Sistema de recomanació

Segon lliurament (2.0)

Projectes de programació (PROP)

Grau en Enginyeria Informàtica - FIB - UPC

Curs 2021-2022, Quadrimestre de Tardor

Equip 4.2

Ferran De La Varga Antoja (ferran.de.la.varga)

Alexandru Dumitru Maroz (alexandru.dumitru)

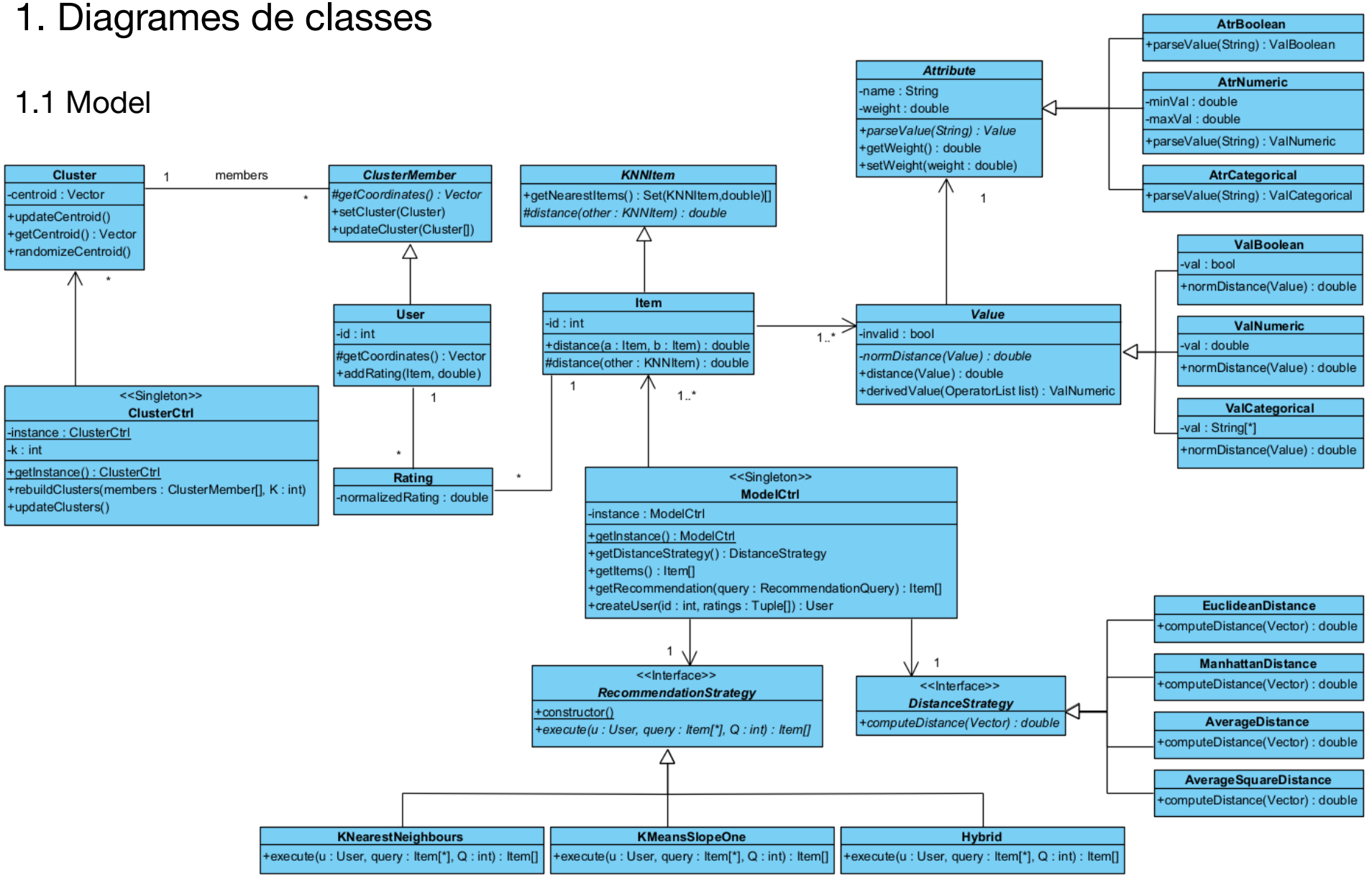
Pablo José Galván Calderón (pablo.jose.galvan)

Pol Rivero Sallent (pol.rivero)

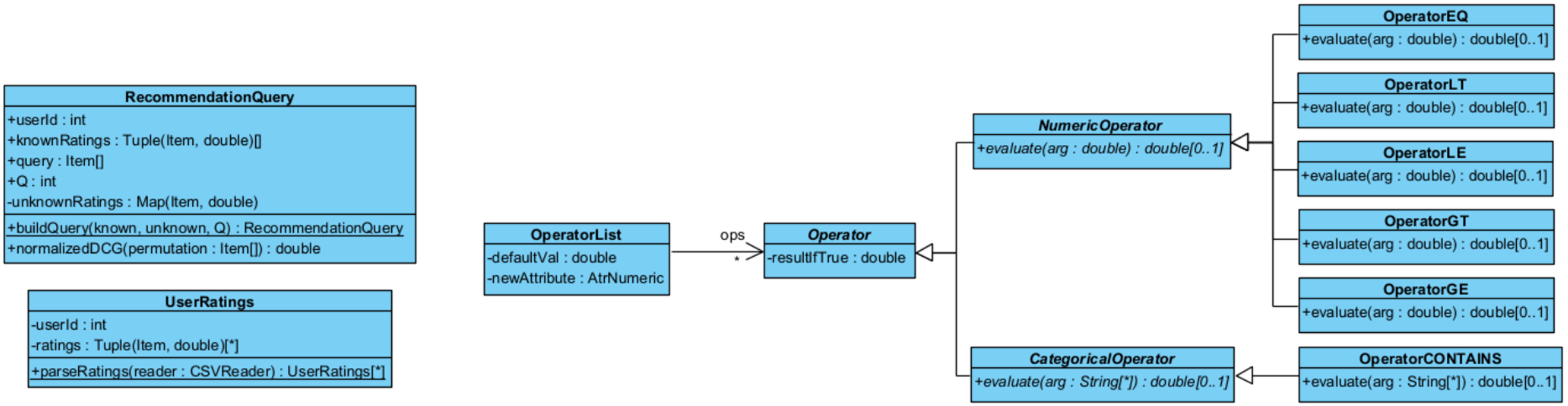
Índex

1. Diagrames de classes

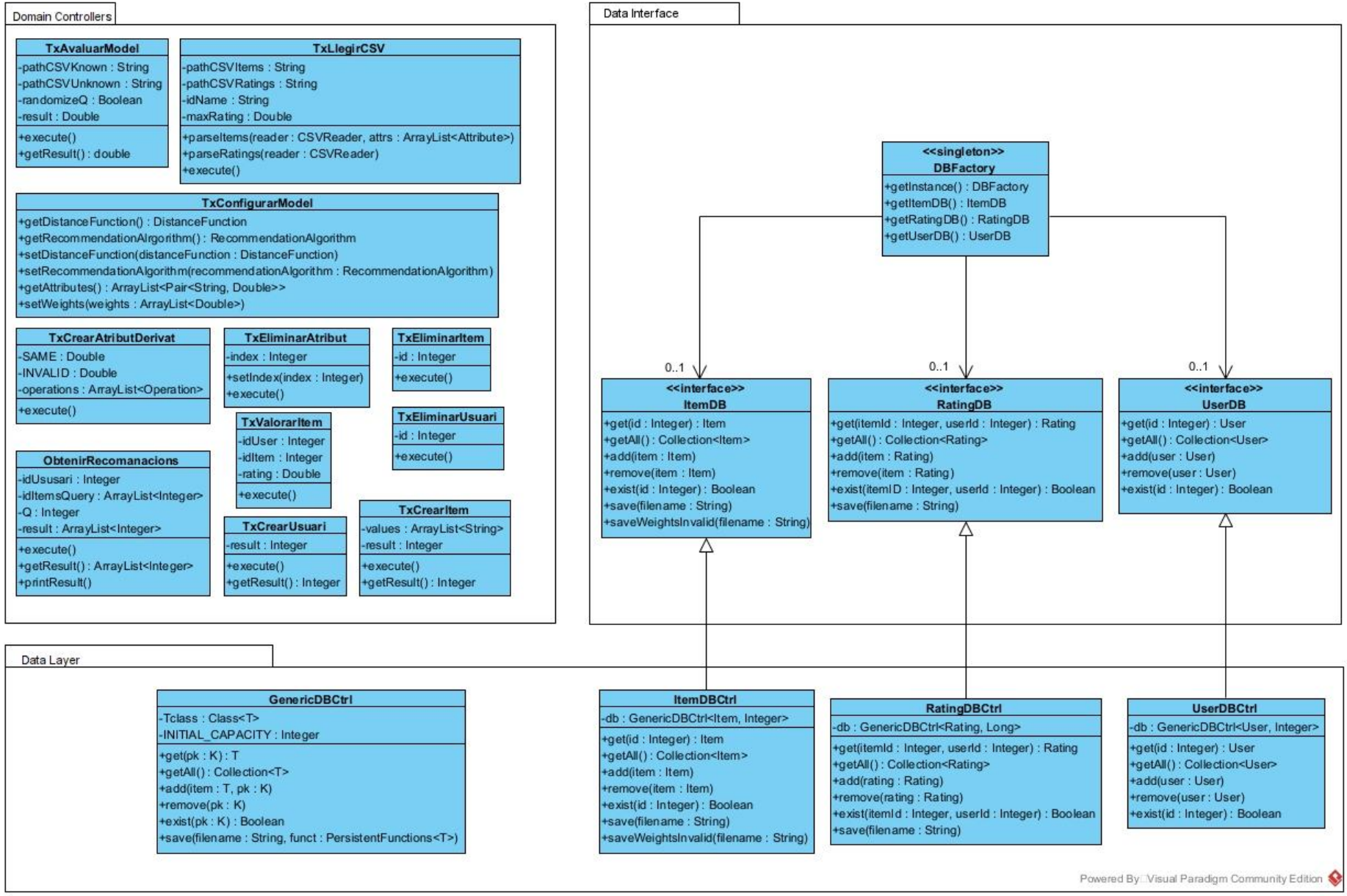
1.1 Model



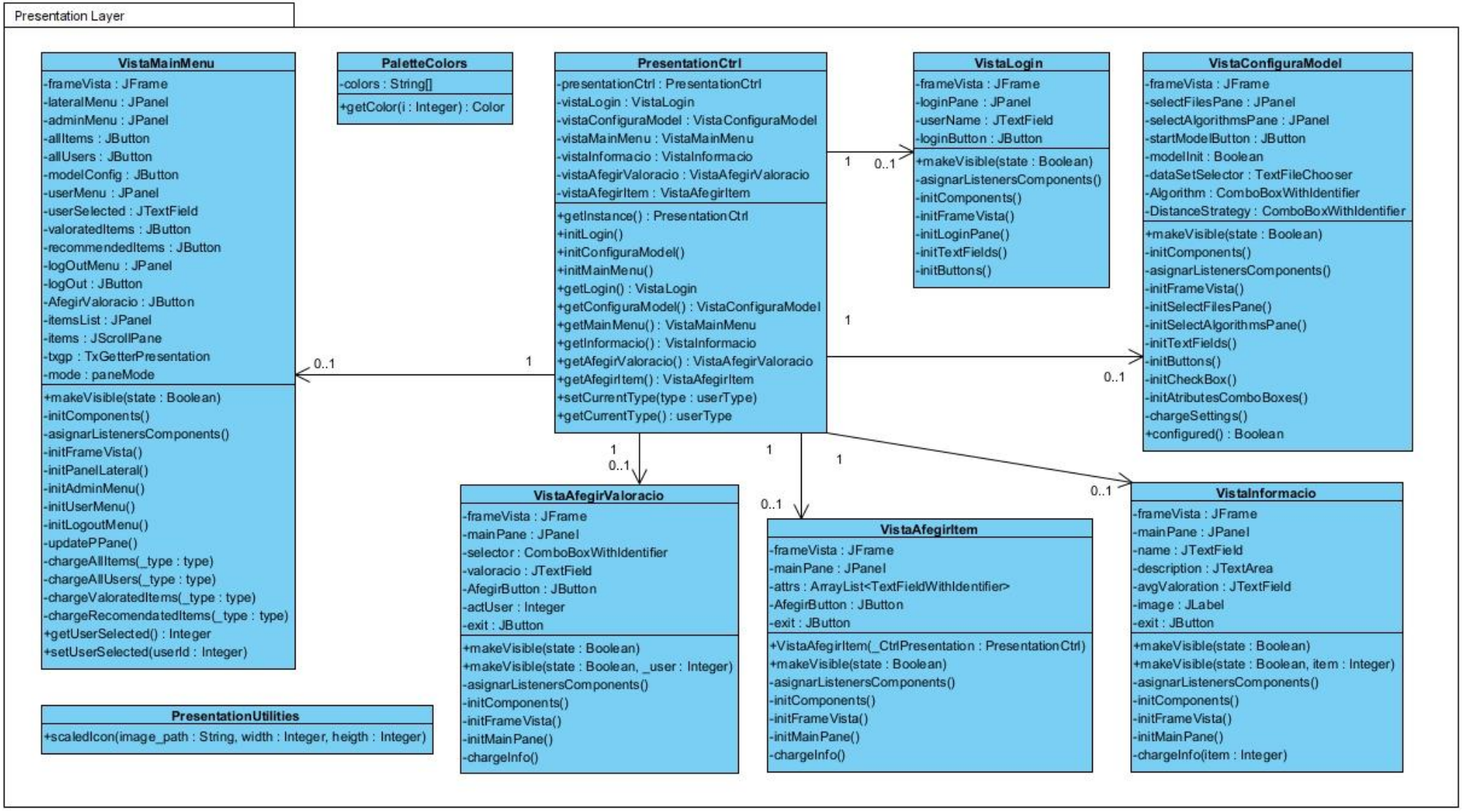
1.2 Model (classes auxiliars)



1.3 Capa de dades i Capa de domini (sense el diagrama de classes del model)



1.4 Capa de Presentació



Repartiment de classes:

- **Value (+fills):** Pol Rivero
- **Attribute (+fills):** Pol Rivero
- **DistanceStrategy (+fills):** Pol Rivero
- **ModelCtrl:** Pol Rivero
- **ClusterCtrl:** Alexandru Dumitru
- **Cluster (+ClusterMember):** Alexandru Dumitru
- **User:** Pol Rivero
- **Rating:** Pol Rivero
- **Item:** Pol Rivero
- **KNNItem:** Pablo José Galván
- **Recommendation (KNN):** Pablo José Galván
- **Recommendation (Slope1):** Ferran de la Varga
- **Recommendation (Mixt):** Pablo José Galván
- **Capa dades:** Pol Rivero
- **Capa de presentació:** Alexandru Dumitru
- **Driver:** Ferran de la Varga

Cal tenir en compte que aquest repartiment de classes no necessàriament és representatiu de la feina que ha fet cada persona, ja que no té en compte la redacció de la documentació o altres elements.

2. Descripció dels atributs i mètodes

Nota: Descripció a alt nivell de la funcionalitat bàsica de cada classe. S'ometen *setters*, *getters* i mètodes instanciadors d'altres classes.

2.1 Domini

Item: Ítem genèric guardat en el sistema (pot ser una pel·lícula, llibre, objecte...).

- *Atributs*: Identificador numèric.
- *Mètodes*: Calcular distància entre 2 Ítems, a partir de les distàncies entre els seus atributs.

KNNItem: Classe genèrica sobre la qual treballa l'algorisme K-nearest neighbours (no necessàriament ha de ser un Item, però en el nostre sistema sempre ho serà).

- *Mètodes*: Calcular els k KNNItems més propers.

User: Persona enregistrada al sistema.

- *Atributs*: Identificador numèric.
- *Mètodes*: Obtenir els Ratings fets als Ítems valorats, calcular la distància entre 2 usuaris a partir de les valoracions que han fet.

Rating: Puntuació que un Usuari dona a un Ítem.

- *Atributs*: Puntuació normalitzada (de 0.0 a 1.0).

Cluster: Encarregada de gestionar i agrupar els membres.

- *Atributs*: Membres pertanyents a aquest clúster (`members`) i coordenades del centroide del cluster (`coord`).
- *Mètodes*:
    - Operacions per calcular el nou centroide del Cluster, a partir de la mitjana de totes les coordenades dels seus membres (normalitzades a [0, 1.0]).
    - Operacions per calcular la diferència dels centroides respecte la iteració anterior. Si la variació és menor a la tolerància, s'acaba l'algorisme.
    - Operacions per inicialitzar el Cluster amb un centroide aleatori.

ClusterCtrl: L'encarregat de gestionar l'estat dels clústers, executant l'algorisme K-means.

- *Atributs*: El conjunt de clústers que hi ha al sistema (`clusters`). Una flag que indica que els clústers estan actualitzats (`initialized`), de forma que K-Means només s'executi quan és necessari.
- *Mètodes*:
    - Operacions per inicialitzar tots els clústers amb una coordenada base.
    - Operacions per executar l'algorisme kmeans (`run_kmeans`).
    - Diverses operacions per aproximar un bon valor a K (`compute_silhouette` i `compute_wss`).

ClusterMember: Classe genèrica sobre la qual treballa el ClusterCtrl (no necessàriament ha de ser un User).

- Atributs: El cluster al qual pertany el membre
- *Mètodes*:
    - Operacions per calcular el nou cluster al qual pertany el membre.
    - Operació abstracta (delegada a User) per obtenir les coordenades del membre.
    - Diverses operacions que ajuden al ClusterCtrl a aproximar un bon valor de K.

Attribute: Paràmetre que té un ítem (per exemple, la durada de la pel·lícula). Pot ser de tipus booleà (`true`/`false`), numèric (coma flotant) o categòric (llista d'etiquetes).

- *Atributs*: Nom del paràmetre, pes en el càlcul de recomanacions.
- *Mètodes*: Operacions per obtenir una llista d'atributs a partir dels valors llegits d'un CSV.

Value: Valor que pren un Atribut en un Ítem concret. Pot ser de tipus booleà, numèric o categòric (el mateix tipus que l'atribut corresponent).

- *Atributs*: El valor corresponent.
- *Mètodes*: Calcular distància entre 2 valors (cal que siguin del mateix tipus). Crear un valor derivat a partir d'una llista d'operadors (veure apartat 3.7: Creació d'un atribut derivat).

ModelCtrl: L'encarregat de gestionar l'estat del model i emmagatzemar les opcions escollides per l'Admin.

- *Mètodes*: Obtenir les recomanacions per un usuari actiu, gestionar les estratègies per calcular la distància i obtenir recomanacions.

RecommendationStrategy: Interfície que implementen els diferents algorismes de recomanació que es vulguin afegir al sistema.

- *Mètodes*: Executar algorisme de recomanació.

DistanceStrategy: Interfície que implementen els diferents algorismes de càlcul de distància que es vulguin afegir al sistema.

- *Mètodes*: Calcular distància, donat un vector de distàncies parcials.

KMeansSlopeOne: És on es troba l'algorisme SlopeOne

- *Mètodes:* Implementació de la interfície RecommendationStrategy i diverses operacions auxiliars per cridar K-Means al principi i executar SlopeOne (veure apartat 3.2: Recomanació basada en K-Means i Slope1).

KNearestNeighbours: És on es troba l'algorisme KNearestNeighbours.

- *Mètodes:* Implementació de la interfície RecommendationStrategy, i operacions auxiliars per implementar KNN utilitzant les operacions de KNNItem (veure apartat 3.3: Recomanació basada en K-Nearest Neighbours).

ItemAffinity: És un parell amb un ítem i un real que representa la mitjana ponderada d'afinitats estimades d'aquest ítem a un usuari actiu, utilitzada pels algorismes KNearestNeighbours i híbrid.

- *Mètodes:* Afegir nova mostra d'afinitat a la mitjana amb un pes.

AffinityList: És una llista d'instàncies d'ItemAffinity respecte un usuari actiu utilitzada pels algorismes KNearestNeighbours i híbrid.

- *Atributs:* Mètode a utilitzar per calcular les afinitats.
- *Mètodes:* Actualització de les afinitats de la llista a partir dels ítems valorats per un nou usuari amb funcions auxiliars, ordenació de la llista segons l'afinitat i conversió a llista d'ítems ordenada.

Hybrid: És on es troba l'algorisme híbrid que combina KMeans + KNN.

- *Atributs:* Una llista d'afinitats que s'actualitza amb els ítems valorats pels usuaris del clúster de l'usuari actiu, i el mètode a utilitzar per calcular les afinitats segons la distància de l'usuari actiu a la resta.
- *Mètodes:* Implementació de la interfície RecommendationStrategy, i operacions auxiliars per assegurar que els clústers estan actualitzats i cridar KNN sobre tots els membres del clúster (veure apartat 3.4: Recomanació basada en un mètode híbrid).

RecommendationQuery: Classe intermitja que conté informació sobre una query a l'algorisme de recomanació.

- *Atributs*: Valoracions conegudes de l'usuari, llista d'ítems de la query, valor Q i, opcionalment, valoracions desconegudes de l'usuari.
- *Mètodes*: Operacions per construir la query, operacions per calcular el NDCG d'una permutació (només si es tenen les valoracions desconegudes).

UserRating: Classe intermitja que ajuda a implementar les operacions en les quals cal llegir un CSV de ratings. Conté un userId i un conjunt de valoracions que aquest usuari ha fet a diferents items.

- *Atributs*: Identificador de l'usuari i una array de valoracions (parelles ítem-puntuació).
- *Mètodes*: Operació per construir el conjunt de valoracions, donats els valors llegits en un CSV. Setters i getters per gestionar els continguts de l'objecte.

Operator: Les operacions són normes que es poden aplicar sobre un atribut per crear un atribut derivat (veure apartat 3.7: Creació d'un atribut derivat). Operator és l'arrel de l'arbre d'herència amb els diferents operadors.

- *Atributs*: Valor numèric del nou atribut si la condició es compleix. Els fills contenen l'argument de l'operador.
- *Mètodes*: No té cap operació. Els fills implementen una operació per comprovar si la condició (operador + argument) es compleix.

OperatorList: Classe intermitja que conté la informació necessària per derivar un nou valor a partir d'un valor de l'atribut base.

- *Atributs*: Una llista d'operadors, el valor per defecte si no es compleix cap operador, i una referència al nou Atribut derivat (cal afegir un punter a Attribute a les noves instàncies de Value).

OperatorFactory: S'encarrega de crear Operators a partir de strings.

- *Mètodes*: Una única operació `getOperatorList` per, donada una lista de strings, retornar una OperatorList amb tots els operadors instanciats.

Controladors transacció: Les classes del package *TransactionControllers* (`Tx*.java`) ofereixen les operacions que la capa de presentació necessita, de forma que aquesta no hagi d'accedir al model.

- *Atributs*: Strings i tipus de dades bàsics necessaris per executar la transacció. En alguns casos, una variable *result* on es guarda el resultat.
- *Mètodes*: L'única operació pública d'aquestes classes és execute, que no retorna res (però pot deixar el resultat a *result*).

Interfície de dades: Les classes del package *DataInterface* (`*DB.java`) són interfaces amb les operacions de la capa de dades, per minimitzar l'acoblament amb aquesta.

- *Mètodes*:
    - Obtenir una instància (donada la seva clau primària)
    - Comprovar si una instància existeix (donada la seva clau primària)
    - Obtenir totes les instàncies en el sistema
    - Afegir i eliminar instàncies.
    - Guardar la base de dades en un fitxer CSV

DBFactory: Singleton encarregat de crear els controladors de dades i retornar una referència (de la interface) a les classes del domini.

- *Mètodes*: Getters de cada tipus de controlador de BD.

2.2 Dades

GenericDBCtrl: Controlador de BD de tipus genèric <T,K> (guarda objectes de tipus T identificats per claus de tipus K). S'utilitza com a base per implementar les bases de dades concretes.

- *Atributs*: La base de dades està implementada amb un HashMap<K,T>.
- *Mètodes*:
    - Obtenir una instància de T (donada la seva clau primària de tipus K)
    - Comprovar si una instància existeix (donada la seva clau primària)
    - Obtenir totes les instàncies de T en el sistema
    - Afegir i eliminar instàncies.
    - Guardar la base de dades en un fitxer CSV

ItemDBCtrl: Adaptador del GenericDBCtrl als tipus T=Item, K=Integer (itemId).

- *Atributs*: Una instància de `GenericDBCtrl<Item,Integer>`.
- *Mètodes*: Els mateixos que la base de dades genèrica. A més, operació per guardar els pesos dels Atributs i quins Valors són invàlids.

UserDBCtrl: Adaptador del GenericDBCtrl als tipus T=User, K=Integer (userId).

- *Atributs*: Una instància de `GenericDBCtrl<User,Integer>`.
- *Mètodes*: Els mateixos que la base de dades genèrica, però no conté cap operació per guardar usuaris en un fitxer, ja que aquests van implícits en el fitxer de ratings.

RatingDBCtrl: A diferència de *ItemDBCtrl* i *UserDBCtrl*, no conté una base de dades de Ratings, ja que fer-ho consumeix molta memòria i gairebé mai es fan queries sobre la BD de ratings. En lloc d'això, es recolza en la funcionalitat de les bases de dades de Item i User.
Per exemple, `getRating(userId,itemId)` estaria implementat com `u=getUser(userId)`, `i=getItem(itemId)`, `u.getRating(i)`.

- *Mètodes*: Els mateixos que la base de dades genèrica, implementats de la forma explicada amb *ItemDBCtrl* i *UserDBCtrl*. A més, una operació per guardar el CSV amb tots els Ratings.

2.3 Presentació

Nota: La GUI del programa encara s'està desenvolupant i pot patir canvis importants abans de la tercera entrega.

PaletteColors: Conté constants de colors utilitzats en el programa.

PresentationCtrl: Conté la lògica bàsica per inicialitzar i gestionar les views de la capa de presentació.

VistaAfegirItem: S'encarrega del diàleg en el que l'usuari introdueix els valors per crear un nou ítem.

VistaAfegirValoració: S'encarrega del diàleg en el que l'usuari introdueix la nova valoració d'un ítem.

VistaConfiguraModel: S'encarrega del diàleg inicial en el que es seleccionen els fitxers necessaris per configurar el model.

VistaCreateDerivated: S'encarrega del diàleg en el que l'usuari introdueix els operadors per crear un nou atribut derivat.

VistaInformació: S'encarrega del diàleg que mostra la informació bàsica d'un ítem.

VistaLogin: S'encarrega del diàleg que es mostra a l'inici, en el que s'introdueix l'identificador d'usuari per accedir al sistema.

VistaMainMenu: S'encarrega del menú principal de l'aplicació, que consisteix en un conjunt de pestanyes amb llistes que mostren els ítems, usuaris, etc. guardats en el sistema.

VistaManageAttributes: S'encarrega de la pantalla que mostra els atributs del model i dona diverses opcions per crear-ne, eliminar-ne, etc.

PresentationUtilities: Conjunt d'operacions útils per a totes les classes de la capa de presentació.

2.4 Altres

2.4.1 Utilities

ConfigFileReader: El sistema ha de ser molt flexible sense obligar a l'usuari a entrar tots els paràmetres del model (nom de tots els CSV, atributs a utilitzar com a ID, algorismes per defecte, etc.), de forma que la configuració de cada dataset estarà guardada en un fitxer `dataset.info`. Aquesta classe s'encarrega de llegir aquest fitxer i retornar els parells de Clau+Valor.

- *Atributs*: Un *TreeMap* que conté, per cada clau del fitxer, quin valor conté.
- *Mètodes*: Operacions per llegir el fitxer i per consultar el valor d'una clau.

ConfigFileWriter: Fa la feina inversa al ConfigFileReader. Quan es guarden les dades del sistema en un nou dataset, s'utilitza aquesta classe per escriure el fitxer `dataset.info`.

- *Atributs*: Una llista de claus i una llista de valors.
- *Mètodes*: Operacions per afegir un nou parell Clau+Valor, per afegir una línia en blanc, i per escriure el resultat en un fitxer.

CSVReader: Classe encarregada de llegir un fitxer CSV i retornar cada línia d'aquesta en forma d'array de Strings.

- *Atributs*: Un Scanner per gestionar la lectura del fitxer.
- *Mètodes*: Operacions per obrir/tancar el fitxer, per llegir la següent línia del CSV i per comprovar si s'ha arribat al final. Operacions privades per processar les dades llegides i gestionar valors amb comes o salts de línia.

CSVWriter: Classe encarregada de convertir arrays de Strings al format CSV i escriure-les en un fitxer.

- *Atributs*: Un BufferedWriter per gestionar l'escriptura del fitxer.
- *Mètodes*: Operacions per obrir/tancar el fitxer, i per escriure una línia al fitxer. Una operació privada per gestionar valors amb comes o salts de línia.

Pair: Tupla genèrica `<T1,T2>` que només té com a objectiu contenir 2 objectes dels tipus corresponents.

- *Atributs*: Només té un atribut de tipus T1 i un de tipus T2. Els dos són públics.

RandomNumber: Col·lecció de rutines per generar enters i doubles aleatoris de forma senzilla.

- *Mètodes*: Generar un double aleatori. Generar un enter aleatori.

2.4.2 Exceptions

ImplementationError: Error que es llença quan es detecta una condició que mai s'hauria de complir, independentment de l'entrada de l'usuari. La causa d'aquest error només pot ser una mala implementació d'alguna operació.

DBException: Excepció llançada dins d'una de les bases de dades.

ObjectAlreadyExistsException: Tipus de *DBException* que indica que estem intentant introduir una instància a la base de dades però ja existia una entrada amb la mateixa clau externa.

ObjectDoesNotExistException: Tipus de *DBException* que indica que l'objecte que estem intentant llegir no existeix (no hi ha cap entrada amb la clau externa indicada).

2.4.3 Executables

Driver: Interfície de línia de comandes (CLI) de l'aplicació. Admet un fitxer de comandes com a argument, i executa totes les comandes contingudes en el fitxer (com si les hagués teclejat l'usuari) abans de passar el control a l'usuari. En el fitxer README hi ha informació sobre les comandes acceptades.

GUI: Interfície gràfica (GUI) de l'aplicació (l'únic que fa és inicialitzar la capa de presentació i cridar-la). Permet realitzar les mateixes funcionalitats que el Driver, des d'una interfície més còmoda.

JUnitDistance: Joc de proves unitari de totes les classes que implementen la interfície Distance, utilitzant JUnit.

TestAlgorithm: Petit programa per comprovar el funcionament dels algorismes utilitzant un fitxer `queries.txt`, en lloc de donar els CSV de known i unknown.

Main *: Diferents executables per comprovar el funcionament dels algorismes i el seu rendiment.

3. Documentació

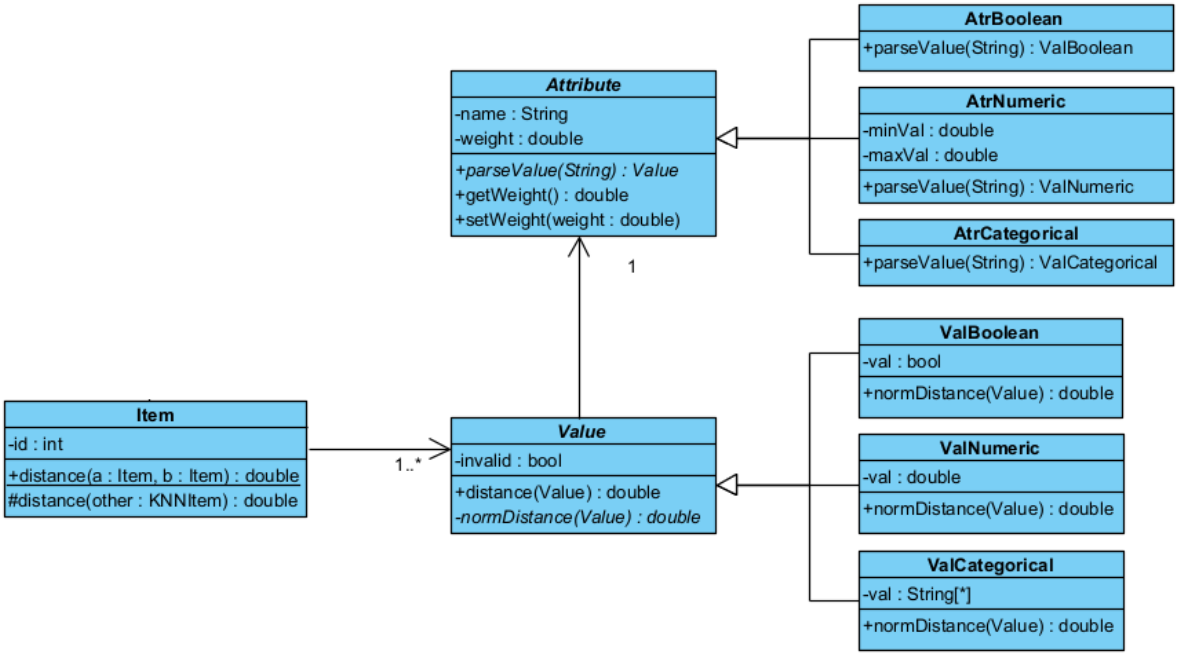
Les noves funcionalitats (no presents a la primera entrega) més importants són l'algorisme de recomanació híbrid (apartat 3.4) i la creació d'un atribut derivat (apartat 3.7). Els altres apartats s'han copiat de la primera entrega i han estat revisats per millorar-ne la redacció o corregir errors. En especial, s'han revisat els costos dels algorismes de recomanació.

Nota: A la primera entrega s'especifiquen els casos d'ús "Guardar recomanacions" i "Llegir recomanacions", els quals guarden el resultat de l'algorisme de recomanacions en un fitxer per llegir-lo posteriorment.
Hem considerat que aquests casos d'ús no aporten cap valor al nostre sistema i són prescindibles, ja que l'execució dels algorismes és prou ràpida i no hem trobat cap situació real on un usuari pugui desitjar guardar una llista de noms en un fitxer.

3.1 Funcionalitat principal / Model

El model es podria dividir en 2 estructures de dades principals: la matriu de valors que prenen els atributs en cada ítem i la matriu de valoracions que els usuaris han fet de cada pel·lícula. Cal tenir en compte que la primera matriu és dispersa (*sparse*), mentre que generalment la segona és densa (*dense*).

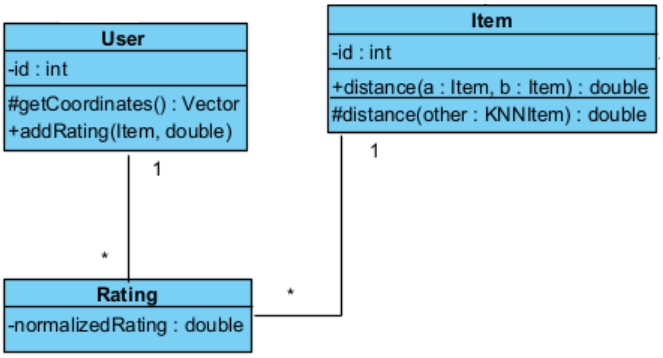
La part del model encarregada d'implementar la matriu d'ítems i atributs és la següent:



Cada ítem té navegabilitat a tots els seus valors, que són d'un dels tipus acceptats (booleà, numèric o categòric). Des del valor es té navegabilitat a l'atribut corresponent, que ha de ser del mateix tipus que el valor.

Per exemple, el ítem amb id 123 té navegabilitat a un valor categòric "`Titanic`" (que apunta al atribut categòric amb nom "Títol"), un valor numèric `120` (que apunta al atribut numèric amb nom "Duració") i un valor booleà `False` (que apunta al atribut booleà amb nom "Adulta").
L'administrador del sistema podrà invalidar els valors d'un atribut per a un conjunt d'ítems (de forma que no s'utilitzin en l'algorisme) i també podrà modificar el pes de cada atribut en l'algorisme de recomanacions.

La matriu *sparse* de valoracions de cada usuari s'implementa com una associació binària entre `User` i `Item`, amb una classe associativa `Rating` que conté la valoració normalitzada (entre 0.0 i 1.0) que l'usuari ha fet. La versió de disseny d'aquesta part del model és la següent:



Finalment, l'administrador ha de poder guardar les preferències de funció de distància i algorisme de recomanació a utilitzar. La classe encarregada de gestionar aquesta funcionalitat és un Singleton, tal com està explicat a la primera entrega.

3.2 Recomanació basada en K-Means i Slope1

3.2.1 Algorisme K-Means

L'algorisme K-Means serà l'encarregat d'agrupar usuaris sobre la base d'uns paràmetres, com per exemple, la valoració que han donat a uns certs ítems. Una de les característiques d'aquest algorisme és que l'espai sobre el que s'executa pot tenir un nombre arbitrari de dimensions. En el nostre cas, aquest espai té tantes dimensions com ítems hi ha al sistema.

El funcionament de l'algorisme és el següent: tenint les valoracions de cada usuari aconseguim les seves coordenades en un espai N-Dimensional (sent N la quantitat de diferents variables/ítems que existeixen). A continuació, a través de fer la mitjana de totes aquestes coordenades, aconseguirem el centre de la mostra, que aplicant uns números aleatoris aconseguim generar uns centroides que ens permeten agrupar a tots els usuaris.

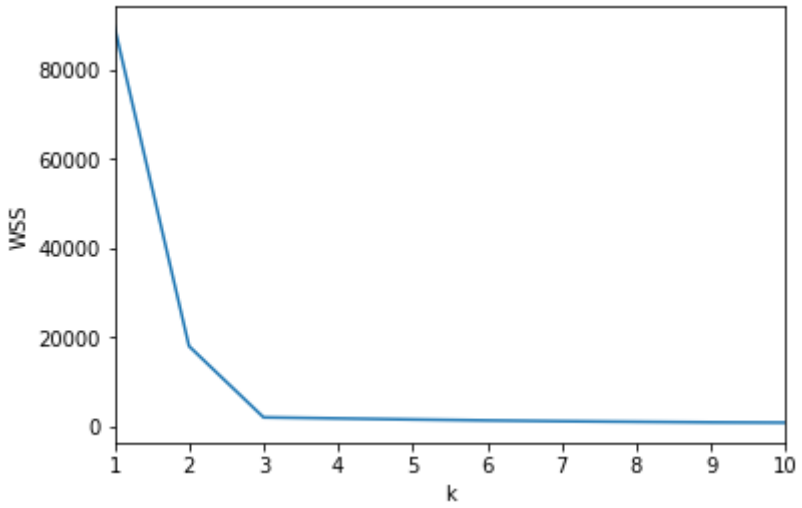
La quantitat de centroides que existiran depèn de la mostra oferta a l'algorisme, per aquest motiu, a través d'algorismes com Elbow Method o Silhouette Method, podem computar com seria l'ideal.

Tornant al funcionament de l'algorisme, el que anirà fent serà el següent:

1. Iterar a través de tots els usuaris calculant la seva distància entre tots els centroides i triant la menor. Destacar que quan hi hagi algun usuari que per a una de les variables no tingui valor, aquesta mateixa s'ignorarà.
2. Recalcular tots els centroides fent la mitjana de coordenades de tots els usuaris pertanyents a ell.
3. Repetir pas 1 i 2 fins a arribar al límit assignat en el programa o fins que, després de recalcular els centroides, no hi hagi un canvi molt significatiu.

D'altra banda, per a fer l'algorisme més independent, hem implementat que l'algorisme s'executi automàticament cada vegada que es crea un nou usuari. D'aquesta manera, ho tindrem actualitzat en tot moment.

En quant als algorismes per a identificar el k ideal, la primera implementació ha estat el Elbow Method, que sol ser el més utilitzat ja que la seva implementació és molt senzilla i eficient algorítmicament parlant. Aquest algorisme consisteix en calcular l'error al quadrat de cada punt cap al seu centroide i sumar-lo a una variable comuna. Una vegada executat per a diferents k, hem d'observar el punt on la funció comença a "aplanar-se" fent una forma de colze, com podem observar en la següent gràfica on la k ideal seria 3.



En quant al silhouette, encara que estigui implementat i pot arribar a ser automàtic, és molt ineficient ja que és un algorisme $\Theta(N^2)$ i fins i tot estant paral·lelitzat, continua trigant 10 minuts per cada k en un dataset gran, com per exemple Movielens 6750. Quant a la part automàtica, és molt més fàcil identificar la k ideal amb aquest algorisme ja que solament hem d'identificar el número màxim dels quals ens retorna l'algorisme, que treballa dins d'un rang [-1, 1].

Finalment, per l'eficiència, tenim les següents variables:

- Nombre de clústers (k). Podem assumir que és una constant.
- Nombre d'iteracions màximes (iter). Podem assumir que és una constant.
- N, nombre total d'usuaris al sistema.
- Dim, nombre de variables (nombre d'ítems en el sistema).
- Val, nombre total de valoracions al sistema.

Assumint que k i iter són valors constants, la implementació original d'aquest algorisme tenia una complexitat $\Theta(N \cdot Dim)$, ja que per cada un dels N usuaris és necessari calcular la distància amb tots els clústers (operació que originalment estava implementada sobre un vector de Dim elements).
Aprofitant que la matriu de valoracions és *sparse*, s'ha millorat l'implementació d'aquest algorisme passant només un mapa amb les valoracions fetes (en lloc del vector sencer). Això redueix la complexitat a $\Theta(N + Val)$.

3.2.2 Algoritme Slope1

Donat un usuari *u* i un ítem *i*, l'algorisme Slope1 s'encarrega d'obtenir la possible valoració que donarà l'usuari *u* a l'ítem *i*.

Per dur-ho a terme, l'algorisme es basa en les valoracions que ha fet l'usuari *u* de tots els ítems que hi ha en el sistema (incloent-hi l'ítem *i*); i també es basa en les valoracions que han fet un grup d'usuaris de tots els ítems (incloent-hi també *i*).

Primer de tot vam pensar que podríem utilitzar com a grup d'usuaris tots els usuaris del sistema, però gràcies a l'algorisme K-Means, ens vam adonar que és millor utilitzar com a grup d'usuaris el clúster al qual pertany l'usuari *u*. D'aquesta manera, la possible valoració d'*u* sobre l'ítem *i* és més fiable, ja que només es té en compte les valoracions d'usuaris semblants a *u*, amb els mateixos interessos. D'altra banda és més eficient perquè només es recorren un subconjunt dels usuaris del sistema.

Més profundament, SlopeOne fa es basa en la regressió lineal: $f(x) = n \cdot \Delta x + m$. En aquest algorisme, $\Delta x = 1$ i la funció esdevé $n + m$.
Primer de tot, hi ha una funció `get_recommendations()` que retorna les possibles valoracions de tots els ítems que *u* no ha valorat. Aquesta, per a cada valoració no feta, crida a `slope_one()`, que retorna la possible valoració de l'ítem corresponent. Per a cada ítem del sistema *j* que l'usuari *u* ha valorat, es calcula una predicció de la valoració de *i*: és la suma de la valoració que l'usuari *u* ha fet a *j* (equivaldria a la *m* de la fórmula) més el que retorna l'`average_slope()` (que equivaldria la *n*). La predicció final és la mitjana de totes les prediccions.
Finalment, en quant a `average_slope()`, aquesta funció calcula el pendent mitjà entre *i* i *j*. Per cada usuari (del clúster al qual pertany *u*), calcula la diferència entre la valoració de l'ítem *j* i la valoració de l'ítem *i*.

Per tal que SlopeOne funcioni correctament, l'usuari *u* ha d'haver valorat almenys un ítem i com a mínim hi haurà un usuari en el clúster (l'usuari *u*). Evidentment, com més usuaris hi hagi en el clúster i com més valoracions hagin fet aquests, més precís serà la possible valoració de l'usuari *u*.

Sigui:

- uc el nombre d'usuaris del clúster
- it el nombre d'ítems
- $ival$ el nombre d'ítems valorats per l'usuari actiu

el cost original de predir la valoració de 1 item amb SlopeOne era: $\Theta(it + ival \cdot uc)$.
Aprofitant el fet que la matriu de recomanacions és *sparse*, hem optimitzat l'algorisme per només comprovar els ítems valorats, reduïnt el cost a $\Theta(ival \cdot uc)$.

3.3 Recomanació basada en K-Nearest Neighbours

Per obtenir recomanacions per a l'usuari actiu basades en els ítems que li han agradat, es retorna una llista d'ítems ordenada descendentment segons la probabilitat estimada que a l'usuari li agradin a partir d'una llista d'ítems valorats per aquest usuari, un subconjunt dels ítems existents i un nombre d'ítems a seleccionar per la recomanació. Això s'aconsegueix fent ús de l'algorisme *K-Nearest Neighbours*.

3.3.1 Algorisme K-Nearest Neighbours

Per començar, *K-Nearest Neighbours* té com a funcionalitat trobar els k (o menys si la mostra no en té suficients elements) ítems a menys distància d'un altre ítem donat. Aquest paràmetre k és arbitrari, i dependrà de si volem que un ítem valorat afecti a més o menys ítems del conjunt a ordenar.

Per aconseguir aquest resultat, `KNNItem` té implementat un mètode `getNearestItems`, el qual, donada una llista d'n ítems i un enter k, retorna una llista de tuples de valors enters i reals: l'enter indica l'índex d'un ítem a la llista, i el real la distància a la que es troba aquest element de l'ítem sobre el que s'executa el mètode. La llista de tuples tindrà k elements com a màxim (només en tindrà menys si n és menor que k, i en aquest cas es retornaran n tuples), i representarà els ítems de la llista més propers a l'ítem de referència (que anomenarem **veïns**) en ordre creixent segons la distància.

Aquest mètode està implementat mitjançant una `PriorityQueue` que s'ordena mitjançant un *heap* (de manera que, considerant que el nombre d'atributs és constant, afegir un element té cost $O(log(n))$), afegir els n elements de la llista d'entrada a la cua té cost $O(n{\cdot}log(n))$. L'única altra acció del mètode amb cost no constant és la conversió de `PriorityQueue` a `ArrayList`, on simplement es crea una nova instància d'`ArrayList` i s'hi afegeixen els elements de la cua un a un, i això també té cost $O(n{\cdot}log(n))$. Per tant, el cost total de l'algorisme és $O(n{\cdot}log(n) + n{\cdot}log(n)) = O(n{\cdot}log(n))$.

Cal tenir en compte que aquest càlcul suposa que el nombre d'atributs és constant. Considerem que aquesta suposició és raonable, ja que en datasets reals el nombre d'usuaris i ítems pot variar i arribar a ser molt gran, mentre que el nombre d'atributs acostuma a mantenir-se aproximadament sobre els 12 (en comparació amb el nombre de ítems, és una mida extremadament petita).

3.3.2 Content-based filtering: càlcul d'afinitats

No obstant tot això, l'algorisme *K-Nearest Neighbours* sol, no aconsegueix el nostre objectiu, ja que és capaç d'ordenar una llista d'ítems segons la seva semblança a un ítem concret, però nosaltres necessitem un algorisme que faci això segons tots els ítems que li agradin a l'usuari actiu.

Per aquesta raó, hem creat un algorisme per obtenir aquest resultat executant *K-Nearest Neighbours* per cada ítem `i` valorat per l'usuari (que tingui una valoració superior o igual a un valor arbitrari anomenat `THRESHOLD`) i obtenint un valor (que anomenarem **afinitat**) per cada veí d'aquest ítem `i`, que és major com més probable és que a l'usuari actiu li agradi l'ítem (per defecte és -1). Al final de l'execució, els ítems seran ordenats segons la mitjana de les afinitats trobades per cada ítem.
Sigui:

- I_i un ítem valorat per l'usuari en la posició i de la llista amb valoració R_i superior o igual a `THRESHOLD`.
- V_j el j veí més llunyà d'aquest ítem.
- $D(I_i, V_j)$ la distància entre I_i i V_j.
- A_{V_j} la nova afinitat per afegir a les afinitats trobades per aquest veí.

Tenim 3 mètodes per trobar les afinitats dels ítems:

- `NORMAL`: $A_{V_j} = R_i$. Aquest mètode és el més senzill i ràpid, però té un problema greu: les distàncies dels veïns als ítems valorats en si no tenen cap pes en l'afinitat (és a dir, que, siguin V_0 i V_1 dos veïns d'I_i, sempre es compleix que $A_{V_0} = A_{V_1}$, fins i tot si $D(I_i, V_1)$ és molt més gran que $D(I_i, V_0)$). Per tant, aquest mètode pot no reflectir correctament les diferències entre els ítems valorats i els seus veïns.
- `LOG`: $A_{V_j} = \frac{R_i}{log(10 + D(I_i, V_j))}$. Aquesta forma d'obtenir l'afinitat no és excessivament lenta, però, a diferència de l'anterior mètode, sí utilitza les distàncies dels veïns per reduir l'afinitat. El motiu pel qual s'utilitza un logaritme i se suma 10 al seu paràmetre i no es divideix directament per la distància és per dos motius: d'una banda, a priori no sabem quina pot ser la grandària de les distàncies, i pot donar-se el cas que aquestes són massa grans, causant que les afinitats siguin massa petites, cosa que es pot alleujar utilitzant una escala logarítmica per la distància; d'altra banda, d'aquesta manera s'evita la possibilitat de divisió entre zero, ja que el paràmetre del logaritme serà 10 com a mínim, de forma que el valor mínim del logaritme és 1 (i així, si la distància és 0, l'afinitat serà exactament R_i, ja que si dos ítems són iguals, tindria sentit que rebessin la mateixa valoració).
- `ORDERED`: el mètode `LOG` és ràpid i pot donar bons resultats, però es pot donar el cas on no vulguem descartar un ítem de la llista a ordenar si és veí d'un ítem ben valorat per l'usuari actiu, fins i tot si aquest ítem és llunyà, però `LOG` penalitza molt els ítems distants. Per provar si l'alternativa proposada funciona millor, s'ha inclòs el

mètode ORDERED, que permet que ítems distants tinguin una bona posició si són veïns d'ítems ben valorats.

Aquest mètode comença ordenant la llista d'ítems valorats per l'usuari segons les valoracions en ordre decreixent. A continuació, per cada I_i troba la distància màxima D_{max} a la qual es troba d'un veí seu, i per cada V_j calcula un nombre $x = (R_i - R_{i+1}) \cdot \frac{D(I_i, V_j)}{D_{max}}$, on R_{i+1} o bé és la valoració immediatament inferior de la llista o bé, si no hi ha més valoracions superiors o iguals a TRESHOLD a continuació, simplement és TRESHOLD. L'afinitat nova serà $A_{V_j} = R_i - x$, que sempre donarà un nombre en l'interval $[R_{i+1}, R_i]$ i estarà interpolat segons la distància del veí al ítem valorat: si aquesta és D_{max} l'afinitat serà R_{i+1}, i si aquesta és 0 l'afinitat serà R_i.

Es pot veure que aquest algorisme és més costós que els altres dos, però ens permetrà comprovar si l'alternativa explicada dona millors resultats que LOG.

Finalment, sigui $|R|$ el nombre d'ítems valorats per l'usuari actiu, $|Q|$ la mida de la llista d'ítems a partir de la qual es genera la recomanació i k el nombre de veïns per ítem valorat màxims, el cost temporal d'aquest algorisme és $O(|R| \cdot |Q| \cdot log(|Q|))$, ja que per cada ítem valorat es calculen els seus veïns més propers amb l'algorisme de *K-Nearest Neighbours*, i cada càlcul d'aquests té, com s'ha calculat a la secció anterior, un cost temporal de $O(|Q| \cdot log(|Q|))$. Hi ha altres bucles que iteren sobre els veïns o els ítems valorats, i en el mètode ORDERED també s'ordenen els ítems valorats al principi, però cap d'aquests supera el del càlcul d'afinitats, de forma que el cost total és $O(|R| \cdot |Q| \cdot log(|Q|))$.

3.4 Recomanació basada en un mètode híbrid

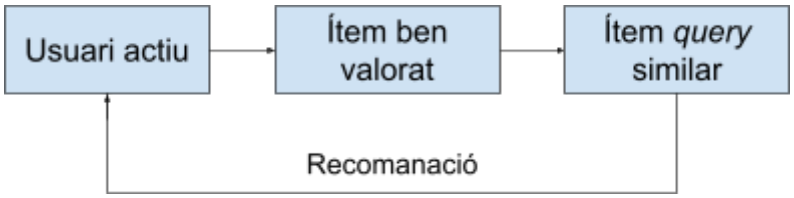
Els dos algorismes de recomanació utilitzats fins ara tenen els seus avantatges i inconvenients.

D'una banda, la recomanació basada en *collaborative filtering* permet mostrar a l'usuari actiu contingut diferent al que està acostumat i, per tant, facilita que descobreixi nous interessos. Això és positiu, ja que si es recomana el mateix tipus d'ítems a l'usuari constantment és probable que acabi trobant les recomanacions repetitives i poc interessants. No obstant això, no es tenen en compte els ítems valorats per l'usuari actiu, causant que els ítems recomanats no siguin coherents amb el que li agrada realment a aquest usuari.

D'altra banda, la recomanació amb *content-based filtering* és més ràpida que l'esmentada anteriorment, i utilitza les pròpies preferències de l'usuari actiu, aconseguint que els ítems recomanats s'apropin més al que l'usuari sol valorar positivament. Això, però, pot resultar monòton per l'usuari, ja que pot recomanar ítems massa semblants entre ells, i tampoc explora altres possibilitats que a l'usuari li puguin agradar encara més que el que sol consumir.

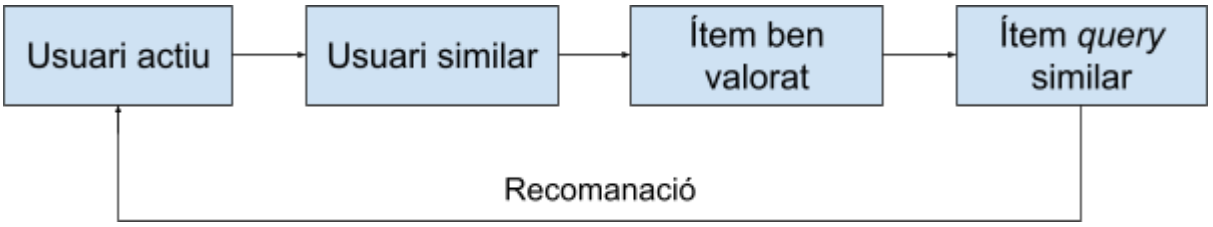
Per tractar d'arreglar els inconvenients de cada estratègia, s'ha utilitzat un algorisme híbrid, el qual intenta combinar els punts forts de cada algorisme per trobar ítems que li agradin a l'usuari amb els menors biaixos possibles.

Aquesta estratègia es basa en canviar els ítems sobre els que opera el nostre algorisme de *content-based filtering*. Amb *KNearestNeighbours*, cada recomanació segueix el següent procés:



On es busquen els ítems ben valorats per l'usuari actiu i es troben els ítems donats per la *query* que s'hi assemblen, els quals s'ordenen segons la valoració que l'usuari actiu ha donat als ítems similars.

Com s'ha mencionat, aquest mètode té l'inconvenient que no es tenen en compte altres valoracions fora del que l'usuari actiu coneix. Per solucionar aquest problema, cada recomanació de l'algorisme híbrid substitueix el procés anterior per aquest:



Amb el qual es troben els usuaris similars a l'actiu i es comparen els ítems que els hi han agradat amb els de la *query*. Com més s'assembli un ítem de la *query* als ítems millor valorats pels usuaris similars, més recomanable serà i se situarà abans a la llista ordenada de recomanacions.

Per aconseguir aquest procés, s'utilitza una llista d'afinitats mitjanes com a l'estratègia de *content-based filtering*, se seleccionen els usuaris que pertanyin al mateix clúster que l'actiu (que són considerats els usuaris similars) i s'executa l'algorisme basat en *KNearestNeighbours* per cadascun d'ells, actualitzant la llista d'afinitats ja existent. Les afinitats trobades per cada usuari similar es multiplicaran per un nombre en el rang (0, 1], el qual determinarà el pes d'aquestes afinitats segons l'usuari. Aquest multiplicador d'un usuari i l'anomenarem m_i. Per trobar-lo hi ha dos mètodes:

- `NORMAL`: $m_i = 1$. Aquest és òbviament el mètode més senzill, i té com a inconvenient que no fa cap distinció entre els usuaris dels quals s'obtenen les afinitats, encara que uns siguin més diferents a l'actiu que altres. No obstant això, en ser un mètode immediat, és també el més ràpid, ja que no ha de fer cap càlcul complex.
- `LOG`: $m_i = \frac{1}{log(10 + D(i,j))}$, on $D(a, b)$ és la distància entre dos usuaris calculada respecte a les seves valoracions per cada ítem (dos usuaris seran molt similars si han valorat els mateixos ítems de forma molt semblant), i és l'usuari semblant sobre el qual s'està iterant i j és l'usuari actiu. Aquí tenim l'avantatge que un usuari menys semblant a l'actiu no influirà en l'afinitat d'un ítem tan positivament com un de més proper. Malgrat aquest avantatge, aquest mètode resulta ser més lent que l'anterior a causa del càlcul de la distància, i ja que s'ha de iterar sobre un nombre possiblement molt gran d'usuaris que poden haver valorat una quantitat significativa d'ítems, això causa una pèrdua de velocitat notable.

Finalment, sigui $|I|$ el nombre total d'ítems en el sistema, $|C|$ el nombre d'usuaris per clúster, $|Q|$ la mida de la llista d'ítems a partir de la qual es genera la recomanació i k el nombre de veïns per ítem valorat màxim a l'algorisme de *KNearestNeighbours*, considerant que el nombre d'atributs de cada ítem és constant, el cost de l'algorisme híbrid serà de $O(|C|\cdot|I|\cdot|Q|\cdot log(|Q|))$, ja que s'iterarà sempre sobre el nombre d'usuaris del clúster de l'usuari actiu i per cadascun s'executarà l'algorisme de *KNearestNeighbours*, el qual, com ja s'ha calculat anteriorment, tindrà un cost temporal de $O(|R_i|\cdot|Q|\cdot log(|Q|))$, on $|R_i|$ és el nombre de valoracions fetes per l'usuari i, i cada usuari tindrà com a màxim $|I|$ valoracions. En el cas del mètode `LOG`, també s'iterarà sobre els ítems valorats de cada usuari una vegada més, però si hi afegim aquest cost $O(|I|)$, l'expressió és equivalent a l'esmentada.

3.5 Funcions de distància

L'administrador pot escollir entre 4 funcions diferents que els algorismes de recomanació utilitzaran per calcular la distància 2 punts (vectors N-dimensionals). El procés sempre és el mateix:

1. Restar els vectors per obtenir el vector de distàncies parcials. Aquest vector conté, per cada component, la variació (delta) entre els 2 punts.
2. Calcular el mòdul del vector, utilitzant la funció escollida per l'usuari. Cal tenir en compte que alguns components del vector poden ser invàlids (representats amb NaN en el sistema).

Distància Euclídea: Consisteix en aplicar el Teorema de Pitàgores. És a dir, el resultat és la suma de quadrats de tots els factors (ignorant els invàlids).
En una implementació tradicional d'aquest algorisme es faria l'arrel quadrada de la suma, però en el nostre cas no ens importa la magnitud total de les distàncies (només volem saber si una distància és major o menor que una altra). Com que aquesta operació es pot cridar moltes vegades i l'arrel quadrada és una operació costosa, hem decidit eliminar aquesta arrel quadrada final.

Distància Manhattan: Similar a la distància Euclídea, però es realitza el sumatori dels valors absoluts dels factors (no dels seus quadrats).

Distància Mitjana: Aquesta noció de distància busca mitigar un defecte dels dos algorismes anteriors. En molts casos els vectors tenen components invàlids, que les distàncies anteriors simplement ignoren. Això pot donar resultats inesperats, ja que no sumar una distància és equivalent a que aquesta sigui 0 (de forma que un vector amb tots els components invàlids estaria a distància 0 de tots els punts de l'espai).
La distància Mitjana consisteix en fer la mitjana del valor absolut de les distàncies parcials vàlides, de forma que els valors invàlids deixen de ser equivalents a un 0.
Per exemple, la mitjana de $(1,\ 2,\ 3,\ invalid)$ seria $6/3 = 2$, mentre que la mitjana de $(1,\ 2,\ 3,\ 0)$ seria $6/4 = 1.5$.

Distància Mitjana al quadrat: Similar a la distància Mitjana, però no es sumen els valors absoluts dels factors, sinó els quadrats d'aquests.
Per exemple, la mitjana al quadrat de $(1,\ 2,\ 3,\ invalid)$ seria $(1 + 4 + 9)/3 = 4.66$.

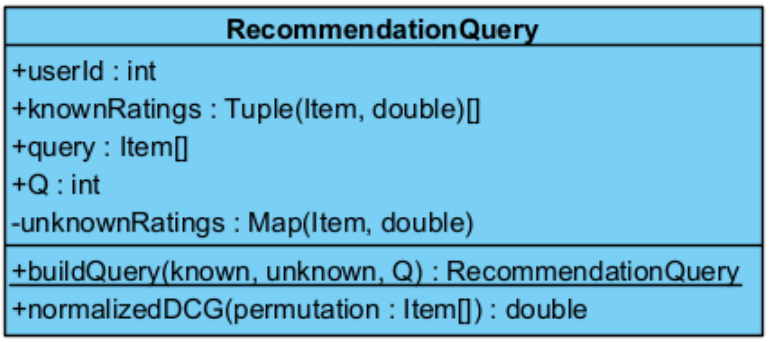
En la nostra experimentació no hem notat cap diferència notable en els resultats obtinguts amb els diferents tipus de distància. Si bé els resultats canvien lleugerament en alguns casos, no hi ha cap distància que sigui constantment millor que les altres.

3.6 Avaluació d'un conjunt de recomanacions

Un cop s'ha executat l'algorisme en qüestió, es puntua el seu resultat calculant el *Discounted Cumulative Gain (DCG)* de la permutació que ha retornat.

Per avaluar el model, l'usuari ha de llegir 2 fitxers. El primer codifica una matriu *Known* que conté un conjunt d'usuaris nous (no existeixen al model) i les valoracions que han fet a alguns ítems (ja existents). Aquestes valoracions seran utilitzades per l'algorisme de recomanacions a l'hora de fer prediccions.
El segon fitxer codifica una matriu *Unknown* que conté, per cada usuari del conjunt anterior, una sèrie de valoracions que han fet a ítems ja existents (diferents dels anteriors). Aquestes valoracions són les que l'algorisme haurà de predir (i aquestes "autèntiques valoracions" són les que s'utilitzen per avaluar la qualitat del model).

El sistema compta amb una classe especialitzada per gestionar aquest procés. Donats els conjunts de valoració *known* i *unknown*, i la mida Q del resultat, crea les queries en el format compatible amb els algorismes de recomanació i guarda l'autèntic resultat per calcular posteriorment el DCG.



El `ModelCtrl` rep les `RecommendationQuery` i s'encarrega de crear l'usuari actiu (identificat pel camp `userId` de la query) i actualitzar les estructures de dades dels algorismes de recomanació, en cas que sigui necessari. Posteriorment, passa l'usuari, els ítems de la query i la mida Q als algorismes de recomanació. Finalment, esborra l'usuari actiu i retorna la permutació calculada per l'algorisme de recomanació.

És aleshores quan s'utilitza `RecommendationQuery::normalizedDCG()` per calcular la qualitat de la permutació retornada, en forma de DCG normalitzat.

Donada una permutació `Perm` amb Q elements dels quals coneixem l'autèntica puntuació donada per l'usuari (guardada a `RecommendationQuery`), el càlcul del DCG consisteix en realizar el següent sumatori:

$$\sum_{i=1}^{Q} \frac{2^{Perm[i].trueRating}-1}{log(i+1)}$$

Per rebre una puntuació normalitzada entre 0.0 i 1.0, podem calcular el DCG ideal (és a dir, el DCG de la permutació òptima, aquella que està ordenada segons les autèntiques valoracions) i fer la divisió:

$$NormalizedDCG = \frac{DCG(Perm)}{DCG(Permutació\ òptima)}$$

3.7 Creació d'un atribut derivat

La creació d'un atribut derivat funciona d'una forma similar a la *access list* d'un firewall. L'usuari ha d'introduir una llista de condicions, cada una acompanyada del valor que prendrà el nou atribut si la condició és certa. Per cada un dels ítems, les condicions es comproven de forma seqüencial i, si una retorna *true*, s'agafa el valor d'aquesta per al nou atribut. Si totes les comprovacions retornen *false*, s'utilitza el valor per defecte introduït per l'usuari.
Per exemple, donada una llista d'operacions:

Operador	**Argument**	**Valor si *true***
<	10	50
<=	20	150
=	30	123
Valor per defecte: 10		

El fragment rellevant de la matriu de ítems resultant seria:

itemId	**Atribut A**	**Nou atribut derivat**	**Notes**
1	5	50	Es compleix 5<10
2	10	150	Falla 10<10, es compleix 10<=20
3	20	150	Falla 20<10, es compleix 20<=20
4	30	123	Fallen els 2 primers, es compleix 30=30
5	40	10	Fallen totes les comprovacions

També es poden utilitzar els valors INVALID (per fer que el valor del nou atribut no es tingui en compte en el càlcul de distàncies) i SAME (per copiar el valor de l'atribut antic en el nou). Cal tenir en compte que els atributs creats sempre són numèrics, de forma que SAME només es pot utilitzar si l'atribut original és numèric.

Per exemple, podem modificar l'exemple anterior per utilitzar SAME i INVALID:

Operador	**Argument**	**Valor si *true***
<	10	50
<=	20	**SAME**
=	30	**INVALID**
Valor per defecte: 10		

El fragment rellevant de la matriu de ítems resultant seria:

itemId	**Atribut A**	**Nou atribut derivat**	**Notes**
1	5	50	Es compleix 5<10
2	10	**10**	Falla 10<10, es compleix 10<=20 -> A=10
3	20	**20**	Falla 20<10, es compleix 20<=20 -> A=20
4	30	**[INVALID]**	Fallen els 2 primers, es compleix 30=30
5	40	10	Fallen totes les comprovacions

Cal recordar que no és possible crear un atribut derivat a partir de un atribut booleà, ja que aquest no conté informació suficient. Ara bé, sí que és possible derivar a partir d'atributs categòrics, però l'únic operador disponible (de moment) és ∈ (CONTÉ).

Operador	**Argument**	**Valor si *true***
∈	cat	10
∈	es	20
Valor per defecte: INVALID		

itemId	**Atribut B**	**Nou atribut derivat**	**Notes**
1	cat ; es	10	Es compleix cat ∈ {cat, es}
2	es ; cat	10	Es compleix cat ∈ {es, cat}
3	es ; en	20	Falla la primera, Es compleix es ∈ {es, en}
4	en	[INVALID]	Fallen totes les comprovacions

En quant a les estructures de dades, aquest mecanisme està implementat amb una jerarquia de classes, que té la classe abstracta Operator com a *root*.
Un Operator conté el valor numèric a assignar al nou atribut si l'operator es compleix, i pot ser del tipus NumericOperator o CategoricalOperator.
Cada operador disponible (OperatorLT, OperatorLE, OperatorEQ, OperatorCONTAINS, etc.) hereda del tipus corresponent d'operador (de moment l'únic fill de CategoricalOperator és OperatorCONTAINS) i conté la lògica per retornar cert (nou valor per l'atribut derivat) o fals (*null*, per indicar que cal seguir executant operacions).

La classe Value rep una llista de Operators abstractes i s'encarrega d'executar-los fins que un d'ells no retorna null. Per crear un nou atribut, es repeteix aquesta operació per tots els *values* de l'atribut base que es vol derivar.