# Algorismes i estructures de dades

## K-means

Aquest algorisme ens serveix per separar els usuaris del sistema en k clústers segons el grau de "proximitat" entre les seves valoracions. En primer lloc, s'assigna a cada usuari un punt en un sistema de coordenades on cada valoració d'un ítem es correspon amb una dimensió. Després, a cada un dels k clústers se li assigna un centroide en el mateix sistema de coordenades (primerament s'agafen les coordenades de k usuaris a l'atzar). A continuació, per a cada usuari es mesura la seva distància als centroides i s'assigna aquell usuari al clúster més proper. Una vegada s'ha fet la primera iteració, es recalculen les coordenades dels k centroides fent la mitjana (d'aquí, k-means) de les coordenades de tots els usuaris que conté el seu clúster. Aquest procés es repeteix fins que els clústers son estables, llavors es considera que cada usuari està al clúster que li correspon i l'algorisme acaba retornant els clústers amb els seus usuaris i els corresponents centroides.

### Cost de l' algorisme

Quan comença l'algorisme, primerament hi ha un for per fer els new dels k clústers de l'algorisme. Com que per la precondició de la funció és que k <= n (on n és el nombre total d'usuaris del sistema), primerament observem que el cost d'aquesta part és de O(n).

A continuació, ens trobem que per als n usuaris calculem les seves coordenades sparse (ja que només es consideren les puntuacions dels ítems que l'usuari ha valorat), de manera que tenim dos fors (un que recorre els n usuaris dels fitxers del sistema) i un altre que calcula, per a cadascun, les seves com a molt m coordenades (on m és el nombre d'ítems que hi ha al sistema). Per tant, el cost d'aquesta part és de O(n*m).

Després, s'assignen les coordenades als k centroides (recordem que k <= n) recorrent el conjunt de n usuaris i escollint-ne un a l'atzar. Novament, tenim dos fors, un que assign les coordenades a cada centroide i un altre que recorre el conjunt d'usuaris: és a dir, el cost d'aquesta part és de $O(n^2)$.

Tot seguit hi ha un while que com a molt fa 100 iteracions per tal d'anar reorganitzant els n usuaris segons les seves coordenades al clúster que li pertoca. Per això, dins cada iteració del while es recorren en un for tots els usuaris i per a cadascun es mira amb un altre for quin és el clúster que té el centroide més proper a ell i si és necessari es fa el canvi de clúster. Aquesta part té cost $O(n^2)$.

Encara dins del while, després de reassignar els usuaris als clústers que pertoqui, es recalculen les posicions dels k (k <= n ) centroides, sumant per a cadascun dels com a molt n usuaris del clúster corresponent les seves coordenades (amb com a molt m dimensions), per després dividir-ho tot entre el nombre d'usuaris que hi ha al

clúster. És per això que aquesta part pot ser una de les més costoses, amb un cost de $O(n*n*m) = O(m*n^2)$.

Finalment, ja fora del while, es prepara el conjunt de retorn amb dos fors de com a molt n iteracions cadascun, de manera que aquesta part tindria un cost de $O(n^2)$.

En conclusió, el cost final de l'algorisme K-means és de $O(m*n^2)$.

## Principals estructures de dades

- **HashMap<Integer, HashMap<String, double>> CoordenadesUsuaris**
    - *És un map que conté, per a tots els usuaris del sistema, les seves coordenades. La clau és l'identificador d'un usuari, i al hashmap s'emmagatzemen les seves coordenades, és a dir, valoracions.*
- **HashMap<Integer, HashMap<String,Double>>[] clusters**
    - *És un array de k maps on cada un dels maps es correspon amb un clúster. La clau de cada entrada dels maps és l'identificador d'un usuari, i al hashmap s'emmagatzemen les seves coordenades, és a dir, valoracions.*
- **HashMap<String, double>[] centroides**
    - *És un array de k Hashmaps on cada un dels hashmaps representa les coordenades d'un centroide, de manera que centroides[i] conté les coordenades del centroide que correspon a clusters[i].*
- **Vector<UsuariFitxers>[] clustersUsuaris**
    - *S'usa per fer el retorn de l'algorisme. Consisteix en un array de k vectors d'usuaris, un per a cada clúster. El contingut de clustersUsuaris[i] són els usuaris identificats per les mateixes claus que hi ha a les entrades de clusters[i].*

## Slope one

Aquest algorisme s'usa per estimar o predir la puntuació que un usuari en concret donaria a un ítem donat, a partir de les valoracions que altres usuaris han fet d'aquell ítem. Es complementa amb l'algorisme K-means esmentat anteriorment i la funció de Collaborative Filtering calculaParticio de manera que les valoracions de l'ítem que s'agafen com a referència siguin les dels usuaris que estan en el mateix clúster (per tant, amb gustos similars) formant així l'algorisme Collaborative Filtering que explicarem més endavant.
L'algorisme Slope One segueix l'esquema seria el següent:

```
Donat un item que l'usuari no ha valorat:
    per cada item(item2) si valorat per l'usuari {
        per cada usuari de la partició de l'usuari que hagi valorat el item1 i el item 2{
            es resta la puntuació del item1 - item2
            s'afegeix el resultat a un sumatori
            s'incrementa el nombre d'usuaris que han valorat els dos items
        }
        es divideix el resultat del sumatori entre el nombre d'usuaris del sumatori =
        desviació mitjana
        se li suma a la desviació mitjana la valoració de l'usuari del item2  =
        resultat parcial
        s'afegeix el resultat parcial a un sumatori de resultats parcials
        s'incrementa el nombre de resultats parcials
    }
    el resultat final ( i doncs la valoració estimada per aquell item ) és el sumatori
    dels resultats parcials entre el nombre d'aquests.
    Es retorna aquesta divisió.
}
```

### Cost de l' algorisme

A l'algorisme de Slope one hi trobem dos fors, un dins de l'altre.
El primer for recorre el Hashmap de valoracions fetes per l'usuari passat per paràmetre, així doncs el cost d'aquest recorregut ve donat pel tamany del Hashmap de valoracions de l'usuari, al que anomenarem v, per tant, el cost és O(v).
Dins del segon for trobem el tercer for, que recòrre els usuaris del vector anomenat uparticio. El cost de recorrer un vector ve donat per la seva mida (que és el nombre d'usuaris que pertanyen a la partició) al que anomenarem u, doncs el cost és O(u). Dins d'aquest tercer for fem 4 cerques a Hashmap amb cost O(1) i altres operacions matemàtiques amb cost constant.

Per tant, el cost de SlopeOne és O(v*u). Sent v el nombre d'ítems valorats per l'usuari donat i u el nombre d'usuaris pertanyents a la mateixa partició a la qual pertany l'usuari donat.

Podem observar que, com major sigui la k de k-means, menor serà u, ja que el nombre d'usuaris a les particions, a priori, es repartirà i doncs disminuirà. Doncs a major k, menor el cost de SlopeOne, però a la vegada si augmentem massa k, menor serà el DCG. Per tant, s'ha de trobar un punt mig on la k que ens dongui un bon DCG i un bon temps d'execució.

### Principals estructures de dades:

- **Vector<UsuariFitxers> uparticio**
    - *És un vector d'Usuaris que representa els usuaris que pertanyen a la mateixa partició que l'usuari entrat per paràmetre a Slope One.*

## Collaborative Filtering

Aquest algorisme s'usa per fer una recomanació a un usuari donat, estimant o predint la puntuació que l'usuari en concret donaria a un conjunt d'ítems passat per paràmetre. Aquest conjunt d'ítems conté ítems pels quals l'usuari no ha fet una valoració.
Collaborative Filtering primerament cerca en quin clúster es trobaria l'usuari, cridant a la funció calculaParticio, que donats els centroides extrets de kmeans (que s'han calculat en algun moment previ a demanar la recomanació) i l'usuari per al qual fer la recomanació, calcula la distància de l'usuari a cada un dels centroides i retorna el número de centroide amb distància mínima.
Un cop troba el clúster al qual pertany l'usuari, recorre els ítems passats per paràmetre i, per cada ítem, crida Slope One, que retorna la puntuació predita per a aquell usuari i aquell ítem concret.
Amb aquesta puntuació, l'algorisme crea una Valoració amb predictiva cert, la puntuació predita, l'ítem que estem tractant en la iteració concreta i l'usuari per al qual es fa la recomanació.
Aquesta Valoració s'insereix a un TreeSet que serà el que es retorna finalment. El TreeSet ordenarà el resultat descendentment per puntuació de valoració.

### Cost de l' algorisme

Per tal de saber el cost de l'algorisme cal saber el cost de les funcions que crida. Com ja hem explicat anteriorment en aquest document, el cost de l'algorisme SlopeOne és O(v*u) sent v el nombre de valoracions fetes per l'usuari donat i u el nombre d'usuaris que pertanyen a la mateixa partició que l'usuari per al qual estem fent la recomanació.

El cost de la funció calculaParticio és O(k * O(v)), on k és el nombre de centroides i O(v) correspon al cost del càlcul de Distància per qualsevol estratègia de distància, on v és el nombre d'ítems valorats per l'usuari a recomanar. Per tant, O(k*v)

I doncs el cost de Collaborative filtering es, el cost de calcular la partició en la qual pertany l'usuari més el cost de cridar tantes vegades Slope One com ítems hi ha a l'entrada. Per tant, el cost és O(k*v)+O(m*v*u)
Sent m el nombre d'ítems de l'entrada.

Com que k com a molt és el nombre d'usuaris u, llavors el cost quedaria com O(u*v)+O(m*v*u) = O(m*v*u)

**Principals estructures de dades**

- **Vector <UsuariFitxers>[] clusters**
  - S'usa per trobar la pertinença de l'usuari donat als diferents clústers. Consisteix en un array de k vectors on cada un dels vectors es correspon amb un clúster, i conté les diferents instàncies d'usuaris que pertanyen a la partició. La clau de les entrades és l'identificador d'un usuari.

- **HashMap<String, double>[] centroides**
  - És un array de k Hashmaps on cada un dels hashmaps representa les coordenades d'un centroide, de manera que centroides[i] conté les coordenades del centroide que correspon a clusters[i].

- **Vector<UsuariFitxers> uparticio**
  - És un vector d'Usuaris que representa els usuaris que pertanyen a la mateixa partició que l'usuari entrat per paràmetre a Slope One.

- **TreeSet<Valoració> result**
  - Aquesta estructura és la que s'usa per el return de Slope One. Representa un conjunt de Valoracions predites per a l'usuari entrat per paràmetre. Tot i que bastaria retornar els id dels ítems i després ordenar-los per la puntuació predita, es retorna un TreeSet de Valoracions. Una valoració té, un usuari actiu, un ítem i una puntuació, d'aquesta manera retornant un objecte (dins d'un TreeSet) podem obtenir tota la informació necessària per si més endavant es vol treballar amb la valoració. El TreeSet es fa servir per poder tenir les Valoracions ordenades, per puntuació, de major a menor. De manera que no cal tocar més tard l'ordre en què es retornen els ítems.

## Content-based filtering i K-nearest neighbours

Content based s'usa per predir valoracions d'un usuari sobre uns ítems concrets. Aquests ítems concrets s'extreuen de K-nearest, algorisme que es crida dins de Content-Based filtering.

L'algorisme de k-nearest ens serveix per cercar, per a cada ítem que li agrada a l'usuari d'entrada, els k ítems més semblants a aquests.
Un cop content based té els k ítems més semblants per cada ítem que li agrada a l'usuari, calcula una Valoració predictiva per aquest ítem semblant, tenint en compte la similitud i la nota de l'ítem agradat per l'usuari, essent la Valoració final la nota de l'ítem agradat afegint-li una desviació tipus calculada a partir de la similitud, de manera que a similitud màxima, la nota seria la mateixa que l'agradat, i a similitud mínima (no s'assemblen absolutament res) seria una Valoració aleatòria. Finalment, s'ordenen per puntuació aquestes valoracions i es retornen.

### Cost de l'algorisme

En cas pitjor KNearests recorre tots els ítems del sistema O(i) i per cada ítem que no hagi valorat l'usuari s'afegeix en una priorityQueue amb cost O(log(n)) on n és el nombre actual de valors a la cua de prioritat. Per tant, el cost de recórrer tots els ítems i afegir-los en una priorityQueue és el sumatori de log(1)+log(2)+...+log(i-1)+log(i) on el cost asimptòtic d'aquesta expressió és O(i*log(i)). Després els k primers elements de la priorityQueue els movem a una priorityQueue amb cost O(k*log(k)) (seguint el mateix raonament d'abans) però com que i és més gran que k, podem ignorar aquest cost. Per tant, el cost de KNearests és O(i*log(i)).

L'algorisme de Content Based filtering comença a la funció Filtering. L'operació Filtering rep per paràmetres una instància d'Usuari i un HashMap amb tots els ítems del sistema. Aquesta operació recorre un map de m elements que són les valoracions de l'usuari (en el cas pijtor l'usuari haurà valorat tots els ítems, per tant, tindrà una mida de i) rebut per paràmetre, per tant, el cost de recorrer aquest Map és O(i). Per a cada valoració que ha fet l'usuari s'elimina l'item valorat del HashMap amb tots els items, això té cost O(2) (ja que és un HashMap), també per a cada item que li agrada a l'usuari s'afegeix la valoració a un vector, aquesta operació és O(1) però en el cas pitjor es faria 'i' vegades i, per tant, té cost O(i). Per tant, el cost de recórrer el map de valoracions de l'usuari té cost O(i).

A continuació, recorrem el vector de les 'i' valoracions que li han agradat a l'usuari, només el cost de recórrer el vector és cost O(i). Per cada element de valoracions fem una crida a KNearests que té cost O(i*log(i)) → O(i*i*log(i)), com ja hem vist abans. Mentre la cua de prioritat de k elements retornada per KNearests no sigui buida afegim o modifiquem els ítems de la priorityQueue i els hi assignem una nota

en un HashMap amb cost O(k) i com que això es farà per cada ítem té cost O(k*i*i*log(i)) i, aleshores, el cost d'aquesta part és de O(k*i2*log(i)).

Finalment, un cop ja tenim tots els ítems amb una nota calculada en un HashMap de e elements el recorrerem per afegir-los en una cua de prioritat. El cost d'aquesta part final té cost O(e*log(e)).

En conclusió, el cost de l'algorisme Content Based serà O(i)+O(k*i2*log(i))+O(e*log(e)) però com que i és molt més gran que m i molt més gran que e, aleshores l'algorisme de Content Based és de O(k*i2*log(i)).

**Principals estructures de dades**

- **HashMap<Integer,Valoració>**
    - Aquesta estructura de dades s'utilitza perquè és l'atribut que conté l'usuari actiu per tal d'accedir a les valoracions d'aquest. El que es fa a l'algorisme és simplement recòrrer-ho per tal de trobar les millors valoracions.
- **Vector<Valoració>**
    - Aquest vector és simplement per emmagatzemar les millors valoracions de l'usuari actiu per a després tractar-les.
- **HashMap <Item, Pair<Integer, Double> >**
    - Aquest HashMap serveix per emmagatzemar, per tots els ítems millor valorats per l'usuari actiu, els k més semblants amb la seva similitud (el double) i, com que és possible que un altre ítem sigui semblant a aquest, l'integer és el nombre de repetits. Es fa servir aquest HashMap per maximitzar l'eficiència de les cerques per detectar repetits.
- **PriorityQueue< Pair<Double,Item> >**
    - Utilitzem la priority queue per, primerament ficar tots els ítems amb els quals es compara l'ítem comparat i, finalment, extreure els k més semblants, amb l'avantatge d'aquesta estructura de que el pas d'extreure els k més semblants és de cost O(k Log(i)) on "i" és el nombre d'ítems, i perquè les insercions també són prou barates en temps (O(log(n)).

- **TreeSet<Valoració>**
    - Finalment, fiquem les valoracions a un TreeSet per tal que quedin ordenades per puntuació i les retornem al mètode de Filtering.

## Valorar recomanació (DCG)

El DCG (Discounted Cumulative Gain) s'usa per mesurar la qualitat d'una recomanació oferida per un algorisme a un usuari, consistint aquesta en una llista dels ítems "recomanats". Aquesta llista d'ítems se suposa ordenada de forma descendent, segons la puntuació amb què l'algorisme prediu que l'usuari valoraria aquell ítem. Per aquest algorisme és necessari conèixer la valoració "real" que hauria de predir l'algorisme per als ítems de la llista. Per calcular el dcg, hem usat la fórmula següent:

$$DCGp = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

On $rel_i$ és la puntuació real de l'ítem que està en la posició ***i*** a la llista de recomanacions. El nombre d'ítems de la llista de recomanats és ***p*** i, ***i*** correspon a la posició de dit element a la llista començant per 1. Aquest sumatori serà major com més encertada sigui l'ordenació dels ítems a la llista de recomanació (de manera que es penalitza si un element amb puntuació "real" baixa és recomanat abans a la llista que un altre ítem amb puntuació "real" major) De cara a la tercera entrega tenim pensat normalitzar aquesta dada de manera que el seu valor oscil·li entre 0 i 1. Això ho farem dividint el DCG obtingut per un DCG "ideal" calculat a partir de la llista dels p ítems millor valorats segons les valoracions "reals" d'un usuari, ordenats de forma decreixent.

### Cost de l' algorisme

Donat que en el càlcul d'aquest valor només s'han de recórrer els p elements de la llista un cop (només hi ha un bucle), i que p <= m (on m és el nombre total d'ítems del sistema); podem afirmar que el cost d'aquest càlcul és de O(m). Per al cas del DCG normalitzat previst per a la tercera entrega, el cost ascendiria a O(m * log(m)) degut al cost de realitzar l'ordenació decreixent.

### Principals estructures de dades

- **Vector<String> itemsRecomanats**
    - *És un vector que conté els noms dels ítems recomanats per un algorisme a un usuari, ordenats descendentment segons la puntuació que prediu que aquest usuari els donaria.*
- **Map<String, Valoració> LT**
    - *És un map on la clau és el nom d'un ítem i que conté les valoracions "reals" amb què l'usuari els puntuaria.*
- **ArrayList<Double> idealPermutation**
    - *És una llista ordenada de forma decreixent de totes les puntuacions "reals" de l'usuari que s'usa per calcular el DCGIdeal necessari per normalitzar el valor del DCG.*

## Hybrid Approaches

L'algorisme que hem dissenyat per al hybrid approaches fa servir una combinació dels algorismes de SlopeOne i K-Nearests. L'algorisme funciona de la següent manera; Inicialment per a predir la nota dels ítems els quals no ha valorat l'usuari executem l'algorisme de Collaborative Filtering amb una modificació, quan aquest executa tants cops com sigui necessari l'algorisme Slope One, si per un ítem a predir la nota, no hi ha cap parella per la qual es troba almenys un usuari dins del clúster que hagi valorat la parella, en comptes d'assignar una nota aleatòria per a aquell ítem (com es fa a l'algorisme Slope one de Collaborative Filtering original), executem la funció hybrid. Hybrid crida a l'algorisme KNearests per obtenir els k items més semblants a l'ítem pel qual volem predir la nota que li posarà aquell usuari. Un cop té els k ítems més semblants, per a cada un d'ells fa SlopeOne (l'original), per predir per cada ítem retornat per Knearest la nota que li donaria l'usuari pel qual estem fent la recomanació, i fa la mitja de les notes que els k ítems han obtingut per tal d'assignar-li a l'ítem pel qual no havíem pogut trobar valoració amb Slope One sol. El valor d'aquesta mitjana, i doncs la nota predita, es retorna, a través del mètode Hybrid, al primer SlopeOne executat i aquest la retorna a Collaborative Filtering. Collaborative Filtering segueix la seva execució normal.

### Cost de l'algorisme

Per aquest algorisme, el cas pitjor es donarà quan per tots els ítems de l'entrada (ítems no valorats per l'usuari donat), Slope one no sigui capaç de treure'n una nota

predictiva i doncs es cridi a hybrid tants cops com nombre d'ítems té l'entrada. Per tant, el cost de l'algorisme, en aquest cas, és el cost de Collaborative Filtering (que hem vist anteriorment que és O(n*v*u) sent n el nombre d'ítems de l'entrada, v el nombre d'ítems si valorats per l'usuari donat i u el nombre d'usuaris que pertanyen a la mateixa partició que la de l'usuari donat) més el cost de la funció Hybrid, ja que el slope one està modificat respecte al que s'usa originalment i cridarà, per cada execució seva, a hybrid en el cas pitjor.

Cost de la funció Hybrid:
Com es pot veure fàcilment, el cost d'aquesta funció vindrà donat pel cost de fer K-nearest i pel cost de fer Slope One (l'original, sense funció hybrid dins), en cas pitjor, k vegades, una per cada ítem retornat per Knearest (ja que en cas pitjor cap dels ítems retornats per knearest haurà estat valorat per l'usuari).
Per tant, com hem vist anteriorment a l'explicació de l'algorisme content based filtering i com hem vist a l'explicació del cost de Slope One, el cost de la funció hybrid serà k*O(*v*u) + O(i*log(i)))

Per tant, el cost total de Hybrid Approaches en cas pitjor és O(n*v*u*el cost de la funció hybrid) = O(n*v*u*O(k*O(*v*u) + O(i*log(i))))).

Les estructures principals d'aquest algorisme son les mateixes que a SlopeOne i ContentBased.

## Distancia Ponderada

Per al càlcul de les distàncies s'ha escollit usar un algorisme al qual hem anomenat Distancia Ponderada.
Aquest algorisme rep per paràmetre dos HashMaps, amb clau String i valor Double. Dins de cada HashMap es representen valoracions d'un usuari concret, on el String és l'identificador de l'ítem i el Double és la puntuació donada per aquell ítem. A aquestes valoracions se'ls anomena també coordenades, ja que a kmeans s'interpreten com a coordenades en un espai de dimensions. Aquests Hashmap seran o bé valoracions reals d'usuaris de ratings o bé valoracions de centroides, calculades a partir de valoracions d'usuaris de ratings (quan es recalculen els centroides es tenen en compte totes les valoracions de tots els usuaris del cluster i doncs pot ser que no coincideixi el centroide amb cap usuari). La pre d'aquest algorisme és que, si algun dels dos HashMap representa un centroide, aquest sigui el segon. Això és necessari que es compleixi, ja que hem de recorrer un dels HashMaps i interessa recórrer els de l'usuari real, perquè el HashMap del centroide pot arribar a tenir un nombre gran d'entrades (tantes com número d'items té el sistema) a causa del procés de recalculament de centroides de l'algorisme Kmeans. Així doncs disminuïm el cost de l'algorisme.

L'algorisme recorre el HashMap coordenades1, que representa les coordenades i doncs valoracions d'un usuari. Per cada element del Hashmap es comprova si la clau d'aquest està continguda en coordenades2. Si és així es calcula una part de la distància euclidiana, fent el quadrat de la resta entre els dos valors i s'acumula el resultat en una variable local inicialitzada amb valor 0 fora del bucle. A més a més s'incrementa una variable local que representa el nombre d'elements en comú.

Un cop s'ha acabat de recórrer el Hashmap coordenades1, es calcula la distància euclidiana total, fent l'arrel quadrada de l'acumulació dels quadrats de les restes.
Per tal de ponderar aquesta distància i penalitzar segons el nombre de valoracions que no tenen en comú, es calcula aquest nombre d'ítems valorats no en comú i es multiplica la distància euclidiana pel logaritme d'aquest.

## Cost de l'algorisme

Per l'explicació anterior es pot veure que el cost de l'algorisme es el cost de recórrer les valoracions de l'usuari. Doncs el cost és O(n) sent n el nombre de valoracions de l'usuari i doncs n és coordenades1.size().

## Perquè hem escollit aquesta representació de les dades?

A l'anterior entrega, calculàvem les coordenades d'un usuari de manera que sempre es tenien tantes coordenades com ítems en el sistema. Es tenien en compte, per calcular les coordenades, les valoracions que havia fet l'usuari, però també, si no havia valorat un ítem i doncs no tenia coordenada per una dimensió, se li atorgava un nombre aleatori.
Després de fer testing (i per mala sort després de fer la primera entrega) ens vam adonar que aquest sistema d'usar tantes coordenades com ítems hi ha i atorgar tants valors aleatoris feia que els valors es contrarestassin entre ells i doncs tots els usuaris tinguessin distàncies molt semblants i acabessin doncs tots en un mateix clúster.
Amb aquesta entrada ens assegurem que això no passa, ja que en usar per al càlcul de totes les distàncies un sistema de coordenades on es té en compte només les valoracions que sí ha fet l'usuari, no es creen valors aleatoris que contraresten els valors reals i doncs les distàncies surten prou diferents perquè quedin tots els clústers amb un nombre decent d'usuaris.

De moment no hem pogut ficar-nos d'acord en quina distància és la que dona millors resultats, ja que estem modificant el sistema per poder comprovar-ho de manera massiva, i doncs encara no hem pogut extreure conclusions.

## Altres estructures de dades rellevants del sistema:

També hi ha altres estructures de dades rellevants que s'usen en diferents classes del sistema (la majoria, dins del controlador de la capa de Domini), i són les que s'esmenten a continuació:

- **Map<String, Item> Items**
    - *És un map on la clau és l'identificador d'un ítem i el valor és l'Item en qüestió.*
- **Vector<Columna> Columnes**
    - *És un vector on es guarden les Columnes corresponents als Atributs que defineixen un Item. És a dir, per a cada Atribut diferent que pot tenir un Item, hi ha un objecte Columna que l'emmagatzema.*
- **HashMap<Integer, UsuariFitxers> Usuaris**
    - *És un map on la clau és l'identificador d'un usuari i que conté tots els usuaris que apareixen al fitxer de ratings.*
- **HashMap<Integer, UsuariFitxers> UsuarisKnown**
    - *És un map on la clau és l'identificador d'un usuari i que conté tots els usuaris que apareixen al fitxer de Known.*
- **HashMap<Integer, UsuariFitxers> UsuarisUnknown**
    - *És un map on la clau és l'identificador d'un usuari i que conté tots els usuaris que apareixen al fitxer d'Unknown. Tot i que l'usuari en si apareix al map de UsuarisKnown, en aquest map la instància té unes valoracions atorgades diferents, les d'unknown. Aquesta manera de guardar la informació de unknown ens permet un accés fàcil a l'hora d'accedir per usuari a les dades que es troben en el fitxer. Serveix per diferenciar clarament quina informació es "coneguda" d'un usuari i quina és "desconeguda".*
- ***Pair<T,T>***
    - *Durant la programació de les funcionalitats vam trobar que l'ús de pairs facilitaria molt la feina a fer i faria més fàcil els retorns d'algunes funcions. És per això que es va crear una classe Pair genèrica.*

Com ja hem esmentat a l'algorisme de DistanciaPonderada, estem modificant la funcionalitat que ens permetia calcular el DCG del recomanador per tal que ens serveixi també veure quins algorismes donen millor resultat, per tant, encara no podem saber al 100% quins són. A l'última entrega trobaràs un document anomenat Decisions, on hi haurà escrit quins hem decidit que siguin els algorismes predeterminats i per què.