo item del training set e l'ultima prende tutti gli elementi del test set per ogni user;

5. **rival-example** è in pratica il manuale di RiVal, e mostra tramite esempi come utilizzare il toolkit in maniera programmatica;

6. **rival-package** infine, è il modulo di configurazione per costruire distribuzioni di RiVal.

3.4 Perchè RiVal?

Dai precedenti due paragrafi si evince che il protocollo proposto in [13] è facilmente attuabile tramite l'utilizzo di RiVal.

Il motivo principale dell'utilizzo di questo framework per con durre l'esperimento descritto in questa tesi è proprio la possibilità di definire canoni standard nelle varie fasi di valutazione, in modo da ottenere risultati confrontabili con la letteratura e che possano eventualmente essere estesi in futuro.

Un'altro fattore importante che ha influito nella scelta è l'alta estendibilità di RiVal. Questo toolkit consente infatti di modifi care le operazioni effettuate nelle diverse fasi di valutazione in un ambiente intuitivo e ben documentato, rendendo più semplice la progettazione dell'esperimento.

La procedura di valutazione presa in considerazione in questa tesi infatti, necessita di uno split che specifica delle profile size nel training set; questo tipo di data splitting non è contemplato in Ri Val (possiamo infatti gestire solamente variabili come la percentuale di utenti da inserire nel training set, oppure decidere se dividere il dataset per utenti o per preferenze tramite le classi già implementa te) ma può essere facilmente costruito estendendo le classi astratte che RiVal mette a disposizione senza andare ad compromettere l'uso delle strutture standard disponibili.

Tutto ciò è possibile grazie all'ottimo livello di astrazione delle

classi che gestiscono le varie fasi di valutazione. Queste classi infat to sono ben divise e indipendenti e permettono l'implementazione di qualsiasi processo di valutazione costringendo chi le utilizza a rispettare il protocollo descritto prima, lasciando allo stesso tempo libertà di decidere i dettagli dell'esperimento in modo versatile.

Questa possibilità in particolare permette una sperimentazio ne, che comunque non è convenzionale, in un ambiente standard, rendendo i risultati confrontabili e soprattutto riproducibili.

Capitolo 4

Descrizione dell'esperimento

L'esperimento in oggetto in questa tesi ha lo scopo di valutare l'al goritmo SeedRecommender in un ambiente controllato. Si tenta di osservare il comportamento dell'algoritmo in diversi stati del sistema cercando di rispecchiare possibili situazioni reali.

Sono stati utilizzati diversi algoritmi come parametro di con fronto, in modo da avere un riscontro facilmente visualizzabile.

In questo capitolo si andranno a descrivere il dataset utilizza to, gli agoritmi usati come baseline, le metriche per valutare le raccomandazioni e il protocollo sperimentale.

4.1 Dati

Il dataset utilizzato proviene dal popolare sito di streaming musi cale LastFm[1]. Questo sito utilizza un recommender system musi cale chiamato Audioscrobbler tramite il quale costruisce un profilo dettagliato per ogni utente tenendo traccia delle canzoni che l'u tente ascolta, sia da stazioni radio Internet, sia dal lettore musicale dell'utente che da altri sistemi di streaming musicali. Queste in

[1] http://www.last.fm

formazioni sono trasferite al database di LastFm tramite il lettore musicale stesso (Spotify, Rdio, Clementine, Amarok, MusicBee) o tramite un plugin. I dati sono visualizzati sulla pagina del profilo dell'utente e compilati per creare pagine che riferiscono a artisti in dividuali. Il sistema fornisce un sistema wiki analogo a Wikipedia a cui tutti gli utenti possono partecipare per inserire informazioni utili al recommender sugli artisti (bibliografia, tag...).

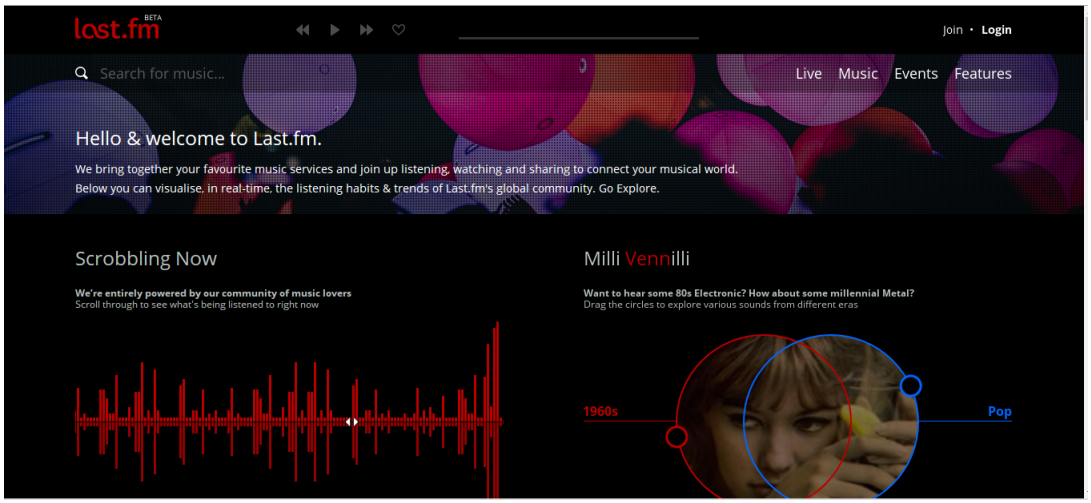


Figura 4.1: Home page di LastFm

Il dataset è composto da preferenze espresse da utenti rispetto a degli artisti. Queste preferenze si trovano nel file datasetFull.dat e sono scritte nel formato:

userId \t *itemId* \t *rating*

Le caratteristiche del dataset sono riassunte nella seguente ta bella:

Binario	SI
Numero item	7000
Numero utenti	1796
Numero rating	66289
Media rating per utente	36,9

Per quanto riguarda i documenti descrittivi degli artisti, si è scelto di utilizzare l'insieme dei tag per ogni artista presente nel le pagine wiki del sito, le caratteristiche del content scaricato da LastFm sono riassunte nella seguente tabella:

Content	Tag
Media numero tag per artista	4.7
Deviazione standard	0.8

Per recuperare i tag per ogni artista è stato implementato un piccolo crawler in Java che utilizza le API[2] messe a disposizione da LastFm per il download delle informazioni.

Questo crawler prende in input il file artist.dat che contiene la lista dei nomi degli artisti associati al corrispondente itemId, e crea un file di testo per ogni artista, contenente i tag, denominato «use rId.txt». Per l'implementazione ho utilizzato delle classi wrapper[3] fornite da Johann Kovacs[4]. Il metodo principale *getInfo* prende in

[2]http://www.last.fm/it/api
[3]https://github.com/jkovacs/lastfm-java
[4]https://github.com/jkovacs

input un HashMap le cui chiavi sono gli itemId e i valori sono i no mi degli artisti e restituisce un'altro HashMap in cui i valori sono istanze di classi wrapper Artist che contengono tutte le informazioni degli artisti (tra cui i tag). Il codice è il seguente:

```
1  public static HashMap<Integer,Artist> getInfo(HashMap<String,Integer> artists,String apiKey){
2
3          HashMap<Integer,Artist> infoArtists = new HashMap<Integer,Artist>();
4
5          for(String name   artists keySet()){
6                  Artist art = Artist getInfo(name, apiKey);
7                  if(art  = null)
8                          infoArtists put(artists get(name), art);
9                  else{
10                         String corrected = Artist getCorrection(name, apiKey) getName();
11                         art = Artist getInfo(corrected, apiKey);
12                         if(art  = null)
13                                         infoArtists put(artists get(name),art);
14                         else
15                                         notLoaded add(name);
16                         }
17         }
18
19         return infoArtists;
20 }
```

Come si può vedere, il metodo prevede un tentativo di correzione nel caso in cui il nome sia scritto male sul file; se anche questo tentativo di corrrezione non va a buon fine, il nome dell'artista va in un file chiamato notLoaded.txt per la successiva ricerca a mano, se possibile. Durante l'estrazione delle informazioni nove utenti sono finiti su questo file, e di questi nove solamente per quattro non è stato possibile trovare informazioni.

4.2 Algoritmi

Le prestazioni dell'algoritmo sono state confrontate con cinque altri algoritmi di cui due sono algoritmi classici presenti in letteratura e tre sono delle baseline molto semplici in grado di fornire racco mandazioni anche ad utenti senza preferenze espresse. Tutti gli algoritmi sono stati implementati in Lenskit:

- **Item-Item Collaborative Filtering**[4], implementato con l'algoritmo *k nearest neighbor*, in cui la similarità è calcolata utilizzando la similarità del coseno e il vicinato considerato ha dimensione fissa pari a 30, riporto la configurazione utilizzata in Lenskit:

```
1  config.bind(ItemScorer.class).
2          to(ItemItemScorer.class);
3  config.within(ItemScorer.class).
4          bind(VectorSimilarity.class).
5                  to(CosineVectorSimilarity.class);
6  config.within(ItemScorer.class).
7          bind(UserVectorNormalizer.class).
8                  to(BaselineSubtractingUserVectorNormalizer.class);
9  config.within(ItemScorer.class).
10         within(UserVectorNormalizer.class).
11                 bind(BaselineScorer.class,ItemScorer.class).
12                         to(ItemMeanRatingItemScorer.class);
13 config.within(ItemScorer.class).
14         within(UserVectorNormalizer.class).
15                 set(MeanDamping.class).to(5.0);
16 config.within(ItemScorer.class).
17         set(ModelSize.class).to(500);
18 config.within(ItemScorer.class).
19         set(NeighborhoodSize.class).to(30);
20 config.bind(UserMeanBaseline.class,ItemScorer.class).
```

```
21          to(ItemMeanRatingItemScorer.class);
22  config.set(MeanDamping.class).to(5.0);
```

- **FunkSVD**[2], l'algoritmo vincitore del Netflix Prize, implementato con l'utilizzo della Singolar Value Decomposition per creare il modello:

```
1  config.bind(ItemScorer.class).
2           to(FunkSVDItemScorer.class);
3  config.bind(BaselineScorer.class,ItemScorer.class).
4           to(UserMeanItemScorer.class);
5  config.bind(UserMeanBaseline.class,ItemScorer.class).
6           to(ItemMeanRatingItemScorer.class);
7  config.set(MeanDamping.class).to(5.0);
8  config.set(FeatureCount.class).to(30);
9  config.set(IterationCount.class).to(150);
```

- **Popularity**, semplice baseline che restituisce per qualsiasi utente la lista degli n item più popolari, definiti come gli item con il maggior numero di rating, la classe PopularityRec implementa un recommender che non fa altro che ordinare gli item per popolarità e restituire i primi n :

```
1  config.bind(ItemRecommender.class).
2           to(PopularityRec.class);
3  config.bind(ItemScorer.class).
4           to(UserMeanItemScorer.class);
5  config.bind(UserMeanBaseline.class,ItemScorer.class).
6           to(ItemMeanRatingItemScorer.class);
7  config.set(MeanDamping.class).to(5.0);
```

- **RandomPopularity**, invece restituisce per ogni utente l'item più popolare seguito da *n 1* item scelti a caso:

```
config.bind(ItemRecommender.class).
        to(RandomPopularityRec.class);
config.bind(ItemScorer.class).
        to(UserMeanItemScorer.class);
config.bind(UserMeanBaseline.class,ItemScorer.class).
        to(ItemMeanRatingItemScorer.class);
config.set(MeanDamping.class).to(5.0);
```

- **Co-Coverage**, infine, restituisce gli n item più covotati tra gli utenti:

```
config.bind(ItemRecommender.class).
        to(CoCoverageRec.class);
config.bind(ItemScorer.class).
        to(UserMeanItemScorer.class);
config.bind(UserMeanBaseline.class,ItemScorer.class).
        to(ItemMeanRatingItemScorer.class);
config.set(MeanDamping.class).to(5.0);
```

4.3 Metriche

Le metriche utilizzate per valutare le performance degli algoritmi sono quelle tipiche dell'Information Retrieval. Come insieme di item con cui confrontare le raccomandazioni per ogni utente sono stati considerati tutti gli item rilevanti nel test set da quell'utente. Per item rilevante si intende un item che ha una preferenza di 1, essendo il test binario è stato considerato item rilevante se il rating per quell'item è uguale a 1 e item non rilevante se il rating è uguale a 0. Si è scelto di utilizzare:

- **Precision**: rappresenta la frazione di item raccomandati che sono rilevanti, in pratica definisce quanto è accurata la lista

di raccomandazione confrontando le predizioni con il test set:

$$Precision = \frac{|I_{rel} \cap I_{rec}|}{|I_{rec}|}$$

dove I_{rel} è l'insieme degli item rilevanti e I_{rec} è l'insieme degli item raccomandati dall'algoritmo.

- **Recall**: rappresenta la frazione di item rilevanti che sono stati raccomandati, considera quindi tutti gli elementi rile vanti del test set e conta quanti di questi compaiono nella lista di raccomandazione, questa metrica è più sensibile alla dimensione della lista di raccomandazione:

$$Recall = \frac{|I_{rel} \cap I_{rec}|}{|I_{rel}|}$$

Si è scelto inoltre di utilizzare un'altra metrica, questa volta speci fica per il campo dei recommender system, per vedere quanto sono diversificati i consigli dati dai diversi algoritmi:

- **Aggregate Diversity**: rappresenta la frazione di item di stinti raccomandati rispetto all'insieme di tutti gli item. Que sta metrica in pratica serve a definire quanto è alta la *diversity* di un sistema, questo attributo è considerato di fondamentale importanza per la qualità di un recommender. Possiamo ve dere la aggregate diversity anche come un indice di personaliz

zazione, infatti un'alta diversificazione significa che il sistema riesce a capire bene le particolarità nei gusti dell'utente e che può distinguere adeguatamente utenti diversi. Formalmente la aggregate diversity è definita come segue:

$$AggregateDiversity = \frac{|\cup_{u \in U} L_N(u)|}{|I_{tot}|}$$

dove I_{tot} è l'insieme di tutti gli item, U l'insieme di tutti gli utenti e $L_N(u)$ è l'insieme degli item raccomandati per l'utente u.

4.4 Protocollo sperimentale

Il protocollo adottato mira a valutare il sistema in diverse situazioni che ci si potrebbero aspettare in un contesto reale. In particolare la variabile chiave è il numero di utenti in cold start. Con questo esperimento si cerca di capire come l'algoritmo si comporta in si tuazioni in cui la matrice user item è molto sparsa, fino a situazioni a regime del sistema in cui solo una percentuale degli utenti ha espresso nessuna o poche preferenze.

Innanzitutto il dataset descritto prima viene diviso in un trai ning set e un test set nel seguente modo:

- **Training Set**: questo insieme per ogni run dell'esperimento conterrà una certa percentuale di utenti in cold start. Un

utente è definito in cold start se ha espresso un numero di rating compreso tra 0 e 19. Questa percentuale è ulteriormente divisa equamente in tre trache con profile size diverse nel seguente modo: un terzo avrà profile size comprese tra 0 e 5, un terzo tra 6 e 12 e un terzo tra 13 e 19. In ogni tranche la dimensione del profilo è scelta in maniera casuale.

- **Test Set**: il test set invece contiene solo i rating degli utenti in cold start che non si trovano già nel training set. Questo è il motivo per cui è stato necessario togliere dal dataset tutti gli utenti con meno di 20 rating, in modo da avere almento un rating di test. In pratica però quasi tutti gli utenti hanno 50 rating, assicurando così un buon numero di rating per il test set.

Gli algoritmi utilizzano come sorgente dati il training set e le raccomandazioni vengono effettuate sugli utenti presenti nel test set e i risultati confrontati con i rating del test set.

L'esperimento è stato eseguito per percentuali di utenti in cold start uguali a 20%, 50%, 80%, 100%. Ognuna di queste run inoltre è stata ripetuta per liste di raccomandazione di dimensioni uguali a 10, 20, 30, 40 e 50.

L'algoritmo infine, viene analizzato in due varianti, la cui discriminante è l'utilizzo dell'insieme dei seed standard.

Capitolo 5

Progettazione e implementazione dell'esperimento

5.1 Descrizione della pipeline e implementazione

L'esperimento è stato scritto tramite le classi fornite da RIval, o comunque estensioni o implementazioni di queste classi. I dettagli configurativi dell'esperimento vengono racchiusi in un file di testo che verrà analizzato nei prossimi paragrafi.

La pipeline dell'esperimento si sviluppa in quattro fasi distinte:

5.1.1 Splitting

Inizialmente si creano i test e training set come spiegato nel capito lo precedente. Si avranno un file di test e uno di training per ogni percentuale di utenti in cold start. Per ottenere questo risultato si è implementata l'interfaccia *Splitter* di RIval in una classe denominata **ColdSplitter**. ColdSplitter implementa il metodo *split* che prendendo in input il data set completo restituisce i due file di train e test. Come prima cosa viene selezionata la percentuale di utenti da porre in cold start, dopodichè questi utenti vengono divisi in tre insiemi di profile size differenti in modo random, ma bilanciato nei tre insiemi, tramite il metodo *splitColdUsers*:

```
private HashMap<U,Integer> splitColdUsers(HashSet<U> csUsers){
int[] profileSizes1 = {0,1,2,3,4,5,6};
int[] profileSizes2 = {7,8,9,10,11,12,13};
int[] profileSizes3 = {14,15,16,17,18,19};

ArrayList<U> listUsers = new ArrayList<U>();
HashMap<U,Integer> userToPsz = new HashMap<U,Integer>();

for(U u:csUsers){
        listUsers.add(u);
        Collections.shuffle(listUsers);
        Random r = new Random();
        int csu0_5, csu6_12, csu13_19;

        if(csUsers.size() % 3 == 2){
                csu0_5 = csu6_12 = csUsers.size()/3 +1;
                csu13_19 = csUsers.size()/3;
        }else if(csUsers.size() % 3 == 1){
                csu0_5 = csUsers.size()/3 +1;
                csu6_12 = csu13_19 = csUsers.size()/3;
        }else
                csu0_5 = csu6_12 = csu13_19 = csUsers.size()/3;

        int p=0;
        for(U u:listUsers){
                if(csu0_5==0){
                        p++;
                        csu0_5=-1;
                }
                if(csu6_12==0){
                        p++;
                        csu6_12=-1;
                }

        int ps = 0;

        switch(p){
                        case 0:
                                ps=profileSizes1[r.nextInt(profileSizes1.length)];
                                csu0_5--;
                                break;
```

```
                        case 1:
                                ps=profileSizes2[r.nextInt(profileSizes2.length)];
                                csu6 12--;
                                break;
                        case 2:
                                ps=profileSizes3[r.nextInt(profileSizes3.length)];
                                csu13 19--;
                                break;
                }
        userToPsz.put(u, ps);
        }

return userToPsz;
}
}
```

A questo punto per ogni utente designato in cold start si considera l'insieme dei rating per quell'utente presenti nel dataset. Si scrivono un numero di rating uguale alla profile size assegnata a quell'utente nel training set, e i rimanenti vengono aggiunti test set. I rating degli utenti che non sono stati selezionati come cold start vengono invece aggiunti tutti al training set.

ColdSplitter viene utilizzata da un'altra classe, **ColdSplitterRunner**, che si occupa di leggere i parametri di configurazione, istanziale ColdSplitter, effettuare lo splitting e salvare gli insiemi ottenuti su file.

5.1.2 Generazione delle raccomandazioni

La fase successiva consiste nella generazione delle raccomandazioni. A questo scopo è stata estesa la classe *LenskitRecommenderRunner*

di RIval in una classe denominata **LenskitBaselineRecommenderRunner** che si occupa di configurare un recommender Lenskit, istanziarlo e generare un certo numero di raccomandazioni per gli utenti che si trovano nel test set utilizzando come sorgente dati il training set. All'interno della classe vi sono le configurazioni del le baseline descritte nel capitolo precedente più la configurazione del SeedRecommender. LenskitBaselineRecommenderRunner ge nera raccomandazioni per un algoritmo alla volta utilizzando uno switch su un parametro letto dal file di configurazione per la scelta delle componenti di Lenskit da istanziare. Per generare le racco mandazioni di tutti gli algoritmi in una volta è stata implementata la classe **MultipleBaselineRunner** utilizzando come modello la classe di Lenskit *MultipleRecommendationRunner*.

5.1.3 Generazione degli item candidati

A questo punto si definisce la strategia con cui si scelgono gli item da considerare per la fase di valutazione.

A questo scopo si utilizza la classe fornita da RIval, *UserTest,* che seleziona per item candidati solo quegli item presenti nel test set per quel determinato utente.

Viene intersecato l'insieme di item presenti nel test set con l'in sieme di item raccomandati dall'algoritmo e viene generato un file che contiene solo quelle raccomandazioni che contengono un item

presente nell'intersezione.

5.1.4 Valutazione

Infine la fase di valutazione viene eseguita utilizzando due classi fornite da RIval, **Precision** e **Recall**, e una classe che estende *AbstractRankingMetric* e implementa *EvaluationMetric*, denominata **AggregateDiversity**.

Queste tre classi servono a calcolare le omonime misure di valutazione e implementano metodi che restituiscono il valore globale, il valore per utente e il valore ad una certa cutoff.

Riporto l'implementazione del metodo *compute* di AggregateDiversity, che calcola il valore globale dell'aggregate diversity e genera l'HashMap contenente i valori delle cutoff che si vogliono considerare, il valore per utente è semplicemente la cutoff diviso il numero di item nel catalogo, dato che il recommender genera raccomandazioni distinte per uno stesso utente:

```
public void compute() {
        if ( Double isNaN(getValue())) {
                // since the data cannot change, avoid re-doing the calculations
                return;
        }
        iniCompute();
        Map<U, List<Pair<I, Double>>> data = processDataAsRankedTestRelevance();
        valuesAt = new HashMap<Integer,Double>();
        HashSet<I> distinctItems = new HashSet<I>();
        double value;

        for(Map Entry<U, List<Pair<I, Double>>> e   data entrySet()){
                List<Pair<I, Double>> sortedList = e getValue();
for(Pair<I,Double> items   sortedList){
                        distinctItems add(items getFirst(););
                }
        }

        for(int at   getCutoffs()){
                HashSet<I> distinctItemsAt = new HashSet<I>();
                for(Map Entry<U, List<Pair<I, Double>>> e   data entrySet()){
List<Pair<I, Double>> sortedList = e getValue();
                        int rank = 0;
                                for(Pair<I,Double> item   sortedList){
                                        rank++;
                                        if(at >= rank)
                                                distinctItemsAt add(item getFirst());
                                }
                }

        valuesAt put(at, (double)distinctItemsAt size()/NUMITEMS);
        }
value = (double)distinctItems size() / NUMITEMS;
setValue(value);
}
```

Per eseguire la valutazione di tutti gli algoritmi per tutte le percentuali di utenti in cold start è stata implementata la classe **MultipleEvaluationMetricPercentageRunner** che si occupa di mappare i test file con i file di raccomandazione giusti, di istanziare le classi per il calcolo delle metriche e di salvare i risultati su file.

5.2 Diagramma delle classi

5.3 Configurazione

Per configurare vari parametri dell'esperimento viene utilizzata la classe *Properties* di Java. Questa classe rappresenta un insieme persistente di proprietà, è composta da una lista di elementi chiave valore, entrambi stringhe, che possono essere salvati su uno stream o caricati da uno stream. In questo caso gli elementi di Properties sono caricati da un file di testo che contiene tutte le variabili che cambiano in run diverse dell'esperimento. Riporto un esempio di configurazione:

```
1  #Splitting
2  dataset.file=./data/LastFm/7000ItemsDatasetClean.dat
3  dataset.parser=net.recommenders.rival.core.SimpleParser
4  split.output.folder=./data/RIval/splits/RecListSize30/
5  split.output.overwrite=true
6  split.training.prefix=lastFm
7  split.training.suffix= train
8  split.test.prefix=lastFm
9  split.test.suffix= test
10 split.percentages=20,50,80,100
11
12 #Recommendation
13 framework=lenskit
14 output=./data/RIval/recommendations/RecListSize30
15 baselines=ItemItem,FunkSVD,RandomPopularity,Popularity,Cocoverage,SeedRec
16 factors = 30
17
18 #Strategy generation
19 split.folder=./data/RIval/splits/RecListSize30/
20 recommendation.folder=./data/RIval/recommendations/RecListSize30/
21 output.ranking.folder=./data/RIval/ranking/RecListSize30/
22 output.groundtruth.folder=./data/RIval/groundtruth/RecListSize30/
23 output.format=SIMPLE
24 recommendation.suffix=tsv
25 strategy.relevance.thresholds=0
26 strategy.classes=net.recommenders.rival.evaluation.strategy.UserTest
27
28 #Evaluation
29 evaluation.pred.folder=./data/RIval/ranking/RecListSize30/
30 evaluation.pred.format=SIMPLE
31 evaluation.output.folder=./data/RIval/results/RecListSize30/
32 evaluation.classes=it.maivisto.rivalEvaluation.evaluate.ranking.AggregateDiversity,
33           net.recommenders.rival.evaluation.metric.ranking.Precision,
34           net.recommenders.rival.evaluation.metric.ranking.Recall
35 evaluation.ranking.cutoffs=30
36 evaluation.relevance.threshold=0
37 evaluation.peruser=true
```

Il file è diviso in quattro parti:

- La parte di splitting specifica: il file del data set considera to(2), il parser da utilizzare(3), dove salvare i file di training e test(4), se sovrascrivere o meno i file già presenti(5), il pre fisso e il suffisso del nome da assegnare al file di training(6,7), il prefisso e il suffisso del nome da assegnare al file di test(8,9) e infine le percentuali di utenti da porre in cold start(10).

- La parte di recommendation specifica: il framework in cui è implementato il recommender(13), dove salvare le raccoman dazioni generate(14), gli algoritmi da configurare e che an dranno ad effettuare le raccomandazioni(15) e la lunghezza della lista di raccomandazione da restituire(16).

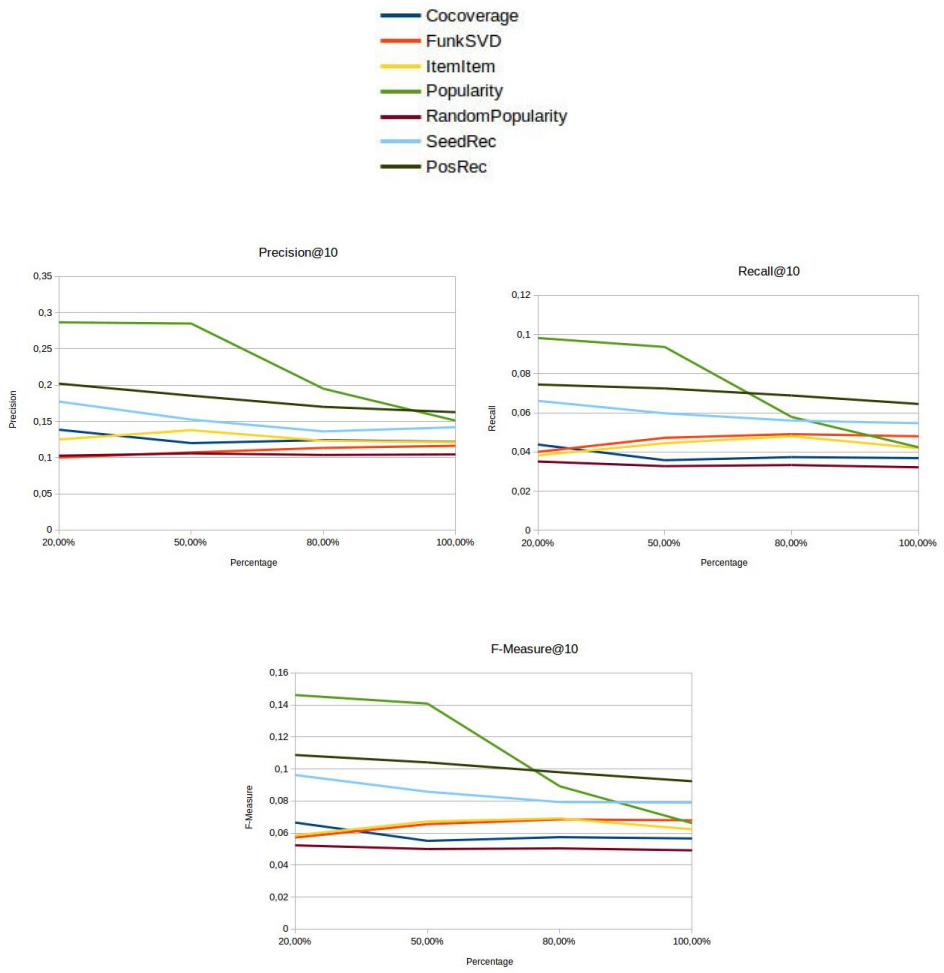
- La parte di generazione della strategia specifica: dove sono salvati gli split(19), dove sono salvate le raccomandazioni(20), dove salvare i file di ranking da generare(21), dove salvare i file di groundtruth(22), il formato in cui sono salvate le rac comandazioni(23), il suffisso del nome dei file di raccomanda zione(24), la relevance threshold da considerare(25) e la classe da utilizzare per generare la strategia(26).

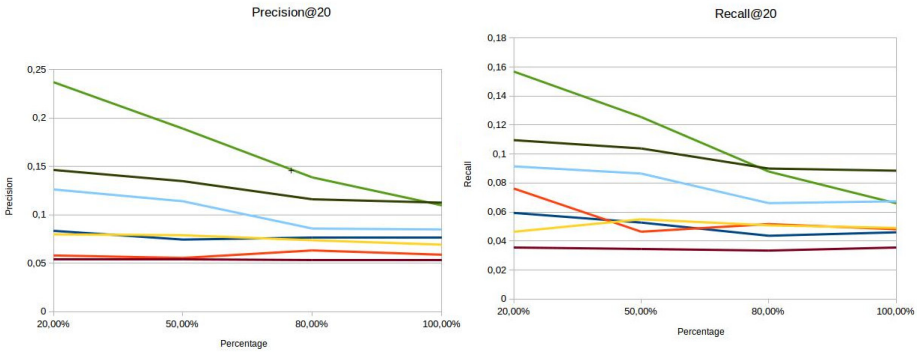
- La parte di valutazione specifica: dove sono salvati i file di ranking(29), il formato dei file di ranking(30), dove salvare i risultati(31), le classi che calcolano le metriche che si deside

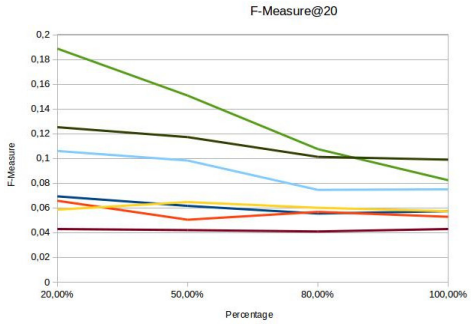
ra misurare(32), i cutoffs che si vogliono considerare(35), la relevance threshold da utilizzare(36) e se si vogliono salvare o meno i risultati per ogni utente(37).

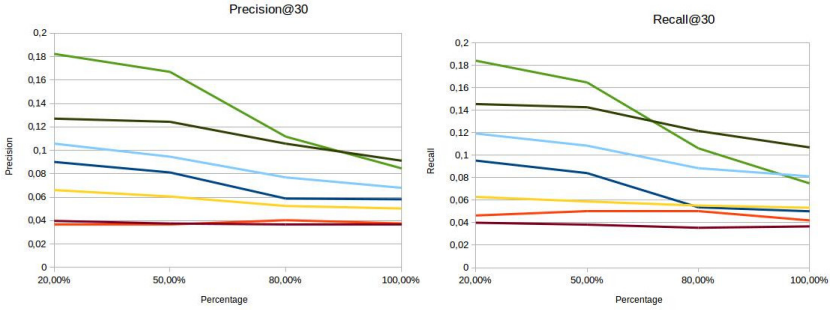
Capitolo 6

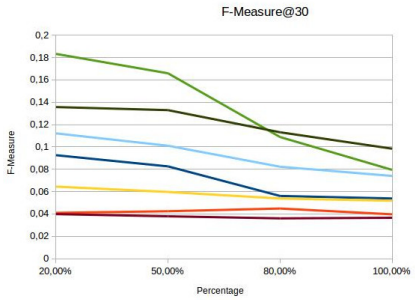
Risultati

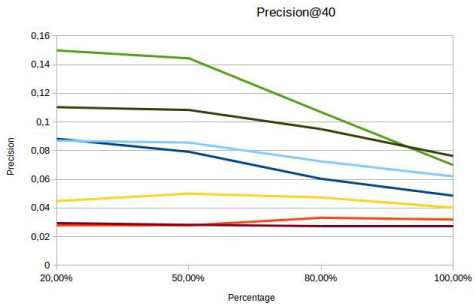


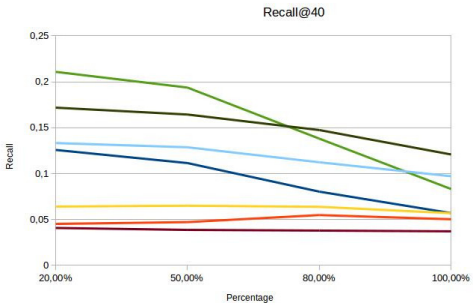

Precision@20
Precision
0,25
0,2
0,15
0,1
0,05
0
20,00%
50,00%
80,00%
100,00%
Percentage
Recall@20
Recall
0,18
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
20,00%
50,00%
80,00%
100,00%
Percentage

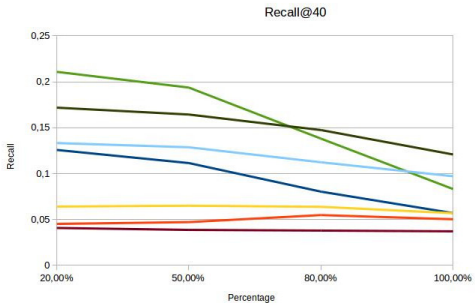

F-Measure@20
F-Measure
0,2
0,18
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
20,00%
50,00%
80,00%
100,00%
Percentage

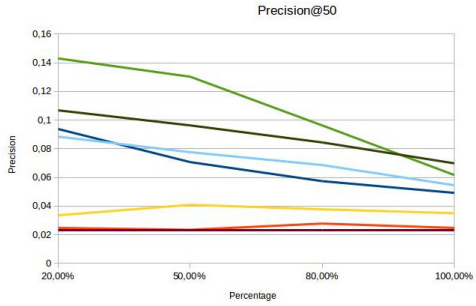

Precision@30
Precision
0,2
0,18
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
20,00%
50,00%
80,00%
100,00%
Percentage
Recall@30
Recall
0,2
0,18
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
20,00%
50,00%
80,00%
100,00%
Percentage

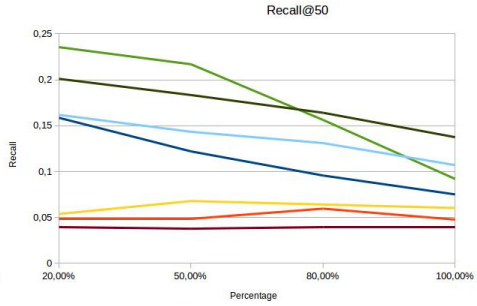

F-Measure@30
F-Measure
0,2
0,18
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
20,00%
50,00%
80,00%
100,00%
Percentage

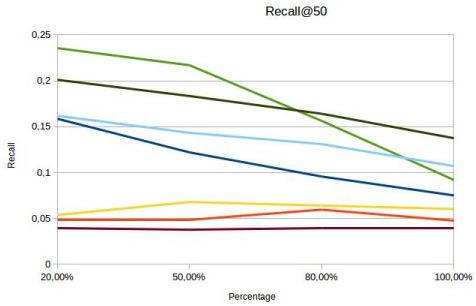

Precision@40
Precision
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
20,00%
50,00%
80,00%
100,00%
Percentage


Recall@40
Recall
0,25
0,2
0,15
0,1
0,05
0
20,00%
50,00%
80,00%
100,00%
Percentage


Recall@40
Recall
0,25
0,2
0,15
0,1
0,05
0
20,00%
50,00%
80,00%
100,00%
Percentage


Precision@50
Precision
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
20,00%
50,00%
80,00%
100,00%
Percentage


Recall@50
Recall
0,25
0,2
0,15
0,1
0,05
0
20,00%
50,00%
80,00%
100,00%
Percentage


Recall@50
Recall
0,25
0,2
0,15
0,1
0,05
0
20,00%
50,00%
80,00%
100,00%
Percentage

Dai risultati sembra che l'algoritmo Popularity abbia le presta zioni migliori.

Questo algoritmo non fa altro che raccomandare gli n item più popolari a tutti gli utenti in modo completamente spersonalizzato. Le buone prestazioni sono date dal fatto che LastFm tende a mo strare artisti molto ascoltati a utenti che si sono appena registrati. La probabilità che un utente voti artisti molto popolari è alta e quindi è facile che un algoritmo così definito riesca ad indovinare un buon numero di raccomandazioni.

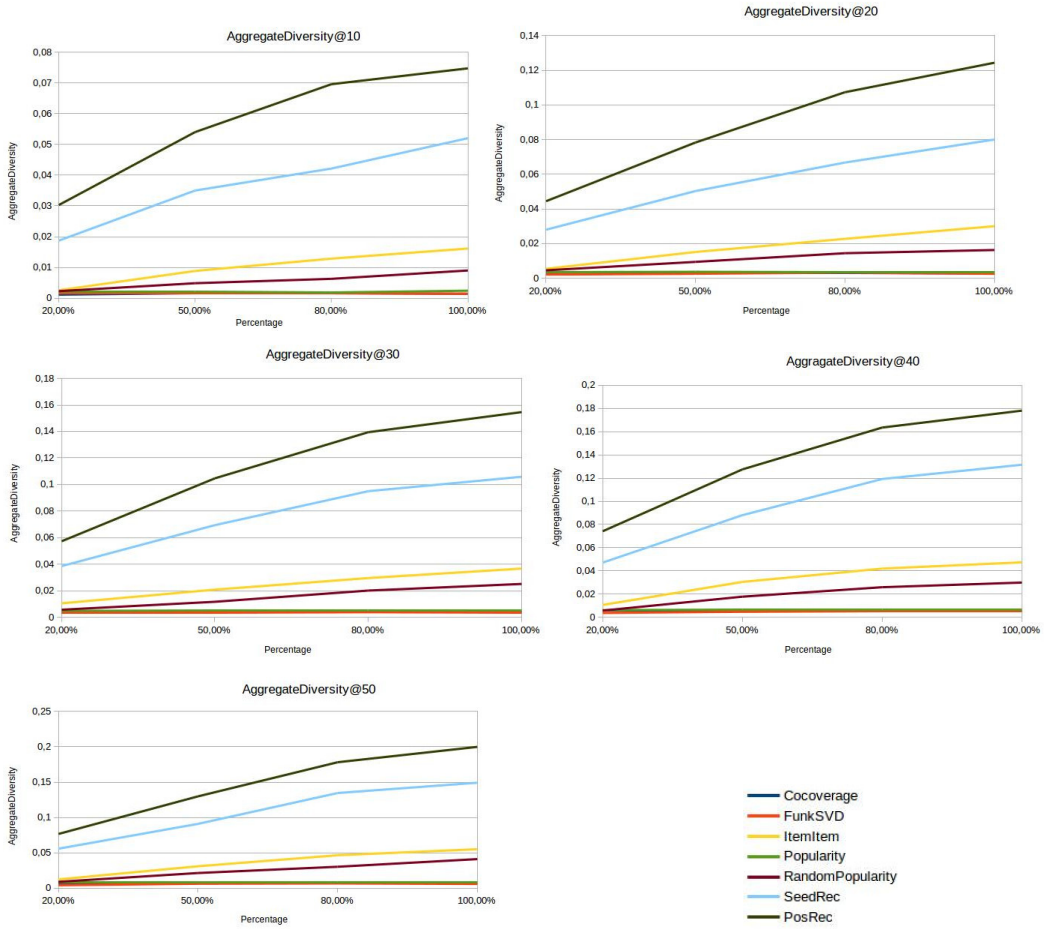
Vediamo inoltre che la curva delle prestazioni di Popularity scen de molto all'aumentare della percentuale di utenti in cold start, da to che l'algoritmo non utilizza per niente le informazioni riguardo le preferenze di un dato utente, se non per eliminare gli oggetti già votati dalla lista di item raccomandati, l'unica spiegazione è che l'algoritmo peggiora all'aumentare delle dimensioni del test set. Questo fattore è ovvio se si pensa che l'insieme di item che Po pularity raccomanda è oggetto di pochissime o nulle variazioni nel tempo, e quindi se si prende un insieme più grande di item rilevanti per l'utente il numero di raccomandazioni indovinate è lo stesso, perciò il calo di prestazioni.

Il nostro algoritmo invece, in entrambe le varianti, mantiene una prestazione abbastanza costante all'aumentare della percentuale di utenti in cold start.

Le prestazioni in generale sono superiori a quelle di tutte le baseline tranne Popularity, per i motivi spiegati prima.

Sorprendenti i risultati riguardanti PosRec, ovvero la variante dell'algoritmo in cui non sono considerati i seed standard, che so no nettamente migliori rispetto a quelli di SeedRec, sembra infatti che i seed rappresentino solamente rumore nella generazione delle raccomandazioni.

Vediamo ora un'altra metrica che riguarda la diversificazione delle raccomandazioni generate dagli algoritmi.



Possiamo vedere come Popularity abbia un coefficiente di diver sificazione vicino allo zero, in quanto le raccomandazioni sono quasi uguali per tutti gli utenti.

Il nostro algoritmo invece mostra un valore abbastanza alto ad indicare che le raccomandazioni generate sono molto personalizzate e varie, in particolare all'aumentare della percentuale di utenti in cold start, la diversificazione aumenta.

Conclusioni e sviluppi futuri

La tesi si è incentrata sull'analisi e l'esecuzione di un esperimento in un ambiente standard e open source.

L'obbiettivo principale è stato quello di ottenere una sperimen tazione che sia facilmente riproducibile ed estendibile. Tutto il co dice utilizzato, compreso di crawler per scaricare i dati da LastFm è liberamente scaricabile da GitHub[1].

Il lavoro esposto inoltre si inserisce nello sviluppo della compo nente di raccomandazione del progetto MAIVISTO, finanziato dal MIUR nell'ambito StartUp.

Questa componente fa uso di diversi criteri di similarità per predire lo score da assegnare ad un item, ognuno di questi criteri definisce un livello di astrazione dell'item, che è quindi visto come un oggetto a più dimensioni.

Nell'implementazione considerata i criteri di similarità non so no niente di innovativo, si utilizzano infatti le comuni definizioni di similarità disponibili in letteratura, come la similarità del co seno, il valore di co occorrenza tra item e il random indexing per la similarità tra contenuti. La parte interessante è l'algoritmo di raccomandazione vero e proprio, infatti le diverse similarità vengo

[1] https://github.com/Mattia26/RivalEvaluation

no combinate per generare una lista di raccomandazione che tenga conto di diverse proprietà dell'item. SeedRecommender infatti uti lizza le diverse matrici per generare un vicinato che, a differenza della definizione classica di vicinato, potrebbe contenere più di un occorrenza dello stesso item. Questi item che compaiono più volte sono i candidati migliori poichè risultano rilevanti per più di un livello di astrazione dell'item.

I risultati della sperimantazione mostrano che l'algoritmo fun ziona meglio se non si utilizzano i seed standard, questi quindi rap presentano un elemento di disturbo nella generazione delle predizio ni, notiamo dai grafici del capitolo precedente che il calo di presta zioni riguarda sia l'accuratezza che la novelty delle raccomandazioni generate.

Vediamo inoltre che l'algoritmo ha prestazioni abbastanza co stanti all'aumentare della percentuale di utenti in cold start nel sistema. Questo ci porta a pensare che l'insieme di seed standard non è necessario o utile nell'affrontare il problema del cold start, se non quando un utente non ha espresso nessun rating.

I valori ottenuti sono comunque accettabili e SeedRecommender ha prestazioni migliori di tutte le baseline considerate tranne Po pularity nel contesto dell'accuratezza. Popularity comunque sfrutta il fatto che molti sistemi mostrano gli item più popolari all'utente per ovviare il problema del cold start e quindi l'algoritmo sembra

essere molto efficiente.

Inoltre l'accuratezza di un algoritmo non è l'unico parametro da tenere in considerazione per giudicare la qualità delle raccoman dazioni.

Un importante elemento di un recommender di qualità è la di versità. Una bassa diversità infatti indica un grado di spersona lizzazione delle raccomandazioni che sono sempre uguali, l'utente in questo modo vede sempre le stesse cose di cui probabilmente conosce già l'esistenza.

Non per niente la popolarità è una strategia scarsamente utiliz zata nella pratica, anche se ottima in campo di ricerca come termine di confronto.

Restano da valutare infine, le diverse configurazioni dei pesi del le matrici, cosa succederebbe infatti se si utilizzasse di più la simi larità del content e meno quella del coseno? Possibili sviluppi com prendono quindi sperimentazioni con parametri di configurazione diversi.

Sarebbe opportuno inoltre, utilizzare una metrica di similarità del content più specifica per il contesto di applicazione reale. In questa tesi il content è costituito da un insieme di tag per artista che non hanno un contesto in una frase, ma sono completamente indipendenti. Nell'ambito in cui verrà utilizzato l'algoritmo invece potrebbero esserci delle descrizioni di trama, o parole chiave come

il genere o il regista che rappresentano un tipo di contenuto a più livelli con un conseguente aumento della complessità dei dati.

Per quanto riguarda i seed standard invece è chiaro che quelli scelti non sono ottimali, un possibile sviluppo può essere la ricerca di un insieme di seed più performante tramite la scelta di criteri euristici diversi.

Infine sarebbe sicuramente interessante effettuare un test nel contesto reale, dopo che l'algoritmo è stato inserito nella piattaforma e il sistema contiene un buon numero di utenti e rating. La valutazione online tramite giudizi espliciti da parte degli utenti sul la qualità delle raccomandazioni è sicuramente il modo migliore per valutare un algoritmo del genere.

Bibliografia

[1] Michael D. Ekstrand, John T. Riedl and Joseph A. Kon stan.Collaborative Filtering Recommender Systems. Journal Foundations and Trends in Human Computer Interaction archive Volume 4 Issue 2, February 2011 Pages 81 173.

[2] Simon Funk, "Netflix update: Try this at home" http://sifter.org/~simon/journal/20061211.html, December 2006.

[3] J. Herlocker, J. A. Konstan, and J. Riedl. An empirical analysis of design choices in neighborhood based collaborative filtering algorithms. Information Retrieval, vol. 5, no. 4, pp. 287 310, 2002.

[4] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item Based Collaborative Filtering Recommendation Algorithms. In ACM WWW '01, pp. 285 295, ACM, 2001.

[5] Pasquale Lops, Marco De Gemmis, and Giovanni Semeraro.Content based recommender systems: State of the art and trends. In Recommender systems handbook., 2011, pp. 73 105.

[6] Rocchio, J. Relevance Feedback Information Retrieval. In: G. Salton (ed.) The SMART retrieval system experiments in automated document processing, pp. 313 323. PrenticeHall, Englewood Cliffs, NJ (1971)

[7] Seung taek Park, David Pennock, Omid Madani, and Nathan Good. Naï ve filterbots for robust cold start recommendations. Proceedings of the 12th ACM SIGKDD international confe rence on Knowledge discovery and data mining, pp. 699 705, 2006.

[8] Mohammad Hossein Nadimi Shahraki and Mozhde Bahador pour. Cold start Problem in Collaborative Recommender Systems: Efficient Methods Based on Ask to rate Techni que. Journal of Computing and Information Technology, pp. 105 113, 2014.

[9] Micheal Elahi, Francesco Ricci, and Neil Rubens. Active Lear ning Strategies for Rating Elicitation in Collaborative Filte ring: A System Wide Perspective. ACM Transactions on In telligent Systems and Technology (TIST) Special Section on Intelligent Mobile Knowledge Discovery and Management Sy stems and Special Issue on Social Web Mining, vol. 5 (1), no. 13, Dicembre 2013.

[10] Will Lowe. Towards a theory of semantic space. Proceedings of the 23rd conference of the cognitive science society, pp. 576 581, 2001.

[11] Michael D. Ekstrand. Towards Recommender Enginee ring: Tools and Experiments for Identifying Recommender Differences. Ph.D Thesis, University of Minnesota, 2014.

[12] Michael D. Ekstrand, Michael Ludwig, Joseph A. Konstan, and John T. Riedl. Rethinking the Recommender Research Ecosystem: Reproducibility, Openness, and LensKit. RecSys '11, October 23 27, 2011.

[13] Alan Said and Alejandro Bellogín. 2014. Comparative recom mender system evaluation: benchmarking recommendation frameworks. In Proceedings of the 8th ACM Conference on Recommender systems. ACM, 129 136.

[14] P. Cremonesi, Y. Koren, and R. Turrin. Performance of recom mender algorithms on top n recommendation tasks. In RecSys, pages 39 46, 2010.