Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Projecte final de carrera:

# Un sistema escalable per a recomanacions en temps real

**Alex Catarineu Morales**

Director: Ricard Gavaldà Mestre

Departament de Llenguatges i Sistemes Informàtics (LSI)

# Dades del Projecte

*Títol:* **Un sistema escalable per a recomanacions en temps real**

*Nom de l'estudiant:* **Alex Catarineu Morales**
*Titulació:* **Enginyeria en Informàtica Superior**
*Crèdits:* **37,5**
*Director/Ponent:* **Ricard Gavaldà Mestre**
*Departament:* **Llenguatges i Sistemes Informàtics (LSI)**

# Membres del tribunal

*President:* **Antoni Lozano Bojados**

*Signatura:*

*Vocal:* **Jan Graffelman**

*Signatura:*

*Secretari:* **Ricard Gavaldà Mestre**

*Signatura:*

# Qualificació

*Qualificació numèrica:*
*Qualificació descriptiva:*

*Data:*

*M'agradaria agrair al meu tutor de projecte Ricard Gavaldà l'ajuda i consells que he rebut per part seva en la realització del projecte i la memòria.*

# Índex

# Capítol 1

# Introducció

## 1.1 Motivació

Avui en dia, els usuaris tenen a la seva disposició una enorme quantitat d'ítems per a consumir (pel·lícules, música, llibres, videojocs...). Amb aquest oceà d'opcions, descobrir nous productes que s'adaptin al seu gust pot esdevenir una tasca llarga i difícil.

Aquí es on entren en joc els anomenats sistemes recomanadors. Aquests sistemes s'encarreguen d'analitzar el comportament d'un usuari per tal de poder fer-li recomanacions personalitzades que siguin del seu interès. Si són ben fets, poden resultar de gran utilitat, permetent-li trobar ítems interessants de forma instantània, que d'una altra forma potser hagués trigat molt de temps en descobrir.

Si anem una mica més enllà, podem posar-nos a l'altra banda i plantejarnos: com es pot crear un bon sistema recomanador? Quins algoritmes utilitzar? Com tractar la enorme quantitat de dades que hi pot haver? Com tenir en compte el *feedback* que els usuaris ens van donant per a deduïr els seus gustos i millorar les recomanacions?

Aquest projecte pretén estalviar totes aquestes preguntes a un desenvolupador que vulgui crear un sistema recomanador de qualitat, oferint les eines per a fer-ho de forma senzilla.

## 1.2 Objectius i etapes del projecte

L'objectiu principal del projecte és el desenvolupament d'una plataforma (o llibreria) que pugui ser utilitzada per a crear sistemes recomanadors escalables, eficients i que reaccionin en temps real als diferents esdeveniments que hi pugui haver, cosa que implica, entre d'altres, adaptar-se instantàniament als gustos dels usuaris.

Un altre objectiu, un cop implementada la llibreria, és utilitzar-la per a desenvolupar un sistema recomanador específic amb una interfície web senzilla, a fi d'avaluar la plataforma

desenvolupada en un domini concret i comprensible.

S'ha dividit el procés de desenvolupament del projecte en les etapes següents.

**Estudi de l'estat de l'art en sistemes recomanadors** Primer de tot, s'ha de realitzar un procés de documentació que ens permeti estudiar amb una certa profunditat el món dels sistemes recomanadors. Per una altra banda, també ens ha d'ajudar a fixar l'abast del projecte, donada la gran diversitat en quant al tipus de sistemes recomanadors que existeixen.

**Recerca d'un algoritme de recomanació que compleixi els requisits desitjats** Es pot considerar una part de l'estudi de l'estat de l'art, però és prou important per a destacar-ho. Es tracta de trobar un algoritme de recomanació que compleixi amb els requisits que ens hem fixat. Aquests es detallaran més endavant però, resumint, l'algoritme hauria ser eficient, escalable, i ser capaç d'incorporar en temps real nou *feedback* dels usuaris per a millorar les recomanacions.

**Disseny i especificació de l'arquitectura del sistema** Tot i que el projecte està més enfocat a la part algorísmica, és important realitzar un bon disseny de l'arquitectura del sistema. L'objectiu principal és el desenvolupament d'una llibreria, pel que s'ha de tenir cura de fer un disseny modular, fàcilment extensible i el més genèric possible.

**Implementació de les classes obtingudes** A partir del disseny, i dels algoritmes que s'han decidit incorporar, s'ha de realitzar la implementació del sistema. Un dels aspectes més delicats d'aquesta implementació s'espera que sigui el relacionat amb la gestió de les dades, ja que el sistema ha de ser capaç de suportar un nombre considerable d'usuaris i ítems.

**Avaluació dels algoritmes implementats** Un cop implementada la llibreria, es farà una avaluació de les recomanacions que s'obtenen amb els diferents algoritmes, utilitzant dades reals (usuaris, ítems, puntuacions) disponibles lliurement. La metodologia seguida ha de garantir que les proves realitzades són justes i objetives per a tots els algoritmes.

**Implementació d'un sistema recomanador concret utilitzant la nostra plataforma** S'implementarà un sistema recomanador utilitzant la plataforma creada. Com s'explicarà als corresponents apartats, s'ha decidit crear un recomanador de grups musicals i artistes, basat en el *dataset* de *Yahoo* [1].

**Avaluació del sistema recomanador implementat** Realitzar una avaluació del sistema recomanador concret que s'ha implementat. La intenció és que aquesta avalució sigui més qualitativa que l'avaluació genèrica de la llibreria mencionada anteriorment. És a dir, una prova on s'analitzin, aplicant el sentit comú i un cert coneixement dels grups musicals, si els ítems recomanats realment s'adequen al perfil de l'usuari que s'estigui utilitzant.

## 1.3 Projectes similars

Podem citar un parell de projectes similars al nostre.

**Apache Mahout** Mahout [2] és una llibreria que implementa nombrosos mètodes d'aprenentatge automàtic, sobre llenguatge Java. Molts d'aquests mètodes estan preparats per a ser executats de forma paral·lela sobre Apache Hadoop [3], un framework emprat per executar tasques de forma distribuïda amb grans quantitats de dades, sobre clusters amb molts nodes. Per al que ens interessa a nosaltres, conté alguns algoritmes de recomanació, més concretament de filtratge col·laboratiu. Inclou, per exemple, un algoritme de factorització semblant al que hem implementat nosaltres, preparat per a ser paral·lelitzat. No obstant, a la pròpia pàgina s'adverteix de que al ser un algoritme iteratiu, té un rendiment pobre sobre la seva plataforma Hadoop. A més, podem dir que tracta el problema només des d'un punt de vista *offline*, sense tenir en compte les actualitzacions incrementals del model a mesura que arriben noves dades, que nosaltres sí tractarem.

**MyMediaLite** La llibreria MyMediaLite [4] la vam descobrir amb el projecte ja en una fase final. Implementa algoritmes semblants al nostre, juntament amb molts d'altres. Està implementada en C#, cosa que en la nostra opinió és un petit inconvenient des del punt de vista de la integració en un entorn web, ja que tot i que la plataforma .NET està creixent, encara està molt lluny de Java en aquest aspecte. A favor nostre tenim que, a diferència de la citada llibreria, integrem la gestió de les dades del recomanador (usuaris, ítems, puntuacions), un punt gens trivial a causa del gran volum que poden arribar a assolir.

## 1.4 Estructura del document

El document està estructurat com segueix:

1. **Introducció:** Es descriu la motivació del projecte i les etapes en les quals s'ha dividit.
2. **Preliminars:** Es fa una introducció general als sistemes recomanadors, i s'expliquen els conceptes necessaris que apareixeran al llarg del document.

3. **Anàlisi de requisits:** Es detalla la llista de requisits funcionals i no funcionals que haurà de complir el nostre producte: la llibreria per a implementar sistemes recomanadors.

4. **Algoritmes de predicció de puntuacions:** Es descriuen els algoritmes de predicció de puntuacions estudiats i implementats, el cor del sistema recomanador tal i com l'entenem. També es mencionen estratègies que es van provar però no van funcionar correctament.

5. **Disseny:** Descripció de l'arquitectura i les classes del sistema.

6. **Implementació:** Inclou detalls d'implementació que s'han considerat rellevants.

7. **Experiments i avaluació:** Es detallen els experiments i resultats que s'han obtingut a l'hora d'avaluar els algoritmes i el sistema recomanador implementat.

8. **Planificació i cost:** Planificació i anàlisi del cost del projecte.

9. **Conclusions i treball futur:** Conclusió i descripció de possibles ampliacions o millores del projecte.

10. **Bibliografia**

Finalment, s'inclouen també apèndixs amb la descripció completa de les classes de la llibreria i gràfiques i taules que han resultat dels experiments i l'avaluació.

## 1.5 Planificació i estimació del cost

### 1.5.1 Planificació temporal

En primer lloc, s'ha realitzat una estimació del temps necessari per a dur a terme les diferents etapes del projecte.

| Etapa | Temps (hores) |
|---|---|
| **Definició del projecte** | **30** |
| Definició d'objectius | 5 |
| Anàlisi de requisits | 15 |
| Planificació | 10 |
| **Formació i aprenentatge** | **150** |
| Estudi de l'estat de l'art dels sistemes recomanadors | 60 |
| Recerca d'algoritmes que compleixin els requisits | 60 |
| Investigació sobre la tecnologia adient per a persistir les dades | 30 |
| **Especificació i disseny** | **50** |
| **Implementació** | **300** |
| Implementació de la llibreria | 250 |
| Implementació del sistema recomanador concret | 50 |
| **Proves** | **70** |
| Proves de la llibreria | 50 |
| Proves del sistema recomanador concret | 20 |
| **Documentació** | **200** |
| **Total** | **800** |

Taula 1.1: Estimació del temps necessari per dur a terme el projecte

En total, s'estima una duració de 800 hores. Si suposem una dedicació de 5 hores al dia, treballant 20 dies al mes, tenim una duració aproximada de 8 mesos.

### 1.5.2 Estimació del cost

El cost de software és nul, ja que s'utilitzarà únicament programari lliure. Per una altra banda, s'estimarà el cost que suposaria si en comptes d'haver estat realitzat per una persona hagués estat realitzat per un equip.

- **Cap de projecte:** s'encarregaria de coordinar tot el grup, i realitzar la documentació.
- **Analista:** s'encarregaria del treball de investigació, formació i aprenentatge, així com de dissenyar l'arquitectura i les classes del sistema.
- **Programador:** s'encarregaria de implementar les classes derivades de la especificació i disseny del sistema.
- **Dissenyador web:** s'encarregaria d'implementar la interfície web del sistema recomanador implementat.
- **Tester:** s'encarregaria de realitzar les proves del sistema.

I fent l'assignació d'hores a cadascun dels rols:

| **Treballador** | **Temps (hores)** | **Preu/hora** | **Cost total** |
|---|---|---|---|
| Cap de projecte | 230 hores | 50€/hora | 11500€ |
| Analista | 200 hores | 30€/hora | 6000€ |
| Programador | 250 hores | 20€/hora | 5000€ |
| Programador web | 50 hores | 15€/hora | 750€ |
| Tester | 70 hores | 20€/hora | 1400€ |
| **Total** | **800 hores** | **31€/hora** | **24650€** |

Taula 1.2: Estimació del cost del projecte

El cost total del projecte resultaria en uns 24650€.

# Capítol 2

# Preliminars

## 2.1 Introducció als sistemes recomanadors

Els sistemes recomanadors són eines que tenen com objectiu suggerir ítems que poden ser d'utilitat a un usuari.

Aquest tipus de sistemes han esdevingut força populars els darrers anys, aplicats en àmbits molt diversos. Alguns exemples coneguts:

- **Amazon:** La coneguda botiga online ofereix recomanacions a partir dels productes que l'usuari ha comprat o valorat positivament anteriorment.
- **Netflix:** Aquesta plataforma, un servei de subscripció que ofereix pel·lícules online a través de *streaming*, mostra recomanacions de pel·lícules que els usuaris encara no han vist, juntament amb una predicció de la seva puntuació.
- **Filmaffinity:** Lloc web dedicat al món del cinema, també ofereix recomanacions de pel·lícules, però en aquest cas sense comercialitzar cap tipus de producte.
- **Tastekid:** Lloc web que recomana música, pel·lícules, sèries, llibres, autors i videojocs, basant-se en els respectius ítems que els usuaris han marcat com *m'agrada* o *no m'agrada*.
- **Bananity:** Xarxa social on els usuaris poden marcar com a *m'encanta* o *ho odio* tot tipus d'ítems. Ens permet conèixer gent amb gustos similars als nostres, i ens recomana ítems que ens poden agradar.

Aquests exemples ens suggereixen algunes categories genèriques on els sistemes recomanadors poden ser aplicats:

- **Entreteniment:** pel·lícules, música, programes, sèries de televisió, etc.
- **Continguts:** articles, documents, notícies, llocs web, etc.

- **Comerç electrònic:** productes per a consumidors com ordinadors, llibres, gadgets, càmeres, etc.

- **Serveis:** plans de viatge, experts en algun tema, cases per comprar, per trobar parella, etc.

Una divisió alternativa dels exemples mencionats podria ser: aquells sistemes recomanadors que s'han implementat sobre una aplicació ja existent, com un servei addicional, i aquells casos on l'aplicació està pensada al voltant d'un sistema recomanador des de l'inici.

Fixem-nos ara en el primer cas, els sistemes recomanadors que han estat afegits a una aplicació per a oferir una millora en el servei. Podem fer un petit anàlisi de les raons que poden haver portat als proveïdors a implementar un sistema recomanador:

- **Augmentar vendes:** En el cas d'una botiga online, per exemple, pot ser interessant implementar un recomanador que permeti a l'usuari descobrir productes que li poden ser d'utilitat, que d'altra forma potser no n'hauria tingut coneixement i per tant, no hagués adquirit.

- **Diversificar vendes:** A banda d'augmentar el nombre de vendes, també pot interessar diversificar-les, en el sentit de no només vendre els productes més populars, sinó també productes menys coneguts però que, mitjançant la personalització que permeten els sistemes recomanadors, descobrim que poden ser d'utilitat per a determinats usuaris.

- **Augmentar la satisfacció de l'usuari:** La funció del sistema recomanador pot ser simplement millorar l'experiència de l'usuari amb l'aplicació. Si les recomanacions són útils, la opinió de l'usuari sobre l'aplicació millorarà, l'utilitzarà amb més freqüència i serà més fidel.

- **Entendre millor les preferències de l'usuari:** Analitzant les dades recol·lectades pel sistema recomanador, es pot arribar a extreure informació interessant sobre les preferències dels usuaris. En una botiga online, per exemple, es podria utilitzar aquesta informàcio per a decidir quins nous tipus de productes posar en venda, tenint en compte les preferències de conjunts d'usuaris rellevants.

Sembla evident que el món dels sistemes recomanadors és realment complex. Poden ser utilitzats en contextos molt diversos i per motius completament diferents. Per tant, el nostre objectiu no serà abarcar-los a tots. El que intentarem serà definir formalment el problema a resoldre (apartat 2.3), especificar una llista de requisits que volem que es compleixin (secció 3) i finalment implementar una plataforma (o llibreria) que resolgui el problema i compleixi amb els requisits especificats.

Per al lector que vulgui conèixer més en detall tot l'espectre dels sistemes recomanadors, recomanem el que considerem un dels llibres més complets sobre la matèria: [5]. En el nostre cas, ens ha resultat de gran utilitat per a la documentació d'aquest projecte.

## 2.2 El concurs *Netflix Prize*

Per ser concrets, perquè el referenciarem en diversos punts de la memòria i perquè ha estat potser l'aparició més mediàtica dels sistemes recomanadors creiem important explicar en què va consistir el *Netflix Prize*.

L'any 2006, *Netflix*, una empresa dedicada al lloguer *online* de pel·lícules en DVD (tot i que més recentment ha fet un gir cap al *streaming*) va posar en marxa una competició anomenada *Netflix Prize*.

El concurs oferia 1 mil·lió de dòlars a aquell equip que aconseguís millorar en un 10% la precisió del model que *Netflix* utilitzava per a fer les seves recomanacions. Concretament, van fer públic un conjunt de dades d'entrenament (consistent en 100.480.507 puntuacions que 480.189 usuaris van donar a 17.770 pel·lícules) amb el qual els participants podien entrenar els seus models. Un equip participant podia proposar una sol·lució enviant el codi font a *Netflix*, que n'avaluava la precisió amb un conjunt de dades de prova privat. L'any 2009, l'equip anomenat "BellKor's Pragmatic Chaos" va aconseguir superar aquesta barrera i guanyar el premi.

Deixant de banda el discutible èxit comercial per a *Netflix* d'aquest concurs, és innegable que va suposar un gran impuls per a la literatura sobre sistemes recomanadors, almenys per aquells algoritmes on es fan servir puntuacions numèriques d'usuaris sobre ítems.

En la nostra opinió, una de les idees claus que va sorgir gràcies al *Netflix Prize* va ser la d'aplicar un algoritme de factorització de matrius basat en una tècnica de *gradient descent* [6]. Tot i que l'autor d'aquesta idea no va guanyar el concurs, moltes de les millors solucions l'han implementat com una part del seu model, incloent la solució guanyadora. De fet, un cop acabat el concurs, *Netflix* va comunicar que havia analitzat el codi font de l'equip guanyador, però havien arribat a la conclusió que el model era massa complex com per a ser implementat de forma eficient (consistia d'una barreja de 107 algoritmes diferents). Finalment, van decidir implementar els dos algoritmes dels 107 que per separat oferien millor resultats, sent el que millor funcionava el citat mètode de factorització de matrius [7].

Més endavant, a l'apartat 4.4, veurem que aquesta tècnica sorgida gràcies al concurs de *Netflix* ha estat la base de l'algoritme principal que hem implementat a la nostra llibreria.

## 2.3 Definició del problema de recomanació

En aquest apartat intentarem definir formalment el problema que ens interessa resoldre: el problema de recomanació.

Denotem per $U$ el conjunt de tots els usuaris, i per $I$ el conjunt d'ítems que poden ser recomanats. Sigui $w\colon U \times I \to \mathbb{R}$ una funció que mesura la utilitat d'un cert ítem per a un usuari. Aleshores, el sistema avaluarà la funció $w(u, i)$ per a tots els ítems dins d'un

conjunt de candidats $C \in I$ i recomanarà a l'usuari $u$ els $k \ll |I|$ ítems amb utilitat més gran.

Evidentment, hi ha recomanadors que queden fora d'aquesta definició. Un sistema recomanador no ha d'emprar necessàriament una funció d'utilitat per a decidir els ítems que li poden agradar a un usuari. Per exemple, per a recomanar ítems a un cert usuari, es podria triar subconjunt d'ítems que hagin agradat a usuaris semblants, i utilitzar algunes regles predefinides o altres mètodes *ad hoc* per a reduïr aquest subconjunt i quedar-nos amb els $K$ millors. No obstant, tot i no incloure tots els recomanadors (cosa que per una altra banda sembla impossible) hem considerat que aquesta formalització del problema n'inclou a la gran majoria.

Amb aquesta definició, se situa el càlcul de la funció d'utilitat $w$ com el nucli d'un sistema recomanador. Per a fer una bona predicció d'aquesta funció ha d'existir algun mecanisme que permeti a un usuari expressar una opinió respecte els ítems.

Aquestes opinions poden tenir diverses formes:

- Puntuacions numèriques, per exemple dins d'un rang de 0 a 100.
- Puntuacions ordinals, com "m'agrada molt", "està bé", "no m'agrada", "no m'agrada gens".
- Puntuacions binàries, on l'usuari expressa que quelcom li agrada o li desagrada.
- Puntuacions unàries, on l'usuari indica que quelcom li agrada.

Per tant, en molts casos la funció $w$ serà una estimació de la puntuació que l'usuari li donaria a un cert ítem, i que en general desconeixem abans que l'usuari la declari. De fet, nosaltres només contemplarem aquest tipus de funcions d'utilitat i per tant substituirem d'ara en endavant $w(u, i)$ per l'expressió $\hat{r}_{u,i}$, que denota la puntuació estimada que l'usuari $u$ li donaria a l'ítem $i$.

Segons el mètode utilitzat per a realitzar aquesta predicció podem obtenir una classificació dels sistemes recomanadors:

- Basats en el contingut
- Col·laboratius
- Híbrids

## 2.4 Tipus de sistemes recomanadors

### 2.4.1 Basats en el contingut

Es recomanen ítems semblants a altres ítems que han agradat a l'usuari. Per a poder calcular aquesta similaritat entre ítems, s'ha de disposar d'un cert coneixement dels mateixos,

de les seves característiques.

En alguns casos tindrem per cadascun un conjunt d'atributs que el defineixen. Per exemple, en un recomanador de pel·lícules, alguns exemples d'atributs adients podrien ser: actors, directors, gèneres, temàtica/es, etc. En aquest cas, aquest conjunt seria el mateix per a tots els ítems, i els valors que podrien prendre serien coneguts. Aquest tipus de representació dels ítems l'anomenem *estructurada*.

No sempre es pot disposar d'una representació d'aquest tipus. Per sort, hi ha tècniques per a extreure automàticament característiques a partir del contingut de certs tipus d'ítems.

Per exemple, són molt comuns els ítems basats en contingut textual, com articles, notícies, anuncis, etc. Emprarem el terme general *document* per a denotar aquests tipus d'ítems. Aleshores, podem definir formalment la representació d'un document com segueix. Sigui $D = \{d_1, d_2, ..., d_n\}$ el conjunt de tots els documents del sistema, i $T = \{t_1, t_2, ..., t_m\}$ un diccionari de termes comú per a tots els documents. Podem representar un document $d_i \in D$ com un vector $m$-dimensional, $\vec{v_i} = (w_1, w_2, ..., w_n)$ on cada component $w_j$ representa el pes del terme $j$-èsim del diccionari a dins del document $d_i$.

Aquests pesos intenten mesurar la importància d'un terme per a un document. Una manera típica de calcular-los és mitjançant l'anomenat *TF-IDF* (*term frequency-inverse document frequency*). Es defineix com el producte de dos factors: la freqüència del terme i la freqüència de documents inversa,

$$\mathrm{tf} * \mathrm{idf}(t, d) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$$

$$\mathrm{tf}(t, d) = \frac{\mathrm{tc}(t, d)}{\max\{\mathrm{tc}(w, d) : w \in d\}}$$

$$\mathrm{idf}(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

on $\mathrm{tc}(t, d)$ denota el nombre d'aparicions del terme $t$ al document $d$. Podem veure la freqüència del terme, $\mathrm{tf}(t, d)$, com el nombre d'aparicions de $t$ a $d$, normalitzat per evitar biaxos a favor de documents llargs. I la freqüència de documents inversa, $\mathrm{idf}(t, D)$, és una mesura que indica si el terme és comú o estrany a tots els documents del sistema. Aleshores, amb aquest càlcul un terme tindrà un pes elevat si apareix força vegades al document i és relativament poc comú a tots els documents del sistema en general.

Amb aquesta representació vectorial, podem emprar la similaritat cosinus (angle entre dos vectors) com a similaritat entre els documents:

$$\mathrm{sim}(d_i, d_j) = \frac{\vec{v_i} \cdot \vec{v_j}}{||\vec{v_i}|| \cdot ||\vec{v_j}||}$$

I amb aquesta noció de similaritat, i les puntuacions que un usuari ha donat, podem utilitzar fórmules típiques del filtratge col·laboratiu basat en ítems (veure apartat 4.3) per a predir les puntuacions per a ítems desconeguts.

### 2.4.2 Col·laboratius (Collaborative filtering)

En aquest cas, per a predir puntuacions no es té cap tipus de coneixement sobre el contingut dels ítems. Per a aconseguir-ho, s'utilitzen les opinions de molts usuaris (col·laboració). Hi ha diverses maneres de fer-ho, però bàsicament les podem dividir en dues categories.

#### 2.4.2.1 Basats en memòria

S'utilitzen les puntuacions disponibles al sistema per a calcular la similaritat entre usuaris i ítems. Aleshores, es poden agregar les puntuacions d'usuaris o ítems similars per a predir una puntuació. Precisament distingirem dues versions, una basada en usuaris i l'altra basada en ítems.

A la versió basada en usuaris, per a predir $\hat{r}_{u,i}$ es combinen les puntuacions d'usuaris similars a $u$ sobre l'ítem $i$. Una manera usual de realitzar el càlcul seria amb la fórmula:

$$\hat{r}_{u,i} = \frac{\sum\limits_{u' \in U_i} sim(u,u') r_{u',i}}{\sum\limits_{u' \in U_i} sim(u,u')}$$

$r_{u,i}$ denota la puntuació que l'usuari $u$ li ha donat a l'ítem $i$, i $U_i$ el conjunt d'usuaris que ha puntuat l'ítem $i$. Altres variants només inclouen usuaris que superin un cert llindar en la similaritat, per tal d'intentar evitar el soroll produït per puntuacions d'usuaris poc similars.

La similaritat entre usuaris $sim(u,u')$ es pot calcular a partir de les puntuacions de $u$ i $u'$, aplicant mecanismes com la *correlació de Pearson*,

$$sim(u,u') = \frac{\sum\limits_{i \in I_{u,u'}} (r_{u,i} - \bar{r_u})(r_{u',i} - \bar{r_{u'}})}{\sqrt{\sum\limits_{i \in I_{u,u'}} (r_{u,i} - \bar{r_u})^2 \sum\limits_{i \in I_{u,u'}} (r_{u',i} - \bar{r_{u'}})^2}}$$

on $\bar{r_u}$ denota la puntuació mitjana de l'usuari $u$ i $I_{u,u'}$ el conjunt d'ítems que han puntuat tant l'usuari $u$ com $u'$, o la similaritat cosinus,

$$sim(u,u') = \frac{\vec{u} \cdot \vec{u'}}{||\vec{u}|| \cdot ||\vec{u'}||} = \frac{\sum\limits_{i \in I_{u,u'}} r_{u,i} r_{u',i}}{\sqrt{\sum\limits_{i \in I_{u,u'}} r_{u,i}^2} \sqrt{\sum\limits_{i \in I_{u,u'}} r_{u',i}^2}}$$

on $\vec{u}$ denota el vector de puntuacions de l'usuari $u$ [1].

A la versió basada en ítems, en canvi, es combinen puntuacions d'ítems similars a $i$ que l'usuari $u$ ha puntuat,

$$\hat{r}_{u,i} = \frac{\sum_{i' \in I_u} sim(i,i') r_{u,i'}}{\sum_{i' \in I_u} sim(i,i')}$$

on $I_u$ és el conjunt d'ítems que ha puntuat l'usuari $u$, i la similaritat entre ítems es calcula de forma anàloga a l'anterior.

Com hem vist, per a calcular la similaritat entre dos usuaris es necessita un cert nombre d'ítems que hagin votat ambdós. Després, per a fer la predicció també és necessari que hi hagi usuaris similars que hagin votat l'ítem pel qual estem estimant la puntuació. En definitiva, si l'ítem o usuari pel qual volem fer la predicció no disposa de prou puntuacions, és difícil fer una bona predicció.

En general, els algoritmes de filtratge col·laboratiu pateixen si hi ha aquest problema d'escassetat de dades. No obstant, si tots els usuaris i ítems tenen un nombre raonable de puntuacions, acostumen a donar bons resultats.

#### 2.4.2.2 Basats en un model

S'utilitzen mètodes de mineria de dades i aprenentatge automàtic per a entrenar un model que s'utilitzarà per a realitzar prediccions. Les dades d'entrenament seran totes les puntuacions que els usuaris han donat als ítems.

És una categoria que pot englobar molts tipus de recomanadors, ja que els models que es poden utilitzar poden ser molt diversos. Com a exemples, podríem citar les Xarxes Bayesianes, models basats en Clustering d'usuaris i ítems, descomposició en valors singulars (SVD), etc.

Un dels avantatges del filtratge col·laboratiu basat en model és que és molt més resistent a problemes d'escassetat de dades que els basats en memòria.

A continuació entrarem en més detall en un tipus concret, anomenat model de factors latents.

**Models de factors latents** Els models de factors latents s'han popularitzat molt arran del concurs *Netflix Prize*, mencionat a l'apartat 2.2. Consisteixen en caracteritzar cada usuari i cada ítem amb vectors d'uns pocs factors (diguem, de 20 a 100), deduïts a partir dels patrons i relacions ocultes de totes les puntuacions del sistema. La predicció de la

[1] Anomenem vector de puntuacions d'un usuari al vector que té un component per cada ítem del sistema, contenint la puntuació que li ha donat a l'ítem en qüestió, o bé un 0 si no l'ha puntuat. El vector de puntuacions d'un ítem seria anàleg, amb un component per cada usuari del sistema

puntuació d'un usuari $u$ a un ítem $i$ es realitzarà amb un producte escalar dels respectius vectors de $u$ i $i$.

Intuitivament, cada component d'aquests vectors (factor) acabarà significant alguna cosa pròpia del context del recomanador. Per exemple, si es tracta d'un recomanador de pel·lícules, el primer component dels vectors dels ítems podria indicar la quantitat d'acció que conté la pel·lícula en concret, el segon la quantitat de comèdia, etc. Aleshores, el primer component dels vectors dels usuaris indicaria com d'important és per l'usuari l'acció, el segon com d'important és per l'usuari la comèdia, etc. D'aquesta forma, fent el producte escalar del vector d'un usuari pel d'un ítem es sumarien tots aquests factors per a obtenir una estimació de la puntuació.

Formalment, si $f$ és el nombre de factors, volem construir un vector $p_u \in \mathbb{R}^f$ per a cada usuari $u$ i un vector $q_i \in \mathbb{R}^f$ per a cada ítem i, de tal forma que la predicció d'una puntució es faci amb el producte escalar

$$\hat{r}_{u,i} = p_u \cdot q_i$$

Una manera de calcular aquests vectors és definint una mesura de l'error que cometen en predir les puntuacions, i intentar minimitzar-lo progressivament.

L'algoritme que s'ha implementat en aquest projecte, com una solució eficient, escalable i amb bona qualitat en les seves prediccions és una variant d'aquest tipus de model, i s'explica detalladament a la secció 4.4.

### 2.4.3 Híbrids

Els sistemes híbrids tracten de combinar diversos tipus d'algoritmes per a aprofitar el millor de cadascun. Per exemple, un recomanador que complementi algoritmes de filtratge col·laboratiu amb algoritmes basats en el contingut és capaç de resoldre els problemes que tenen els primers per a recomanar nous ítems. En aquests casos, el filtratge col·laboratiu no pot funcionar ja que no es disposa de cap puntuació per a l'ítem, però la recomanació basada en el contingut sí que funcionarà, ja que disposem de la informació sobre les seves característiques, l'únic que necessitem per a fer prediccions de puntuacions amb aquest mètode.

Una manera típica de combinar algoritmes de predicció és *barrejar* els seus resultats (*blending*). Si tenim $n$ predictors, el que necessitem per a combinar-los és una funció $f : \mathbb{R}^n \to \mathbb{R}$ que accepti com entrada un vector amb les prediccions de cada algoritme, i retorni una predicció final. Una opció seria utilitzar una funció lineal, assignant a cada predictor un pes de forma i calculant la predicció final com una suma ponderada de les prediccions d'entrada. Sigui del tipus que sigui, la funció $f$ s'ha d'entrenar per a optimitzar l'error utilitzant un conjunt de prova amb moltes instàncies de prediccions i els seves corresponents puntuacions reals donades per l'usuari.

Aquests mètodes per a barrejar predictors van ser molt utilitzats al concurs *Netflix Prize*, mencionat anteriorment. De fet, la solució guanyadora és una barreja de 107 algoritmes diferents [8].

# Capítol 3

# Anàlisi de requisits

En aquest apartat s'indiquen els requisits funcionals i no funcionals que ens hem proposat per a la llibreria de sistemes recomanadors. Aquests requisits han marcat clarament el seu desenvolupament, sobretot pel que fa a la tria de l'algoritme de predicció de puntuacions.

## 3.1 Funcionals

La llibreria ha de suportar les següents operacions o events:

**Nou usuari** Un nou usuari es registra al sistema, possiblement amb un conjunt de puntuacions a ítems inicial.

**Nou ítem** Un ítem s'introdueix al sistema, possiblement amb un conjunt de puntuacions d'usuaris inicial.

**Nova puntuació** S'afegeix una nova puntuació d'un usuari a un ítem, possiblement sobreescrivint una de ja existent (canvi d'opinió).

**Eliminar usuari** S'elimina un usuari del sistema, i tota la informació corresponent (per exemple, les seves puntuacions).

**Eliminar ítem** S'elimina un ítem del sistema, i tota la informació corresponent (per exemple, les seves puntuacions).

**Eliminar puntuació d'un usuari a un ítem** S'esborra la puntuació que un usuari havia donat a un ítem.

**Generar recomanacions per un usuari** Es recomanen $K$ ítems a un cert usuari.

Un altre requisit funcional és que les recomanacions han de ser bones, la qual cosa es tradueix bàsicament en que les prediccions de les puntuacions d'usuaris a ítems també ho siguin. El concepte de "bones" no està definit amb precisió, però per tenir-ne una idea, s'espera que les prediccions siguin millors que les d'algoritmes bàsics com els de filtratge col·laboratiu basat en usuaris o ítems.

## 3.2 No funcionals

**Escalabilitat** Ha de poder suportar un elevat creixement de nombre d'usuaris, ítems i puntuacions sense patir pèrdues de rendiment notables (per problemes de falta de memòria, per exemple). Concretant una mica més, hauria de ser capaç de suportar fins a un milió d'usuaris i ítems, i sobre els mil milions de puntuacions. Més enllà d'aquests nombres, segurament caldria distribuïr les dades, cosa que queda fora de l'abast del projecte.

**Eficiència** Els algoritmes emprats han de ser eficients en funció del nombre de usuaris, ítems i puntuacions del sistema.

**Temps real** Els possibles events generats per usuaris s'han de tractar en temps real, és a dir, el recomanador s'ha de mantenir sempre actualitzat. Per exemple, s'ha de tenir en compte cada nova puntuació que arriba, i adaptar-se conseqüentment als gustos de l'usuari.

**Modularitat** S'ha de seguir un disseny modular de les classes de la llibreria. És a dir, separar tant com es pugui components que realitzin funcionalitats independents i complementàries.

**Extensibilitat** Ha de permetre fàcilment a un programador modificar i ampliar les seves funcionalitats.

**Genericitat** S'ha de poder implementar tot tipus de sistemes recomanadors, independentment de la classe d'ítems que s'estiguin recomanant.

**Llenguatge Java** La llibreria s'ha d'implementar en llenguatge Java. Per a prendre aquesta decisió, s'han tingut en compte diversos factors. Per una banda, els recomanadors gairebé sempre s'utilitzen en aplicacions o entorns web. Per tant, el llenguatge triat ha de ser una bona opció per a programar en aquests tipus de sistemes. Java ho és, i disposa d'una gran quantitat de *frameworks* amb els quals es poden desenvolupar aplicacions web de gran qualitat (*Google Web Toolkit*, *Java Server Faces*, *Struts*...). Per una altra banda, es tenia la intenció, en un primer moment, d'intentar a aplicar alguna tècnica de les contingudes a la llibreria *MOA*, que implementa algoritmes de mineria de dades sobre fluxes de dades massius. Tot i que finalment no es va usar, aquest va ser un altre motiu per la tria de Java, juntament amb el fet que es tenia força experiència, i molt positiva, en el desenvolupament d'aplicacions amb aquest sistema.

# Capítol 4

# Algoritmes de predicció de puntuacions

En aquest apartat es donaran detalls sobre els algoritmes de predicció de puntuacions implementats a la llibreria, que corresponen a les implementacions de la interfície *RatingPredictor* llistades a la secció A.1.2.

No cal repetir que l'algoritme de predicció de puntuacions, segons el nostre concepte de recomanador, és la seva peça més important. Per tant, creiem que dedicar-li un apartat específic dins de la memòria és coherent amb això i amb el fet que el treball de recerca i documentació que s'ha fet en aquest sentit ha suposat gran part del temps de realització del projecte.

S'ha descartat estudiar algoritmes basats en el contingut, ja que per les seves característiques són massa depenents del tipus d'ítems a recomanar, i busquem una solució genèrica. Per tant, ens hem decidit per algoritmes de filtratge col·laboratiu, que en usar només les puntuacions d'usuaris són completament independents dels ítems que es recomanen.

L'algoritme que s'ha implementat com una bona solució respecte els requisits que ens havíem plantejat (escalabilitat, eficiència, bones recomanacions) s'anomena *BRISMF*, de *Biased Regularized Incremental Simultaneous Matrix Factorization* [9].

També s'han implementat algoritmes més senzills com el filtratge col·laboratiu basat en usuaris o basat en ítems de l'apartat 2.4.2.1. La idea és oferir algunes alternatives, i així tenir també la possibilitat de fer una comparativa justa amb el *BRISMF*.

Per últim es mencionen alguns algoritmes i estratègies que s'han provat en el procés de recerca i que s'han descartat per no haver donat resultats satisfactoris.

## 4.1 Algoritme simple

S'ha implementat aquest predictor com a *baseline*, és a dir, com un punt de referència per a comparar com és la millora d'algoritmes més complexos respecte un de molt simple.

La predicció de la puntuació que un usuari $u$ li donaria a un ítem $i$ es realitza amb la següent fórmula,

$$\hat{r}_{u,i} = \bar{r}_{*,i} + \bar{r}_{u,*} - \bar{r}$$

on $\bar{r}$ denota la mitjana de totes les puntuacions del sistema, $\bar{r}_{*,i}$ és la puntuació mitjana de l'ítem $i$ i $\bar{r}_{u,*}$ és la puntuació mitjana de l'usuari $u$.

Intuitivament, el que es fa és sumar a la puntuació mitjana de l'ítem la diferència entre la puntuació mitjana de l'usuari i la puntuació mitjana global (si l'usuari puntua per sobre de la mitja, la quantitat que s'afegirà serà positiva, si no negativa).

No obstant, les mitjanes $\bar{r}_{*,i}$ i $\bar{r}_{u,*}$ no es calculen simplement amb la suma de puntuacions dividida per el nombre de puntuacions. En cas que el nombre de puntuacions d'un usuari o ítem fos petit, una mitjana calculada d'aquesta forma podria no ser massa fiable. En comptes d'això, es pondera amb la mitjana global de la següent forma:

$$\bar{r}_{*,i} = \frac{K \cdot \bar{r} + \sum_{r \in R_i} r}{K + |R_i|}$$

$$\bar{r}_{u,*} = \frac{K \cdot \bar{r} + \sum_{r \in R_u} r}{K + |R_u|}$$

On $R_i$ denota el conjunt de puntuacions de l'ítem $i$, $R_u$ el conjunt de puntuacions de l'usuari $u$ i $K$ és una constant positiva que controla la importància que té la mitjana global $\bar{r}$.

A partir d'ara, si no s'indica el contrari se suposarà que les mitjanes $\bar{r}_{*,i}$ i $\bar{r}_{u,*}$ es calculen d'aquesta forma.

## 4.2 Filtratge col·laboratiu basat en usuaris

En aquesta implementació, la predicció de la puntuació d'un usuari $u$ per a un ítem $i$ es realitza amb una suma de les puntuacions de tots els usuaris $u'$ que han puntuat el mateix ítem $i$, ponderada per la similaritat entre $u'$ i $u$. Intuitivament, el que es fa és utilitzar la opinió d'usuaris similars per a estimar la puntuació d'un ítem.

$$\hat{r}_{u,i} = \bar{r}_{u,*} + \frac{\sum_{u' \in U} sim(u,u')(r_{u',i} - \bar{r}_{u',*})}{\sum_{u' \in U} sim(u,u')}$$

De fet, a la fórmula es fa un petit ajustament per a compensar els biaixos dels usuaris. És a dir, no seria just comptar de la mateixa forma la puntuació d'un usuari que de mitjana puntua un 3 i la d'un que de mitjana puntua un 4. Pel tant, el que es fa és una suma ponderada de les diferències entre la puntuació i la mitjana de cada usuari $u'$, i s'afegeix a la puntuació mitjana de l'usuari $u$.

La similaritat entre usuaris és quelcom que es pot calcular de diverses maneres. Per al nostre algoritme hem emprat l'anomenada similaritat cosinus, vista a la secció 2.4.2.1. Si representem totes les puntuacions d'un usuari en un vector, podem calcular la similaritat entre dos usuaris com el cosinus de l'angle entre els seus respectius vectors.

Tanmateix, es podrien implementar moltes altres alternatives per la similaritat entre dos usuaris, per exemple mètodes que utilitzin coneixement més enllà de les puntuacions, dades demogràfiques, etc.

## 4.3 Filtratge col·laboratiu basat en ítems

Aquest algoritme és anàleg a l'anterior, però en aquest cas centrant-nos en els ítems. És a dir, s'utilitzen les puntuacions d'ítems similars que l'usuari ha puntuat per a estimar la puntuació de l'ítem.

$$\hat{r}_{u,i} = \bar{r}_{*,i} + \frac{\sum_{i' \in I} sim(i,i')(r_{u,i'} - \bar{r}_{*,i'})}{\sum_{i' \in I} sim(i,i')}$$

També s'utilitza el mecanisme de compensació pel possible biaix de les puntuacions dels ítems.

## 4.4 BRISMF

Aquest és l'algoritme que s'ha implementat amb la idea de satisfer tots els requisits que ens havíem plantejat. El significat del les sigles és (*Biased Regularized Incremental Simultaneous Matrix Factorization*).

L'algoritme es descriu a [9], però també s'han aplicat algunes idees de [10], sobretot pel que fa a l'entrenament incremental. Es pot classificar dins la categoria de models de factors latents, descrita breument a la secció 2.4.2.2.

Els algoritmes d'aquest tipus tenen en comú que la predicció de la puntuació d'un usuari $u$ a un ítem $i$ es realitza amb

$$\hat{r}_{u,i} = p_u \cdot q_i \tag{4.1}$$

on $p_u \in \mathbb{R}^f$ és un vector que caracteritza l'usuari $u$ i $q_i \in \mathbb{R}^f$ un vector que caracteritza l'ítem i. Suposarem que $f$, el nombre de components (factors) dels vectors, és constant.

Els vectors característics de cada usuari i ítem es calculen a partir de totes les puntuacions usuari-ítem del sistema. Evidentment, tota la importància de l'algoritme recau en la manera de calcular aquests vectors.

Els següents sub-apartats explicaran en detall l'algoritme utilitzat. Se seguirà un procediment incremental, començant pel model més bàsic i afegint millores fins arribar a l'algoritme final. Així, es començarà per l'*ISMF* (*Incremental Simultaneous Matrix Factorization*), se seguirà amb el *RISMF* (*Regularized Incremental Simultaneous Matrix Factorization*) i finalment es concluirà amb la versió definitiva *BRISMF* (*Biased Regularized Incremental Simultaneous Matrix Factorization*).

### 4.4.1 Algoritme bàsic - *ISMF*

En primer lloc cal aclarir, pel nom de l'algoritme, què és la factorització de matrius. La idea és molt simple: donada una matriu $R \in \mathbb{R}^{n \times m}$ l'objectiu és obtenir dues matrius $P \in \mathbb{R}^{n \times f}$ i $Q \in \mathbb{R}^{m \times f}$ tal que $R \simeq PQ^T$, on $f \ll min(n, m)$ denota el nombre de factors de la factorització.

En el nostre cas, la matriu $R$ correspondria a la matriu de puntuacions, amb les files representant usuaris i les columnes ítems. Aleshores, cada fila $p_u$ de $P$ correspondria a un vector característic de l'usuari $u$ i cada fila $q_i$ de $Q$ correspondria a un vector característic de l'ítem $i$.

No obstant, per a nosaltres la matriu $R$ a factoritzar és dispersa, és a dir, només alguns dels seus elements (puntuacions) estan definits. Aquí és on algoritmes clàssics de factorització de matrius, com el *Singular Value Decomposition*, tenen problemes. Les puntuacions desconegudes no es poden representar a la matriu simplement com zeros, ja que la factorització quedaria molt deformada i en conseqüència les prediccions serien molt dolentes. Per tant, per aplicar algoritmes com l'esmentat s'haurien d'omplir els forats de les puntuacions desconegudes amb estimacions realitzades per models més simples (p.e. secció 4.1). Això augmentaria la complexitat de l'algoritme (s'ha de factoritzar la matriu sencera, en comptes d'usar només les puntuacions conegudes) i es perdria precisió, en fer estimacions de puntuacions desconegudes.

Per tant, necessitem un algoritme que sigui capaç de realitzar aquestes factoritzacions tenint només en compte els elements de la matriu de puntuacions que coneixem. És discutible si amb aquestes restriccions el problema es pot seguir considerant de factorització

de matrius, ja que, sent estrictes, $R$ no podria ser una matriu en tenir elements no definits. Ens ha semblat important aclarir la relació entre la factorització de matrius i el problema que ens ocupa, el de trobar els vectors característics de cada usuari i ítem per al nostre model, ja que són termes que apareixen molt lligats a la literatura quan es fa referència a aquests tipus d'algoritmes. No obstant, a partir d'ara deixarem de banda el concepte de matriu de puntuacions i factorització de matrius, ja que creiem que només complicaria la notació i fórmules utilitzades.

Tornem al problema de trobar els conjunts de vectors $p_*$ i $q_*$ tal que les prediccions realitzades amb ells siguin bones. Aplicarem, però, una petita variació en la manera de calcular les prediccions respecte la fórmula 4.1.

$$\hat{r}_{u,i} = \bar{r} + p_u \cdot q_i \tag{4.2}$$

Al producte escalar dels vectors li afegim $\bar{r}$, la puntuació mitjana global, que serà constant. És un truc que ens donarà algunes avantatges pràctiques. La primera és que per les característiques de l'algoritme, resulta més eficient el càlcul dels vectors $p$ i $q$, ja que en aquest cas s'encarregaran d'expressar la diferència de les puntuacions respecte la mitjana, magnituds molt més petites que les puntuacions en sí mateixes. La segona és que en cas d'un nou usuari o ítem amb poques (o cap) puntuacions, aquesta fórmula resulta molt més estable, fent una predicció prop de la mitjana, que en l'altra versió podria quedar molt deformada.

Un cop aclarida aquesta variació, procedim a formalitzar el problema que volem resoldre. Ens cal introduïr alguna mesura, quelcom que ens indiqui com de bones son les prediccions del nostre model. Usarem el *RMSE* (*Root-Mean-Square Error*). Es tracta d'una mesura de l'error que comet el model a l'hora de fer prediccions, calculada usant les puntuacions reals ($r_{u,i}$) i les predites ($\hat{r}_{u,i}$).

$$RMSE = \sqrt{\frac{\sum_{r_{u,i} \in R} (r_{u,i} - \hat{r}_{u,i})^2}{|R|}}$$

On $R$ és el conjunt de puntuacions del sistema.

Ara que tenim una mesura de la precisió del nostre model, podríem definir la tasca de calcular els vectors $p$ i $q$ d'usuaris i ítems com un problema d'optimització:

$$\underset{p_*, q_*}{min} \sum_{r_{u,i} \in R} (r_{u,i} - \hat{r}_{u,i})^2 \tag{4.3}$$

És a dir, el problema quedaria reduït a trobar el conjunt de vectors $p_*$ i $q_*$ tal que la predicció realitzada amb ells tingui *RMSE* mínim. A la fórmula anterior ens hem estalviat

dividir per $|R|$ i aplicar l'arrel quadrada, ja que de totes maneres és equivalent a optimitzar el RMSE.

De fet, l'objectiu de l'algoritme d'aquest apartat (*ISMF*) és precisament resoldre el problema d'optimització esmentat. Per a fer-ho, s'emprarà una tècnica coneguda, que va ser utilitzada per primera vegada en aquest context per Simon Funk [6] .

**Gradient Descent** El mètode del *Gradient Descent* ens permet trobar un mínim local d'una funció $F\colon \mathbb{R}^n \to \mathbb{R}$ diferenciable. La idea intuïtiva és, a partir d'un punt inicial, anar fent passos proporcionals al gradient negat de la funció en el punt actual. Si els passos són prou petits, ens anirem acostant successivament a un mínim local.

Més formalment, sigui $x_0 \in \mathbb{R}^n$ un punt inicial. Aleshores podem definir la seqüència

$$x_{n+1} = x_n - \alpha \nabla F(x_n), n \geq 0$$

on $\nabla$ denota com és usual el gradient d'una funció. Si $\alpha$ és prou petit (es pot considerar infinitesimal) aleshores es complirà

$$F(x_0) \geq F(x_1) \geq F(x_2) \geq ... \tag{4.4}$$

D'aquesta manera, un possible algoritme consistiria en anar fent iteracions fins que $F(x_{n-1}) - F(x_n)$ fos més petit que una certa constant $\epsilon$.

Intentem ara aplicar aquest mètode al nostre problema. En el nostre cas, la funció a minimitzar és la que apareix a la equació 4.3.

Si $I$ denota el conjunt d'ítems, i $U$ el d'usuaris, tenim $|I|\,|U|\,f$ variables escalars, ja que cada vector $p$ i $q$ té $f$ components. Per a calcular-ne el gradient, hauríem de derivar respecte cada una d'aquestes variables.

Intentem calcular-lo per a un sumand de la funció, és a dir, deixant fixat $r_{u,i}$.

$$g(p_u, q_i) = (r_{u,i} - \hat{r}_{u,i})^2 = (r_{u,i} - \bar{r} - p_u \cdot q_i)^2 \tag{4.5}$$

i expandint els $f$ components dels vectors $p_u$ i $q_i$

$$g(p_{u,1}, p_{u,2}, ..., p_{u,f}, q_{i,1}, q_{i,2}, ..., q_{i,f}) = (r_{u,i} - \bar{r} - p_{u,1} \cdot q_{i,1} - p_{u,2} \cdot q_{i,2} - ... - p_{u,f} \cdot q_{i,f})^2 \tag{4.6}$$

Calculem ara les derivades respecte components $p_{u,n}$ i $q_{i,n}$ qualsevols

$$\frac{dg}{dp_{u,n}} = -2(r_{u,i} - \bar{r} - p_{u,1} \cdot q_{i,1} - p_{u,2} \cdot q_{i,2} - ... - p_{u,f} \cdot q_{i,f})q_{i,n} = -2(r_{u,i} - \hat{r}_{u,i})q_{i,n} \tag{4.7}$$

$$\frac{dg}{dq_{i,n}} = -2(r_{u,i} - \bar{r} - p_{u,1} \cdot q_{i,1} - p_{u,2} \cdot q_{i,2} - ... - p_{u,f} \cdot q_{i,f})p_{u,n} = -2(r_{u,i} - \hat{r}_{u,i})p_{u,n} \quad (4.8)$$

Cal recordar que estem calculant les derivades de la equació 4.5, no pas de la funció objectiu sencera de 4.3. Per calcular les derivades d'aquesta darrera funció respecte $p_{u,n}$ o $q_{i,n}$ s'hauria d'aplicar les derivades de 4.7 o 4.8 a tots els sumands corresponents a les puntuacions de l'usuari $u$ o l'ítem $i$, respectivament, i la resta de sumands quedarien anulats.

No obstant, hi ha una variant d'aquest algoritme que simplifica notablement aquestes operacions, amb resultats molt similars. Es tracta de l'anomenat *stochastic gradient descent*.

**Stochastic gradient descent** Aquesta variant del *gradient descent* s'utilitza per a minimitzar funcions que estan escrites com una suma de funcions diferenciables. Fem notar que, malgrat el nom *stochastic*, l'algoritme tal com aquí el descrivim és determinista. En el *gradient descent* clàssic, a la $i$-èsima iteració actualitzaríem el punt actual $x_i$ calculant el gradient de la funció objectiu sencera sobre aquell punt. Al *stochastic gradient descent*, en canvi, si la funció està escrita com $n$ sumes de funcions diferenciables, a la iteració $i$-èsima fem $n$ actualitzacions del punt $x_i$, una per cada suma que composa la funció objectiu, calculant el gradient de cadascuna per separat. És a dir, si tenim la funció

$$F(x) = \sum_{i=1}^{n} F_i(x)$$

composta per $n$ sumes de funcions diferenciables, l'algoritme que s'aplicaria a cada iteració seria

**for** $i = 1, 2, ..., n$ **do**
  $x_{n_{it}} \leftarrow x_{n_{it}} - \alpha \nabla F_i(x_{n_{it}})$
**end for**

El mètode confia en que, malgrat els canvis, l'equació 4.4 se seguirà complint i per tant s'acabarà arribant a un mínim local.

Aquest algoritme ens permet una implementació molt senzilla per al nostre problema, ja que en aquest cas sí que podem utilitzar les equacions 4.7 i 4.8, perquè només necessitem calcular els gradients dels sumands per separat, un per cada puntuació $r_{u,i}$.

**Algoritme d'entrenament *offline*** Un cop introduït el mètode de *stochastic gradient descent*, podem detallar l'algoritme d'entrenament *offline*. El seu objectiu és entrenar el model per a poder fer prediccions de puntuacions, és a dir, calcular els vectors $p_u$ i $q_i$ per a tots els usuaris i ítems del sistema. Diem que és *offline* perquè és un procés que entrena el model des de zero, utilitzant totes les puntuacions que hi ha fins al moment al sistema. És

**Algoritme 4.1** Entrenament *offline* - *ISMF*

**Entrada:** Conjunt $R$ de totes les puntuacions del sistema
**Sortida:** Vectors característics $p_*$ i $q_*$ de tots els ítems i usuaris

1: inicialitzar els vectors de $p_*$ i $q_*$ amb valors aleatoris molt petits
2: $T \leftarrow$ extreure aleatòriament $n_{test}$ puntuacions de $R$
3: $lastError \leftarrow \infty$
4: $lastImprovement \leftarrow \infty$
5: **while** $lastImprovement > \epsilon$ **do**
6:  **for all** $r_{u,i} \in R$ **do**
7:  $\hat{r}_{u,i} \leftarrow \bar{r} + p_u \cdot q_i$
8:  $err \leftarrow r_{u,i} - \hat{r}_{u,i}$
9:  **for** $k = 1, 2, ..., f$ **do**
10:  $p'_{u,k} \leftarrow p_{u,k}$
11:  $p_{u,k} \leftarrow p_{u,k} + 2 \cdot \gamma \cdot err \cdot q_{i,k}$
12:  $q_{i,k} \leftarrow q_{i,k} + 2 \cdot \gamma \cdot err \cdot p'_{u,k}$
13:  **end for**
14:  **end for**
15:  $curError \leftarrow 0$
16:  **for all** $r_{u,i} \in T$ **do**
17:  $\hat{r}_{u,i} \leftarrow \bar{r} + p_u \cdot q_i$
18:  $curError \leftarrow curError + (r_{u,i} - \hat{r}_{u,i})^2$
19:  **end for**
20:  $lastImprovement \leftarrow lastError - curError$
21:  $lastError \leftarrow curError$
22: **end while**

a dir, aquest mètode no permet incorporar nous usuaris, nous ítems o noves puntuacions a un model ja entrenat. El pseudocodi es presenta a l'algoritme 4.1.

De $\gamma$ en direm la **constant d'aprenentatge**, i serà quelcom que tindrà força influència en el resultat de l'algoritme. Per tant, és una bona idea que a la implementació es permeti canviar amb facilitat.

Simplificant, l'algoritme va fent iteracions fins que detecta que la millora respecte la iteració anterior és massa petita. A cada iteració, s'aplica *stochastic gradient descent*, recorrent totes les puntuacions del conjunt d'entrenament $R$ i calculant les derivades tal i com s'ha vist a les equacions 4.7 i 4.8. Immediatament després, es calcula l'error de la iteració actual en predir les puntuacions del conjunt de prova $T$. Aquesta partició entre conjunt d'entrenament i conjunt de prova es fa per a poder detectar el sobreentrenament.

El sobreentrenament o *overfitting* es un fenòmen força comú a l'aprenentatge automàtic. Succeeix quan el model s'especialitza massa en les dades d'entrenament (en el nostre cas, totes les puntuacions conegudes) i acaba generalitzant malament, donant com a resultat males prediccions de les mostres noves (en el nostre cas, puntuacions desconegudes). Per tant, no seria estrany que l'error en predir les puntuacions de $R$ seguís millorant i al mateix temps l'error del conjunt $T$ empitjorant. Aquest mecanisme d'aturar l'algoritme quan detectem que les prediccions del conjunt de prova empitjoren també és conegut i s'anomena *early stopping*. La idea és que la mida d'aquest conjunt, $n_{test}$, sigui un nombre prou significatiu per a que l'error calculat sigui fiable, però al mateix temps sigui molt més petit que $|R|$.

L'algoritme és capaç d'entrenar el model en temps $O(|R| \cdot f \cdot n_{it})$. Però $f$, el nombre de factors, és constant. I les proves realitzades indiquen que el nombre d'iteracions $n_{it}$ en general és petit. Per tant, podem dir que $|R|$ és el factor determinant per a la complexitat de l'algoritme.

Per una altra banda, un cop entrenat el model cada predicció es realitza en temps $O(f)$, ja que només es necessita calcular el producte escalar dels vectors característics del corresponent usuari i ítem.

**Algoritme d'entrenament *online*** L'algoritme anterior té la limitació de no poder incorporar nous usuaris, ítems o puntuacions al model, sense haver de tornar-lo a entrenar tot sencer. Per això, necessitem un algoritme eficient capaç d'actualitzar un model ja calculat amb nova informació.

A continuació presentem dos algoritmes que, partint d'un model ja entrenat, permeten recalcular el vector característic d'un usuari o ítem. Ens basarem en l'algoritme anterior, però en comptes d'iterar sobre totes les puntuacions, ho farem només sobre les puntuacions de l'usuari o ítem en concret. D'aquesta manera, podrem incorporar nous usuaris, ítems i puntuacions al model sense haver de tornar-lo a entrenar des de zero cada cop. El pseudocodi es presenta als algoritmes 4.2 i 4.3. Les idees d'aquests dos algoritmes i de

les estratègies per a mantenir actualitzat el model de forma eficient quan arriba nova informació s'han obtingut de [10].

**Algoritme 4.2** Reentrenament d'un usuari - *ISMF*

**Entrada:** Conjunt $R_u$ de les puntuacions de l'usuari $u$, nombre d'iteracions $n_{it}$
**Sortida:** Vector cacterístic de l'usuari $p_u$

```
1: inicialitzar el vector p_u amb valors aleatoris molt petits
2: for it = 1, 2, ..., n_it do
3:    for all r_{u,i} ∈ R_u do
4:       r̂_{u,i} ← r̄ + p_u · q_i
5:       err ← r_{u,i} − r̂_{u,i}
6:       for k = 1, 2, ..., f do
7:          p_{u,k} ← p_{u,k} + 2 · γ · err · q_{i,k}
8:       end for
9:    end for
10: end for
```

A diferència de la versió *offline* vista a l'algoritme 4.1, aquí només s'actualitza el vector de l'usuari que es vol entrenar, ignorant les actualitzacions dels vectors dels ítems.

Una altra particularitat és que en aquest cas no és tan senzill calcular la millora de precisió del model a cada iteració. A l'altre algoritme es divideixen les puntuacions en un conjunt d'entrenament i un altre de prova, però aquí estem iterant sobre les puntuacions d'un usuari. Això vol dir que el nombre de puntuacions del conjunt de prova hauria de ser molt reduït, fent que les mesures de precisió fossin estadísticament poc fiables.

Per tant, hem de fixar el nombre de iteracions $n_{it}$ com un paràmetre d'entrada. Totes les proves realitzades indiquen que el nombre òptim d'iteracions disminueix a mesura que $|R_u|$ augmenta. No obstant, el $n_{it}$ òptim dependrà sempre del tipus de recomanador i de les seves dades. Per tant, la única forma d'estimar aquest nombre és mitjançant proves sobre les dades del recomanador en qüestió, per exemple entrenant molts usuaris i ítems amb diferents valors de $n_{it}$ i intentar deduïr una funció que a partir del nombre de puntuacions sobre les que s'entrena dóni el nombre òptim d'iteracions.

**Actualitzacions del model en temps real** Un cop introduïts els algoritmes per reentrenar un usuari o ítem sense haver de tornar a entrenar tot el model, podem plantejar un esquema on s'hi puguin incorporar tots els canvis en temps real. Veiem tots els possibles events i la forma de gestionar-los eficientment.

- **Arriba una nova puntuació** $r_{u,i}$: una manera de tractar aquest event seria reentrenant primer l'usuari $u$ i després l'ítem $i$. No obstant, la implementació per defecte és una mica més eficient. S'aprofita la idea de que quan més gran sigui $|R_u|$ o $|R_i|$ menys important serà reentrenar l'usuari o ítem, ja que les noves puntuacions afectaran molt poc. Aleshores, quan arriba una nova puntuació es fa en primer lloc un

**Algoritme 4.3** Reentrenament d'un ítem - *ISMF*

**Entrada:** Conjunt $R_i$ de les puntuacions de l'ítem $i$, nombre d'iteracions $n_{it}$
**Sortida:** Vector cacterístic de l'ítem $q_i$

```
1: inicialitzar el vector q_i amb valors aleatoris molt petits
2: for it = 1, 2, ..., n_it do
3:    for all r_{u,i} ∈ R_u do
4:       r̂_{u,i} ← r̄ + p_u · q_i
5:       err ← r_{u,i} − r̂_{u,i}
6:       for k = 1, 2, ..., f do
7:          q_{i,k} ← q_{i,k} + 2 · γ · err · p_{u,k}
8:       end for
9:    end for
10: end for
```

sorteig amb probabilitat $\alpha^{|R_u|}$ i en cas positiu es reentrena l'usuari $u$. A continuació, es fa un altre sorteig amb probabilitat $\alpha^{|R_i|}$ i en cas positiu es reentrena l'ítem $i$. La idea és que $\alpha$ sigui un nombre molt proper a 1, com 0.99 o 0.98.

- **Es modifica o elimina una puntuació** $r_{u,i}$: exactament de la mateixa forma que amb una nova puntuació, es fan els dos sortejos i es decideix si es reentrena l'usuari $u$ i l'ítem $i$.
- **Arriba un nou usuari** $u$: s'inicialitza el seu vector $p_u$ amb valors molt petits, de forma que totes les prediccions estiguin properes a la mitjana (no coneixem els seus gustos encara). A mesura que vagi puntuant ítems, la informació ja s'anirà incorporant al model mitjançant l'event de *nova puntuació*.
- **Arriba un nou ítem** $i$: cas anàleg al de nou usuari.
- **S'elimina un usuari** $u$: s'elimina el vector característic de l'usuari, però no reentrenem els ítems que havia puntuat.
- **S'elimina un ítem** $i$: cas anàleg a l'anterior.

Cal insistir en què aquesta és la implementació per defecte, però es dóna la opció a l'usuari de desactivar les actualitzacions automàtiques del model, de forma que pugui decidir amb criteris arbitraris quan reentrenar usuaris i ítems.

Pot cridar l'atenció que l'algoritme d'entrenament *offline* no apareixi com a resposta a cap event que es gestionen. Les proves demostren que es pot mantenir actualitzat el model només amb operacions incrementals des de zero, mantenint una qualitat de prediccions excel·lent. No obstant, l'execució de l'algoritme d'entrenament *offline* sempre proporcionarà una lleugera una millora de precisió, ja que les actualitzacions *online* són, encara que per poc, sub-òptimes. Per tant, és aconsellable, sobretot al principi de la vida del recomanador (quan hi ha poques puntuacions), anar executant un entrenament complet cada cert temps. Es deixa a criteri de l'usuari decidir si (i quan) realitzar aquests entrenaments

i per tant no s'ha tingut en compte a la implementació per defecte.

### 4.4.2 Afegint regularització - *RISMF*

A la pràctica, els algoritmes anteriors acostumen a patir d'*overfitting*, sobretot per a usuaris amb poques puntuacions. Per exemple, si estem entrenant un usuari que només ha puntuat un ítem, és d'esperar que alguns components del seu vector característic creixin desmesuradament, adaptant-se en excés a l'únic exemple d'entrenament del que es disposa, i generalitzant malament.

Una millora que ajuda a pal·liar aquests problemes consisteix en tenir en compte la norma dels vectors, a més de l'error de predicció, a l'hora d'optimitzar. Concretament, podem redefinir el problema d'optimització com:

$$\min_{p_*,q_*} \sum_{r_{u,i}\in R} (r_{u,i} - \hat{r}_{u,i})^2 + \lambda(||p_u||^2 + ||q_i||^2) \tag{4.9}$$

Amb aquesta nova magnitud es penalitza la norma dels vectors d'usuaris i ítems. Aquest mecanisme de prevenció d'*overfitting* s'anomena *regularització*. La idea intuïtiva és intentar que els vectors es mantinguin "petits", al mateix temps que es millora la precisió del model a cada pas d'optimització. La importància que se li dona a la norma dels vectors a l'hora d'optimitzar està controlada per la constant $\lambda$, anomenada *factor de regularització*.

Procedim de forma anàloga a l'apartat anterior, calculant les derivades respecte factors $p_{u,n}$ i $q_{i,n}$ qualsevols, només d'un sumand (suposant $r_{u,i}$ fixat).

$$g(p_u, q_i) = (r_{u,i} - \hat{r}_{u,i})^2 = (r_{u,i} - \bar{r} - p_u \cdot q_i)^2 + \lambda(||p_u||^2 + ||q_i||^2) \tag{4.10}$$

i expandint els $f$ components dels vectors $p_u$ i $q_i$

$$g(p_{u,1}, p_{u,2}, ..., p_{u,f}, q_{i,1}, q_{i,2}, ..., q_{i,f}) = (r_{u,i} - \bar{r} - p_{u,1} \cdot q_{i,1} - p_{u,2} \cdot q_{i,2} - ... - p_{u,f} \cdot q_{i,f})^2 + \lambda({p_{u,1}}^2 + {p_{u,2}}^2 + ... + {p_{u,n}}^2 + {q_{i,1}}^2 + {q_{i,2}}^2 + ... + {q_{i,n}}^2) \tag{4.11}$$

$$\frac{dg}{dp_{u,n}} = -2(r_{u,i} - \bar{r} - p_{u,1} \cdot q_{i,1} - p_{u,2} \cdot q_{i,2} - ... - p_{u,f} \cdot q_{i,f})q_{i,n} + 2\lambda p_{u,n} = -2(r_{u,i} - \hat{r}_{u,i})q_{i,n} + 2\lambda p_{u,n} \tag{4.12}$$

$$\frac{dg}{dq_{i,n}} = -2(r_{u,i} - \bar{r} - p_{u,1} \cdot q_{i,1} - p_{u,2} \cdot q_{i,2} - ... - p_{u,f} \cdot q_{i,f})p_{u,n} + 2\lambda q_{i,n} = -2(r_{u,i} - \hat{r}_{u,i})p_{u,n} + 2\lambda q_{i,n} \tag{4.13}$$

Aleshores, per aplicar la regularització només cal modificar els algoritmes de l'apartat anterior de forma que utilitzin aquestes dues darreres fórmules per actualitzar els vectors d'usuaris i ítems. Obtindrem el *RISMF*, o *Regularized Incremental Simultaneous Matrix Factorization*.

En aquest cas, de $\lambda$ en direm **factor de regularització**. Igual que la constant d'aprenentatge, tindrà importància en els resultats de l'algoritme, per tant la implementació haurà de permetre que es pugui canviar fàcilment.

### 4.4.3 Afegint biaixos - *BRISMF*

A la pràctica, sovint succeeix que hi ha usuaris que tendeixen a puntuar per sobre o per sota de la mitjana, i anàlogament ítems que poden ser molt (o gens) populars. És a dir, podem dir que usuaris i ítems poden tenir un cert biaix per sí mateixos. Una manera de realitzar les prediccions tenint en compte aquest biaix seria amb

$$\hat{r}_{u,i} = b_u + b_i + \bar{r} + p_u \cdot q_i \tag{4.14}$$

on $b_u$ denota el biaix de l'usuari $u$ i $b_i$ el de l'ítem $i$.

No obstant, per a la implementació s'ha utilitzat un truc consistent en deixar fixat a 1 el primer component dels vectors d'usuaris i el segon component dels vectors d'ítems, deixant-los sense entrenar per tal que es mantinguin sempre amb aquest valor. Així, a l'hora de fer el producte escalar obtindrem

$$p_u \cdot q_i = q_{i,1} \cdot 1 + 1 \cdot p_{u,2} + p_{u,3} \cdot q_{i,3} + p_{u,4} \cdot q_{i,4} + ... + p_{u,f} \cdot q_{i,f}$$

i $p_{u,2}$ estarà fent de biaix de l'usuari $u$ i $q_{i,1}$ de l'ítem $i$. D'aquesta manera, podem incorporar els biaixos sense haver de canviar la manera com es fan les prediccions de puntuacions i les actualitzacions dels vectors als algoritmes d'optimització.

Aquest és l'algoritme definitiu, obtingut a partir d'aplicar les millores de regularització i biaixos a l'algoritme bàsic *ISMF*.

## 4.5 Comparativa de l'eficiència dels algoritmes

La taula a continuació compara el temps que triguen les diferents implementacions en tractar els principals events que poden succeïr al recomanador. Suposarem que quan arriba una nova puntuació s'entrena el corresponent usuari i ítem sempre, cosa que no és exactament certa a la implementació per defecte, com s'indica al pàrraf *Actualitzacions del model en temps real* de l'apartat 4.4.1.

| | **Baseline** | **UserBased** | **ItemBased** | **BRISMF** |
|---|---|---|---|---|
| **Predicció de puntuació** $\hat{r}_{u,i}$ | $O(1)$ | $O(\|U\|\|\bar{R}_{u'}\|)$ | $O(\|I\|\|\bar{R}_{i'}\|)$ | $O(1)$ |
| **Nova puntuació** $r_{u,i}$ | $O(1)$ | $O(1)$ | $O(1)$ | $O(\|R_u\| + \|R_i\|)$ |
| **Nou usuari** $u$ | $O(\|R_u\|)$ | $O(\|R_u\|)$ | $O(\|R_u\|)$ | $O(\|R_u\|)$ |
| **Nou ítem** $i$ | $O(\|R_i\|)$ | $O(\|R_i\|)$ | $O(\|R_i\|)$ | $O(\|R_i\|)$ |
| **S'elimina un usuari** $u$ | $O(\|R_u\|)$ | $O(\|R_u\|)$ | $O(\|R_u\|)$ | $O(\|R_u\|)$ |
| **S'elimina un ítem** $i$ | $O(\|R_i\|)$ | $O(\|R_i\|)$ | $O(\|R_i\|)$ | $O(\|R_i\|)$ |
| **Entrenament del model sencer** | $O(\|R\|)$ | $O(\|R\|)$ | $O(\|R\|)$ | $O(\|R\|)$ |

Taula 4.1: Comparativa de la complexitat per gestionar els events del recomanador

Recordem que $|U|$ denota el nombre d'usuaris del recomanador, i $|I|$ el de ítems. També, utilitzem $|\bar{R}_{u'}|$) per denotar el nombre mitjà de puntuacions d'un usuari qualsevol, i $|\bar{R}_{i'}|$) el d'un ítem.

L'algoritme més eficient és el *baseline*, com era d'esperar. De fet, la única diferència en complexitat rau en la predicció d'una puntuació i en tractar una nova puntuació. A l'hora de realitzar prediccions, els algoritmes *UserBased* i *ItemBased* són molt més lents que el *BRISMF*, havent d'iterar sobre tots els usuaris o ítems, i calcular-ne cadascun la similaritat amb el propi usuari o ítem del que es fa la predicció.

En canvi, tracten les noves puntuacions en temps constant, mentre que l'algoritme *BRISMF* ho fa en temps $O(|R_u| + |R_i|)$. Tot i així, la operació de predicció de puntuacions es realitzarà molt més cops que la de nova puntuació, i per una altra banda no és estrictament necessari entrenar l'usuari i ítem cada cop que arriba una nova puntuació, amb la qual cosa la complexitat es pot reduïr encara més. Per tant, podem dir que l'algoritme *BRISMF* és molt més eficient i escalable per a les nostres necessitats que els altres algoritmes, sense comptar el *baseline*, que és clarament inferior en quant a qualitat de prediccions.

De moment només podem fer una comparativa a nivell teòric, però a l'apartat *Experiments i avaluació* es comparen en més detall, tenint en compte aspectes pràctics que no es poden tenir en compte aquí, com la qualitat de les prediccions i recomanacions.

## 4.6 Algoritmes i idees descartades

En aquest apartat es comentaran algunes idees de les que es van provar abans d'arribar a l'algoritme *BRISMF*, però que no van donar els resultats esperats. No s'entrarà en un detall excessiu, només per tenir una idea del què s'intentava i per què no va funcionar.

### 4.6.1 *Locality-sensitive hashing*

Anomenem anomenem funció *hash* a una funció que assigna elements d'un conjunt d'entrada, típicament infinit, a un conjunt de sortida finit:

$$H\colon A \to B$$

on $|A| \gg |B|$ i $B$ és finit.

Algunes d'aquestes funcions poden satisfer certes propietats. Per exemple, una funció hash pot complir que tots els elements del conjunt de sortida tenen la mateixa probabilitat de ser assignats un element d'entrada qualsevol (els elements d'entrada es reparteixen uniformement als elements de sortida). O per exemple, si el conjunt d'entrada són els nombres naturals, i el de sortida un subconjunt del conjunt d'entrada, podria ser interessant que petites variacions en dos nombres d'entrada donin grans diferències en els seus respectius nombres de sortida.

El *Locality-sensitive hashing* (*LSH*) és un tipus de *hashing* on es fan servir funcions *hash* que compleixen amb que dos elements del conjunt d'entrada seran assignats als mateix conjunt de sortida amb una probabilitat proporcional a la seva similaritat.

Formalitzant, podem considerar que els elements mencionats són en realitat punts a $\mathbb{R}^d$. Sigui $sim\colon \mathbb{R}^d \times \mathbb{R}^d \to [0,1]$ una funció de similaritat entre dos elements, que prendria un valor real entre 0 i 1, amb 0 en el cas que fossin totalment diferents i 1 en el cas oposat. Aleshores, anomenem *esquema de funcions LSH* a una família de funcions $H$ tal que si es tria una funció $h \in H$ a l'atzar, es satisfa la propietat $Pr_{h\in H}[h(a) = h(b)] = sim(a,b)$ per tot $a, b \in \mathbb{R}^d$.

Una de les aplicacions més comunes d'aquest algoritme consisteix en la cerca eficient dels $K$ elements més semblants a un cert element. Si recordem els algoritmes de filtratge col·laboratiu basats en usuaris o ítems de l'apartat 4.2 veurem que, a priori, poder realitzar aquest tipus d'operacions de forma eficient ens pot resultar molt útil per al càlcul de les prediccions de puntuacions. Per exemple, podem representar fàcilment un usuari en aquest espai euclidià: un vector on a l'$i$-èsim component hi apareixi la puntuació que l'usuari li ha donat a l'$i$-èsim ítem, o 0 en cas que no l'hagi puntuat. Aplicant aquest esquema podríem obtenir una bona aproximació dels $K$ usuaris o ítems més semblants a un cert usuari o ítem, en temps $O(K)$.

És evident que les funcions *hash* a utilitzar han dependran de la funció de similaritat. En aquest cas, per comparar dos usuaris o ítems havíem triat la similaritat cosinus entre els seus respectius vectors. Construirem, per tant, una família de funcions *hash* compatibles amb la similaritat cosinus.

**Funcions LSH basades en la projecció aleatòria** La idea intuïtiva és que si tenim dos punts a $\mathbb{R}^d$ i un hiperplà, si els punts són similars (l'angle entre els vectors que

defineixen és petit) quedaran amb molta probabilitat al mateix "costat" de l'hiperplà, mentre que si són molt diferents és molt probable que quedin separats per ell.

Podem representar un hiperplà en aquest espai com un vector a $\mathbb{R}^d$ normalitzat. Es pot comprovar fàcilment a quin costat queda un punt respecte aquest hiperplà, a partir del signe del seu producte escalar. Aleshores, sigui $v \in \mathbb{R}^d$ un vector d'entrada, i $r \in \mathbb{R}^d$ un vector normal que representa un hiperplà, podem definir la funció $h(v, r) = signe(v \cdot r)$, on *signe* val 0 en cas que el nombre sigui negatiu o igual a 0, i 1 altrament.

Amb això, aconseguim una família de funcions LSH compatible amb la similaritat cosinus. Concretament, aquesta família tindrà tantes funcions com vectors unitaris a $\mathbb{R}^d$.

**Aproximació eficient als $K$ veïns més propers** Donada una família $F$ de funcions LSH, podem construïr una estructura de dades que ens permeti obtenir de forma eficient una aproximació als $K$ elements més propers donat un element d'entrada.

L'algoritme té dos paràmetres: l'amplitud $k$ de les funcions hash i el nombre de taules hash $L$. En un primer pas, construïm una nova família $H'$ de $L$ funcions hash, on cada funció $h' \in H'$ es construeix seleccionant $k$ funcions aleatòries de $H$ i concatenant-les. Pensant en termes pràctics, si les funcions de $H$ fossin com les mencionades a l'apartat anterior, cadascuna contribuïria amb un bit a cada funció $h'$, retornant, per tant, nombres de $k$ bits (naturals a l'interval $[0, 2^k - 1]$). L'algoritme mantindrà $L$ taules hash, aplicant a la taula i-èsima la funció $h'_i \in H'$ corresponent.

Per cada nou element que s'hagi d'introduïr, l'algoritme l'emmagatzemarà a cada taula, al forat que indiqui la funció hash corresponent. Donat un element $a$, per a trobar-ne els $K$ més semblants l'algoritme cercarà, a cada taula, al forat on la funció hash assigna l'element $a$, i de tots els elements trobats es quedarà amb els $k$ més propers. La reducció respecte comparar tots els elements del sistema pot ser important.

Cada taula tindrà exactament $|B|^k$ forats on repartir els elements d'entrada, on $B$ és el conjunt de sortida de cada funció hash de la família $F$. La tria de $k$ i $L$ tindrà un impacte important tant en el rendiment de l'algoritme com en la qualitat de l'aproximació. Als algoritmes 4.4 i 4.5 es presenta el pseudocodi per a la inserció de nous ítems a l'estructura de dades que proposem (l'eliminació seria anàloga) i per a obtenir eficientment una bona aproximació al problema dels $K$ veïns més propers.

**Algoritme 4.4** Introducció d'un element a l'estructura de dades LSH

**Entrada:** Un nou element $e$, l'estructura $t$ de $L$ taules hash

1: **for** $i = 1, 2, ..., L$ **do**
2: $\quad n \leftarrow h'_i(e)$
3: $\quad t_{i,n} \leftarrow t_{i,n} \cup e$
4: **end for**

**Algoritme 4.5** Aproximació als $K$ veïns més propers

**Entrada:** Un element $e$, l'estructura $t$ de $L$ taules hash
**Sortida:** Una llista $l$ dels $K$ elements més propers a $e$ (aproximació)

```
1: l ← []
2: for i = 1, 2, ..., L do
3:    n ← h'_i(e)
4:    for e' ∈ t_{i,n} do
5:       if |l| < K then
6:          l ← l ∪ e'
7:       else if ∃e'' ∈ l : sim(e, e') > sim(e, e'') then
8:          l ← l − e''
9:          l ← l ∪ e'
10:      end if
11:   end for
12: end for
```

**Resultats pràctics** L'aplicació d'aquest algoritme al nostre problema (el de trobar els $K$ usuaris o ítems més propers a un de donat) va resultar a la pràctica poc satisfactòria. En primer lloc, per a que l'algoritme donés aproximacions prou bones, es van haver de triar combinacions amb $k$ i $L$ prou grans com perquè el cost d'avaluar les funcions hash no fos gens despreciable. En definitiva, no valia la pena, en termes d'eficiència, pagar el cost extra de mantenir els elements actualitzats dintre d'aquesta estructura de dades cada cop que aquests cambiaven (nous ítems, nous usuaris, noves puntuacions), per un petit guany a l'hora de realitzar prediccions de puntuacions (on es requeria trobar elements similars a un de donat de forma eficient).

Per una altra banda, ens vam adonar que aquesta operació en realitat no resultava tan útil a l'hora de predir les puntuacions. Per exemple, en el cas del filtratge col·laboratiu basat en usuaris, per a estimar la puntuació d'un usuari $u$ a un ítem $i$ necessitem usuaris similars a $u$ però que al mateix temps hagin puntuat $i$. A la pràctica, a no ser que $K$ sigui molt elevat, dels $K$ usuaris més semblants a $u$ molt pocs hauran puntuat $i$, sobretot si $i$ és un ítem prou rar. Això fa que les prediccions realitzades triant els usuaris més similars sense tenir en compte quins d'aquests han puntuat l'ítem no siguin especialment bones. De forma anàloga passa amb l'algoritme basat en ítems.

Per aquests motius, com s'ha indicat al principi de l'apartat, es va decidir descartar aquest algoritme i es va continuar investigant altres alternatives.

### 4.6.2 *Singular Value Decomposition*

La descomposició en valors singulars (*Singular Value Decomposition, SVD*) és un tipus de factorització que descomposa una matriu $M \in \mathbb{R}^{n\times m}$ en $M = USV$, on $U \in \mathbb{R}^{n\times n}, V \in \mathbb{R}^{m\times m}$ són matrius ortogonals i $S \in \mathbb{R}^{n\times m}$ és diagonal, i amb nombres no negatius. Els

elements no nuls de $S$ s'anomenen valors singulars de $M$, i podem suposar sense pèrdua de generalitat que estan ordenats de major a menor, de dalt a baix, a la matriu. De fet, la SVD es pot definir sobre una matriu de complexos, però per simplicitat només contemplarem el cas dels reals.

Aquesta descomposició té un gran nombre d'aplicacions i propietats, però ens centrarem en les realment rellevants per al problema de recomanació. Una d'aquestes propietats és la següent. Suposem que tenim una certa descomposició SVD $M = USV$, de la qual ens quedem amb els $k \ll min(n, m)$ primers valors singulars (els $k$ més grans). És a dir, ens quedem només amb les $k$ primeres columnes de $U$, amb les $k$ primeres columnes i files de $S$ i les $k$ primeres files de $V$. Aleshores, es compleix que la matriu resultant de la nova descomposició $M' = U'S'V'$ és la millor aproximació de $M$ que es pot aconseguir amb una matriu de rang com a molt $k$, en termes de $RMSE$ (recordem l'apartat 4.4).

Suposem ara que $M$ és la nostra matriu de puntuacions, amb una fila per cada usuari i una columna per cada ítem. Aleshores, amb l'aproximació $U'S'V'$ tenim una factorització de la mateixa forma que la de l'algoritme *BRISMF*, on $U'S' \in \mathbb{R}^{n\times k}$ contindria els $n$ vectors característics de cada usuari i $V'^T \in \mathbb{R}^{m\times k}$ els $m$ vectors característics de cada ítem, amb $k$ factors cadascun. De la mateixa forma, la predicció d'una puntuació es faria fent el producte escalar del vector d'un usuari pel d'un ítem.

No obstant, al contrari que a l'algoritme *BRISMF*, a l'hora de fer la factorització s'ha de tenir especial cura amb les puntuacions desconegudes, els buits de la matriu. Amb la SVD es factoritza la matriu complerta (no com al *BRISMF*, on només s'utilitzen les puntuacions conegudes), per tant, no podem substituïr les puntuacions desconegudes per zeros, ja que la descomposició final quedaria totalment distorsionada.

Així, és obligatori omplir els buits emprant algun tipus d'aproximació d'aquestes puntuacions, com per exemple l'algoritme *baseline*. La idea és realitzar la descomposició *SVD* d'aquesta matriu on s'han substituït els buits per aproximacions, i aleshores truncar-la, quedant-nos amb els $k$ factors més importants. S'espera que amb aquest truncament les relacions entre les puntuacions conegudes predominin, ja que les puntuacions omplertes no aportaran informació nova, al haver estat calculades a partir de les puntuacions conegudes amb algun algoritme trivial. Si aquestes relacions es generalitzen prou bé, les puntuacions no conegudes que apareixeran a la matriu aproximada (la obtinguda pel truncament de la factorització) seran bones prediccions.

La manera innocent de calcular la factorització esmentada seria en primer lloc calcular la SVD sencera i aleshores truncar-la. Es pot calcular la SVD d'una matriu $n \times m$ en temps $O(nm^2)$, amb $n > m$. Evidentment, per les dimensions que considerem, i tenint en compte la complexitat, aquest procediment seria excessivament lent.

No obstant, existeixen mètodes que s'aprofiten del fet que no necessitem la descomposició SVD sencera, sinó només els $k$ valors singulars més grans. En el nostre cas, vam estudiar un mètode incremental per al càlcul de SVD's truncades vist a [11].

La idea d'aquest algoritme és permetre, a partir d'una factorització $M' = U'S'V'$ SVD

truncada a $k$ factors ($U' \in \mathbb{R}^{n\times k}, V' \in \mathbb{R}^{k\times m}, S' \in \mathbb{R}^{k\times k}$), realitzar actualitzacions tant en la matriu modificada com en la seva factorització SVD de forma eficient.

Concretament, permet fer modificacions de columnes (afegir, eliminar, modificar una columna) en temps $O(nk)$ i per tant, modificacions de files en temps $O(mk)$. Per tant, és força eficient si $k$ és prou petit i el considerem constant, com el nostre cas.

No entrarem en els detalls de l'algoritme, que es basa en tota una sèrie d'operacions matricials força complicades, i que es pot consultar en detall a [11].

**Resultats pràctics** En un esquema *offline*, on volem entrenar el model des de zero amb totes les puntuacions disponibles, l'algoritme trigaria $O(nmk)$, on $k$ és el nombre de factors que decidim que tingui el nostre model, que es pot considerar una constant petita. Recordem que, al contrari que al *BRISMF*, aquí hem de factoritzar la matriu sencera, omplint les puntuacions desconegudes amb estimacions simples. Aquest argument ja seria prou fort com per abandonar aquest algoritme a favor del *BRISMF*, però en el moment en què es va provar encara es desconeixia aquest algoritme.

Avaluem ara l'adequació de l'algoritme per a un funcionament incremental, com és el que nosaltres considerem. És a dir, ens situem un context on poden arribar constantment events de noves puntuacions, on alguns es traduïran també en events de nou usuari o ítem (quan arriba la primera puntuació de qualsevol d'ells). Per a mantenir actualitzat el nostre model, hem de mantenir actualitzada la nostra matriu de puntuacions i la seva factorització *SVD* truncada a $k$ factors.

Si arriba una nova puntuació, l'algoritme incremental mencionat ens dóna la opció de modificar la corresponent fila (usuari) o columna (ítem) de la matriu de puntuacions i la seva corresponent factorització en temps $O(n+m)$. No és massa positiu el fet que es trigui exactament el mateix en fer una modificació d'una única puntuació que en canviar la fila o columna sencera. Podem millorar aquest inconvenient si en comptes d'actualitzar la columna de l'ítem o la fila de l'usuari cada cop que arribi una puntuació, ignorem algunes actualitzacions, possiblement en funció del nombre de puntuacions que ja tingui l'usuari o ítem (d'una forma similar a l'esquema d'actualitzacions de l'algoritme *BRISMF*, quan més puntuacions tingui l'usuari o ítem menys sovint cal actualitzar).

Des del punt de vista de la qualitat de les prediccions obtingudes, aquestes eren lleugerament superiors (tot i que molt similars) a les que es podien obtenir, usant les mateixes dades, amb algoritmes de filtratge col·laboratiu basats en usuaris o ítems. No obstant, per la lentitud de l'algoritme a la pràctica resultava poc adequat per a un context en temps real, on el model s'ha de mantenir actualitzat en tot moment. Per això, es va descartar i es va decidir seguir investigant.

Una reflexió interessant que es va fer quan es va descobrir l'algoritme *BRISMF* és que com a mínim havia de ser tan bo en termes de qualitat de les prediccions com el model d'una *SVD* truncada. Si al model *BRISMF*, li desactivem la regularització, desactivem el mecanisme de *early stopping* (seguim optimitzant fins que el *RMSE* sigui mínim) i en comptes

d'optimitzar a partir de les puntuacions conegudes, emprem també les desconegudes (amb la mateixa imputació de valors utilitzada al *SVD*) hauriem d'obtenir el mateix model que amb l'algoritme vist a aquest apartat, ja que en els dos casos s'hauria d'obtenir la millor aproximació possible de la matriu en termes de *RMSE*. És a dir, l'algoritme *BRISMF* inclou el d'aquest apartat, tot i que evidentment dóna possibilitats molt més interessants que permeten entrenar només amb les puntuacions conegudes i evitar el sobreentrenament.

# Capítol 5

# Disseny

En aquest apartat intentarem detallar el procés de presa de decisions i els raonaments que s'han seguit fins a arribar al disseny de l'arquitectura del sistema, i al mateix temps fer una breu descripció de les seves principals classes.

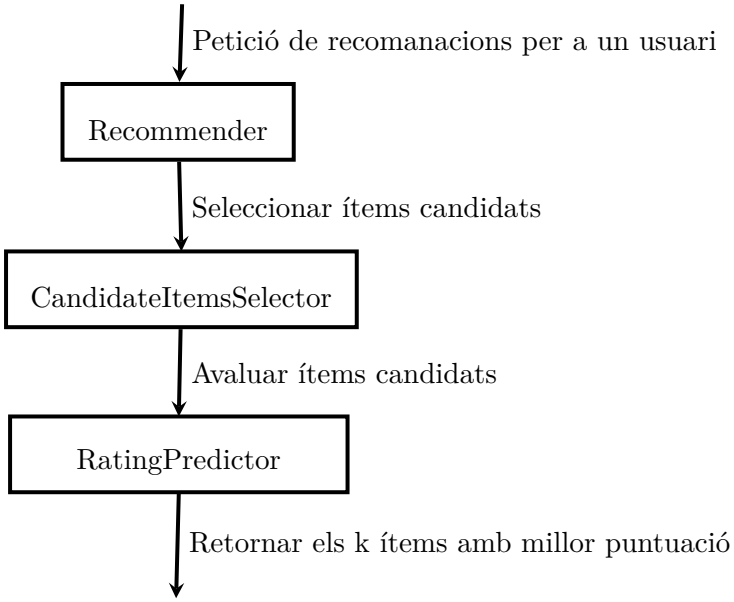
Per al disseny del recomanador, s'ha partit de la definició donada a la secció 2.3. Això resumeix el procés de recomanació en els següents passos:

1. Seleccionar un subconjunt d'ítems candidats a recomanar per a l'usuari

2. Fer una estimació de la puntuació que l'usuari li donaria a cadascun d'ells

3. Recomanar els $k$ ítems amb puntuació estimada més alta

Figura 5.1: Diagrama del procés per a generar recomanacions




D'això ja se'n poden deduïr unes primeres classes. Seguint la premisa d'un disseny modular, tindrem en primer lloc la classe del recomanador (*Recommender*), que tindrà associada una classe encarregada de predir puntuacions d'usuaris a ítems (implementant la interfície *RatingPredictor*) i una classe per a seleccionar un subconjunt d'ítems candidats a recomanar (implementant la interfície *CandidateItemsSelector*).

Pot semblar que amb aquest disseny del recomanador s'està arribant a un nivell d'abstracció innecessari, ja que, per exemple, la implementació de la classe *Recommender* que en resulta és completament trivial. No obstant, ens ha semblat que el fet que tota la complexitat vagi a parar en la forma com les puntuacions són predites, i en menor mesura en com els ítems candidats a recomanar són triats, és coherent amb la definició de recomanador que utilitzem, on el nucli és precisament aquesta tasca. Per tant, s'ha triat aquest disseny per ser conceptualment molt clar respecte l'esquema de recomanador que proposem.

Avançant-nos una mica, podem pensar en les necessitats de les classes que implementin *RatingPredictor* i *CandidateItemsSelector*. És evident que necessitaran accedir a les dades del recomanador (usuaris, ítems, puntuacions, puntuacions mitjanes d'usuaris i ítems...). Per una altra banda, l'accés a aquests dades ha d'estar centralitzat d'alguna forma, ja que en cas contrari estaríem replicant informació. Aquí és on entra en joc la interfície *RecommenderData*. Aquesta interfície defineix totes les operacions de consulta i modificació de les dades del recomanador.

A continuació es farà una descripció de les principals classes i interfícies de la llibreria. Per claredat, s'han omès les especificacions detallades amb diagrames, llista d'atributs i operacions de la majoria de components, exceptuant-ne alguns que s'han considerat

especialment rellevants. Per a una especificació més complerta, vegeu l'apèndix A.

## 5.1 Classes principals

### 5.1.1 Classe *Recommender*

És la classe que implementa les funcionalitats d'un recomanador. Tindrà associades una instància de *CandidateItemsSelector*, per a poder triar el conjunt d'ítems candidats, i una altra de *RatingPredictor*, per a poder predir les seves puntuacions.

Totes les funcions de la classe *Recommender* estan implementades (no és una classe abstracta ni interfície). Seran, per tant, les diferents implementacions de *RatingPredictor* i *CandidateItemsSelector* les que defineixin el comportament del recomanador.

### 5.1.2 Interfície *RatingPredictor*

Aquesta interfície serà utilitzada per la classe *Recommender* per a predir les puntuacions desconegudes. La seva implementació és la part més crítica del recomanador. En bona part, serà el que el defineixi en quant a nivell de les recomanacions, escalabilitat, eficiència...

Tot i ser una interfície, s'ha inclòs l'operació *getData*, ja que s'ha suposat que totes les classes que la implementin tindran associades una instància de *RecommenderData* per a poder accedir a les dades del recomanador (puntuacions, estadístiques d'usuaris i ítems...).

Les classes següents implementen la interfície *RatingPredictor* segons els algoritmes detallats a la secció 4. Totes hereden les operacions de *RatingPredictor*, però algunes n'implementen d'addicionals.

#### 5.1.2.1 Classe *BaselinePredictor*

Aquesta classe implementa l'algoritme simple de la secció 4.1.

#### 5.1.2.2 Classe *UserBasedPredictor*

Aquesta classe implementa l'algoritme de filtratge col·laboratiu basat en usuaris de la secció 4.2. Té associada per una banda una instància de la interfície *Similarity*, que li permetet calcular la similaritat entre dos usuaris, i per una altra una instància de *SimilarUsersSelector*, per a seleccionar un s ubconjunt d'usuaris similars sobre el qual es realitzarà el sumatori de la fórmula de la secció 4.2.

Aquest disseny persegueix la genericitat i modularitat, ja que ens interessa poder implementar qualsevol tipus de similaritat entre usuaris, i qualsevol estratègia per a seleccionar usuaris similars.

#### 5.1.2.3 Classe *ItemBasedPredictor*

Aquesta classe implementa l'algoritme de filtratge col·laboratiu basat en ítems de la secció 4.3, anàleg al *UserBasedPredictor*. També té associada una instància de *Similarity*, que li permet calcular en aquest cas la similaritat entre dos ítems. I anàlogament, la instància associada de *SimilarItemsSelector* li permetrà escollir un conjunt d'ítems similars sobre el qual es farà el sumatori de la fórmula de la secció 4.2.

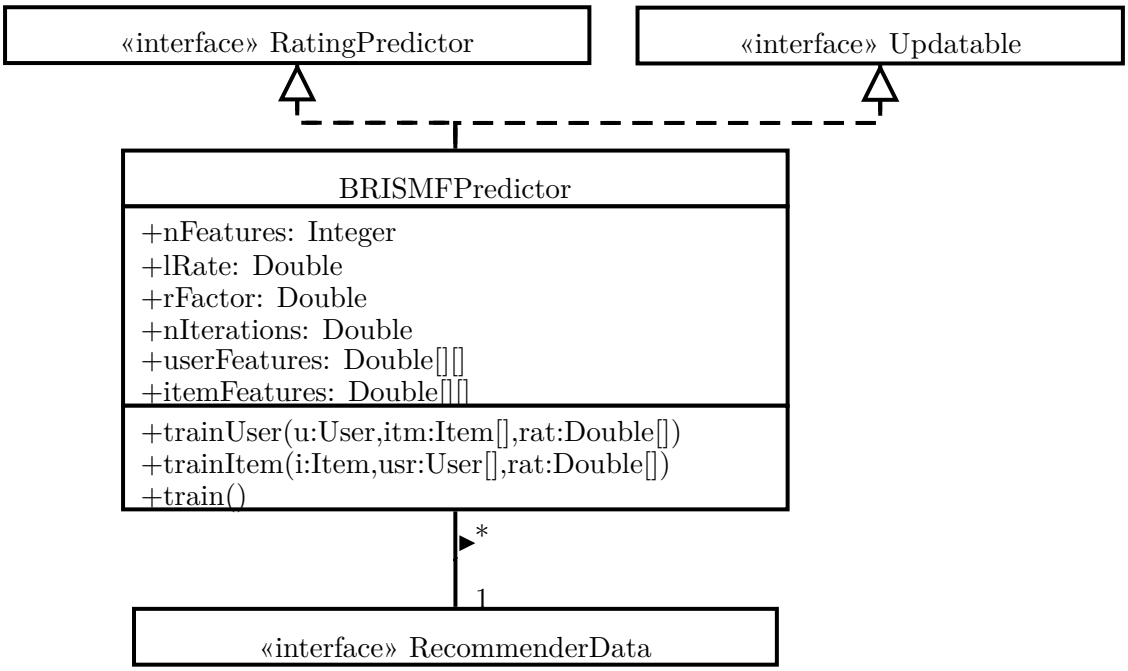
#### 5.1.2.4 Classe *BRISMFPredictor*

Aquesta classe implementa l'algoritme *BRISMF* de l'apartat 4.4. És la opció preferida en quant a qualitat de prediccions, escalabilitat i eficiència. Fent una excepció, es fàra una breu descripció de les operacions que conté (addicionals a les heredades de la interfície *RatingPredictor*), ja que es consideren prou rellevants.

A més d'implementar la interfície *RatingPredictor*, implementa *Updatable*, de forma que pot ser notificada automàticament per un *RecommenderData* quan hi ha un canvi en les dades del sistema (veure A.1.4 i A.2.6). També es farà una breu descripció del comportament d'algunes operacions heredades d'aquesta interfície.

A la descripció de les operacions es farà referència a conceptes explicats a l'algoritme, de forma que s'assumirà que el lector està familiaritzat amb els mateixos.

Figura 5.2: Diagrama de la classe *BRISMFPredictor*




**Atributs**

- **nFeatures:** Nombre de factors (dimensió dels vectors característics d'usuaris i ítems).
- **lRate:** Constant d'aprenentatge.
- **rFactor:** Factor de regularització.
- **nIterations:** Nombre d'iteracions utilitzat per l'entrenament *online* d'usuaris i ítems.
- **userFeatures:** conté els vectors característics de tots els usuaris.
- **itemFeatures:** conté els vectors característics de tots els ítems.

**Operacions**

- **trainUser(u:User,itm:Item[],rat:Double[]):** entrena el vector característic d'un usuari, utilitzant *nIterations* iteracions.
- **trainItem(i:Item,usr:User[],rat:Double[]):** entrena el vector característic d'un ítem, utilitzant *nIterations* iteracions.
- **train():** entrena el model sencer, és a dir, tots els vectors d'usuaris i ítems.

**Operacions heredades de *Updatable***

- **updateNewUser:** entrena el vector del nou usuari.
- **updateNewItem:** entrena el vector del nou ítem.
- **updateRemoveUser:** elimina el vector de l'usuari.
- **updateRemoveItem:** elimina el vector de l'ítem.
- **updateSetRating:** fa el que s'especifica al pàragraf *Actualitzacions en temps real* 4.4.1, és a dir, entrena l'usuari amb probabilitat funció del nombre de puntuacions del mateix, i entrena l'ítem amb probabilitat funció del nombre de puntuacions del mateix.
- **updateRemoveRating:** exactament el mateix que *updateSetRating*, però eliminant la puntuació en comptes d'afegint-la.

### 5.1.3 Interfície *CandidateItemsSelector*

Aquesta interfície s'utilitza per a seleccionar un subconjunt de $k$ ítems candidats a ser recomanats a un usuari $u$. El criteri per a seleccionar els ítems candidats dependrà de la implementació realitzada. A la llibreria, s'inclouen dues implementacions: *SamplingCandidateSelector* i *RandomCandidateSelector*.

#### 5.1.3.1 Classe *RandomCandidateSelector*

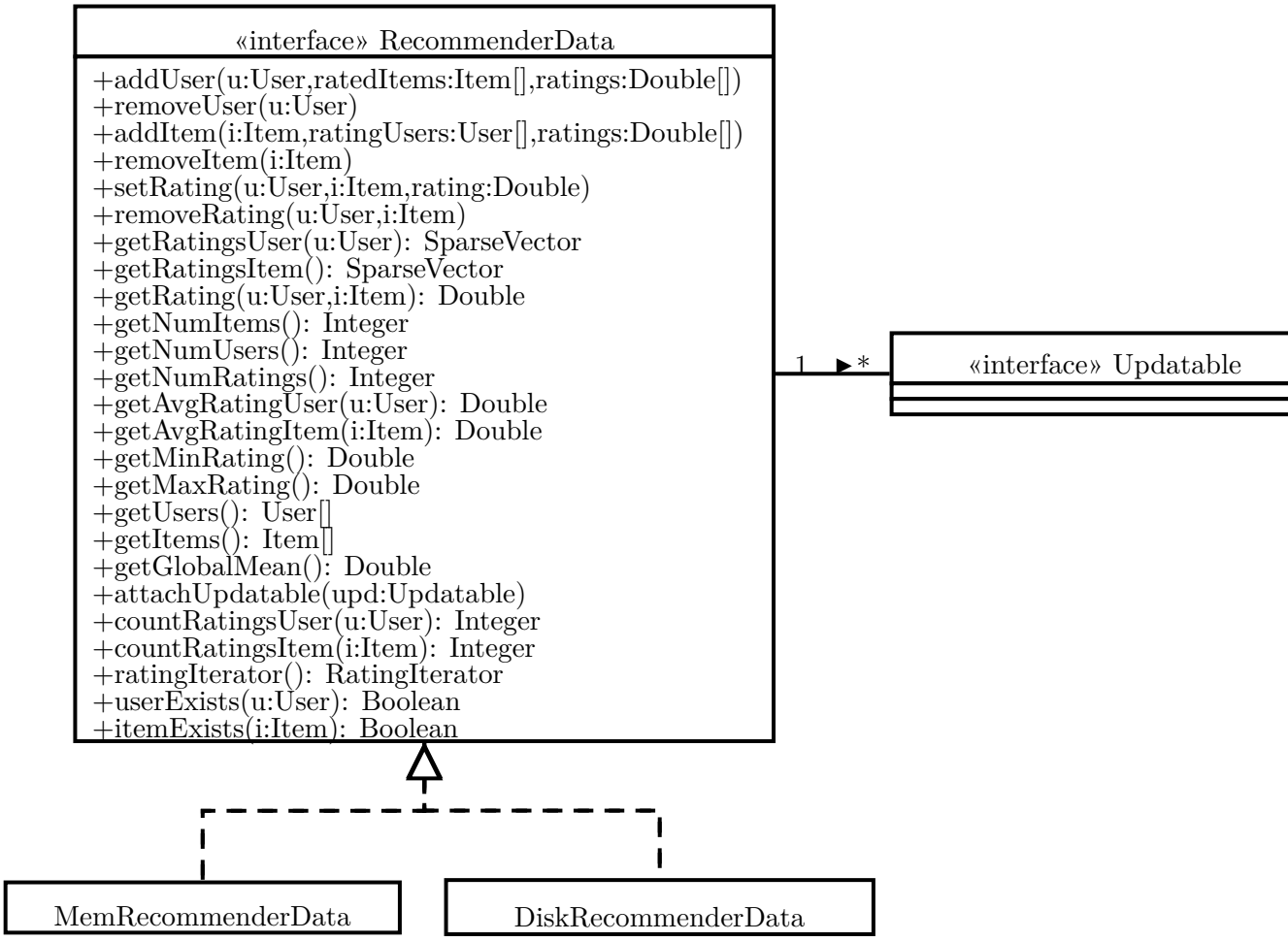
Per generar una llista de candidats, aquesta implementació en tria aleatòriament $k$ d'entre tots els ítems que no han estat puntuats per l'usuari $u$.

#### 5.1.3.2 Classe *MostRatedCandidateSelector*

Per a generar una llista de $k$ ítems candidats per a un usuari $u$, es trien els $k$ ítems amb un nombre de puntuacions més elevat que no hagin estat puntuats per l'usuari $u$.

### 5.1.4 Interfície *RecommenderData*

Figura 5.3: Diagrama de la interfície *RecommenderData*




La interfície *RecommenderData* proporciona les operacions necessàries per a consultar i modificar les dades del recomanador. Tota classe que implementi *RecommenderData* podrà tenir associats una sèrie d'instàncies de *Updatable*. Aquestes instàncies seran notificades quan s'executin certes operacions al *RecommenderData*, seguint d'alguna forma el conegut *patró observador* d'enginyeria del software. La interfície *Updatable* s'explica amb més detall a l'apartat A.2.6.

#### Operacions

- **addUser(u:User,ratedItems:Item[],ratings:Double[]):** afegeix un usuari al sis-

tema, possiblement amb algunes puntuacions inicials (llista d'ítems i llista de les seves respectives puntuacions). S'executa la operació *updateNewUser* per a tots els *Updatable* associats.

- **removeUser(u:User):** elimina un cert usuari del sistema. S'executa la operació *updateRemoveUser* per a tots els *Updatable* associats.
- **addItem(i:Item,ratingUsers:User[],ratings:Double[]):** afegeix un ítem al sistema, possiblement amb algunes puntuacions inicials (llista d'usuaris i llista de les seves respectives puntuacions). S'executa la operació *updateNewItem* per a tots els *Updatable* associats.
- **removeItem(i:Item):** elimina un ítem del sistema. S'executa la operació *updateRemoveItem* per a tots els *Updatable* associats.
- **setRating(u:User,i:Item,rating:Double):** introdueix una nova puntuació al sistema, és a dir: l'usuari que puntua, l'ítem puntuat i el valor de la puntuació. Si l'usuari ja havia puntuat l'ítem, se substitueix el valor de la puntuació antiga. S'executa la operació *updateSetRating* per a tots els *Updatable* associats.
- **removeRating(u:User,i:Item):** Elimina una puntuació del sistema. S'executa la operació *updateRemoveRating* per a tots els *Updatable* associats.
- **getRatingsUser(u:User):** Obté totes les puntuacions que ha realitzat un usuari (codificades com un *SparseVector*).
- **getRatingsItem():** Obté totes les puntuacions d'un ítem (codificades com un *SparseVector*).
- **getRating(u:User,i:Item):** Obté la puntuació de l'usuari $u$ a l'ítem $i$ si existeix, *null* altrament.
- **getNumItems():** Retorna el nombre d'ítems del sistema.
- **getNumUsers():** Retorna el nombred d'usuaris del sistema.
- **getNumRatings():** Retorna el nombre de puntuacions del sistema.
- **getAvgRatingUser(u:User):** Retorna la puntuació mitjana de l'usuari $u$ (calculada segons la fórmula de l'apartat 4.1).
- **getAvgRatingItem(i:Item):** Retorna la puntuació mitjana de l'ítem $i$ (calculada segons la fórmula de l'apartat 4.1).
- **getGlobalMean():** Retorna la puntuació mitjana global.
- **getMinRating():** Retorna la puntuació mínima del sistema.
- **getMaxRating():** Retorna la puntuació màxima del sistema.
- **getUsers():** Retorna el conjunt de tots els usuaris.

- **getItems():** Retorna el conjunt de tots els ítems.
- **attachUpdatable(upd:Updatable):** Afegeix un nou *Updatable* a notificar quan hi hagi modificacions a les dades.
- **countRatingsUser(u:User):** Retorna el nombre de puntuacions de l'usuari $u$.
- **countRatingsItem(i:Item):** Retorna el nombre de puntuacions de l'ítem $i$.
- **ratingIterator():** Retorna un iterador sobre totes les puntuacions del sistema (permet recórrer-les totes sense haver d'emmagatzemar-les en memòria).
- **userExists(u:User):** Indica si un cert usuari existeix al sistema.
- **itemExists(i:Item):** Indica si un cert ítem existeix al sistema.

#### 5.1.4.1 MemRecommenderData

Implementació on totes les dades s'emmagatzemen en memòria. Això fa que per una banda la implementació sigui ràpida, però per l'altra pot provocar problemes de falta de memòria quan el nombre d'usuaris, ítems i puntuacions és relativament elevat.

#### 5.1.4.2 DiskRecommenderData

Implementació que utilitza el disc dur per a emmagatzemar les dades. És més lenta que *MemRecommenderData*, però no dóna problemes de memòria.

## 5.2 Classes auxiliars

En aquest apartat detallarem les classes i interfícies auxiliars utilitzades a les llibreries.

### 5.2.1 Interfície *Similarity*

Interfície que proporciona les operacions per calcular la similaritat entre dos usuaris o dos ítems. La similaritat retornada és un nombre que serà més gran o més petit en funció de si la similaritat és més o menys elevada.

#### 5.2.1.1 Classe *CosineSimilarity*

Implementació que utilitza la similaritat cosinus, mencionada a l'apartat 2.4.2.1 per a mesurar la semblança entre dos usuaris o ítems. S'utilitza una implementació de *FeatureObtainer* associada per a obtenir un vector associat a un usuari o ítem (per exemple, un

vector de puntuacions, o un vector de factors en el cas d'una factorització). Aleshores, per a calcular la similaritat entre dos usuaris o ítems, només cal calcular la similaritat cosinus entre els seus dos vectors associats.

### 5.2.2 Interfície *FeatureObtainer*

Interfície que proporciona operacions per a obtenir vectors associats a un usuari o ítem, de forma que calcular la similaritat entre dos vectors d'un usuari o ítem sigui equivalent a la similaritat entre els seus corresponents usuaris o ítems.

#### 5.2.2.1 Classe *SimpleFeatureObtainer*

Implementació que retorna com a vector associat a un usuari o ítem el seu vector de puntuacions. Si es tracta del vector d'un usuari, hi haurà tants components com ítems, i cadascun representarà una puntuació. Si es tracta del vector d'un ítem tindrà tants components com usuaris. Aquests vectors típicament seran dispersos, és per això que s'implementaran amb la subclasse *SparseVector*.

#### 5.2.2.2 Classe *FactorizationFeatureObtainer*

Utilitza una factorització (per exemple, algoritme *BRISMF* de l'apartat 4.4) i retorna com a vector associat a un usuari o ítem el seu vector de factors que el caracteritza.

### 5.2.3 Interfície *SimilarUsersSelector*

Interfície que proporciona operacions per a trobar un conjunt d'usuaris similars a un donat.

#### 5.2.3.1 Classe *SimpleSimilarUsersSelector*

Implementació que calcula la similaritat de tots els usuaris (amb la instància de *Similarity* associada) amb l'usuari donat i en retorna els *nUsers* més similars.

### 5.2.4 Interfície *SimilarItemsSelector*

Interfície que proporciona operacions per a trobar un conjunt d'ítems similars a un donat.

#### 5.2.4.1 Classe *SimpleSimilarItemsSelector*

Implementació que calcula la similaritat de tots els ítems (amb la instància de *Similarity* associada) amb l'ítem donat i en retorna els *nItems* més similars.

### 5.2.5 Interfície *Updatable*

Interfície que proporciona operacions que han de permetre a una classe que la implementi ser actualitzada automàticament quan hi hagi canvis en les dades del sistema. Intenta aplicar la idea del *patró Observador*, on hi ha un subjecte (*RecommenderData*) que quan pateix un canvi d'estat (operació que modifica les dades) ho notifica a una sèrie d'observadors (varis *Updatable*).

# Capítol 6

# Implementació

## 6.1 Entorn de treball

### 6.1.1 Hardware utilitzat

Al llarg del desenvolupament del projecte s'han emprat dues màquines diferents. La major part ha estat realitzada sobre un ordinador amb processador *Intel Core 2 Quad Q6600*, amb 4GB de RAM, sobre la distribució linux *Xubuntu 11.10*.

A la part final, i per a la realització de les proves, s'ha utilitzat un hardware més modern. Concretament, un PC amb *Intel Core i5 3330* i 4GB de RAM, sobre la distribució linux *Xubuntu 12.10*.

### 6.1.2 Entorn de desenvolupament

Com bé s'ha definit als requisits del projecte, la llibreria s'ha implementat en llenguatge *Java*. Amb aquesta decisió presa, n'hi ha una altra de força important pel que fa al desenvolupament: quin entorn integrat de desenvolupament (IDE) usar.

Per a *Java* disposem de vàries alternatives, sent les més conegudes Eclipse, Netbeans i Intellij IDEA. En el nostre cas, per l'experiència prèvia, i per estar considerat un dels millors entorns de desenvolupament de Java, ens hem decantat per Eclipse.

Concretament, s'ha utilitzat la versió *Eclipse IDE for Java EE Developers*. Aquesta versió inclou la possibilitat d'integrar un servidor Tomcat en el propi entorn de desenvolupament, cosa que ha resultat de gran utilitat a la hora de desenvolupar la implementació de la demostració d'un recomanador amb una interfície web.

## 6.2 Detalls d'implementació rellevants

### 6.2.1 Les dades del recomanador: *RecommenderData*

La implementació d'aquesta classe té molta influència en la escalabilitat de la plataforma. S'ha d'encarregar de gestionar totes les dades del sistema, que poden arribar a ocupar una gran quantitat de memòria. Recordem que s'han implementat dues versions, una on totes les dades s'emmagatzemen en memòria RAM (*MemRecommenderData*), i l'altra on es fa servir el disc dur (*DiskRecommenderData*).

Analitzem en més detall quines dades s'han d'emmagatzemar exactament. Recordem que *RecommenderData* ofereix operacions per a obtenir la puntuació mitjana d'un usuari o ítem. Per tant, per una banda hem de mantenir, per cada usuari i ítem, una estadística del nombre i suma de puntuacions que tenen, que ens permeti calcular la mitjana i actualitzar-la ràpidament quan hi hagi canvis a les puntuacions del sistema.

Per una altra banda, necessitem accés ràpid a les puntuacions d'un usuari o un ítem concret, i també a la puntuació que un usuari li ha donat a un cert ítem. També, hem de tenir constància dels identificadors de tots els usuaris i ítems del sistema.

Veiem les solucions pràctiques a les dues implementacions de *RecommenderData* per a satisfer aquestes necessitats.

#### 6.2.1.1 Implementació de *MemRecommenderData*

Per a poder calcular la puntuació mitjana d'un usuari o ítem en temps constant hem optat per utilitzar dos *HashMap* [1] i mantenir per a cada usuari i ítem el seu nombre i la seva suma de puntuacions.

S'ha decidit representar els identificadors d'usuaris i ítems com un *Integer* (32 bits), i per tant, la clau dels *HashMap* serà d'aquest tipus, i el valor associat una estructura de dades anomenada *EntityStats* que contindrà les estadístiques de l'usuari o ítem, respectivament.

[1] Un HashMap és una col·lecció que ofereix la llibreria *java.util*, que permet associar una clau a un valor, oferint accés eficient (consulta, modificació) per clau. Una mateixa clau no pot estar associada a dos valors.

```
class EntityStats implements Serializable {
    public double sum = 0;
    public double num = 0;
}
//Estadistiques dels usuaris
protected HashMap<Integer, EntityStats> usersStats;

//Estadistiques dels items
protected HashMap<Integer, EntityStats> itemsStats;
```

Si volem calcular la puntuació mitjana d'un usuari, per exemple, només hem de fer:

```
public Double getAvgRatingUser(int userID) {
    EntityStats stats = usersStats.get(userID);
    return stats.sum/stats.num;
}
```

Per a un ítem es faria de forma anàloga. Mantenir actualitzades aquestes estructures de dades quan s'introdueixen o eliminen puntuacions tampoc és difícil: només cal sumar o restar la puntuació a *stats.sum* i incrementar o decrementar *stats.num*.

Aquestes dues estructures de dades, *usersStats* i *itemsStats* també ens permeten obtenir el conjunt d'identificadors d'usuaris i ítems, ja que inclouen una operació anomenada *keySet()* que retorna el conjunt de claus de cadascuna.

Ara necessitem estructures de dades que ens permetin accedir ràpidament a les puntuacions d'un cert usuari o un cert ítem. Altre cop utilitzarem dos *HashMap*, utilitzant també com a clau l'identificador d'usuari i ítem, però que tindran com a valor les seves puntuacions. Aquestes puntuacions es representaran amb un altre *HashMap*. Si es tracta de les puntuacions d'un usuari, la clau seran identificadors d'ítems i el valor la seva puntuació. Si es tracta de puntuacions d'un ítem, la clau seran identificadors d'usuaris i el valor la seva puntuació.

```
//La primera clau es un id. d'usuari, la segona un id. d'item
protected HashMap<Integer, HashMap<Integer, Double>> ratingsUser;

//La primera clau es un id. d'item, la segona un id. d'usuari
protected HashMap<Integer, HashMap<Integer, Double>> ratingsItem;
```

Evidentment, amb aquesta implementació hi ha redundància de dades, ja que estem emmagatzemant dos cops cada puntuació (apareixerà al corresponent usuari a *ratingsUser* i al corresponent ítem a *ratingsItem*). Però aquesta redundància és necessària per a permetre un accés ràpid a les puntuacions tant d'un usuari com d'un ítem.

Si el que volem és accedir a la puntuació d'un usuari a un ítem en concret, tenim dues opcions vàlides: accedir a *ratingsUser* primer a partir de l'usuari i al *HashMap* obtingut a partir de l'ítem, o bé accedir a *ratingsItem* primer per ítem i després per usuari.

Aquesta implementació funciona bé sempre i quan les dades càpiguen a memòria. No obstant, als tests realitzats en molts casos això no és així, i en aquests casos el rendiment sol ser molt pobre. Quan la quantitat de dades a gestionar és considerable, és recomanable usar *DiskRecommenderData*.

#### 6.2.1.2 Implementació de *DiskRecommenderData*

Per a aquesta implementació, les estadístiques se segueixen guardant de la mateixa forma que a *MemRecommenderData*, és a dir, en memòria. Això és a així perquè l'espai que ocupen és assumible (lineal en el nombre d'usuaris i ítems) i a canvi ens ofereix un accés molt ràpid. No obstant, fàcilment es podria passar a disc de forma anàloga al que s'ha fet amb les puntuacions.

De puntuacions, en canvi, n'hi haurà en general un nombre molt més gran que d'usuaris i ítems. És per això que s'ha decidit emmagatzemar-les a disc. Per a fer-ho, s'ha utilitzat la llibreria *JDBM3* [12]. Aquesta llibreria ens proporciona una manera transparent de treballar amb estructures de dades anàlogues a les estàndards de Java (TreeMap, HashMap...), però que estan preparades per a contenir una quantitat molt gran d'elements, emmagatzemant-les a disc. Una de les classes incloses s'anomena *TreeMap*. Proporciona les mateixes operacions que *HashMap*, i algunes d'addicionals, però utilitza fitxers a disc per a guardar les dades. La llibreria gestiona de forma automàtica una *caché* per a millorar la rapidesa dels accessos, i les estructures de dades que ofereix estan preparades per a treballar-hi de forma concurrent. Tot i així, per motius d'eficiència la concurrència (transaccions) s'ha desactivat, ja que suposava una penalització important en el rendiment.

Un primer intent d'implementació va ser aprofitar la classe *MemRecommenderData*, però canviant els *HashMap* corresponents per *TreeMap* de JDBM3. Però el fet que fossin *Maps* anidats (un *Map* dins d'un altre *Map*) va donar certs problemes. La manera de fer la persistència en disc de la llibreria està pensada per a claus i valors petits, però el valor en aquest cas seria un *Map* representant les puntuacions d'un usuari o ítem. Això feia que els accessos fossin molt lents.

Finalment es va decidir substituïr els *HashMap* anidats per *TreeMap* on les claus són la concatenació de l'identificador de l'usuari amb l'identificador de l'ítem (per a *ratingsUser*) i la concatenació l'identificador de l'ítem amb l'identificador de l'usuari (per a *ratingsItem*).

Com que els identificadors d'usuaris i ítems ocupen 32 bits, les claus seran de 64 bits (*Long* a Java), i el valor dels *TreeMap*, un *Double* denotant la puntuació de l'usuari a l'ítem.

```
//key = userID|itemID
protected TreeMap<Long, Double> ratingsUser;

//key = itemID|userID
protected TreeMap<Long, Double> ratingsItem;
```

Els *TreeMap* estan implementats com arbres de cerca balancejats. Això ens permet, a partir d'una clau, trobar el seu corresponent valor en temps logarítmic. I també, un cop obtingut un element, podem obtenir els següents $n$ elements (en ordre de clau), en temps $\Theta(n)$.

Gràcies a les claus que estem usant, podem obtenir eficientment tant les puntuacions d'un cert usuari (cercant al *TreeMap* indexat per *userID|itemID*) com les puntuacions d'un cert ítem (cercant a l'indexat per *itemID|userID*) ja que per la propietat esmentada apareixeran de forma consecutiva a l'arbre.

La implementació resultant és molt ràpida, i els tests realitzats han demostrat que escala bé respecte el nombre de puntuacions, usuaris i ítems.

#### 6.2.1.3 Alternatives provades

Abans de trobar la llibreria *JDBM3* que s'ha usat finalment, es van probar diverses opcions per a emmagatzemar les puntuacions a disc per tal d'alliberar l'ús de memòria.

Podem destacar les bases de dades *MySQL*, *PostgreSQL* i *SQLite*. Vam trobar que tant *MySQL* com *PostgreSQL* utilitzaven un espai en disc excessiu per a emmagatzemar les puntuacions, i el rendiment que oferien, sobretot a mesura que el nombre de puntuacions creixia, era molt inferior al de *JDBM3*. *SQLite*, en canvi, emmagatzemava les dades de forma força compacta, i el rendiment que oferia era prometedor al principi. Però altre cop, a mesura que el nombre de puntuacions emmagatzemades creixia (p.e. a partir de 75.000.000 de puntuacions) aquest empitjorava força, fins i tot per sota del de *MySQL* i *PostgreSQL*.

Possiblement, s'està donant a entendre que el rendiment d'aquestes bases de dades és molt pobre. Però només ho és si es compara amb el que dóna *JDBM3*, que és molt més simple i especialitzada (p.e. els *TreeMap* només permeten guardar parells clau/valor). Per a les proves necessitàvem processar en minuts dades que potser s'han estat recopilant durant mesos. Així que en un escenari real, segurament el rendiment que pot oferir una base de dades és més que suficient per a suportar la càrrega de peticions diàries, obtenint també

els beneficis que dóna una base de dades seriosa com les esmentades: robustesa i bona gestió de la concurrència.

### 6.2.2 Implementació de l'algoritme *BRISMF*

La implementació de l'algoritme *BRISMF* (apartat 4.4) és, en general, força directa i senzilla. No obstant, sí que van sorgir alguns problemes, sobretot per temes de falta de memòria, la solució dels quals és interessant d'esmentar.

Recordem l'entrenament *offline*. Primer de tot, s'havia de dividir el conjunt de totes les puntuacions del sistema en dos: un de molt petit anomenat conjunt de testeig, i la resta de puntuacions a un altre anomenat conjunt d'entrenament. A continuació s'havien de fer vàries passades sobre el conjunt d'entrenament, i utilitzar el conjunt de testeig per avaluar la precisió del model a cadascuna.

A causa de la gran quantitat de puntuacions que hi pot haver al sistema, la implementació de la manera d'iterar sobre totes les puntuacions és crítica per evitar problemes de manca de memòria. És per això que es va incloure la operació *ratingIterator()* a *RecommenderData*. Aquesta operació retorna un iterador que ens permet recórrer totes les puntuacions del sistema una a una, sense haver de guardar-les a memòria.

Per a guardar la factorització, és a dir, els vectors característics de tots els usuaris i ítems s'ha utilitzat de nou la classe *HashMap*. Això ens permet de forma senzilla emmagatzemar per a cada usuari i cada ítem (indexats per l'identificador) el seu corresponent vector característic.

```
//vectors caracteristics dels usuaris
protected HashMap<Integer, float[]> userFeature;

//vectors caracteristics dels items
protected HashMap<Integer, float[]> itemFeature;
```

Cal destacar que s'ha decidit utilitzar *Float* en comptes de *Double* per a representar els components dels vectors. Així, la factorització ocuparà la meitat de memòria (32 bits en comptes de 64 per cada component). En un futur, es podria fer una versió distribuïda o en basada en disc (de forma similar que la classe *DiskRecommenderData*), per a recomanadors amb una quantitat d'usuaris i ítems extremadament gran. No obstant, tenir tota la factorització en memòria suposa un gran avantatge en quant a rendiment, i l'espai ocupat és admissible $O(|U| + |I|)$.

### 6.2.3 Persistència del recomanador

Per a la persistència hem aprofitat el mecanisme de serialització que ofereix el propi llenguatge Java, que posa a la nostra disposició les classes *ObjectOutputStream* i *ObjectInputStream*. Aquests components ens permeten convertir qualsevol classe de Java en una seqüència de bytes, i viceversa, cosa que ens permet, entre d'altres coses, emmagatzemarles en fitxers. La gestió de les dependències entre classes es fa de forma automàtica. L'únic requisit és que les classes a serialitzar implementin la interfície *Serializable*, la qual cosa és una pura formalitat, ja que aquesta interfície no inclou cap operació.

Hi ha hagut, però, alguns problemes respecte a la serialització que s'han hagut de tractar de forma especial. Concretament, pel que fa a la classe *DiskRecommenderData*. Aquesta classe utilitza una persistència de les dades per sí mateixa, ja que les puntuacions d'usuaris a ítems es guarden a disc per evitar problemes de memòria. Simplement, s'ha hagut de fer ús del keyword *transient*, que indica que no s'ha de serialitzar ni deserialitzar una variable en concret d'una classe (en el nostre cas, les variables dels *TreeMap* que contenen les puntuacions, que ja estan a disc). També, si implementant les operacions *readObject(ObjectInputStream in)* i *writeObject(ObjectOutputStream out)* a la pròpia classe, es poden realitzar algunes accions específiques just abans de serialitzar o just després de deserialitzar una classe, respectivament. Amb aquestes dues eines s'ha pogut gestionar correctament la persistència en aquest cas.

## 6.3 Demostració: recomanador de grups musicals

En aquest apartat intentarem explicar el recomanador de grups musicals implementat utilitzant la nostra llibreria. Ens centrarem, en primer lloc, en els detalls referents a la tecnologia utilitzada i a continuació descriurem les funcionalitats implementades.

### 6.3.1 Aspectes tècnics

Per a implementar el recomanador s'ha utilitzat el conjunt de dades de Yahoo!, de grups i artistes musicals, i l'algoritme *BRISMF* de la nostra llibreria. Aquest algoritme és el que clarament ha donat millors resultats a les proves realitzades, complint amb els requisits que ens havíem plantejat.

Per a l'emmagatzematge, s'han utilitzat les estructures de dades persistents a disc implementades a *DiskRecommenderData*, ja que el tamany d'aquest conjunt de dades ho feia necessari. A més, en aquest cas la persistència és desitjable de totes formes, ja que ens interessa poder conservar les dades entre diferents execucions.

S'ha implementat una interfície web senzilla, una part essencial per a poder comprovar els resultats del recomanador. Per a fer-ho, hem emprat un servidor web Tomcat, basat

en Java, que ens ha permès integrar de forma senzilla la nostra llibreria per a sistemes recomanadors.

#### 6.3.1.1 Tomcat i JSP

Tomcat és un servidor web programat en Java, amb soport a Servlets i Java Server Pages (JSP). Un Servlet no és més que una classe que corre dins d'un contenidor de Servlets (Tomcat) i que normalment s'encarrega de generar el codi HTML a partir d'una petició del navegador web. Els Servlets poden, a partir dels paràmetres rebuts, persistir o recuperar dades per a una mateixa sessió web (les sessions es gestionen transparentment pel servidor web). Així, es pot per exemple gestionar de forma fàcil les transaccions d'inici de sessió, recordar una sessió, tancar-la, etc.

Els JSP, per una altra banda, també permeten generar codi web dinàmicament. Un document JSP permet, entre d'altres coses, incrustar codi Java dins de HTML, utilitzant un sistema de plantilles amb una sintaxis específica. Al final, internament el servidor Tomcat acabarà compilant els fitxers JSP a Servlets, de forma que tot el que es pot fer amb Servlets també es pot fer amb aquesta tecnologia, i viceversa. En la nostra opinió, no obstant, resulta molt més còmode d'utilitzar que implementar els Servlets corresponents directament, precisament perquè és molt més senzill haver d'incrustar codi Java dins de HTML, que no al revés, ja que generalment en un sistema web la proporció de HTML és molt més gran.

Per tant, hem utilitzat aquesta tecnologia Tomcat i JSP, per a implementar tota la part del servidor.

A mode d'exemple, aquest seria un exemple de codi JSP per a generar una web amb una llista de les primeres 10 potències de 2:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head><title>Potencies de 2</title></head>
<body>
<h1>Potencies de 2</h1>
<table>
<th>Exponent</th><th>2^Exponent</th>
<% for (int i=0; i<10; i++) {%>
<tr><td><%= i%></td>
<td><%= Math.pow(2, i)%></td>
</tr>
<% } %>
</table>
</body>
</html>
```

I a continuació, la versió anàloga utilitzant un servlet:

```
public class PowersOf2 extends HttpServlet {
    public void service(HttpServletRequest request,
       HttpServletResponse response) throws IOException,
       ServletException {
        response.setContentType("text/html");
        ServletOutputStream out = response.getOutputStream();
        out.print("<html>");
        out.print("<head><title>Potencies de 2</title></head>");
        out.print("<body>");
        out.print("<h1>Potencies de 2</h1>");
        out.print("<table>");
        out.print("<th>Exponent</th><th>2^Exponent</th>");
        for (int i=0; i<10; i++) {
            out.print("<tr><td>" + i + "</td>");
            out.print("<td>" + Math.pow(2, i) + "</td>");
            out.print("</tr>");
        }
        out.print("</table></body></html>");
        out.close();
    }
}
```

#### 6.3.1.2 Integració del servidor Tomcat amb Eclipse

L'IDE Eclipse (for Java EE Developers) permet crear projectes web de forma molt senzilla. N'hi ha prou amb seleccionar la opció "New Dynamic Web Project" i triar la carpeta on hi ha instalat el Servidor Tomcat. Aleshores, podem començar a editar els nostres fitxers JSP, incloure llibreries (la del nostre projecte, per exemple) de la mateixa manera que amb qualsevol altre projecte Java.

El fet que el servidor estigui integrat dins de l'entorn de desenvolupament ens ha estat molt útil per al procés de desenvolupament de la interfície web, per arreglar errors i provar noves funcionalitats de forma ràpida.

#### 6.3.1.3 Integració de la nostra llibreria de sistemes recomanadors

Per a integrar la nostra llibreria de sistemes recomanadors hem creat una classe anomenada *RecommenderManager*, encarregada de fer d'intermediària entre les classes del recomanador i la part pròpia de la interfície web. Aquesta classe implementa el patró *Singleton*, de forma que mentre el servidor Tomcat estigui corrent només n'hi haurà una única instància, accessible des de qualsevol classe utilitzant el mètode estàtic *RecommenderManager.getInstance()*, com és usual en aquest patró de disseny. Això ens permet, d'entrada, assegurar-nos un accés centralitzat i únic al recomanador i les seves dades.

Aquesta classe també s'encarrega de realitzar tasques que escapen a la responsabilitat del recomanador. Una d'elles es la de conèixer l'assignació entre els identificadors d'ítems (enters) que s'utilitzen internament al recomanador amb la seva descripció textual, que en el nostre cas correspon al nom del grup o artista musical. Aquesta assignació ve donada pel propi conjunt de dades, evidentment, en fitxer separat de les puntuacions.

L'assignació de noms d'usuaris (emprats per a iniciar sessió al sistema) amb identificadors d'usuari interns del recomanador (enters) també es gestiona mitjançant aquesta classe.

Per últim, cal dir que no s'ha dedicat cap esforç per a que la implementació sigui completament segura des del punt de vista dels accesos concurrents, donat el seu caràcter de prototip. No obstant, creiem que a la pràctica no costaria massa adaptar la classe *RecommenderManager* tenint en compte aquest aspecte, fent algunes operacions *synchronized*, o usant *locks* a les parts potencialment més problemàtiques. A la pràctica, cal dir que s'ha provat amb varis usuaris i no hi ha hagut cap problema en aquest sentit.

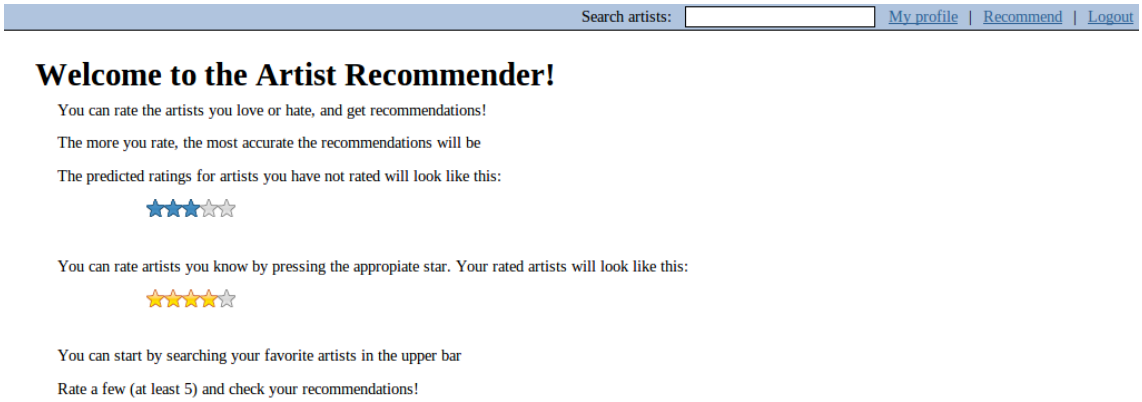
### 6.3.2 Funcionalitats

En aquest apartat veurem, ajudant-nos de diverses captures de pantalla, els casos d'ús del sistema recomanador implementat. Concretament, els casos d'ús seran:

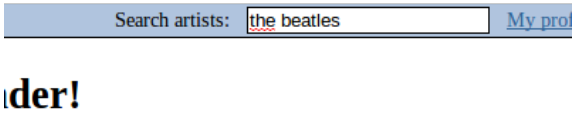
- Cerca d'artistes pel nom

- Valoració d'artistes
- Obtenció de recomanacions personalitzades
- Visualització del perfil d'un artista
- Visualització del perfil del propi usuari

**Cerca d'artistes** En primer lloc, un cop entrem amb el nostre usuari, veurem una pantalla similar a aquesta:

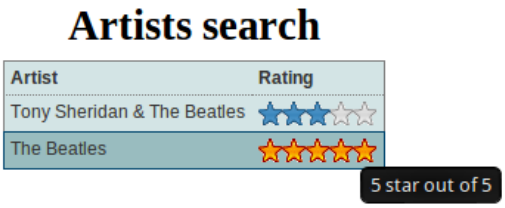



El primer pas que podem fer és començar a buscar artistes, utilitzant la barra de cerca a la part superior:



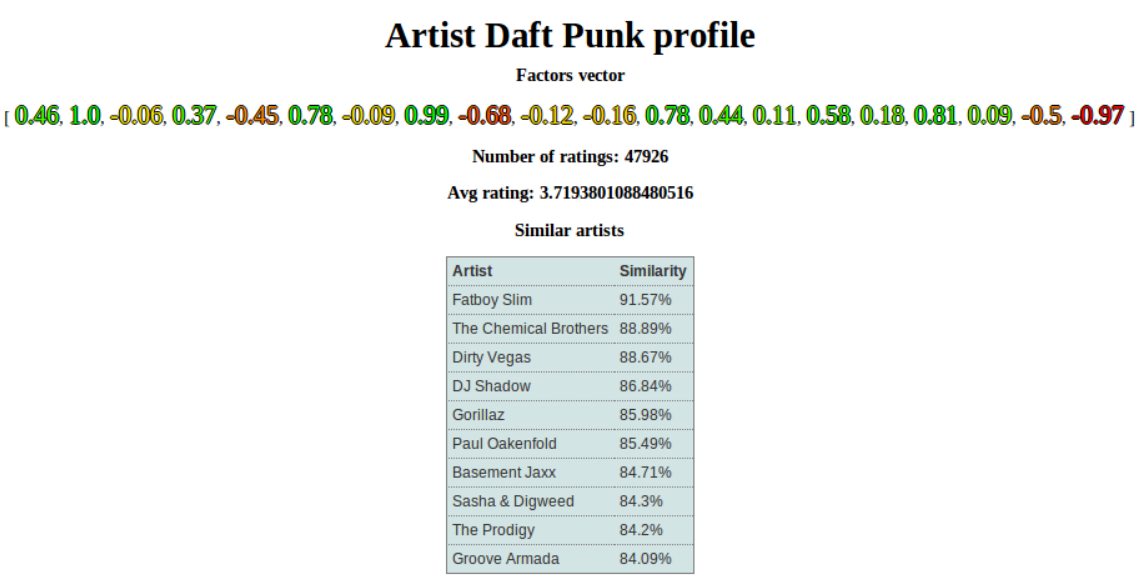

De seguida veurem els resultats. Les estrelles en color blau denoten la predicció de puntuació que fa el sistema per a artistes que no hem valorat, encara que de moment no hem introduït cap puntuació:

**Valoració d'artistes** Podem votar un artista fent clic a la estrella corresponent:




**Recomanació d'artistes** Un cop haguem puntuat alguns dels nostres artistes preferits (o més odiats), podem generar alguns recomanacions, fent clic a "Recommend", a la barra superior. Obtindrem les 10 millors recomanacions segons l'algoritme:


**Visualització del perfil d'un artista** Fent clic al nom d'un artista, podem accedir al seu perfil. Ens mostrarà els 10 grups més similars a aquest artista (calculats comparant la similaritat cosinos entre els vectors característics dels diferents artistes), així com el nombre de puntuacions de l'artista i la seva puntuació mitjana. També podrem veure el seu vector de factors, cosa que per un possible usuari final no tindria massa utilitat, però com a desenvolupadors ens ha semblat interessant d'incloure'l:



## Artist Daft Punk profile

**Factors vector**

[ 0.46, 1.0, -0.06, 0.37, -0.45, 0.78, -0.09, 0.99, -0.68, -0.12, -0.16, 0.78, 0.44, 0.11, 0.58, 0.18, 0.81, 0.09, -0.5, -0.97 ]

**Number of ratings: 47926**

**Avg rating: 3.7193801088480516**

**Similar artists**

| Artist | Similarity |
|---|---|
| Fatboy Slim | 91.57% |
| The Chemical Brothers | 88.89% |
| Dirty Vegas | 88.67% |
| DJ Shadow | 86.84% |
| Gorillaz | 85.98% |
| Paul Oakenfold | 85.49% |
| Basement Jaxx | 84.71% |
| Sasha & Digweed | 84.3% |
| The Prodigy | 84.2% |
| Groove Armada | 84.09% |

**Visualització del perfil del propi usuari** Per últim, podem accedir al perfil del nostre usuari, també a partir de la opció corresponent a la barra superior. Ens mostrarà els artistes que hem puntuat, juntament amb el nostre vector de factors, anàlogament al perfil dels artistes:

## User 370025683 profile

**Factors vector**

[ 1.0, -0.03, 0.02, 0.57, 0.11, -0.0, 0.21, 0.04, -0.57, -0.27, -0.12, 0.23, 0.24, -0.09, 0.23, 0.11, 0.13, -0.23, 0.02, -0.08]

**Rated artists**

| Artist | Rating |
|---|---|
| Johnny Cash | ★★★★☆ |
| The Doors | ★★★★☆ |
| Nirvana | ★★★★★ |
| The Strokes | ★★★★☆ |
| Brak | ★☆☆☆☆ |
| Dire Straits | ★★★★★ |
| Green Day | ★★★☆☆ |
| Fatboy Slim | ★★★★★ |
| David Bowie | ★★★★★ |
| The Who | ★★★★☆ |
| R.E.M. | ★★★★★ |
| Blondie | ★★★★★ |
| OK Go | ★★★★★ |
| U2 | ★★★★☆ |
| Abba | ★★★★★ |
| Queen | ★★★★★ |

# Capítol 7

# Experiments i avaluació

En aquest apartat avaluarem el resultat dels dos objectius del projecte. En primer lloc, provarem els algoritmes implementats a la nostra plataforma. Per a fer-ho, executarem proves automatitzades que mesuraran la precisió dels models a l'hora de predir puntuacions i recomanar productes de diferents conjunts de dades.

En segon lloc, es provarà el sistema recomanador implementat amb una interfície web. Tenint en compte que a la primera part ja s'hauran fet proves a priori objectives, en aquest cas ens centrarem en aspectes no mesurables numèricament. Intentarem fer una avaluació subjectiva dels resultats de les recomanacions, creant perfils d'usuaris i utilitzant uns certs coneixements previs i el sentit comú per a decidir si les recomanacions obtingudes tenen sentit.

Un resum de les proves a realitzar seria:

- Entrenament *offline* dels models i avaluació del *RMSE* resultant (secció 7.2.1)
- Entrenament *online* dels models i avaluació del *RMSE* resultant (secció 7.2.2)
- Precisió a l'hora d'ordenar els ítems segons la preferència de l'usuari, amb la mètrica *Fracció de Parelles Discordants* (secció 7.2.3)
- Precisió i *recall* a l'hora de fer recomanacions (secció 7.2.4)
- Avaluació subjectiva de les recomanacions de grups musicals (secció 7.3.1)
- Avaluació subjectiva dels grups similars calculats (secció 7.3.2)

## 7.1 Conjunts de dades utilitzats

Per a realitzar les proves s'han fet servir diversos conjunts de dades. Es tracta de conjunts que contenen una gran quantitat de puntuacions d'usuaris a ítems d'un cert tipus. Típi-

cament, cadascun d'aquests conjunts té la forma d'un fitxer de text on cada línia conté un identificador d'usuari, un identificador d'ítem i una puntuació. En alguns casos, també s'hi inclou una data, indicant en quin moment l'usuari va puntuar l'ítem, i informació addicional, com una descripció textual per a cada ítem (títol de la pel·lícula, nom de l'artista musical, etc) A continuació fem una breu descripció dels conjunts de dades emprats al llarg d'aquest apartat.

- **Movielens 1M:** conté puntuacions de pel·lícules
- **Jester Dataset 2:** conté puntuacions d'acudits
- **Flixter:** conté puntuacions de pel·lícules, obtingudes de la pàgina Flixter
- **Yahoo! Music:** conté puntuacions de grups i artistes musicals

| | **Movielens 1M** | **Jester Dataset 2** | **Flixter** | **Yahoo! Music** |
|---|---|---|---|---|
| **# puntuacions** | 1,000,209 | 1,761,439 | 8,196,077 | 115,579,438 |
| **# usuaris** | 6,040 | 59,132 | 147,612 | 1,948,882 |
| **# ítems** | 3,706 | 140 | 48,794 | 98,211 |
| **Puntuacions/usuari** | 165 | 29 | 55 | 59 |
| **Puntuacions/ítem** | 269 | 12,581 | 167 | 1,176 |
| **Rang original puntuacions** | $[1, 5]$ | $[-10, 10]$ | $[1, 5]$ | $[0, 100]$ |
| **Puntuació mitjana** | 3.58 | 3.32 | 3.61 | 3.09 |
| **Stdev puntuació** | 1.12 | 1.06 | 1.09 | 1.61 |

Taula 7.1: Informació dels conjunts de dades emprats als experiments

Tot i que el rang de les puntuacions varia entre els diferents conjunts, s'ha decidit normalitzar-les totes dins el rang $[1, 5]$ (les mitjanes i desviacions estàndard mostrades a la taula tenen en compte aquesta normalització). D'aquesta forma, les mètriques utilitzades per a calcular l'error seran comparables entre els diferents conjunts de dades, i els paràmetres numèrics d'alguns algoritmes no hauran de variar tant d'un conjunt a un altre.

## 7.2 Avaluació dels algoritmes

Un cop implementats, és necessari poder avaluar els algoritmes i comparar-ne els resultats. Evidentment, s'espera que destaqui l'anomenat *BRISMF*, però s'ha de comprovar empíricament.

L'aspecte més important a avaluar és evident: la qualitat de les recomanacions obtingudes. Després, també tindrem molt en compte el temps en executar els experiments dels diferents algoritmes, ja que l'eficiència, si recordem, era un dels requisits no funcionals plantejats.

Segons la concepció de recomanador que hem utilitzat, avaluar com de bones són les recomanacions equival a mesurar com de bones son les prediccions de puntuacions que

fan els algoritmes respecte les puntuacions que realment donaran els usuaris. Per tant, té sentit que alguns experiments se centrin en mesurar precisament això, l'error en les prediccions.

Per una altra banda, també s'inclouran tests que evitin la coneguda mètrica RMSE de l'error de predicció de puntuacions, per tal d'avaluar la qualitat de les recomanacions d'una forma alternativa i més directa.

Abans de mostrar els resultats de les proves, recordem els algoritmes implementats a la llibreria i que s'avaluaran en aquesta secció:

- Baseline (secció 4.1)
- Filtratge col·laboratiu basat en usuaris (secció 4.2)
- Filtratge col·laboratiu basat en ítems (secció 4.3)
- BRISMF (secció 4.4)

### 7.2.1 Prova offline

Aquesta prova intentarà avaluar la qualitat de les prediccions en un mode no incremental. Es dividiran les puntuacions del conjunt de dades en un conjunt d'entrenament i un altre de prova, de forma que el d'entrenament contingui aproximadament un 80% de les puntuacions i el de prova el 20% restant. S'entrenen els models utilitzant les puntuacions del conjunt de prova, i a continuació s'avalua l'error RMSE de la predicció de les puntuacions del conjunt d'entrenament.

La divisió de les puntuacions en els dos conjunts esmentats és aleatòria, però idèntica per a tots els algoritmes (s'ha fet servir el mateix generador de nombres aleatoris en tots els casos). Concretament, per a cada puntuació del conjunt de dades, es fa un sorteig amb probabilitat 0.8 i en funció del resultat s'utilitza per entrenar el model o es reserva per avaluar-lo.

En primer lloc presentem una taula amb el resum dels resultats d'aquesta prova. Conté, per a cada conjunt de dades i cada algoritme, els millors resultats obtinguts en termes de RMSE a l'hora de predir les puntuacions.

| | **Baseline** | **UserBased** | **ItemBased** | **BRISMF** |
|---|---|---|---|---|
| **Movielens 1M** | 0.92245 | 0.91325 | 0.87833 | 0.84567 |
| **Jester Dataset 2** | 0.87656 | 0.97767 | 0.94121 | 0.80700 |
| **Flixter** | 0.90933 | 0.95208 | 0.92940 | 0.83421 |
| **Yahoo! Music** | 1.16570 | N/A | N/A | 0.89169 |

Taula 7.2: RMSE dels diferents algoritmes a la prova offline

Fixant-nos en el RMSE, podem veure que clarament els millors resultats obtinguts són els de l'algoritme *BRISMF*, com era d'esperar. Per una altra banda, els resultats de l'algoritme *Baseline* són sorprenentment bons per un algoritme tan simple, superant fins i tot en alguns casos els basats en usuaris o ítems.

Clarament, el conjunt de dades més difícil de predir és el de Yahoo! Music. En aquest cas, la diferència entre l'algoritme baseline i el BRISMF és molt més elevada que en la resta. Però això parla encara més positivament de l'algoritme BRISMF, ja que és capaç d'adaptar-se adequadament a conjunts de dades molt diferents.

Cal dir que en el cas del *UserBased* i *ItemBased*, el nombre de puntuacions avaluades ha estat molt menor que en els altres casos, per límits de temps. Això ha portat a que, en el cas del dataset de Yahoo!, aquest nombre fos tant petit que no tenia cap sentit d'incloure els resultats.

A continuació, es detallaran una mica més els resultats, aclarint algunes particularitats de l'execució d'aquesta prova per a cadascun dels algoritmes avaluats.

#### 7.2.1.1 Baseline

Aquest és l'algoritme que s'explica a la secció 4.1. És un algoritme simple, i per tant extremadament eficient, ja que per a realitzar les prediccions només fa servir mitjanes d'usuaris i ítems.

En aquest cas, el temps d'entrenament del model és 0 en tots els casos, ja que l'únic que es necessita és introduïr totes les puntuacions del conjunt d'entrenament a una instància de la classe *RecommenderData*, una operació que es repetirà per a tots els algoritmes.

| | **Movielens** | **Jester** | **Flixter** | **Yahoo!** |
|---|---|---|---|---|
| **RMSE** | 0.92245 | 0.87656 | 0.90933 | 1.16570 |
| **# puntuacions avaluades** | 200208 | 352487 | 1639695 | 23125491 |
| **Temps d'avaluació** | <0.5s | <0.5s | <0.5s | 4s |

Taula 7.3: Resultats de la prova offline per a l'algoritme baseline

#### 7.2.1.2 Filtratge col·laboratiu basat en usuaris i ítems

En aquest cas hem avaluat conjuntament el filtratge col·laboratiu basat en usuaris (secció 4.2) i el basat en ítems (secció 4.3).

De la mateixa forma que amb el *baseline*, tampoc és necessari un entrenament del model. Les prediccions es fan a partir de les estructures de dades que ofereix la classe *RecommenderData*, i altre cop la única operació necessària és la d'introduïr totes les puntuacions d'entrenament.

No obstant, aquí els temps d'avaluació són força elevats, a causa de la manera en com aquests algoritmes fan les prediccions. Cada predicció implica iterar sobre usuaris o ítems, i calcular les similaritats entre ells és una operació força costosa. Per tant, per alguns conjunts de dades ens hem vist obligats a limitar el nombre de puntuacions avaluades, per motius de temps.

Per una altra banda, els algoritmes tenen un paràmetre lliure, el nombre $K$ d'usuaris (o ítems) més similars sobre els quals calcularem les puntuacions. Recordem les fórmules d'aquests algoritmes:

$$\hat{r}_{u,i} = \bar{r}_{u,*} + \frac{\sum\limits_{u' \in U} sim(u,u')(r_{u',i} - \bar{r}_{u',*})}{\sum\limits_{u' \in U} sim(u,u')}$$

Per a la nostra implementació, en comptes d'iterar sobre tots els usuaris ($u' \in U$), ho farem només sobre els $K$ més similars a $u$.

$$\hat{r}_{u,i} = \bar{r}_{*,i} + \frac{\sum\limits_{i' \in I} sim(i,i')(r_{u,i'} - \bar{r}_{*,i'})}{\sum\limits_{i' \in I} sim(i,i')}$$

I de forma anàloga en el cas basat en ítems, iterarem sobre els $K$ ítems més similars a $i$.

Un cop aclarit aquest punt, vegem els millors resultats obtinguts per aquests algoritmes als diferents conjunts de dades.

| | **Movielens** | **Jester** | **Flixter** | **Yahoo!** |
|---|---|---|---|---|
| **RMSE** | 0.91324 | 0.97767 | 0.95208 | N/A |
| $K$ | 50 | 50 | 50 | N/A |
| **# puntuacions avaluades** | 36694 | 9994 | 3218 | 317 |
| **Temps d'avaluació** | 1800s | 1800s | 1800s | 1800s |

Taula 7.4: Resultats de la prova offline per a l'algoritme basat en usuaris

| | **Movielens** | **Jester** | **Flixter** | **Yahoo!** |
|---|---|---|---|---|
| **RMSE** | 0.87833 | 0.94120 | 0.92940 | N/A |
| $K$ | 30 | 20 | 20 | N/A |
| **# puntuacions avaluades** | 37559 | 5510 | 2946 | 179 |
| **Temps d'avaluació** | 1800s | 1800s | 1800s | 1800s |

Taula 7.5: Resultats de la prova offline per a l'algoritme basat en ítems

Cal destacar que no s'inclouen els resultats de RMSE per al conjunt de dades de Yahoo!, a causa del mínim nombre de puntuacions que s'han pogut avaluar, per la lentitud de l'algoritme. En tots els casos, el temps d'avaluació s'ha limitat a 30 min.

A l'apèndix B.1.1 es troben els resultats complets, amb tots els $K$ provats.

Com es pot veure, el temps a l'hora de fer prediccions de puntuacions és realment elevat. Això fa que aquests algoritmes no siguin bons candidats per a un escenari en temps real. Això, juntament amb el fet que els resultats en quant a RMSE són força discrets, ha fet que prescindim d'aquests algoritmes per a les successives proves d'aquesta secció.

#### 7.2.1.3 BRISMF

En aquest apartat avaluarem l'algoritme vist a la secció 4.4. En aquest cas sí que és necessari un entrenament del model a banda de la introducció de puntuacions a la classe *RecommenderData*. Com es descriu a l'apartat de l'algoritme, hi ha dos modes d'entrenament: *offline* i *online*. El *offline* consteix a reentrenar el model sencer, des de zero, a partir de totes les puntuacions de *RecommenderData*. El *online*, per una altra banda, procura mantenir actualitzat el model cada cop que s'introdueix una puntuació. En aquest apartat desactivarem el mode *online* i ens centrarem en avaluar el primer.

Hi ha tres paràmetres de l'algoritme que s'han de definir: el nombre de factors($F$), la constant d'aprenetatge ($\gamma$) i el factor de regularització ($\lambda$). Hem executat totes les proves amb 20, 30 i 40 factors, excepte per al conjunt de dades de Yahoo!, on només s'han utilitzat 20 factors, per limitacions de la memòria RAM disponible (la factorització s'ha de mantenir en memòria sencera).

| | **Movielens** | **Jester** | **Flixter** | **Yahoo!** |
|---|---|---|---|---|
| **RMSE** | 0.84567 | 0.80700 | 0.83421 | 0.89169 |
| $K$ | 40 | 40 | 40 | 20 |
| $\gamma$ | 0.003 | 0.002 | 0.002 | 0.001 |
| $\lambda$ | 0.02 | 0.02 | 0.02 | 0.01 |
| **# puntuacions avaluades** | 200208 | 352487 | 1639695 | 23125491 |
| **Temps d'entrenament** | 31s | 73s | 322s | 2753s |
| **Temps d'avaluació** | <0.5s | <0.5s | <0.5s | 4s |

Taula 7.6: Resultats de la prova offline per a l'algoritme BRISMF

Aquests són els millors resultats obtinguts explorant sistemàticament un subconjunt de l'espai de paràmetres. Per motius de límits de memòria, i de temps, no s'han usat les mateixes combinacions de paràmetres per a tots els conjunts de dades. A l'apèndix B.1.2 es troben els resultats complets, amb tots tots els paràmetres provats.

### 7.2.2 Prova online

Aquesta prova avaluarà el comportament de la nostra llibreria en un mode incremental, on s'incorporin continuament noves puntuacions, nous usuaris i ítems i el model s'hi hagi d'adaptar en temps real. Com s'ha argumentat a l'apartat anterior, s'ha decidit eliminar els algoritmes basats en usuaris i ítems per a les successives proves, vistos els resultats en quant a eficiència i precisió de les prediccions. Per tant, a partir d'ara compararem només els algoritmes *baseline* i *BRISMF*.

Partirem d'un estat on el sistema no contingui cap dada. A continuació, per a cada puntuació del conjunt de dades, primer en farem una predicció (sense que el model en tingui coneixement) i aleshores introduïrem la puntuació al sistema. Acumularem l'error RMSE d'aquestes prediccions i cada 100,000 el reiniciarem a 0. És a dir, cada punt de les gràfiques mostrades correspondrà a la precisió d'un interval de 100,000 prediccions. Concretament, si té l'abscissa 100,000 correspondrà a la precisió en predir les 100,000 primeres puntuacions, si té la 200,000, a la precisió en predir les puntuacions entre la 100,000 i la 200,000, etc. Però cal remarcar que els errors no són acumulats, sinó que corresponen als intervals esmentats, per tal de poder veure millor la evolució de la precisió dels models. També cal aclarir que només tindrem en compte, per al càlcul d'aquest error, les puntuacions d'usuaris que hagin puntuat més de 5 ítems fins el moment. Creiem que és un mínim raonable, ja que amb menys puntuacions és difícil que les prediccions siguin precises.

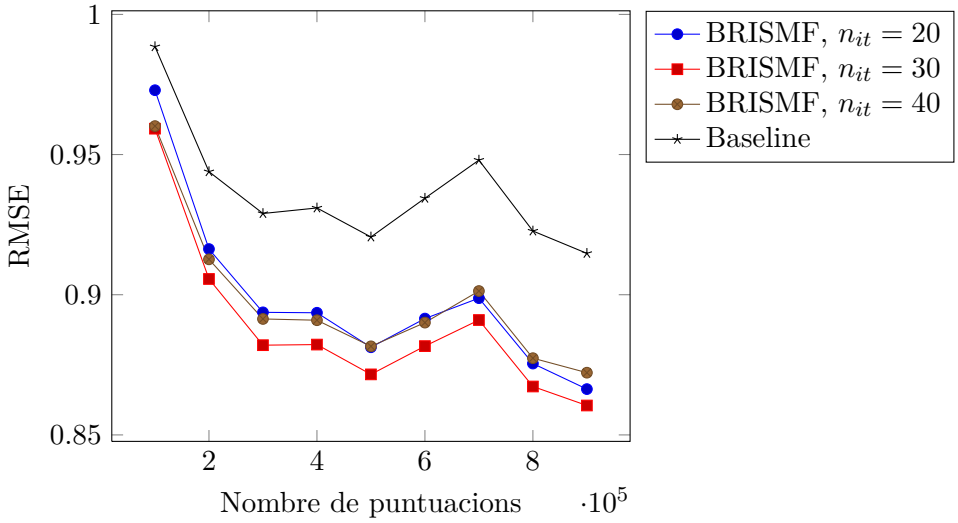
Recordant la secció 4.4 on s'explica l'algoritme, veurem que hi ha un aspecte de l'entrenament incremental que no queda ben resolt. Es tracta de la dificultat de conèixer, a diferència de l'entrenament *offline*, el nombre d'iteracions necessari per a que els vectors quedin ben entrenats, sense arribar al sobreentrenament.

Per a simplificar, deixarem aquest nombre fixat a una constant, $n_{it}$, tant per al reentrenament dels usuaris com per al d'ítems. Realitzarem vàries execucions per a intentar trobar un valor de $n_{it}$ el més òptim possible.

Per últim, i seguint amb l'algoritme *BRISMF*, avaluarem si certs entrenaments *offline* periòdics poden ajudar a millorar la possible pèrdua de precisió de les contínues actualitzacions incrementals.

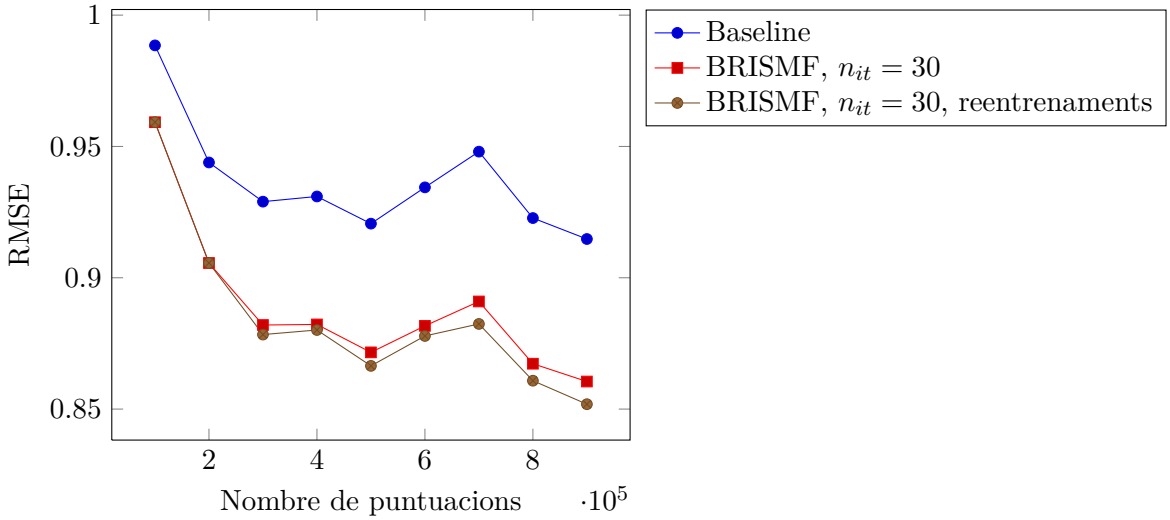
#### 7.2.2.1 Conjunt de dades Movielens

En primer lloc, hem realitzat la prova per a $n_{it} = 20, 30, 40$. A continuació mostrem la gràfica de l'evolució del RMSE per a cada interval de 100000 puntuacions, comparant els algoritmes *BRISMF* i *baseline*.




Es pot observar com l'error de les prediccions millora a mesura que els models disposen de més puntuacions. En els tres casos l'algoritme *BRISMF* supera al *baseline*, tot i que els millors resultats s'obtenen amb $n_{it} = 30$.

A continuació, realitzem el mateix entrenament incremental amb $n_{it} = 30$ però en aquest cas realitzant reentrenaments *offline* del model sencer *BRISMF* cada 200,000 puntuacions introduïdes. Els resultats obtinguts són els següents:




Sembla clar que els reentrenaments periòdics ajuden a millorar lleugerament la precisió del model. No obstant, resulta sorprenent el fet que es pugui entrenar el model des de zero únicament amb actualitzacions incrementals i obtenir una precisió en les prediccions comparable al cas on es fan reentrenaments cada 200,000 puntuacions. Més encara quan s'està fixant el nombre d'iteracions en aquest tipus d'entrenament a un nombre constant.

Podem comparar el RMSE mitjà obtingut al llarg de tota la avaluació online amb el millor obtingut a la prova offline:

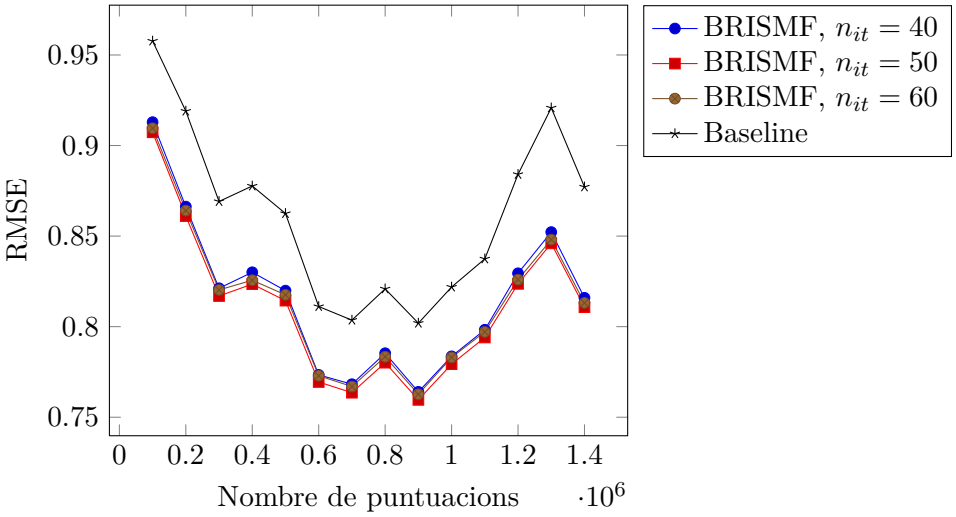
| | **Offline** | **Online** |
|---|---|---|
| **RMSE (BRISMF)** | 0.84567 | 0.88475 |
| **RMSE (Baseline)** | 0.92245 | 0.93698 |

Taula 7.7: Comparativa entre la prova *offline* i *online* per a Movielens

S'ha de tenir en compte que, d'entrada, l'entrenament *offline* sempre serà més precís que les actualitzacions incrementals. A més, l'avaluació de les prediccions a la prova *offline* es fa quan el model ja està completament entrenat, mentre que a l'*online* es realitza durant tot el procés d'entrenament. Amb això, creiem que els resultats de l'entrenament incremental del *BRISMF* són molt satisfactoris.

#### 7.2.2.2 Conjunt de dades Jester

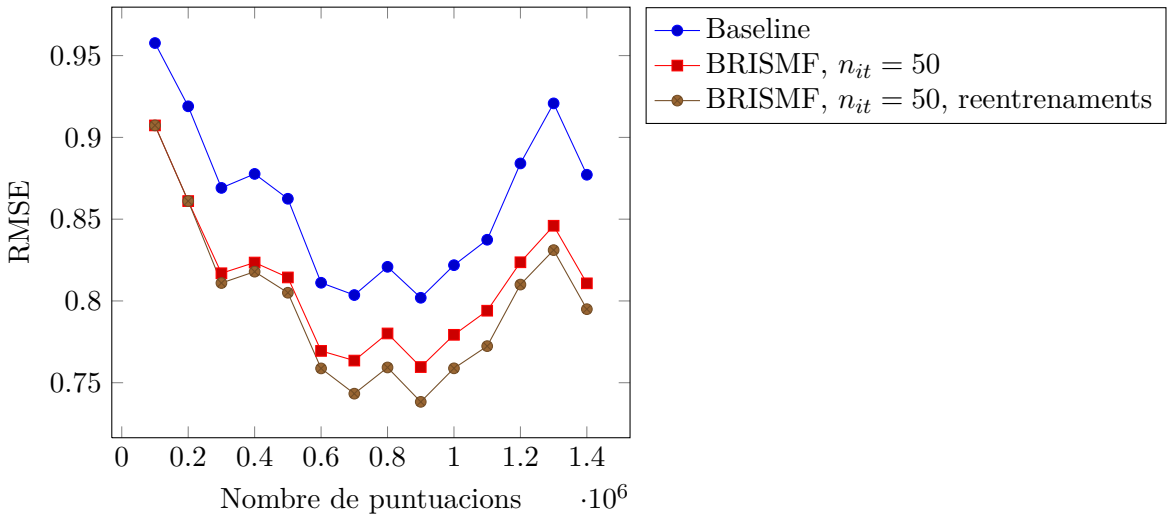
A continuació mostrem la gràfica amb els resultats variant el valor de $n_{it}$ entre 40 i 60:




En aquest cas, els millors resultats els obtenim amb $n_{it} = 50$. Podem observar que l'error en les prediccions s'estabilitza de seguida, a causa segurament del reduït nombre d'ítems (150). Això fa que els vectors característics dels ítems disposin de moltes més puntuacions que estiguin ben entrenats de seguida, ja que disposen de moltes puntuacions, cosa que no succeix de forma tan ràpida als altres conjunts de dades.

Les fluctuacions en la precisió que mostra la gràfica, més que errors en l'entrenament de l'algoritme *BRISMF*, sembla que estan relacionades amb la diferència en la dificultat de predir les puntuacions d'un interval respecte un altre, ja que es pot observar com els pics es repliquen també a l'algoritme *baseline*, que no pot patir de sobreentrenament.

Per últim, s'ha realitzat una altra execució, també amb $n_{it} = 50$ però reentrenaments *offline* cada 200,000 puntuacions. A continuació podem veure la comparativa:




De nou veiem com el reentrenament periòdic del model sencer afecta positivament a la precisió de l'algoritme *BRISMF*. Podem comparar el RMSE mitjà obtingut amb l'entrenament *offline*:

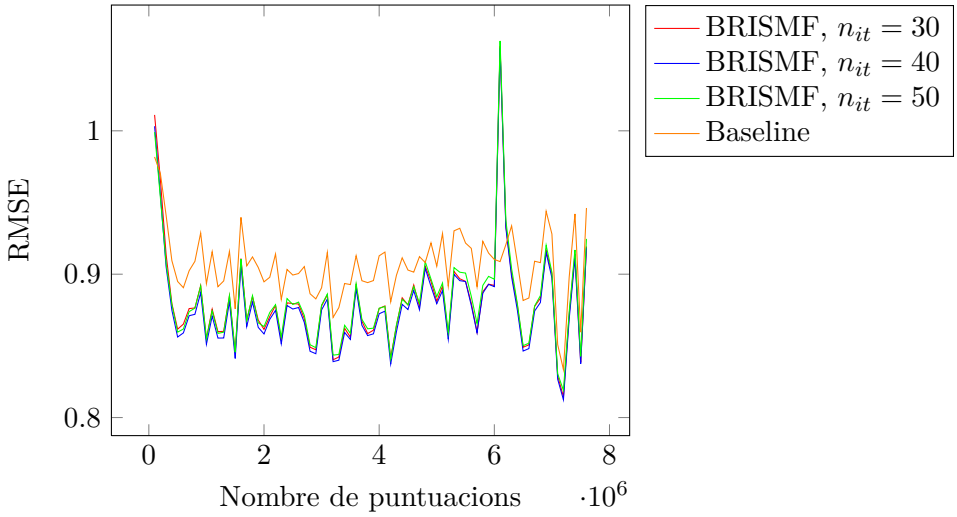
| | **Offline** | **Online** |
|---|---|---|
| **RMSE (BRISMF)** | 0.80700 | 0,79912 |
| **RMSE (Baseline)** | 0.87656 | 0.86303 |

Taula 7.8: Comparativa entre la prova *offline* i *online* per a Jester

En aquest cas, crida la atenció que la predicció mitjana de la prova *online* hagi estat millor que la de la prova *offline*. També succeix per al *baseline*, cosa que indica que el subconjunt utilitzat a la prova *offline* debia ser més difícil que la resta de puntuacions. També pot ser que la restricció a la prova *online* de només tenir en compte puntuacions a partir de que l'usuari ja ha valorat 5 ítems hagi jugat a favor de l'avaluació *online*. En tot cas, és indiscutible que la qualitat de les prediccions obtingudes a partir de l'entrenament incremental és molt bona.

#### 7.2.2.3 Conjunt de dades Flixster

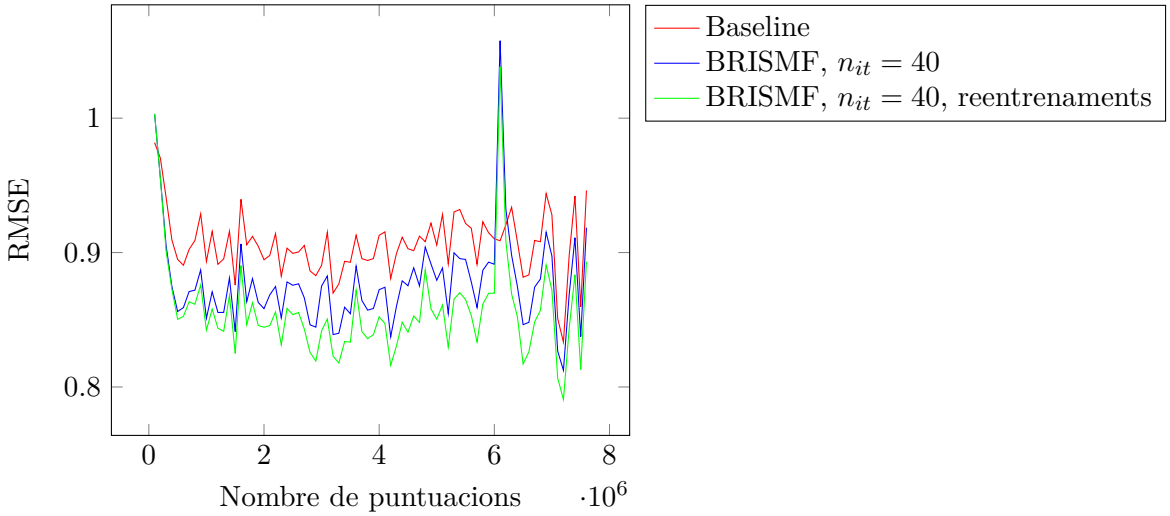
A continuació mostrem la gràfica amb els resultats variant el valor de $n_{it}$ entre 30 i 50:




Es pot detectar una anomalia respecte al vist fins ara, que consisteix en que a l'interval entre les 600,000 i 700,000 puntuacions, l'error RMSE dels tres algoritmes *BRISMF* creix molt per sobre del de l'algoritme *baseline*.

Per algun motiu en aquest interval les puntuacions són millor predites per l'algoritme *baseline*. Un dels motius que podria fer que això passés seria que en aquest punt s'haguessin introduït al sistema puntuacions aleàtories, un dels possibles atacs que es poden fer a un sistema recomanador per tal de distorsionar els models. En tot cas, sembla ser un cas puntual, ja que a continuació la gràfica continua amb els nivells de precisió típics.

Veiem a continuació la diferència fent reentrenaments *offline* del model cada 200,000 puntuacions:




Com era d'esperar, el reentrenament periòdic torna a millorar els resultats. Per últim, comparem el RMSE mitjà obtingut per a la millor execució *online* amb el RMSE obtingut

a la prova *offline*:

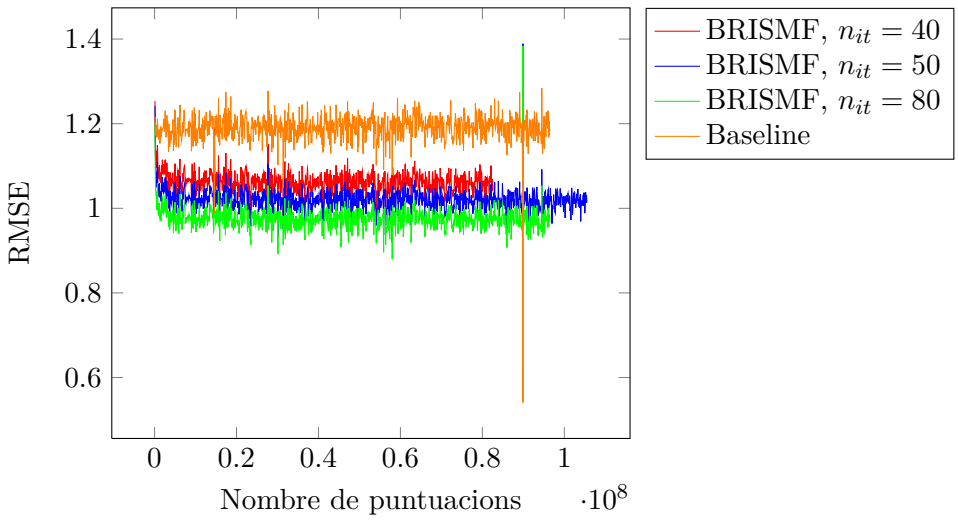
| | **Offline** | **Online** |
|---|---|---|
| **RMSE (BRISMF)** | 0.83421 | 0.85740 |
| **RMSE (Baseline)** | 0.90933 | 0.90617 |

Taula 7.9: Comparativa entre la prova *offline* i *online* per a Flixster

En aquest cas, veiem que els resultats de l'entrenament *online* del *BRISMF* són lleugerament inferiors al de l'*offline*, però la diferència és mínima i per tant els podem valorar molt positivament.

#### 7.2.2.4 Conjunt de dades Yahoo!

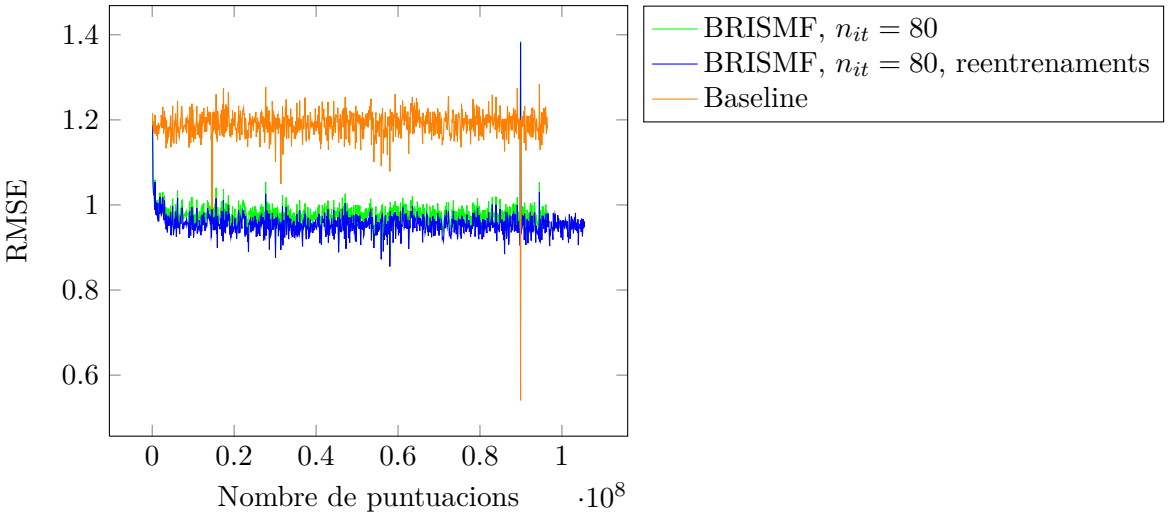
A continuació mostrem la gràfica amb algunes execucions variant el valor de $n_{it}$ entre 40 i 80:




A causa del temps elevat que es triga per a cadascuna d'aquestes execucions, es pot veure que en algunes no s'ha arribat fins el final (p.ex. a la de $n_{it} = 40$) en veure clarament que podien ser superades augmentant el nombre d'iteracions.

Cal destacar una anomalia semblant a la del dataset *Flixster*, que consisteix en que sobre les $9 \cdot 10^8$ puntuacions hi ha un pic d'error dels algoritmes *BRISMF* i en canvi el del *baseline* baixa dràsticament. La única explicació que veiem és la mateixa, que es tracti d'un atac on les puntuacions siguin molt extremes (per exemple, un usuari que dóna a tots els ítems amb la puntuació més baixa) de forma que al *BRISMF* li costi molt d'adaptar-s'hi i en canvi el *baseline*, en tenir en compte que la mitjana de l'usuari és molt baixa, obtingui una precisió molt alta. No tenim una explicació definitiva, però en tot cas la gràfica continua normalment a partir d'aquest punt.

Veiem a continuació la diferència fent reentrenaments *offline* del model cada 200,000 puntuacions:




Podem observar una petita millora en el cas dels reentrenaments periòdics. Per últim, comparem l'error mitjà RMSE obtingut a la millor execució amb el de la prova *offline*:

| | **Offline** | **Online** |
|---|---|---|
| **RMSE (BRISMF)** | 0.89169 | 0.95435 |
| **RMSE (Baseline)** | 1.16570 | 1.19015 |

Taula 7.10: Comparativa entre la prova *offline* i *online* per a Yahoo!

En aquest cas, la diferència entre l'entrenament incremental del *BRISMF* i l'*offline* és més gran que en els conjunts de dades previs. També és cert que aquest sembla ser el més difícil de predir, tenint en compte els resultats de l'algoritme *baseline*. Per això, creiem que encara podem valorar positivament l'entrenament incremental, tenint en compte que els resultats obtinguts segueixen sent molt millors que la precisió (i el rendiment) que ofereixen els altres algoritmes provats sobre el mateix conjunt de dades. Això sí, és possible que es necessitéssin algunes proves més per intentar calibrar els paràmetres, i veure si la precisió a l'entrenament *online* es pot acostar una mica més a la de l'*offline*.

### 7.2.3 Prova d'ordenació

Per tal de mesurar la qualitat dels predictors de puntuacions sense haver de passar per l'error RMSE, introduirem una mètrica alternativa vista a [13]. La idea és intentar mesurar com de bones son les prediccions a l'hora d'ordenar els ítems segons les preferències d'un

usuari. Definim el nombre de parelles concordants de puntuacions com

$$n_c = \sum_u |\{(i,j)|\hat{r}_{u,i} > \hat{r}_{u,j} \text{ and } r_{u,i} > r_{u,j}\}|$$

i de forma similar, el nombre de parelles discordants.

$$n_d = \sum_u |\{(i,j)|\hat{r}_{u,i} \geq \hat{r}_{u,j} \text{ and } r_{u,i} < r_{u,j}\}|$$

Aleshores, sumant per a tots els usuaris, podem definir la Fracció de Parelles Discordants, que mesura la proporció de puntuacions ordenades correctament.

$$FPC = \frac{n_c}{n_c + n_d}$$

Cal destacar que en aquest cas, a diferència del RMSE, com més gran és la magnitud, millor.

Seguirem la mateixa divisió entre conjunt d'entrenament i conjunt d'avaluació de la prova *offline*. Un cop entrenat el model, triarem $n_u$ usuaris a l'atzar del conjunt d'avaluació i calcularem el $FPC$ a partir de les seves puntuacions en aquest conjunt.

Avaluarem, altre cop, l'algoritme *baseline* i el *BRISMF*. Per al *BRISMF* es triaran els paràmetres que hagin donat millors resultats a la prova *offline*.

| | **Movielens** | **Jester** | **Flixter** | **Yahoo!** |
|---|---|---|---|---|
| **Baseline** | 0.7313 | 0.6104 | 0.5697 | 0.5683 |
| **BRISMF** | 0.7714 | 0.6403 | 0.6434 | 0.7796 |

Taula 7.11: $FPC$ per a cada dataset i algoritme

Els resultats mostren que l'algoritme *BRISMF* és millor que el *baseline* en la tasca d'ordenar els ítems segons la preferència dels usuaris. No obstant, la diferència no es tan gran com cabria esperar, amb l'excepció del cas de Yahoo!, on sí és notable (com també ho era en l'error RMSE obtingut).

Crida l'atenció que els millors resultats del *BRISMF* s'obtinguin en aquest conjunt de dades, amb un 77.96% de parelles de puntuacions ordenades correctament, quan s'ha obtingut l'error RMSE més elevat de tots els datasets. Es podria pensar que el rang original de les puntuacions, que era a $[0, 100]$, afavoreix la facilitat d'ordenar els ítems. No obstant, en el dataset Jester era $[-10, 10]$ i no s'han obtingut millors resultats que als altres. Segurament tingui a veure amb qualitats pròpies del conjunt de dades en qüestió.

### 7.2.4 Prova de recomanació

En aquesta prova intentarem mesurar directament com de bones són les recomanacions dels nostres algoritmes, amb idees basades en [14].

Emprarem dues mètriques molt utilitzades en tasques de classificació, les anomenades *precision* i *recall*. Siguin $tp$ el nombre de positius encertats, $tn$ el nombre de negatius encertats, $fp$ el nombre de positius fallats i $fn$ el nombre de negatius fallats. Definim les mètriques esmentades com:

$$\text{recall} = \frac{tp}{tp + fn} \tag{7.1}$$

$$\text{precision} = \frac{tp}{tp + fp} \tag{7.2}$$

Per aplicar-les al nostre problema, seguirem el següent procediment. Partirem de la mateixa partició entre conjunt d'entrenament i avaluació que en la prova *offline* i fixarem els paràmetres de l'algoritme *BRISMF* als millors obtinguts. Aleshores, per a cada usuari del conjunt d'entrenament generarem $n_r$ recomanacions. Per cada ítem del conjunt d'entrenament que a l'usuari li agradi i estigui a la llista de recomanacions, incrementarem $tp$. Per cada ítem del conjunt d'entrenament que a l'usuari li agradi i no estigui a la llista, incrementarem $fn$. I per cada ítem del conjunt d'entrenament que a l'usuari no li agradi i estigui a la llista dels recomanats incrementarem $fp$.

Decidim que a un usuari $u$ li agrada un ítem $i$ si i només si es compleix la següent condició:

$$r_{u,i} > 3 \wedge r_{u,i} > \bar{r_u} \tag{7.3}$$

És a dir, si la puntuació de l'ítem és més gran que 3 (recordem que s'ha normalitzat el rang de totes les puntuacions a $[1, 5]$) i la puntuació està per sobre de la mitjana de l'usuari.

Per una altra banda, també cal tenir en compte com es fa la selecció dels ítems candidats, és a dir, el subconjunt d'ítems (no han puntuats per l'usuari) per als quals predirem les puntuacions i en recomanarem els millors. Definirem, per a la resta de secció, $n_r$ com el nombre d'ítems que retornarà el recomanador i $n_c$ com el nombre d'ítems candidats que el recomanador avaluarà. Provarem dues estratègies de selecció d'ítems candidats:

- **Tots els ítems ($n_c \geq |I|$):** si el nombre de candidats és més gran o igual al nombre d'ítems al sistema, tot el pes de la recomanació racau en els algoritmes de predicció de puntuacions, ja que estem avaluant tots els ítems possibles.
- **Seleccionem els $10 \cdot n_r$ ítems més puntuats:** en aquest cas, seleccionarem com a ítems candidats els $n_c = 10 \cdot n_r$ ítems amb més puntuacions del sistema, és a dir, 10 vegades més que el nombre que finalment recomanarem.

S'han realitzat proves per a aquestes estratègies i per als algoritmes *baseline* i *BRISMF*, variant el nombre d'ítems recomanats $n_r$ de 10 a 100, en increments de 10 en 10. A l'apèndix B.2 es poden veure els resultats per a tots els conjunts de dades. En aquest apartat, a les taules 7.12 i 7.13 podem veure els resultats per a $n_r = 10 i n_r = 100$ i el conjunt de dades Movielens, i a les figures 7.1 i 7.2 les gràfiques dels resultats d'aquest

| | **Baseline** | | **BRISMF** | |
|---|---|---|---|---|
| | Precisió | Recall | Precisió | Recall |
| **Tots els ítems** | 3.71% | 87.38% | 4.86% | 92.83% |
| **Més votats** ($n_c = 10n_r$) | 5.48% | 84.15% | 7.58% | 89.54% |

Taula 7.12: Precisió/recall del dataset Movielens, per a $n_r = 10$

| | **Baseline** | | **BRISMF** | |
|---|---|---|---|---|
| | Precisió | Recall | Precisió | Recall |
| **Tots els ítems** | 18.68% | 81.08% | 21.51% | 86.48% |
| **Més votats** ($n_c = 10n_r$) | 21.06% | 80.53% | 27.44% | 84.69% |

Taula 7.13: Precisió/recall del dataset Movielens, per a $n_r = 100$

mateix conjunt de dades en funció de $n_r$. Els resultats per als altres conjunts de dades varien (de la mateixa forma que ho fa el nombre d'ítems dels mateixos), però l'ordre relatiu dels algoritmes i estratègies de selecció pel que fa a la precisió i *recall* es conserva.

Podem comprovar com, independentment de la estratègia de selecció d'ítems, l'algoritme *BRISMF* obté millors resultats que el *baseline* tant pel que fa a *recall* com per a la precisió.

Comparant les estratègies de selecció d'ítems utilitzades, la guanyadora pel que fa al *recall* és la segona, on s'avaluen els $10 \cdot n_r$ ítems més puntuats. No obstant, la situació canvia si ens fixem en la precisió, sent la millor estratègia en aquest cas la que selecciona tots els ítems del sistema. Per una altra banda, la precisió de la primera, i el *recall* de la segona són més que acceptables.

En tots els casos, podem veure que a mesura que augmenta $n_r$, el nombre d'ítems recomanats, també ho fa el *recall*, però al mateix temps la precisió, com és d'esperar, disminueix. Hem limitat els $n_r$ de la prova a 100, ja que creiem que en el context d'un sistema recomanador no té massa sentit recomanar un nombre d'ítems més elevat que aquest.

Podem concloure, pel que fa a aquesta prova, que l'algoritme *BRISMF* supera en precisió i *recall* de les recomanacions al *baseline*. Per una altra banda, la selecció dels ítems candidats també té molta influència en aquestes mètriques. Avaluant tots els ítems amb el *BRISMF* s'obté un bon resultat, però el *recall* millora considerablement si només ho fem sobre un subconjunt amb els $n_c$ ítems més puntuats, a canvi d'una pitjor precisió.

Des del nostre punt de vista, la precisió és més important que el *recall*. Una precisió baixa denota que el nostre sistema està recomanant ítems que no agraden a l'usuari, mentre que un *recall* baix simplement vol dir que no estem recomanant ítems que l'usuari ha vist i puntuat positivament. Però evidentment, una precisió molt alta no serveix de res si tenim un *recall* baix, i viceversa. Al final, qui hagi d'implantar el recomanador haurà de decidir on situar aquest equilibri, cosa que es pot aconseguir variant els paràmetres i l'estratègia de selecció d'ítems candidats, com indiquen les proves.

Figura 7.1: Precisió i recall avaluant tots els ítems, Movielens

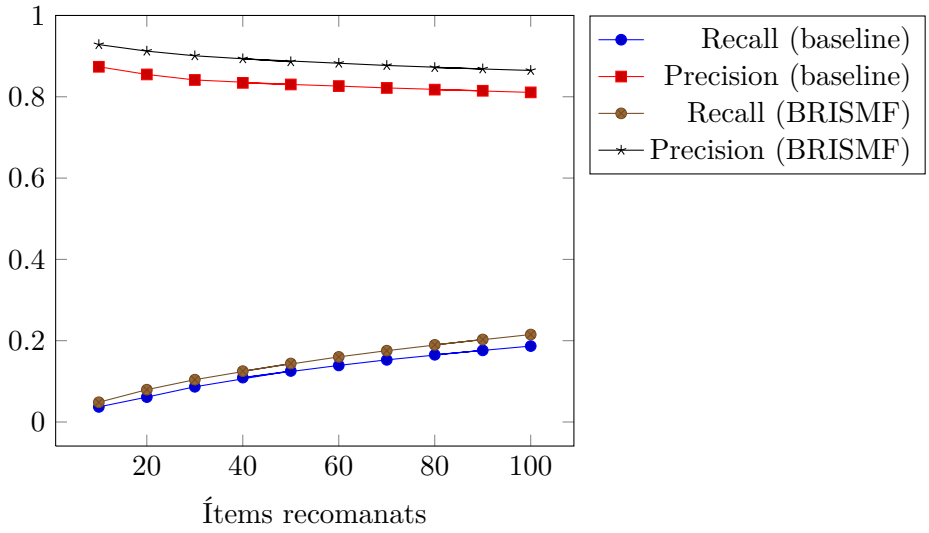
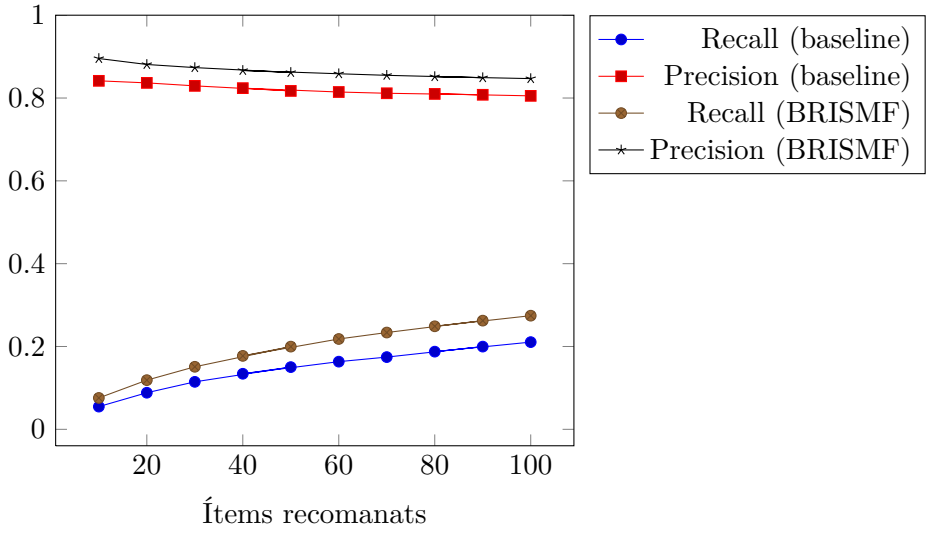



Figura 7.2: Precisió i recall avaluant els $n_c = 10n_r$ ítems més puntuats, Movielens



## 7.3 Avaluació del recomanador implementat

En aquest apartat avaluarem el recomanador de grups musicals implementat, fent-ne una valoració objectiva, al contrari que a la secció anterior. S'ha partit del conjunt de dades de Yahoo!, utilitzant l'algoritme *BRISMF* i els paràmetres que han donat millors resultats a les proves vistes anteriorment.

En primer lloc, provarem la funcionalitat de recomanar grups. Crearem varis perfils estereotipats d'usuari i comprovarem que les recomanacions són coherents. A continuació, analitzarem una funcionalitat addicional del recomanador implementat: la possibilitat de veure els grups o artistes més similars a un de donat, comparant els seus respectius vectors característics de factors.

### 7.3.1 Recomanació d'artistes musicals

Per als tres perfils (rock, pop, música espanyola) hem seguit el mateix procediment. En primer lloc, crear un perfil nou i introduïr algunes puntuacions de grups segons l'estil de música en qüestió. A continuació, generar les 10 millors recomanacions per al perfil. En algun cas, un cop obtingudes aquestes recomanacions, hem puntuat algun grup dels recomanats, per a refinar una mica els resultats.

#### 7.3.1.1 Perfil rock i heavy

Les puntuacions introduïdes apareixen a la següent taula.

| **Artista** | **Puntuació** |
|---|---|
| Guns & Roses | 4 |
| Dire Straits | 5 |
| Metallica | 5 |
| Iron Maiden | 4 |
| Led Zeppelin | 5 |

Taula 7.14: Puntuacions del perfil rock i heavy

I a continuació, les recomanacions obtingudes.

| **Artista** |
|---|
| Pink Floyd |
| Bob Marley And The Wailers |
| Eric Clapton |
| The Doors |
| Eagles |
| John Valby |
| Creedence Clearwater Revival |
| Lynyrd Skynyrd |
| The Rolling Stones |
| 38 Special |

Taula 7.15: Recomanacions del perfil rock i heavy

En general podem considerar que els resultats són coherents amb el tipus de perfil. Sí que crida la atenció la recomanació de *John Valby*, que és un artista que fa paròdies de cançons amb piano. Si mirem el seu perfil al recomanador, veurem que té 96 puntuacions, amb una puntuació mitjana de 4.11. I si ens fixem en el seu vector de factors, veurem que té un biaix de +1.52, és a dir, independentment de l'usuari, a l'avaluar la puntuació sempre se li sumarà 1.52. Això, en perfils d'usuaris amb poques puntuacions, pot suposar que d'entrada se li doni una puntuació molt alta. S'ha comprovat que d'aquests tipus d'artistes (relativament poques puntuacions, mitjana alta, biaix elevat) n'hi ha uns quants, i al principi poden aparèixer a la llista de recomanacions. No obstant, quan es puntuen negativament uns quants, aquests tipus de "outliers" tendeixen a desaparèixer.

#### 7.3.1.2 Perfil pop

Les puntuacions introduïdes per al perfil:

| **Artista** | **Puntuació** |
|---|---|
| Robbie Williams | 5 |
| Maroon 5 | 5 |
| Michael Jackson | 5 |
| Madonna | 4 |
| Shakira | 4 |

Taula 7.16: Puntuacions del perfil pop

I les recomanacions:

| **Artista** |
|---|
| Backstreet Boys |
| *NSYNC |
| Justin Timberlake |
| JC Chasez |
| Bob Marley And The Wailers |
| Thicke |
| Westlife |
| John Mayer |
| B.B. Mak |
| Brak |

Taula 7.17: Recomanacions del perfil pop

En general, els artistes recomanats pertanyen al gènere pop. Però novament, n'hi ha alguns que criden l'atenció. Per exemple, el fet que es recomani *Bob Marley And The Wailers* tant al perfil de pop com al de rock és sorprenent. També la inclusió d'algun "artista" com *Brak*, que de fet són uns dibuixos animats. Aquest darrer cas és anàleg al de *John Valby* de l'apartat anterior, té un biaix molt elevat. Hem decidit marcar aquests dos casos com *outliers* per al perfil pop (tot i que el primer és discutible), i puntuar-los amb un 2 i un 1, respectivament. Les noves recomanacions han estat:

| **Artista** |
|---|
| Backstreet Boys |
| Boyzone |
| *NSYNC |
| Five |
| Britney Spears |
| Steps |
| Westlife |
| O-Town |
| A1 |
| Scene 23 |

Taula 7.18: Segones recomanacions del perfil pop

Al final, veiem un predomini de *boy bands* de pop, juntament amb alguns altres artistes coneguts del gènere. Creiem que la llista és coherent amb el tipus de perfil que hem creat.

#### 7.3.1.3 Perfil grups espanyols

Les puntuacions del perfil:

| Artista | Puntuació |
|---|---|
| Alejandro Sanz | 5 |
| Café Quijano | 4 |
| Mecano | 5 |
| Oreja de Van Gogh | 4 |

Taula 7.19: Puntuacions del perfil de grups espanyols

I les recomanacions obtingudes:

| Artista |
|---|
| Juanes |
| Westlife |
| Tarkan |
| Enanitos Verdes |
| Andrea Bocelli |
| Jaguares |
| Eros Ramazzotti |
| Mana |
| Heroes Del Silencio |
| Ricardo Arjona |

Taula 7.20: Recomanacions del perfil de grups espanyols

Veiem que la majoria de recomanacions són d'artistes espanyols o latinoamericans, la qual cosa és coherent amb el perfil creat.

### 7.3.2 Artistes similars

La manera clàssica de calcular la similaritat entre dos usuaris o ítems, comparant els respectius vectors de puntuacions, pateix d'entrada un problema amb l'escassetat de dades: es necessita que disposin de moltes puntuacions per tal que els dos usuaris o ítems coincideixin en un nombre prou gran d'elles.

L'algoritme *BRISMF* ens ofereix una possibilitat molt interessant en aquest sentit. Gràcies a la representació compacta que fa d'usuaris i ítems (vectors de $F$ factors, on $F$ sol ser relativament reduït) ens permet calcular la similaritat entre dos usuaris o ítems de forma eficient i precisa, a partir de la similaritat cosinus entre els seus respectius vectors. Per exemple, en el cas del recomanador implementat, on s'han utilitzat 20 factors, calcular la similaritat entre dos usuaris o ítems costa poc més que 20 multiplicacions i 20 sumes. A més de resoldre els esmentats problemes d'escassetat de dades (ja que en aquest cas els vectors són densos), té l'avantatge de tenir en compte totes les puntuacions del sistema i

les relacions globals entre elles (per la manera de calcular els vectors característics), mentre que la similaritat utilitzada filtratge col·laboratiu basat en usuaris o ítems només te en compte les puntuacions locals, resultant en un càlcul, per força, molt més imprecís.

En aquest apartat intentarem avaluar com de bona és la similaritat entre artistes calculada a partir de la similaritat cosinus dels respectius vectors característics. Concretament, la funcionalitat implementada ofereix, per a cada artista o grup musical, els 10 artistes més similars. Intentarem fer una avaluació subjectiva, amb sentit comú, dels artistes similars obtinguts.

A continuació mostrem, per a diversos artistes, els 10 més similars que mostra el recomanador.

**The Beatles** Sembla clara la similaritat, com a mínim amb els tres primers artistes, ja que són ex-components del grup. Per als altres, la valoració entraria més dins el terreny de la subjectivitat (i de fet, el grau de similaritat mostrat també baixa) però en tot cas tots els artistes mostrats són raonablement similars.

| **Artista** | **Similaritat** |
|---|---|
| John Lennon | 96.08% |
| Paul McCartney | 91.83% |
| George Harrison | 86.43% |
| Elton John | 79.79% |
| The Mamas & The Papas | 79.28% |
| The Who | 78.33% |
| Simon & Garfunkel | 77.33% |
| Wings | 76.62% |
| Bob Dylan | 76.44% |
| The Doors | 76.34% |

Taula 7.21: Artistes similars a The Beatles

**Britney Spears** En aquest cas els graus de similaritat no són massa elevats, pero tot i així els artistes mostrats es poden considerar força similars a *Britney Spears*, en el sentit que tots són artistes o grups de pop comercial.

| Artista | Similaritat |
|---|---|
| Christina Aguilera | 79.46% |
| Jessica Simpson | 77.91% |
| *NSYNC | 72.2% |
| Dream [Pop] | 71.69% |
| Kelly Clarkson | 70.23% |
| The Party | 70.16% |
| Justin Timberlake | 70.0% |
| Mandy Moore | 69.73% |
| Emma Bunton | 68.03% |
| Backstreet Boys | 67.74% |

Taula 7.22: Artistes similars a Britney Spears

**Mozart** Aquí, els artistes mostrats sens dubte tenen molta relació amb Mozart. Tots pertanyen al món de la música clàssica. Seria discutible si, dins de la música clàssica són similars, però en principi, sense arribar a aquest tipus de subjectivitat, els resultats tenen molt de sentit.

| Artista | Similaritat |
|---|---|
| Beethoven | 99.15% |
| Antonio Vivaldi | 97.15% |
| Chopin | 96.95% |
| Chicago Symphony Orchestra | 96.13% |
| English Chamber Orchestra | 96.11% |
| Boston Pops | 93.75% |
| Luciano Pavarotti | 93.43% |
| Orchestre Symphonique De Montreal | 93.09% |
| Berlin Philharmonic Orchestra | 93.0% |
| Victoria Postnikova | 92.91% |

Taula 7.23: Artistes similars a Mozart

Aquests resultats pel que fa al càlcul de la similaritat obren una altra possibilitat a l'hora de fer recomanacions amb l'algoritme *BRISMF*. Es podria aprofitar el càlcul eficient i precís de la similaritat per a recomanar, simplement, ítems semblants a altres ítems que li hagin agradat a un usuari. És quelcom que no entra dins el paradigma de recomanador que ens havíem fixat en aquest projecte, basat en la predicció de puntuacions, però vistos els resultats valdria la pena tenir-ho en compte.

# Capítol 8

# Conclusions i treball futur

## 8.1 Conclusions generals del projecte

El primer objectiu del projecte era la creació d'una plataforma per a crear sistemes recomanadors escalables, eficients i que s'adaptin en temps real als gustos dels nous usuaris. Segons la nostra definició de sistema recomanador, la part central del procés consisteix en la predicció de puntuacions desconegudes. S'han implementat diversos algoritmes per a realitzar aquesta tasca, però sens dubte el que compleix amb tots els requisits plantejats prèviament és l'anomenat *BRISMF*. Els algoritmes de filtratge col·laboratiu basats en usuaris i ítems, que no requereixen d'entrenament, mostren un rendiment molt pobre a l'hora de predir puntuacions, tant en la qualitat de les mateixes com en el temps necessari per calcular-les. Per una altra banda, el *baseline* és un algoritme molt eficient, tot i que massa simple, produïnt unes prediccions acceptables però força millorables. Les proves han demostrat que, en canvi, el *BRISMF* ofereix unes excel·lents prediccions, i al mateix temps les calcula de forma extremadament ràpida. A canvi, requereix d'un petit cost addicional per a mantenir entrenat el model predictiu que fa servir.

Hi ha dos modes d'entrenament d'aquest model. En primer lloc, l'anomenat *offline*, que consisteix en entrenar-lo des de zero, utilitzant totes les puntuacions del sistema. Tot i ser el més precís, seria inviable realitzar-lo cada cop que s'afegeix alguna puntuació nova. I aquí es on entra en joc l'altre mode d'entrenament, l'incremental. En teoria no tan precís com l'*offline*, ens permet incorporar noves puntuacions de forma eficient a un model, sense haver de reentrenar-lo des de zero.

A la pràctica, ens ha sorprès el bon funcionament de l'entrenament incremental. Les proves mostren que fins i tot es pot arribar a entrenar un model buit només amb actualitzacions d'aquest tipus, obtenint quasi bé la mateixa precisió que s'obtindria entrenant-lo amb el mode *offline*. No obstant, els millors resultats s'han obtingut combinant ambdós modes, és a dir, realitzant per defecte actualitzacions incrementals i aplicant cada cert temps un reentrenament *offline* per a rectificar la possible pèrdua de precisió fruït d'aquestes

actualitzacions.

A banda de l'algoritme *BRISMF*, un altre component que ha fet possible assolir aquest objectiu ha estat sens dubte la utilització de la llibreria *JDBM3* per a l'emmagatzematge de les puntuacions. Quan es va començar a provar amb dades d'un volum considerable (més de 100 milions de puntuacions) de seguida es va comprovar que era inviable intentar mantenir-les a memòria. Es van provar diverses bases de dades, oferint totes un rendiment mediocre a mesura que el nombre de puntuacions introduïdes anava augmentant. Aleshores es va arribar a *JDBM3*, que oferia estructures de dades amb la mateixa interfície que les pròpies de Java (HashMap, Treemap...) i gestionava transparentment la seva persistència a disc, evitant problemes de memòria. Els resultats no podien haver estat millors, permetent-nos el que no vam poder aconseguir amb d'altres llibreries, introduïr i consultar les dades de forma extremadament ràpida, i sense degradació en el rendiment.

El segon objectiu del projecte consistia en la creació d'un sistema recomanador concret utilizant la llibreria implementada. Per a assolir-lo, s'ha utilitzat una interfície web sobre un servidor Tomcat. Ha estat de gran utilitat la facilitat amb què l'entorn de desenvolupament Eclipse permet crear projectes web i executar-los en un servidor Tomcat, tot integrat en el propi IDE. També ha estat de gran ajuda el fet que tot el codi (llibreria, servidor web, interfície web) fos en Java, resultant en una integració molt senzilla. Però, evidentment, la decisió de programar la llibreria en aquest llenguatge ja tenia en compte aspectes com aquest.

En definitiva, es pot afirmar que s'han aconseguit amb èxit els objectius plantejats inicialment. Pel camí, s'han aplicat coneixements generals vistos al llarg de la carrera, especialment els adquirits a les assignatures de programació i algorísmia. Per una altra banda, també han estat útils conceptes d'assignatures d'enginyeria del software, especialment en el disseny de l'arquitectura i les classes de la llibreria.

Però són també molts els coneixements adquirits gràcies a la realització d'aquest projecte. En primer lloc, la introducció al món dels recomanadors ja ha estat de per sí prou interessant. Per una altra banda, l'algoritme de factorització emprat és una tècnica que segur que pot resultar útil en molts altres àmbits. També el *gradient descent*, un mètode que l'autor desconeixia i que també sembla aplicable a molts altres contextes. Però cal destacar, per sobre de tot, l'interès que la realització d'aquest projecte ha despertat en l'autor per a seguir aprenent, tant en el camp dels sistemes recomanadors com en l'aprenentatge automàtic en general.

## 8.2 Treball futur

La realització del projecte ens deixa amb algunes idees de possibles millores, que no s'han pogut dur a terme bé per motius de temps, o bé perquè no entraven dins de l'abast que ens haviem plantejat.

Moltes tenen a veure amb l'algoritme *BRISMF*. En primer lloc, tot i haver donat resultats molts positius, és cert que s'ha hagut de calibrar realitzant diverses proves, per tal d'ajustar els paràmetres a cadascun dels conjunts de dades. Seria extremadament interessant deslliurar a l'implementador d'aquesta selecció de paràmetres, fent que s'adaptessin automàticament a mesura que s'introduissin més i més puntuacions.

Un altre punt problemàtic és, sens dubte, l'entrenament incremental. Concretament, no hem trobat una bona manera de decidir quan els vectors d'usuaris i ítems estan prou entrenats. En el nostre cas, per simplificar, hem fixat el nombre d'iteracions per a aquests entrenaments a una constant. Tot i que s'han obtingut resultats positius, creiem que és una estratègia millorable. Per exemple, la intuició ens diu que segurament els vectors d'ítems necessitin un nombre d'iteracions diferent que els dels usuaris. També és molt possible que aquest nombre vagi canviant a mesura que el nombre de puntuacions del sistema augmenta. En definitiva, creiem que és un aspecte on hi ha molt marge de millora. D'alguna forma està relacionat amb el que s'ha vist al paragràf anterior, ja que són paràmetres que també es podrien calcular adaptativament.

Les proves realitzades amb el recomanador implementat ens han permès veure artistes semblants a d'altres, calculats a partir de la similaritat dels seus vectors de factors. Això obre una nova possibilitat per a la factorització *BRISMF*. Es podrien provar estratègies de recomanació alternatives, on simplement es recomanéssin ítems similars a d'altres que a un usuari li hagin agradat, utilitzant la factorització per a calcular les similaritats. Més interessant seria, segurament, combinar les dues estratègies. És a dir, seleccionar ítems candidats que s'assemblin en més o menys mesura a ítems ben valorats per l'usuari, i aleshores predir les seves puntuacions i recomanar-ne els millors.

Per últim, creiem que seria interessant provar com de bé funcionaria l'algoritme *BRISMF* en un context on les puntuacions siguin unàries, és a dir, només amb *feedback* positiu. Un exemple conegut serien els típics "M'agrada" de la xarxa social *Facebook*. El fet que no hi hagi informació sobre què no li agrada a un usuari (*feedback* negatiu) fa dubtar sobre si el *BRISMF* funcionaria realment en aquest context, però en tot cas valdria la pena aprofundir-hi una mica.

# Bibliografia


[1] Yahoo Research. R1 - yahoo! music user ratings of musical artists, version 1.0. URL `http://webscope.sandbox.yahoo.com/catalog.php?datatype=r`. Citat a la pàgina 12.

[2] Apache Software Foundation. Apache mahout. . URL `http://mahout.apache.org`. Citat a la pàgina 13.

[3] Apache Software Foundation. Apache hadoop. . URL `http://hadoop.apache.org`. Citat a la pàgina 13.

[4] Zeno Gantner, Steffen Rendle, Lucas Drumond, and Christoph Freudenthaler. Mymedialite recommender system library. URL `http://mymedialite.net`. Citat a la pàgina 13.

[5] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors. *Recommender Systems Handbook*. Springer, 2011. ISBN 978-0-387-85819-7. Citat a la pàgina 18.

[6] Simon Funk. Netflix Update: Try this at Home. 2006. URL `http://sifter.org/~simon/journal/20061211.html`. Citat a les pàgines 19 i 36.

[7] Netflix. Netflix recommendations: Beyond the 5 stars (part 1). URL `http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`. Citat a la pàgina 19.

[8] Andreas Töscher, Michael Jahrer, and Robert M. Bell. The bigchaos solution to the netflix grand prize, 2009. Citat a la pàgina 25.

[9] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. Scalable collaborative filtering approaches for large recommender systems. *J. Mach. Learn. Res.*, 10:623–656, June 2009. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1577069.1577091`. Citat a les pàgines 31 i 33.

[10] Steffen Rendle and Lars Schmidt-Thieme. Online-updating regularized kernel matrix factorization models for large-scale recommender systems. In *Proceedings of the 2008 ACM conference on Recommender systems*, RecSys '08, pages 251–258, New York,

NY, USA, 2008. ACM. ISBN 978-1-60558-093-7. doi: 10.1145/1454008.1454047. URL http://doi.acm.org/10.1145/1454008.1454047. Citat a les pàgines 33 i 40.

[11] Matthew Brand. Fast online svd revisions for lightweight recommender systems. In *In SIAM International Conference on Data Mining*, 2003. Citat a les pàgines 48 i 49.

[12] Jan Kotek. Jdbm3 database. URL https://github.com/jankotek/JDBM3. Citat a la pàgina 66.

[13] Yehuda Koren and Joe Sill. Ordrec: an ordinal model for predicting personalized item rating distributions. In *RecSys*, pages 117–124, 2011. Citat a la pàgina 89.

[14] Elica Campochiaro, Riccardo Casatta, Paolo Cremonesi, and Roberto Turrin. Do metrics make recommender algorithms? In *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications Workshops*, WAINA '09, pages 648–653, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3639-2. doi: 10.1109/WAINA.2009.127. URL http://dx.doi.org/10.1109/WAINA.2009.127. Citat a la pàgina 90.

# Apèndix A

# Especificació completa de les classes de la llibreria

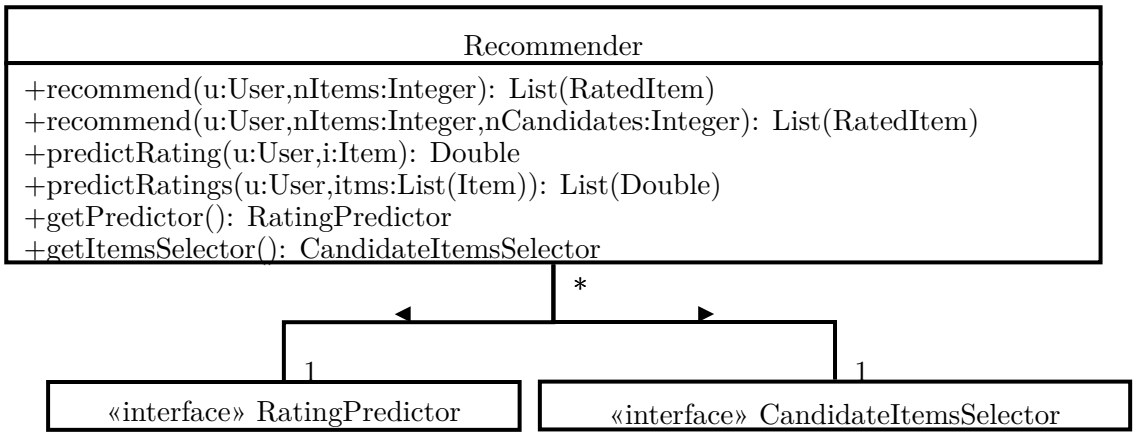
Aquesta secció pretén complementar la descripció de les classes de la llibreria vista a l'apartat 5, incloent els diagrames, descripcions d'operacions i atributs de les classes i interfícies del sistema que s'hi han omès.

## A.1 Classes principals

### A.1.1 Classe *Recommender*

És la classe que implementa les funcionalitats d'un recomanador. El comportament del recomanador vindrà definit per les implementacions de les instàncies de *RatingPredictor* i *CandidateItemsSelector* que rep el recomanador.

Figura A.1: Diagrama de la classe *Recommender*




**Operacions**

- **recommend(u:User,nItems:Integer):** retornar una llista de *nItems* recomanats per a l'usuari *u*.
- **recommend(u:User,nItems:Integer,nCandidates:Integer):** retornar una llista de *nItems* recomanats per a l'usuari *u*, fixant el nombre de ítems candidats que s'avaluaran a *nCandidates*.
- **predictRating(u:User,i:Item):** retornar la predicció de la puntuació que l'usuari *u* li donaria a l'ítem *i*.
- **predictRatings(u:User,itms:List(Item)):** retornar una llista de les prediccions de les puntuacions que l'usuari *u* li donaria als ítems continguts a *itms*.
- **getPredictor():** retorna la classe encarregada de predir les puntuacions.
- **getItemsSelector():** retorna la classe encarregada de seleccionar els ítems candidats a recomanar.

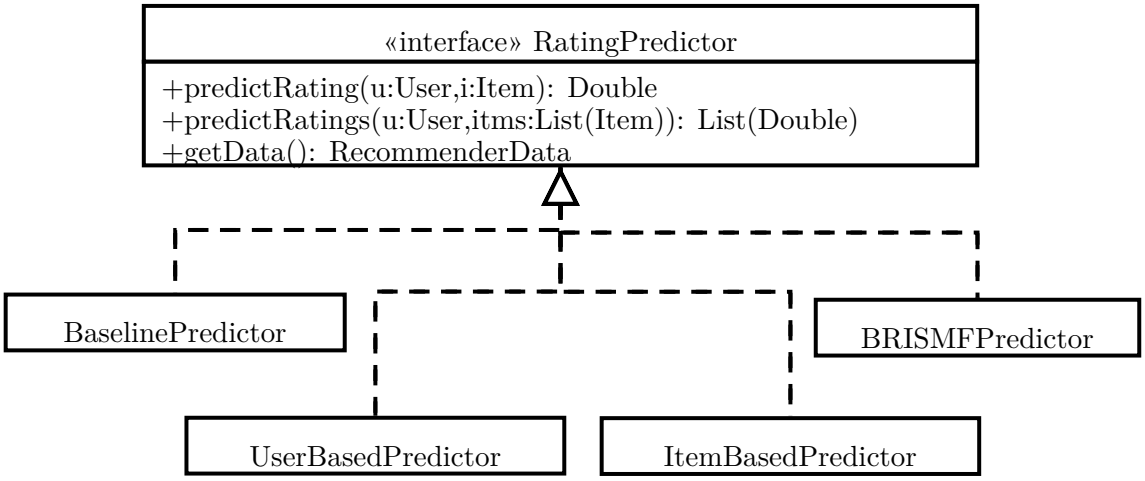
### A.1.2 Interfície *RatingPredictor*

Com s'ha esmentat anteriorment, la interfície serà utilitzada per la classe *Recommender* per a predir les puntuacions desconegudes. La implementació d'aquesta interfície és la part més crítica del recomanador. En bona part, serà el que el defineixi en quant a nivell de les recomanacions, escalabilitat, eficiència...

Tot i ser una interfície, s'ha inclòs l'operació *getData*, ja que s'ha suposat que totes les classes que la implementin tindran associades una instància de *RecommenderData* per a poder accedir a les dades del recomanador (puntuacions, estadístiques d'usuaris i ítems...).

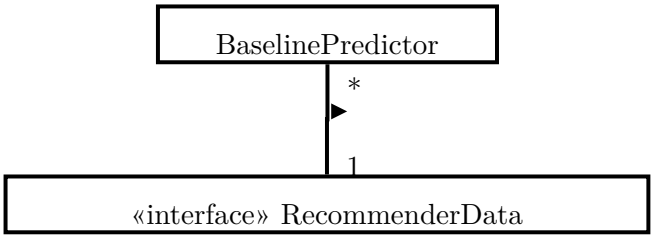
Les classes següents implementen la interfície *RatingPredictor* segons els algoritmes detallats a l'apartat 4. Totes hereden les operacions de *RatingPredictor*, però algunes n'implementen d'addicionals.

Figura A.2: Diagrama de la interfície *RatingPredictor*




#### A.1.2.1 Classe *BaselinePredictor*

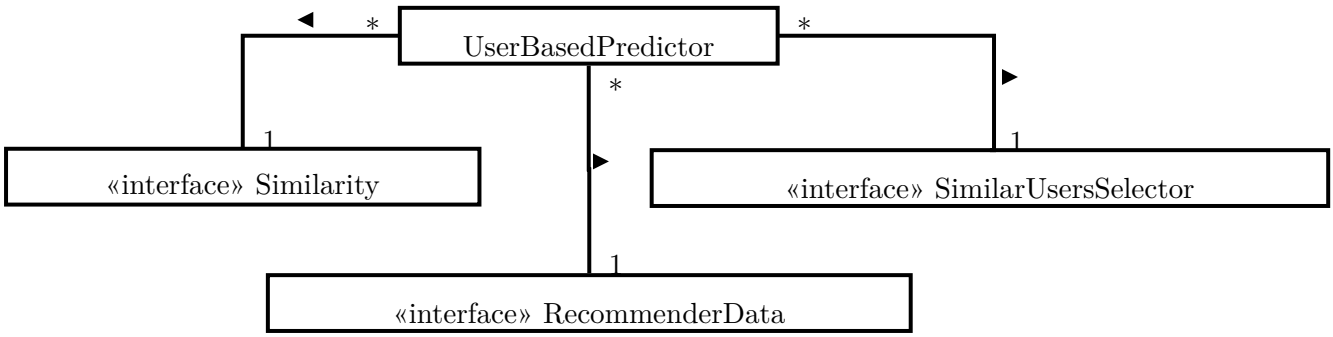
Figura A.3: Diagrama de la interfície *RatingPredictor*




Classe que implementa l'algoritme simple de l'apartat 4.1. No inclou operacions addicionals a la interfície *RatingPredictor*.

#### A.1.2.2 Classe *UserBasedPredictor*

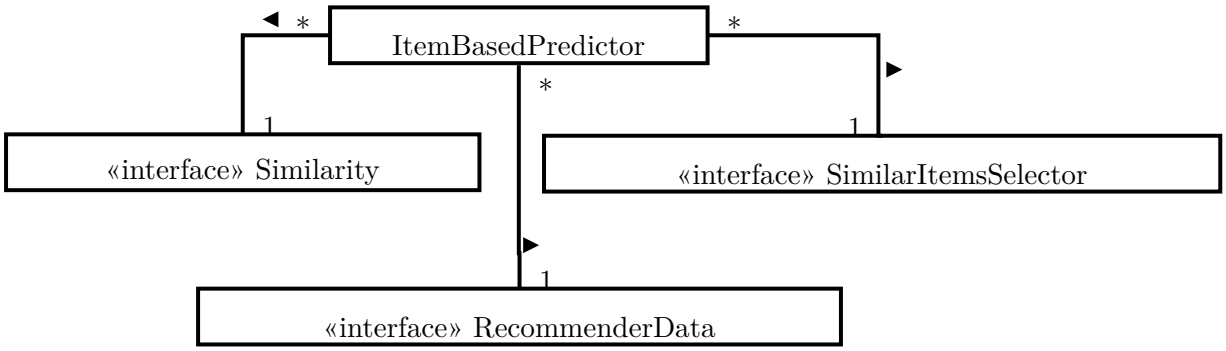
Figura A.4: Diagrama de la classe *UserBasedPredictor*




Classe que implementa l'algoritme de filtratge col·laboratiu basat en usuaris de l'apartat 4.2. La instància de *Similarity* s'utilitzarà per a calcular la similaritat entre dos usuaris, i la de *SimilarUsersSelector* per a seleccionar un subconjunt d'usuaris similars sobre el qual es realitzarà el sumatori de la fórmula (veure algoritme). D'aquesta forma es guanya en genericitat i modularitat, podent implementar qualsevol tipus de similaritat entre usuaris, i qualsevol estratègia per a seleccionar usuaris similars. La classe *UserBasedPredictor* no implementa mètodes addicionals a la interfície *RatingPredictor*.

#### A.1.2.3 Classe *ItemBasedPredictor*

Figura A.5: Diagrama de la classe *ItemBasedPredictor*




Classe que implementa l'algoritme de filtratge col·laboratiu basat en ítems de l'apartat 4.3, anàleg al *UserBasedPredictor*. La instància de *Similarity* s'utilitzarà per a calcular la similaritat entre dos ítems, i la de *SimilarItemsSelector* per a seleccionar un subconjunt d'ítems

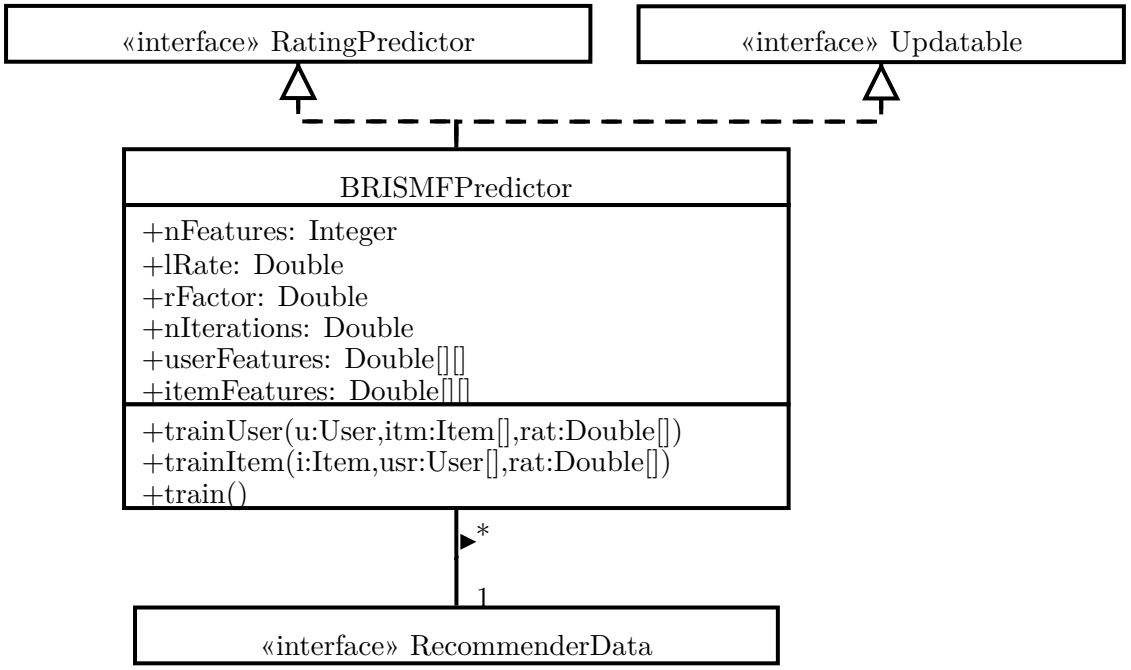
similars sobre el qual es realitzarà el sumatori de la fórmula (veure algoritme). La classe *ItemBasedPredictor* no implementa mètodes addicionals a la interfície *RatingPredictor*.

#### A.1.2.4 Classe *BRISMFPredictor*

Classe que implementa l'algoritme *BRISMF* de l'apartat 4.4. És la opció preferida en quant a qualitat de prediccions, escalabilitat i eficiència. Es farà una breu descripció de les operacions afegides a les pròpies de *RatingPredictor*, però cal tenir clars els conceptes de l'algoritme.

A més d'implementar la interfície *RatingPredictor*, implementa *Updatable*, de forma que pot ser notificada automàticament per un *RecommenderData* quan hi ha un canvi en les dades del sistema (veure apartats A.1.4 i A.2.6). També es farà una breu descripció del comportament d'algunes operacions heredades d'aquesta interfície.

Figura A.6: Diagrama de la classe *BRISMFPredictor*




**Atributs**

- **nFeatures:** Nombre de factors (dimensió dels vectors característics d'usuaris i ítems).
- **lRate:** Constant d'aprenentatge.
- **rFactor:** Factor de regularització.

- **nIterations:** Nombre d'iteracions utilitzat per l'entrenament *online* d'usuaris i ítems.
- **userFeatures:** conté els vectors característics de tots els usuaris.
- **itemFeatures:** conté els vectors característics de tots els ítems.

**Operacions**

- **trainUser(u:User,itm:Item[],rat:Double[]):** entrena el vector característic d'un usuari, utilitzant *nIterations* iteracions.
- **trainItem(i:Item,usr:User[],rat:Double[]):** entrena el vector característic d'un ítem, utilitzant *nIterations* iteracions.
- **train():** entrena el model sencer, és a dir, tots els vectors d'usuaris i ítems.

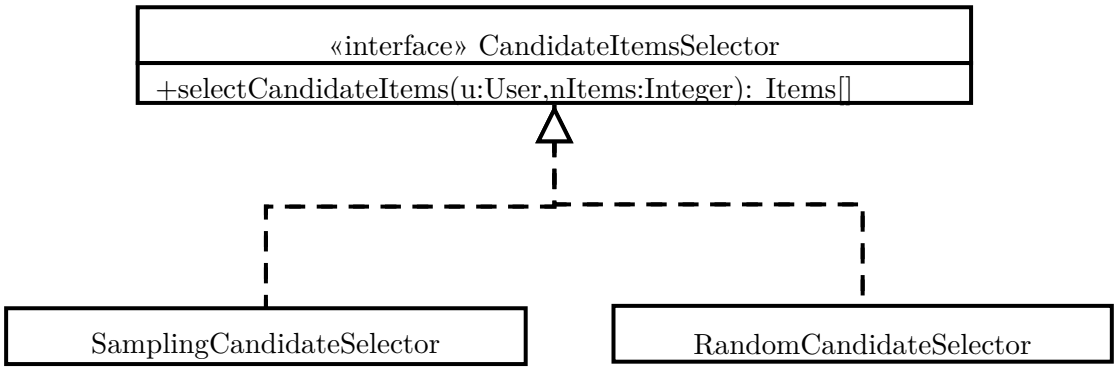
**Operacions heredades de *Updatable***

- **updateNewUser:** entrena el vector del nou usuari.
- **updateNewItem:** entrena el vector del nou ítem.
- **updateRemoveUser:** elimina el vector de l'usuari.
- **updateRemoveItem:** elimina el vector de l'ítem.
- **updateSetRating:** fa el que s'especifica al pàragraf *Actualitzacions en temps real* 4.4.1, és a dir, entrena l'usuari amb probabilitat funció del nombre de puntuacions del mateix, i entrena l'ítem amb probabilitat funció del nombre de puntuacions del mateix.
- **updateRemoveRating:** exactament el mateix que *updateSetRating*, però eliminant la puntuació en comptes d'afegint-la.

### A.1.3 Interfície *CandidateItemsSelector*

Aquesta interfície s'utilitza per a seleccionar un subconjunt d'ítems candidats a ser recomanats, típicament per a ser avaluats (puntuats per un *RatingPredictor*) i recomanar-ne els millors.

Figura A.7: Diagrama de la interfície *CandidateItemsSelector*



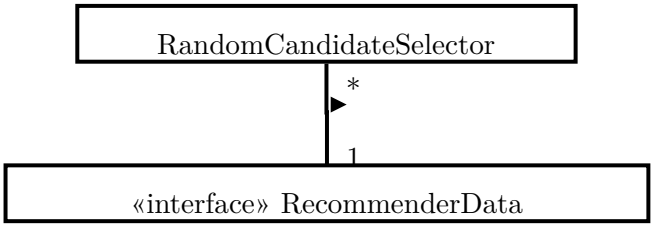

**Operacions**

- **selectCandidateItems(u:User,nItems:Integer):** retorna *nItems* ítems seleccionats amb algun criteri depenent de la implementació, amb la única restricció de que cap d'ells hagi estat puntuat per l'usuari $u$.

#### A.1.3.1 Classe *RandomCandidateSelector*

Per generar una llista de candidats, aquesta implementació en tria aleatòriament $k$ d'entre tots els ítems que no han estat puntuats per l'usuari $u$.

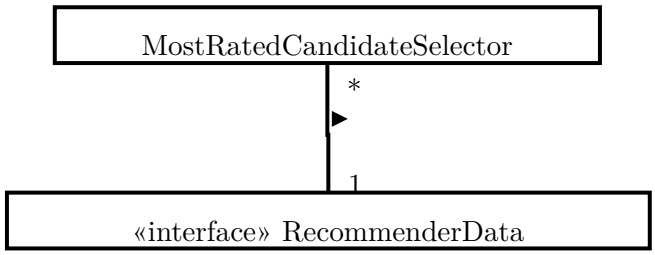
Figura A.8: Diagrama de la interfície *CandidateItemsSelector*




#### A.1.3.2 Classe *MostRatedCandidateSelector*

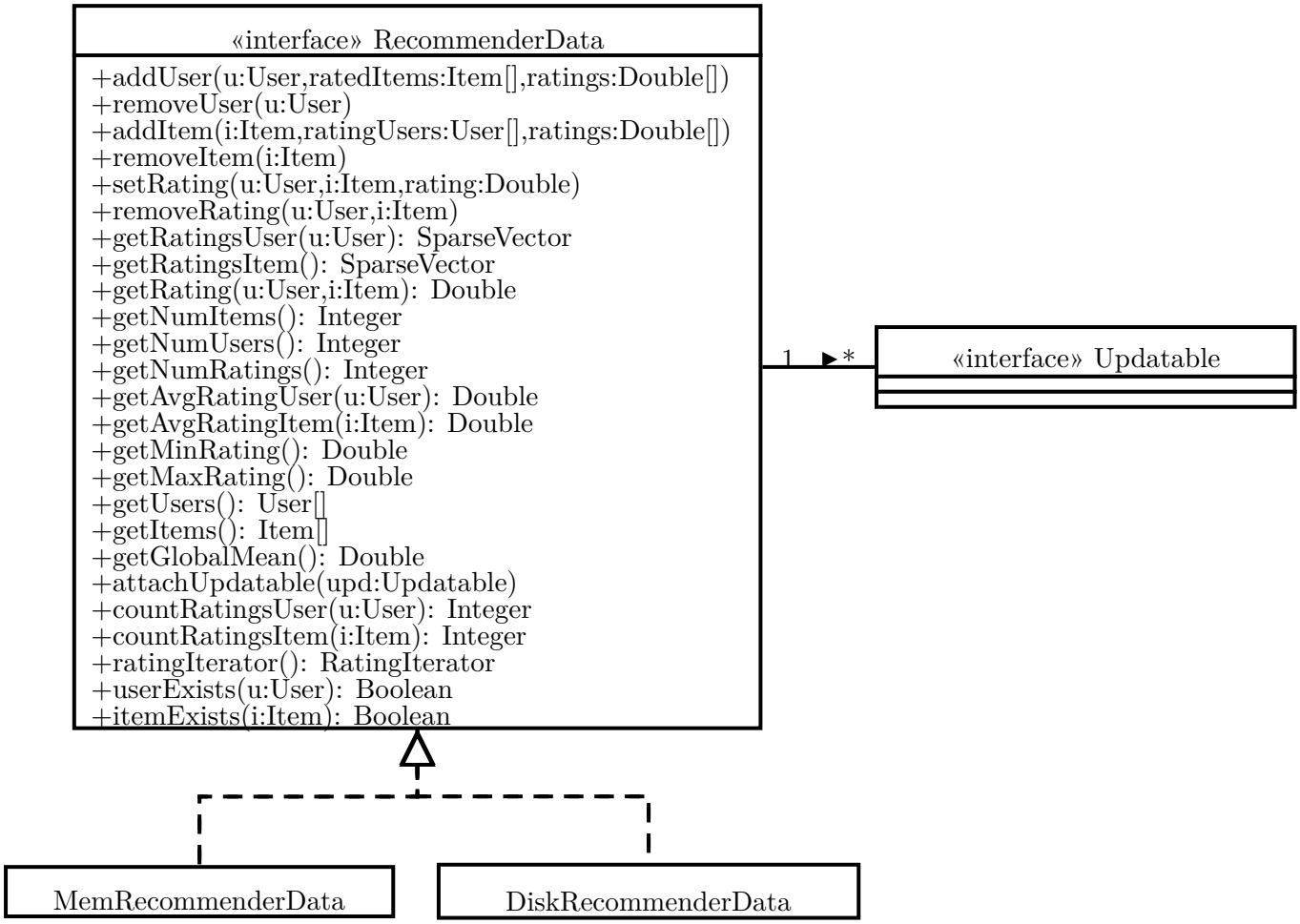
Per a generar una llista de $k$ ítems candidats per a un usuari $u$, es trien els $k$ ítems amb un nombre de puntuacions més elevat que no hagin estat puntuats per l'usuari $u$.

Figura A.9: Diagrama de la interfície *MostRatedCandidateSelector*




### A.1.4 Interfície *RecommenderData*

Figura A.10: Diagrama de la interfície *RecommenderData*



La interfície *RecommenderData* proporciona les operacions necessàries per a consultar i modificar les dades del recomanador. Tota classe que implementi *RecommenderData* podrà tenir associats una sèrie d'instàncies de *Updatable*. Aquestes instàncies seran notificades quan s'executin certes operacions al *RecommenderData*, seguint d'alguna forma el conegut *patró observador* d'enginyeria del software. La interfície *Updatable* s'explica amb més detall a l'apartat A.2.6.

**Operacions**

- **addUser(u:User,ratedItems:Item[],ratings:Double[]):** afegeix un usuari al sistema, possiblement amb algunes puntuacions inicials (llista d'ítems i llista de les seves respectives puntuacions). S'executa la operació *updateNewUser* per a tots els *Updatable* associats.
- **removeUser(u:User):** elimina un cert usuari del sistema. S'executa la operació *updateRemoveUser* per a tots els *Updatable* associats.
- **addItem(i:Item,ratingUsers:User[],ratings:Double[]):** afegeix un ítem al sistema, possiblement amb algunes puntuacions inicials (llista d'usuaris i llista de les seves respectives puntuacions). S'executa la operació *updateNewItem* per a tots els *Updatable* associats.
- **removeItem(i:Item):** elimina un ítem del sistema. S'executa la operació *updateRemoveItem* per a tots els *Updatable* associats.
- **setRating(u:User,i:Item,rating:Double):** introdueix una nova puntuació al sistema, és a dir: l'usuari que puntua, l'ítem puntuat i el valor de la puntuació. Si l'usuari ja havia puntuat l'ítem, se substitueix el valor de la puntuació antiga. S'executa la operació *updateSetRating* per a tots els *Updatable* associats.
- **removeRating(u:User,i:Item):** Elimina una puntuació del sistema. S'executa la operació *updateRemoveRating* per a tots els *Updatable* associats.
- **getRatingsUser(u:User):** Obté totes les puntuacions que ha realitzat un usuari (codificades com un *SparseVector*).
- **getRatingsItem():** Obté totes les puntuacions d'un ítem (codificades com un *SparseVector*).
- **getRating(u:User,i:Item):** Obté la puntuació de l'usuari $u$ a l'ítem $i$ si existeix, *null* altrament.
- **getNumItems():** Retorna el nombre d'ítems del sistema.
- **getNumUsers():** Retorna el nombred d'usuaris del sistema.
- **getNumRatings():** Retorna el nombre de puntuacions del sistema.

- **getAvgRatingUser(u:User):** Retorna la puntuació mitjana de l'usuari $u$ (calculada segons la fórmula de l'apartat 4.1).
- **getAvgRatingItem(i:Item):** Retorna la puntuació mitjana de l'ítem $i$ (calculada segons la fórmula de l'apartat 4.1).
- **getGlobalMean():** Retorna la puntuació mitjana global.
- **getMinRating():** Retorna la puntuació mínima del sistema.
- **getMaxRating():** Retorna la puntuació màxima del sistema.
- **getUsers():** Retorna el conjunt de tots els usuaris.
- **getItems():** Retorna el conjunt de tots els ítems.
- **attachUpdatable(upd:Updatable):** Afegeix un nou *Updatable* a notificar quan hi hagi modificacions a les dades.
- **countRatingsUser(u:User):** Retorna el nombre de puntuacions de l'usuari $u$.
- **countRatingsItem(i:Item):** Retorna el nombre de puntuacions de l'ítem $i$.
- **ratingIterator():** Retorna un iterador sobre totes les puntuacions del sistema (permet recórrer-les totes sense haver d'emmagatzemar-les en memòria).
- **userExists(u:User):** Indica si un cert usuari existeix al sistema.
- **itemExists(i:Item):** Indica si un cert ítem existeix al sistema.

#### A.1.4.1 MemRecommenderData

Implementació on totes les dades s'emmagatzemen en memòria. Això fa que per una banda la implementació sigui ràpida, però per l'altra pot provocar problemes de falta de memòria quan el nombre d'usuaris, ítems i puntuacions és relativament elevat.

#### A.1.4.2 DiskRecommenderData

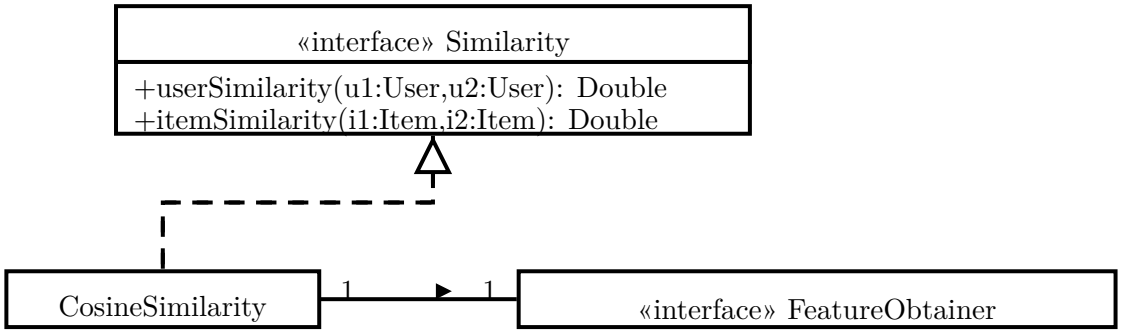
Implementació que utilitza el disc dur per a emmagatzemar les dades. És més lenta que *MemRecommenderData*, però no dóna problemes de memòria.

## A.2 Classes auxiliars

En aquest apartat detallarem les classes i interfícies auxiliars utilitzades a les llibreries.

### A.2.1 Interfície *Similarity*

Figura A.11: Diagrama de la interfície *Similarity*




Interfície que proporciona les operacions per calcular la similaritat entre dos usuaris o dos ítems.

**Operacions**

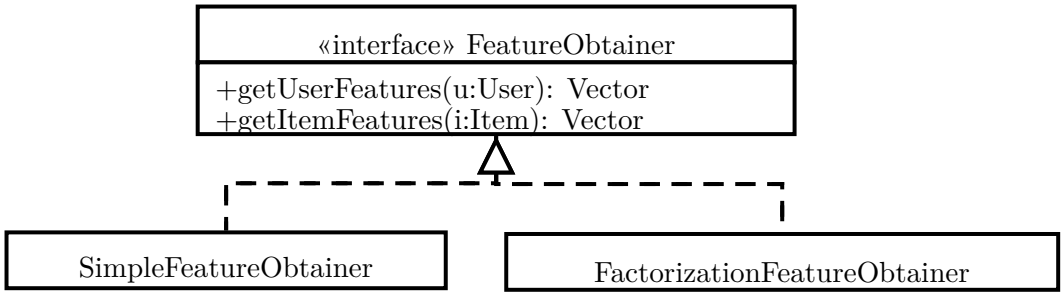
- **userSimilarity(u1:User,u2:User):** retorna la similaritat entre l'usuari $u1$ i $u2$.
- **itemSimilarity(i1:Item,i2:Item):** retorna la similaritat entre l'ítem $i1$ i $i2$.

#### A.2.1.1 Classe *CosineSimilarity*

Implementació que utilitza la similaritat cosinus, mencionada a l'apartat 2.4.2.1 per a mesurar la semblança entre dos usuaris o ítems. S'utilitza una implementació de *FeatureObtainer* associada per a obtenir un vector associat a un usuari o ítem (per exemple, un vector de puntuacions, o un vector de factors en el cas d'una factorització). Aleshores, per a calcular la similaritat entre dos usuaris o ítems, només cal calcular la similaritat cosinus entre els seus dos vectors associats.

### A.2.2 Interfície *FeatureObtainer*

Figura A.12: Diagrama de la interfície *FeatureObtainer*




Interfície que proporciona operacions per a obtenir vectors associats a un usuari o ítem, de forma que calcular la similaritat entre dos vectors d'un usuari o ítem sigui equivalent a la similaritat entre els seus corresponents usuaris o ítems.

**Operacions**

- **getUserFeatures(u:User):** retorna el vector associat a l'usuari $u$.
- **getItemFeatures(i:Item):** retorna el vector associat a l'ítem $i$.

#### A.2.2.1 Classe *SimpleFeatureObtainer*

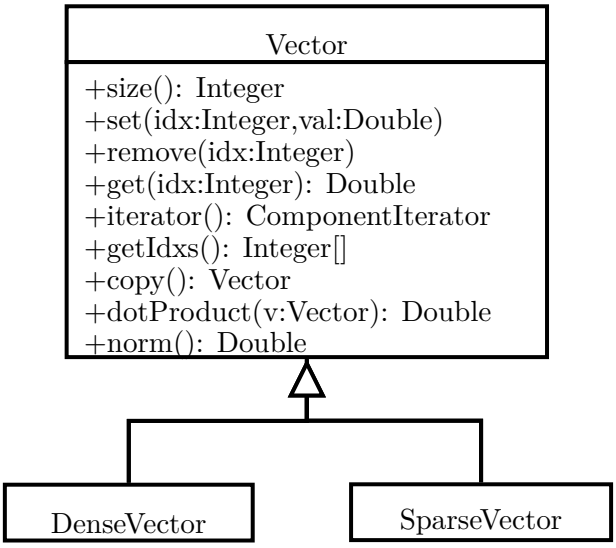
Implementació que retorna com a vector associat a un usuari o ítem el seu vector de puntuacions. Si es tracta del vector d'un usuari, hi haurà tants components com ítems, i cadascun representarà una puntuació. Si es tracta del vector d'un ítem tindrà tants components com usuaris. Aquests vectors típicament seran dispersos, és per això que s'implementaran amb la subclasse *SparseVector*.

#### A.2.2.2 Classe *FactorizationFeatureObtainer*

Utilitza una factorització (per exemple, algoritme *BRISMF* de l'apartat 4.4) i retorna com a vector associat a un usuari o ítem el seu vector de factors que el caracteritza.

### A.2.3 Classe *Vector*

Figura A.13: Diagrama de la classe *Vector*




Classe abstracta (no és interfície perquè implementa *dotProduct* i *norm*) que inclou operacions per operar amb vectors.

**Operacions**

- **size():** retorna el nombre d'elements del vector.
- **set(idx:Integer,val:Double):** introdueix al component número *idx* el valor *val*.
- **remove(idx:Integer):** elimina el component número *idx*.
- **get(idx:Integer):** obté el component número *idx*.
- **iterator():** retorna un iterador de tots els components del vector (parells *idx*, *val*).
- **getIdxs():** obté els índexs de tots els components del vector (no sempre van de 0 a *size()* - 1, a la implementació de vectors dispersos típicament mancaran molts components).
- **copy():** retorna una còpia del vector.
- **dotProduct(v:Vector):** calcula el producte escalar amb el vector $v$.
- **norm():** calcula la norma del vector.

#### A.2.3.1 Classe *DenseVector*

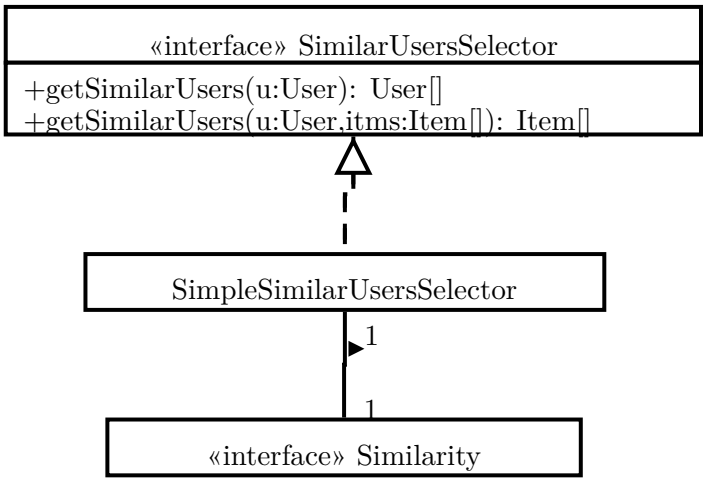
Implementació per a vectors densos, on els índexs dels components seran consecutius.

#### A.2.3.2 Classe *SparseVector*

Implementació per a vectors dispersos, on els índexs dels components poden ser qualsevol nombre arbitrari (de fet, és com un conjunt de parells «clau(*Integer*), valor(*Double*)»).

### A.2.4 Interfície *SimilarUsersSelector*

Figura A.14: Diagrama de la interfície *SimilarUsersSelector*



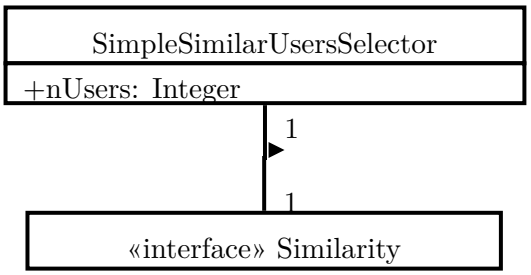

Interfície que proporciona operacions per a trobar un conjunt d'usuaris similars a un donat.

**Operacions**

- **getSimilarUsers(u:User):** obté un conjunt d'usuaris similars a $u$.
- **getSimilarUsers(u:User,i:Item):** obté un conjunt d'usuaris tal que $u$ ha puntuat almenys un ítem de *itms*.

#### A.2.4.1 Classe *SimpleSimilarUsersSelector*

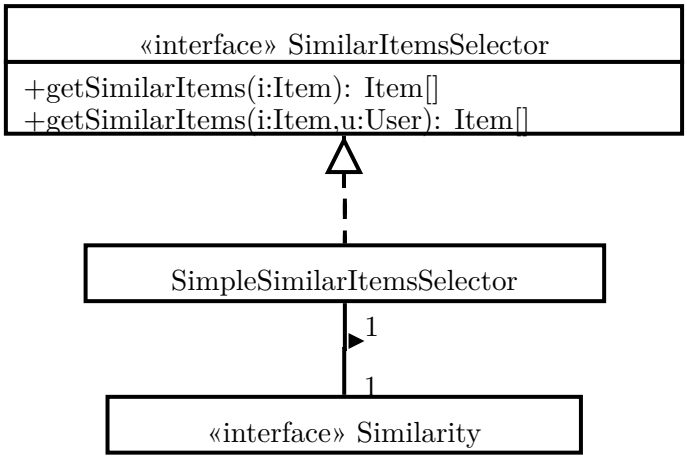
Figura A.15: Diagrama de la classe *SimpleSimilarUsersSelector*




Implementació que calcula la similaritat de tots els usuaris (amb la instància de *Similarity* associada) amb l'usuari donat i en retorna els *nUsers* més similars.

### A.2.5 Interfície *SimilarItemsSelector*

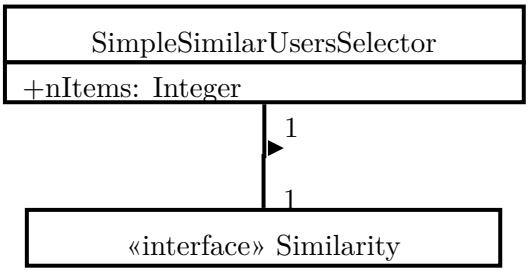
Figura A.16: Diagrama de la interfície *SimilarItemsSelector*




Interfície que proporciona operacions per a trobar un conjunt d'ítems similars a un donat.

#### A.2.5.1 Classe *SimpleSimilarItemsSelector*

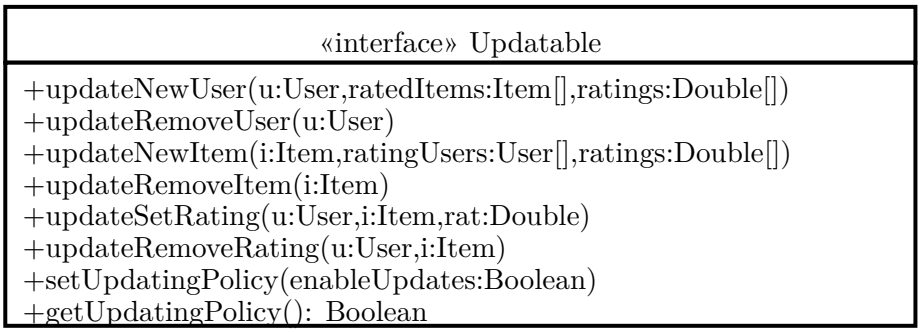
Figura A.17: Diagrama de la classe *SimpleSimilarItemsSelector*




Implementació que calcula la similaritat de tots els ítems (amb la instància de *Similarity* associada) amb l'ítem donat i en retorna els *nItems* més similars.

### A.2.6 Interfície *Updatable*

Figura A.18: Diagrama de la interfície *Updatable*




Interfície que proporciona operacions que han de permetre a una classe que la implementi ser actualitzada automàticament quan hi hagi canvis en les dades del sistema. Intenta aplicar la idea del *patró Observador*, on hi ha un subjecte (*RecommenderData*) que quan pateix un canvi d'estat (operació que modifica les dades) ho notifica a una sèrie d'observadors (varis *Updatable*).

**Operacions**

- **updateNewUser(u:User,ratedItems:Item[],ratings:Double[]):** es crida quan s'afegeix un nou usuari al sistema.
- **updateRemoveUser(u:User):** es crida quan s'elimina un usuari del sistema.

- **updateNewItem(i:Item,ratingUsers:User[],ratings:Double[]):** es crida quan s'afegeix un nou ítem al sistema.
- **updateRemoveItem(i:Item):** es crida quan s'elimina un ítem del sistema.
- **updateSetRating(u:User,i:Item,rat:Double):** es crida quan s'afegeit una puntuació al sistema.
- **updateRemoveRating(u:User,i:Item):** es crida quan s'elimina una puntuació del sistema.
- **setUpdatingPolicy(enableUpdates:Boolean):** permet activar o desactivar les actualitzacions automàtiques per a aquest *Updatable*.
- **getUpdatingPolicy():** pregunta si les actualitzacions automàtiques estan activades o no.

# Apèndix B

# Resultats numèrics de les proves realitzades

## B.1 Prova offline

### B.1.1 Filtratge col·laboratiu basat en usuaris i ítems

| | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| **Movielens** | 0.93526 | 0.91905 | 0.91498 | 0.91333 | 0.91324 |
| **Jester** | 1.00141 | 0.98649 | 0.98095 | 0.97940 | 0.97767 |
| **Flixter** | 0.97609 | 0.96142 | 0.95268 | 0.95272 | 0.95208 |
| **Yahoo!** | N/A | N/A | N/A | N/A | N/A |

Taula B.1: RMSE en funció de $K$ per a l'algoritme basat en usuaris

| | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| **Movielens** | 0.89186 | 0.87973 | 0.87833 | 0.88189 | 0.88588 |
| **Jester** | 0.95480 | 0.94121 | 0.94641 | 0.95307 | 0.95533 |
| **Flixter** | 0.93001 | 0.92940 | 0.93057 | 0.93261 | 0.93222 |
| **Yahoo!** | N/A | N/A | N/A | N/A | N/A |

Taula B.2: RMSE en funció de $K$ per a l'algoritme basat en ítems

### B.1.2 BRISMF

#### B.1.2.1 Conjunt de dades Movielens

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** | **0.004** |
|---|---|---|---|---|
| **0.005** | 0.90656/7s | 0.85488/20s | 0.85475/14s | 0.85478/10s |
| **0.01** | 0.90661/7s | 0.85234/22s | 0.85217/15s | 0.85221/11s |
| **0.015** | 0.90675/9s | 0.90579/6s | 0.85018/16s | 0.85020/12s |
| **0.02** | 0.90717/10s | 0.90623/7s | **0.84845/18s** | 0.84854/14s |

Taula B.3: RMSE/temps d'entrenament, algoritme BRISMF, $F = 20$, dataset Movielens

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** | **0.004** |
|---|---|---|---|---|
| **0.005** | 0.90644/13s | 0.85402/27s | 0.85401/18s | 0.85402/14s |
| **0.01** | 0.90649/13s | 0.85137/30s | 0.85125/20s | 0.85134/15s |
| **0.015** | 0.90657/14s | 0.84922/33s | 0.84912/22s | 0.84920/17s |
| **0.02** | 0.90701/15s | 0.90584/8s | **0.84741/25s** | 0.84747/19s |

Taula B.4: RMSE/temps d'entrenament, algoritme BRISMF, $F = 30$, dataset Movielens

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** | **0.004** |
|---|---|---|---|---|
| **0.005** | 0.90643/16s | 0.85208/35s | 0.85205/24s | 0.85216/18s |
| **0.01** | 0.90648/17s | 0.84949/38s | 0.84943/26s | 0.84957/20s |
| **0.015** | 0.90655/21s | 0.84739/42s | 0.84734/28s | 0.84742/22s |
| **0.02** | 0.90697/25s | 0.84569/46s | **0.84567/31s** | 0.84574/23s |

Taula B.5: RMSE/temps d'entrenament, algoritme BRISMF, $F = 40$, dataset Movielens

#### B.1.2.2 Conjunt de dades Jester

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** | **0.004** |
|---|---|---|---|---|
| **0.005** | 0.81410/38s | 0.81474/20s | 0.81520/14s | 0.81575/11s |
| **0.01** | 0.81168/49s | 0.81204/26s | 0.81235/18s | 0.81274/14s |
| **0.015** | 0.81073/60s | 0.81062/33s | 0.81083/23s | 0.81114/18s |
| **0.02** | 0.81082/70s | 0.81000/40s | **0.81025/27s** | 0.81045/22s |

Taula B.6: RMSE/temps d'entrenament, algoritme BRISMF, $F = 20$, dataset Jester

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** | **0.004** |
|---|---|---|---|---|
| **0.005** | 0.81075/54s | 0.81169/29s | 0.81241/20s | 0.81306/15s |
| **0.01** | 0.80862/70s | 0.80917/38s | 0.80988/25s | 0.81044/19s |
| **0.015** | 0.80801/84s | 0.80811/46s | 0.80863/32s | 0.80912/25s |
| **0.02** | 0.80817/100s | **0.80788/55s** | 0.80826/38s | 0.80861/30s |

Taula B.7: RMSE/temps d'entrenament, algoritme BRISMF, $F = 30$, dataset Jester

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** | **0.004** |
|---|---|---|---|---|
| **0.005** | 0.80962/69s | 0.81054/37s | 0.81124/25s | 0.81186/19s |
| **0.01** | 0.80749/91s | 0.80824/48s | 0.80883/33s | 0.80939/26s |
| **0.015** | 0.80713/109s | 0.80734/60s | 0.80784/42s | 0.80829/33s |
| **0.02** | 0.80732/130s | **0.80700/73s** | 0.80745/51s | 0.80786/39s |

Taula B.8: RMSE/temps d'entrenament, algoritme BRISMF, $F = 40$, dataset Jester

#### B.1.2.3 Conjunt de dades Flixter

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** | **0.004** |
|---|---|---|---|---|
| **0.005** | 0.84247/237s | 0.84247/127s | 0.84268/85s | 0.84300/64s |
| **0.01** | 0.83909/286s | 0.83906/154s | 0.83940/100s | 0.83975/79s |
| **0.015** | 0.83773/333s | 0.83750/170s | 0.83777/119s | 0.83822/91s |
| **0.02** | 0.83720/381s | **0.83680/199s** | 0.83710/137s | 0.83749/107s |

Taula B.9: RMSE/temps d'entrenament, algoritme BRISMF, $F = 20$, dataset Flixter

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** | **0.004** |
|---|---|---|---|---|
| **0.005** | 0.84084/286s | 0.84106/151s | 0.84135/101s | 0.84165/78s |
| **0.01** | 0.83751/349s | 0.83761/184s | 0.83796/125s | 0.83837/98s |
| **0.015** | 0.83586/413s | 0.83588/220s | 0.83622/154s | 0.83665/122s |
| **0.02** | 0.83528/482s | **0.83503/261s** | 0.83532/184s | 0.83578/146s |

Taula B.10: RMSE/temps d'entrenament, algoritme BRISMF, $F = 30$, dataset Flixter

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** | **0.004** |
|---|---|---|---|---|
| **0.005** | 0.83974/360s | 0.83997/186s | 0.84031/123s | 0.84060/93s |
| **0.01** | 0.83650/433s | 0.83665/230s | 0.83703/154s | 0.83750/119s |
| **0.015** | 0.83492/518s | 0.83500/270s | 0.83538/188s | 0.83585/145s |
| **0.02** | 0.83429/611s | **0.83421/322s** | 0.83455/223s | 0.83499/171s |

Taula B.11: RMSE/temps d'entrenament, algoritme BRISMF, $F = 40$, dataset Flixter

#### B.1.2.4 Conjunt de dades Yahoo!

En aquest cas, a diferència dels anteriors, només s'ha provat amb 20 factors, per motius de límits de memòria.

| $\lambda$ \ $\gamma$ | **0.001** | **0.002** | **0.003** |
|---|---|---|---|
| **0.005** | 0.89340/1560s | 0.89425/1080s | 0.89427/765s |
| **0.01** | **0.89169/2753s** | 0.89219/1530s | 0.89370/1158s |
| **0.015** | 0.89212/3566s | 0.89292/1970s | 0.89853/1422s |

Taula B.12: RMSE/temps d'entrenament, algoritme BRISMF, $F = 20$, dataset Yahoo!

## B.2 Prova de recomanació

### B.2.1 Movielens

Figura B.1: Precisió i recall avaluant tots els ítems, Movielens

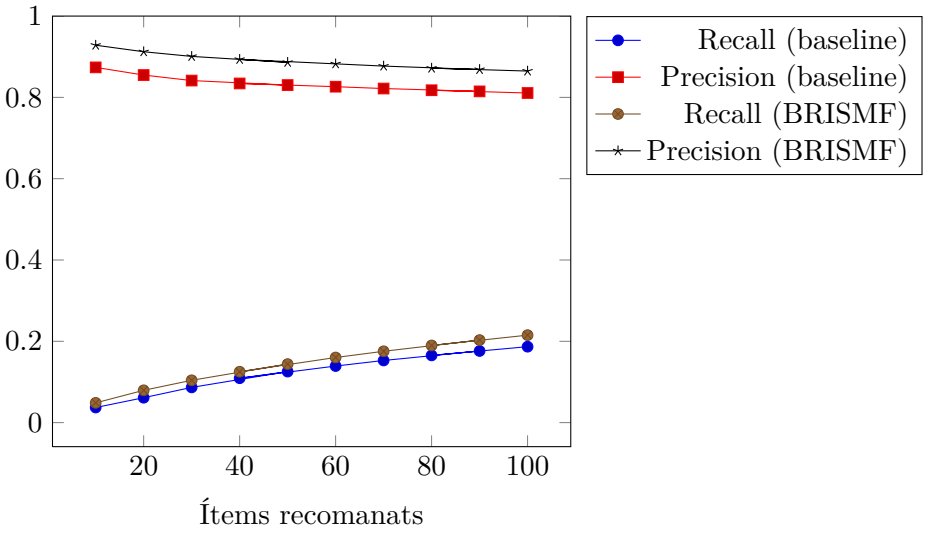
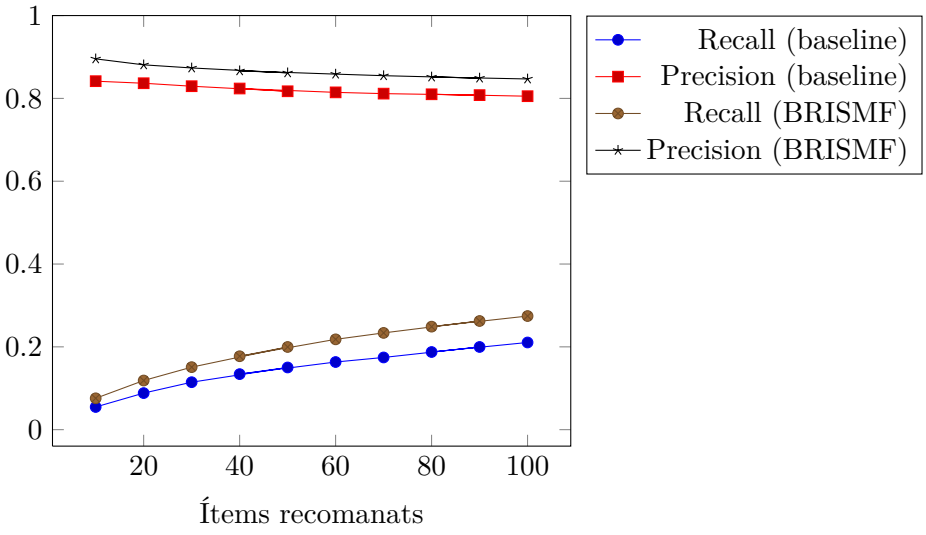



Figura B.2: Precisió i recall avaluant els $n_c = 10n_r$ ítems més puntuats, Movielens



### B.2.2 Jester

Figura B.3: Precisió i recall avaluant tots els ítems, Jester

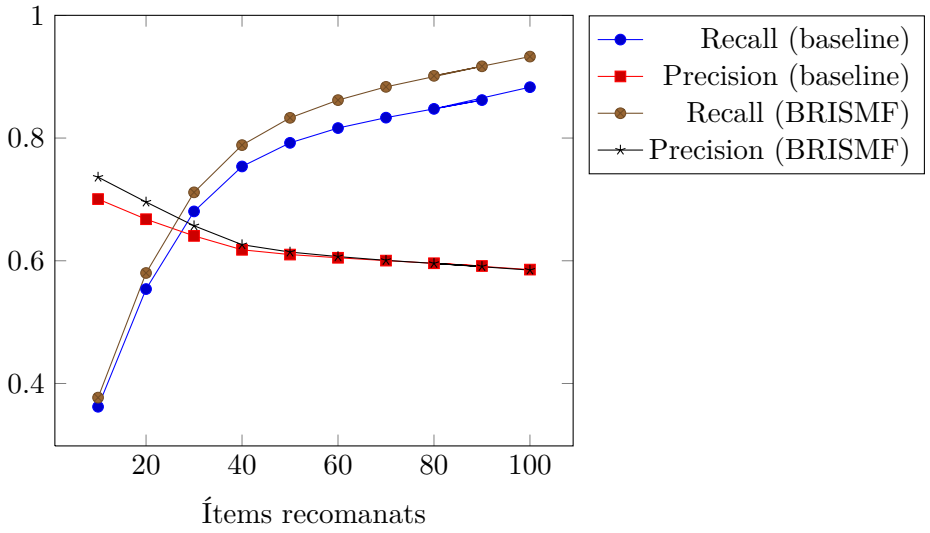
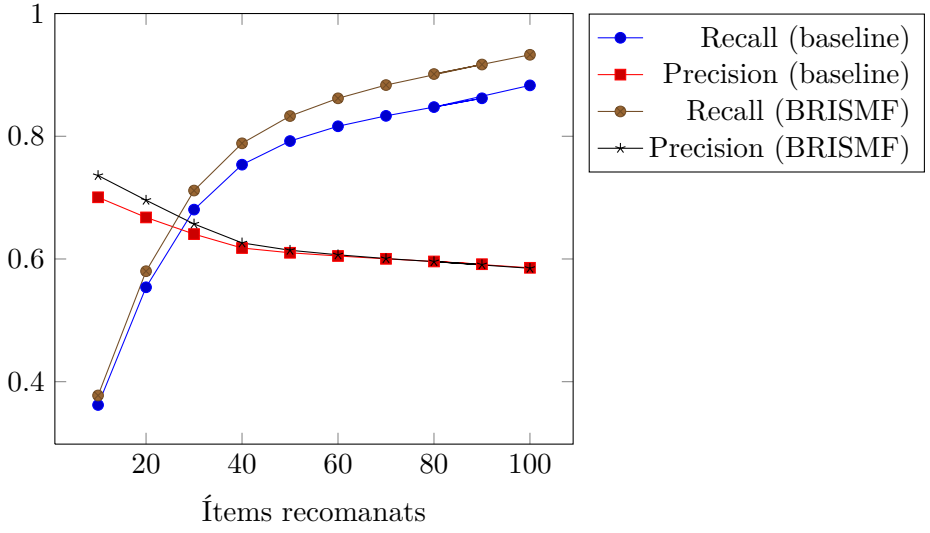



Figura B.4: Precisió i recall avaluant els $n_c = 10n_r$ ítems més puntuats, Jester



### B.2.3 Flixster

Figura B.5: Precisió i recall avaluant tots els ítems, Flixster

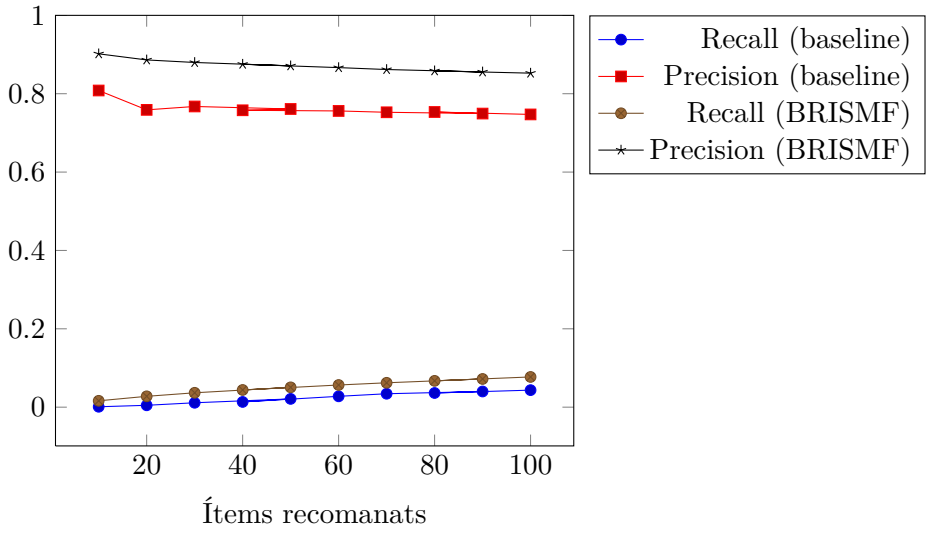
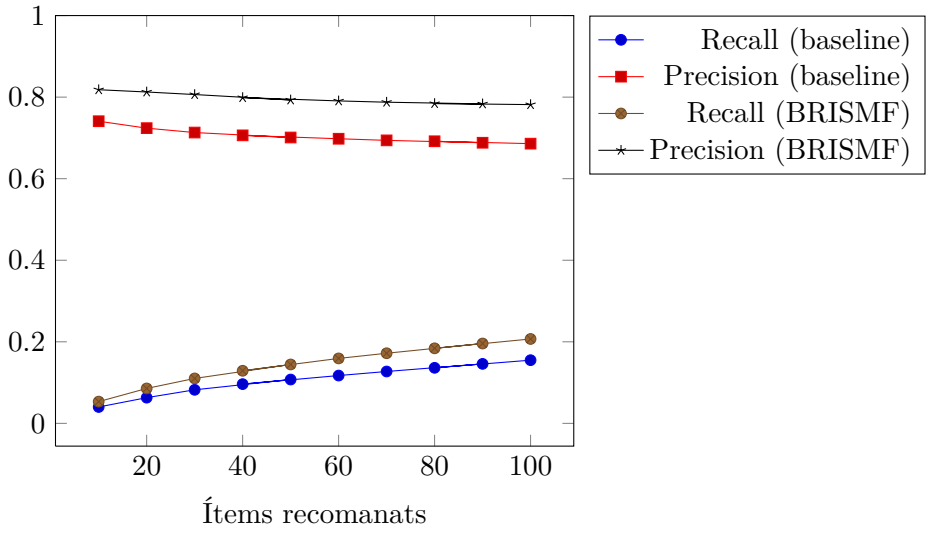



Figura B.6: Precisió i recall avaluant els $n_c = 10n_r$ ítems més puntuats, Flixster



### B.2.4 Yahoo!

Figura B.7: Precisió i recall avaluant tots els ítems, Yahoo!

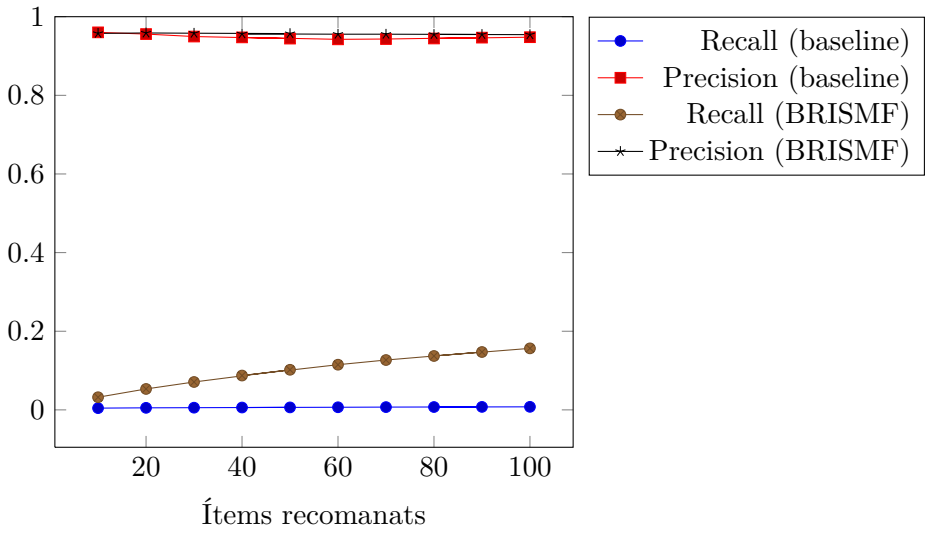
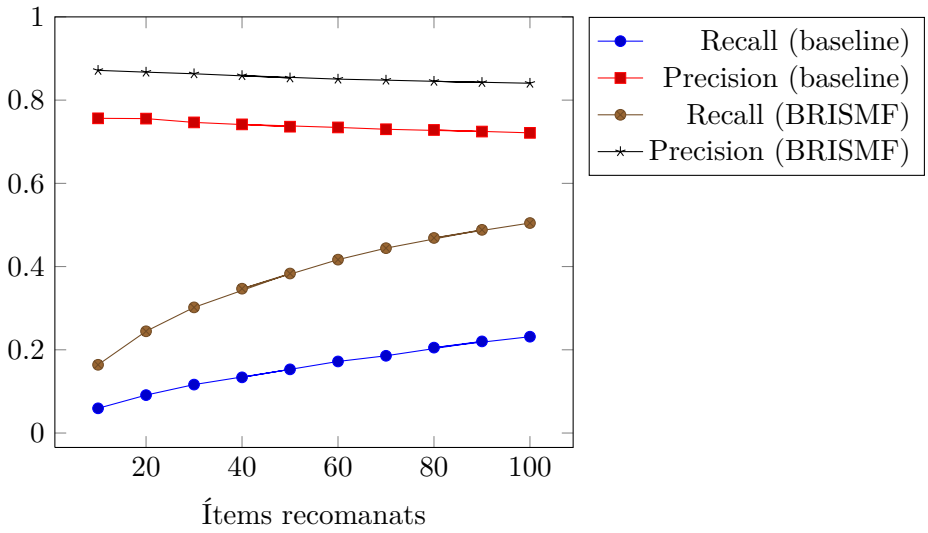



Figura B.8: Precisió i recall avaluant els $n_c = 10n_r$ ítems més puntuats, Yahoo!



# Índex de figures

# Índex de taules

# Índex d'algoritmes