Miércoles 15 Diciembre 2021

Sistema de Recomendación

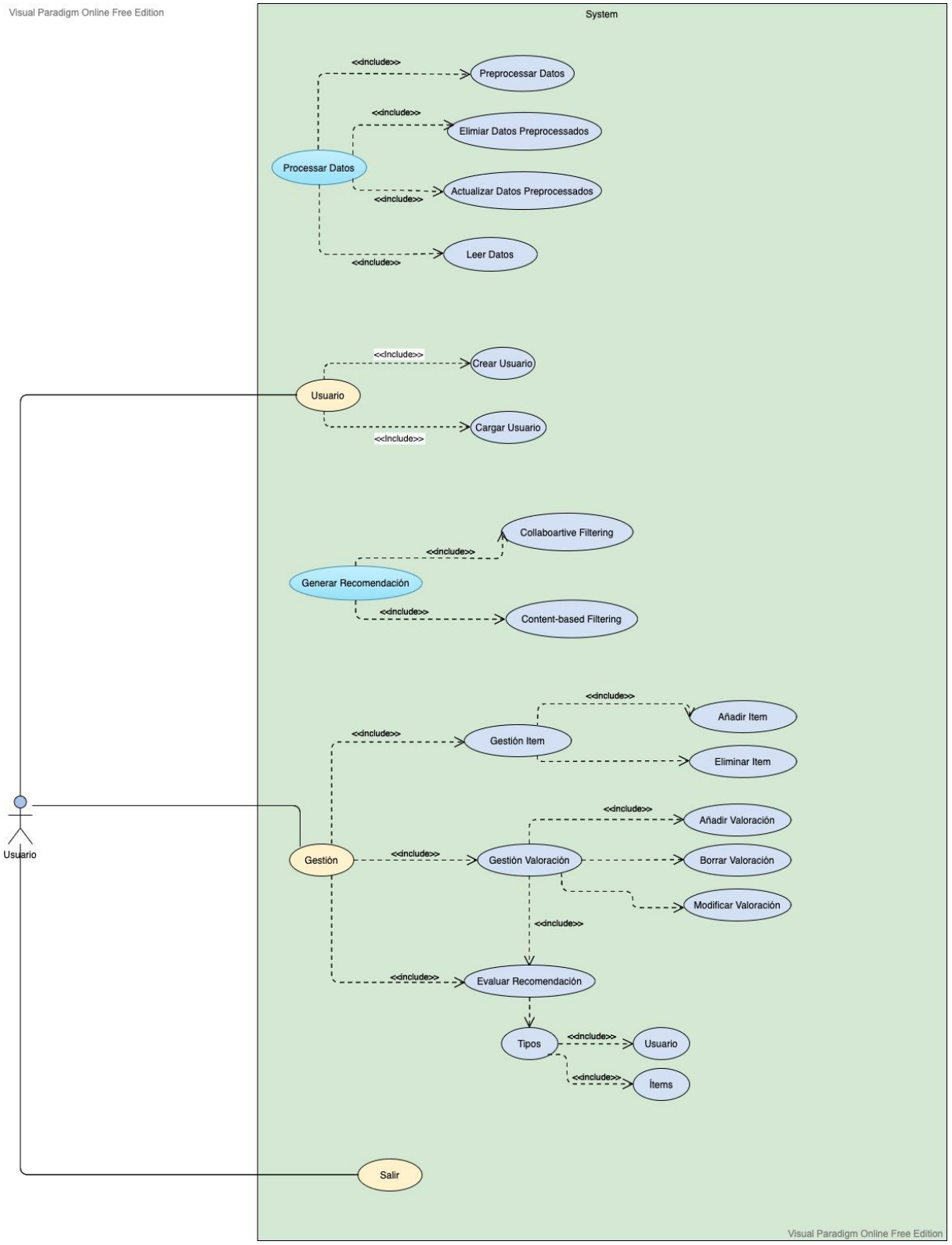
versión entrega 2.0

PROP
Universitat Politècnica de Catalunya
2021-2022 Q1

SUBGRUPO PROP 2.1:
ANDRÉS EDUARDO BERCOWSKY RAMA
CARLA CAMPÀS GENÉ
BEATRIZ GOMES DA COSTA
PAU VALLESPÍ MONCLÚS

1 CASOS DE USO

1.1 DIAGRAMA



1.2 DESCRIPCIÓN

El actor principal en nuestro caso será el gestor del sistema, quien tendrá información de varios usuarios y la ejecutará de forma que podrá controlar el sistema.

1.2.1 Procesar Datos

Al inicializar el escenario que un usuario quiere representar, se tiene que pasar la información necesaria para poder inicializar el escenario. Esto se hace mediante ficheros csv (item, ratings, ratings.known y ratings.unknown).

Preprocesar Datos: Queremos preprocesar los datos que deberá utilizar nuestro sistema para tener más eficiencia en la ejecución de nuestro programa.
Actor Principal: Gestor del sistema
[Precondición]: El sistema acaba de ser inicializado por el usuario, aún no se ha ejecutado ninguna otra instrucción por parte del usuario.
[Detonante]: Inicialización del sistema.
Escenario principal:

- El gestor del sistema inicia el sistema, el sistema lanzará automáticamente el preprocesamiento de datos

[Extensiones]: --

Eliminar Datos Preprocesados: Cuando nuestro sistema, o la sesión activa en la que este esta ya no se requiere, los datos preprocesados que este tiene se pueden borrar ya que ya no són necesarios para le ejecución de nuestro programa.
Actor Principal: Gestor del sistema
[Precondición]: El sistema está activo.
[Detonante]: Salir de la sesión activa del sistema en la que está el usuario.
Escenario principal:

- El gestor del sistema usa el sistema para mirar las recomendaciones que este hace sobre diferentes ítems.
- El gestor del sistema acaba esta sesión.

[Extensiones]: --

Actualizar Datos Preprocesados: Mientras nuestro sistema se está ejecutando ciertos datos se pueden añadir, cambiar o borrar dependiendo de la necesidad del sistema y del usuario.
Actor Principal: Gestor del sistema
[Precondición]: El sistema está activo y el usuario está procesando recomendaciones con este.
[Detonante]: El gestor de sistema introduce un usuario nuevo en el sistema con sus valoraciones respectivas.
Escenario principal:

- El gestor del sistema usa el sistema para mirar las recomendaciones que este hace sobre diferentes ítems.
- El gestor del sistema añade un identificador de un usuario que no ha sido previamente visto por nuestro sistema, y sus respectivos identificadores.
- El sistema crea las recomendaciones para este usuario, y añade al nuevo usuario al grupo de usuarios de nuestro sistema.

[Extensiones]: --

Leer Datos: Leeremos los datos de los cuales se hará el preproceso, este tiene dos facetas. Se leerán los datos de los ficheros csv, y después se leerá el input del canal estándar para la ejecución del sistema de nuestro gestor de sistema.
Actor Principal: Gestor del sistema
[Precondición]: El sistema acaba de ser inicializado por el usuario, aún no se ha ejecutado ninguna otra instrucción por parte del usuario.
Escenario principal:

- El gestor del sistema inicia el sistema, el sistema lanzará automáticamente el preprocesamiento de datos

[Detonante]: la inicialización de nuestro sistema.
[Extensiones]: --

1.2.2 Usuario

Los usuarios del sistema serán esos creados por el gestor, se identificarán por un identificador único, dentro de nuestro sistema será generado por un identificador generado por nuestro preproceso que se mapeara a esos ids usados externamente.

Crear Usuario: Un usuario se crea en cuanto primero vemos a este usuario, y posteriormente se cargan las valoraciones de este.
Actor Principal: Gestor del sistema
[Precondición]: El sistema está activo.
Escenario principal 1:

- El gestor del sistema inicializa el sistema y este empieza a leer los datos y a preprocesar estos.
- Como no se ha visto ningún usuario previamente, todos los usuarios serán nuevos en el sistema este los deberá crear.

Escenario principal 2:

- El gestor del sistema ha inicializado los datos, estos ya se han leído y preprocesado.
- El gestor del sistema ha empezado a ejecutar comandos para el sistema de recomendaciones.
- Si algún usuario no se ha visto antes, el usuario entrará sus valoraciones y el sistema creara un nuevo usuario.

[Extensiones]: --

Cargar Usuario:
Actor Principal: Gestor del sistema
[Precondición]: El sistema está activo y el usuario está procesando recomendaciones con este.
[Detonante]: El usuario pide una recomendación de un usuario que ha sido previamente procesado.
Escenario principal:

- El gestor del sistema ha inicializado los datos, estos ya se han leído y preprocesado.
- El gestor del sistema ha empezado a ejecutar comandos para el sistema de recomendaciones.
- El gestor del sistema pide un usuario que ha sido previamente preprocesado por nuestro sistema, y sus datos vendrán de esos en nuestro sistema.

[Extensiones]: --

1.2.3 Generar Recomendación

El usuario podrá elegir entre dos posibles algoritmos de recomendaciones, una de estas será collaborative filtering que usará K-Means y Slope One para crear las predicciones, y la otra será Content Based Filtering que generará las predicciones basado en el algoritmos K-Nearest Neighbours.

Collaborative Filtering: Collaborative filtering es uno de los sistemas de recomendación que vamos a usar. Este ejecutará inicialmente K-Means y posteriormente Slope One para evaluar esos items que el usuario aún no ha valorado.

Actor Principal: Gestor del sistema

[Precondición]: El sistema está activo y el usuario está procesando recomendaciones con este.

[Detonante]: El gestor del sistema le da un nuevo usuario al sistema de recomendaciones.

Escenario principal:

- El gestor del sistema ha inicializado los datos, estos ya se han leído y preprocesado.
- El gestor del sistema ha empezado a ejecutar comandos para el sistema de recomendaciones.
- El gestor del sistema entra en el sistema un usuario para obtener las recomendaciones
- El sistema inicializa el algoritmo K-Means para romper el grupo de usuarios entre clusters y asignar a nuestro nuevo usuario a uno de estos
- El sistema pasa el cluster de usuarios, y nuestro usuario al algoritmo Slope One y este encuentra las predicciones de valoración de ítems.

[Extensiones]: --

Content Based Filtering: Content Based Filtering es uno de los sistemas de recomendación que vamos a usar. Este ejecuta el algoritmo K-Nearest Neighbors para encontrar similitudes en ítems que le han gustado al usuario.

Actor Principal: Gestor del sistema

[Precondición]: El sistema está activo y el usuario está procesando recomendaciones con este.

[Detonante]: El gestor del sistema le da un nuevo usuario al sistema de recomendaciones.

Escenario principal:

- El gestor del sistema ha inicializado los datos, estos ya se han leído y preprocesado.
- El gestor del sistema ha empezado a ejecutar comandos para el sistema de recomendaciones.
- El gestor del sistema entra en el sistema un usuario para obtener las recomendaciones
- El sistema inicializa el algoritmo K-Nearest Neighbors con los items que queremos encontrar, y este organiza estos ítems de forma “más cercana” a los gustos del usuario.

[Extensiones]: --

1.2.4 Gestión Item

Añadir Item

Actor Principal: Gestor del sistema

[Precondición]: El sistema está activo.

[Detonante]: Lectura y preproceso de datos.

Escenario principal:

- El gestor del sistema ha inicializado el sistema.
- Los datos se están leyendo (concretamente el fichero de ítems)
- Estos ítems se están creando, junto con sus atributos en el momento de lectura de los ficheros.

[Extensiones]: --

1.2.5 Gestión Valoración

Añadir Valoración

Actor Principal: Gestor del sistema

[Precondición]: El sistema está activo.

[Detonante]: Se leen y procesan datos (concretamente de los archivos ratings.*).

Escenario principal 1:

- El gestor del sistema inicializa el sistema y este empieza a leer los datos y a preprocesar estos.
- Como no se ha visto ningún usuario previamente, todos los usuarios serán nuevos en el sistema y por lo tanto este deberá crear valoraciones nuevas para todos las valoraciones de cualquier usuario que leamos.

Escenario principal 2:

- El gestor del sistema ha inicializado los datos, estos ya se han leído y preprocesado.
- El gestor del sistema ha empezado a ejecutar comandos para el sistema de recomendaciones.
- Si algún usuario no se ha visto antes, el usuario entrará sus valoraciones y el sistema creará valoraciones nuevas para estas.

[Extensiones]: --

Eliminar Valoración

Actor Principal: Gestor del sistema

[Precondición]: El sistema está activo.

[Detonante]: Se leen y procesan datos (concretamente de los archivos ratings.*).

Escenario principal 1:

- El gestor del sistema ha inicializado los datos, estos ya se han leído y preprocesado.
- El gestor del sistema ha empezado a ejecutar comandos para el sistema de recomendaciones.
- El usuario tendrá una serie de valoraciones de ítems y querrá eliminar una de estas valoraciones existentes.

[Extensiones]: --

Modificar Valoración

Actor Principal: Gestor del sistema

[Precondición]: El sistema está activo.

[Detonante]: Se leen y procesan datos (concretamente de los archivos ratings.*).

Escenario principal 1:

- El gestor del sistema ha inicializado los datos, estos ya se han leído y preprocesado.
- El gestor del sistema ha empezado a ejecutar comandos para el sistema de recomendaciones.
- El usuario tendrá una serie de valoraciones de ítems y querrá modificar una de estas valoraciones existentes.

[Extensiones]: --

1.2.6 Evaluar Recomendaciones

Actor Principal: Gestor del sistema

[Precondición]: El gestor del sistema ha pedido la evaluación de ítems de un usuario, y ha acabado de solicitar ítems para un usuario.

[Detonante]: Cambiar de usuario/acabar con las recomendaciones de un usuario.

Escenario principal:

- El gestor del sistema ha inicializado los datos, estos ya se han leído y preprocesado.
- El gestor del sistema ha empezado a ejecutar comandos para el sistema de recomendaciones.

- Una vez acaba la lista de ítems que quiere ordenar la lista de mejor predicción a peor, y después el sistema evaluará esta recomendación.

[Extensiones]: --

1.2.7 Salir

Actor Principal: Gestor del sistema

[Precondición]: El sistema está activo.

[Detonante]: El usuario quiere salir del sistema y para su ejecución.

Escenario principal:

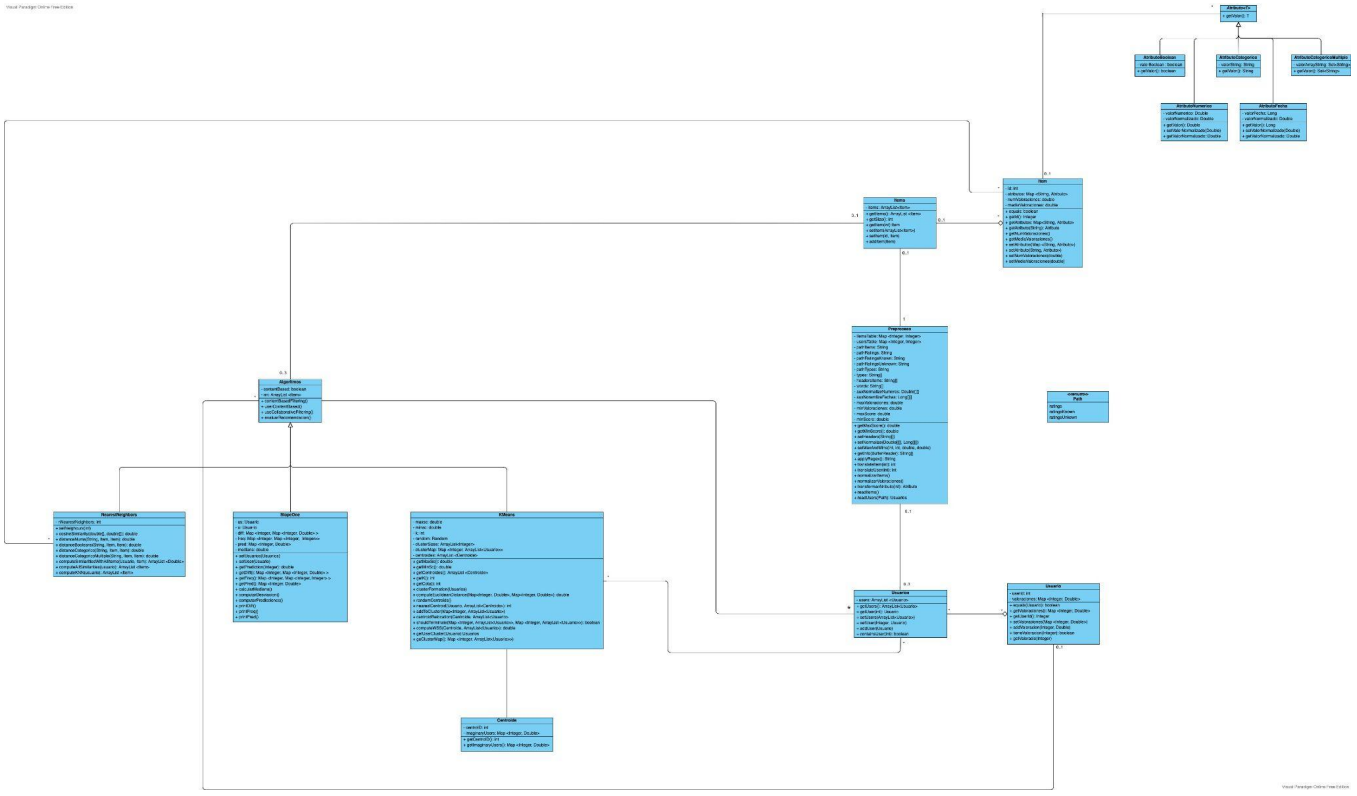
- El gestor del sistema corre el sistema, ejecuta los queries que necesita y cierra el programa al acabar de usarlo.

[Extensiones]:

- Si el fichero de datos que queremos leer no se encuentra, el sistema no se ejecutará.

2 MODELO CONCEPTUAL DE DATOS

2.1 DIAGRAMA



2.2 DESCRIPCIÓN DE CLASES

2.2.1 Algoritmos

Descripción: Clase de control de algoritmos, usa esta clase para controlar el uso de los diferentes algoritmos del sistema de recomendaciones.

Cardinalidad:

Usuarios: La clase algoritmos contiene de cero a un conjuntos de usuarios

Usuario: La clase algoritmos contiene de cero a un usuario (sin contar esos que están en el conjunto - en ese caso de cero a muchos usuarios).

Ítems: La clase de algoritmos contiene una instancia de la clase de items.

[Atributo] Usuarios us

Tipo: protected

Breve descripción: Conjunto de usuarios que vamos a usar para inicializar nuestro algoritmo de collaborative filtering.

Valor por defecto: Clase de usuarios vacía (no hay usuarios)

Estático: No.

[Atributo] Usuario u

Tipo: protected

Breve descripción: Usuario desde el cual vamos a hacer las predicciones de los ítems.

Valor por defecto: Clase de usuario vacía (el usuario está vacío - no tiene id)

Estático: No.

[Atributo] Map <Integer, Double> predictions

Tipo: protected

Breve descripción: Mapa de predicciones de los algoritmos

Valor por defecto: mapa vacío

Estático: No.

[Atributo] Items it

Tipo: protected

Breve descripción: Ítems que queremos usar para las predicciones

Valor por defecto: items vacío

Estático: No.

[***Método***] ***getPrediction()***

Breve descripción: Coger la predicción de un ítem a partir de su id para el algoritmo de collaborative filtering.

Parámetros:

- *Integer idItem*: el id del item para el cual queremos una predicción

Resultados: Se ha devuelto la predicción hecha usando collaborative filtering.

2.2.2 Recomendación

Descripción: Clase de gestión de las llamadas a los algoritmos y las recomendaciones que esta hace hacia el usuario.

Cardinalidad:

Algoritmos: Recomendación hará llamadas a los algoritmos para conseguir estas llamadas
Usuarios: La clase algoritmos contiene de cero a un conjuntos de usuarios
Usuario: La clase algoritmos contiene de cero a un usuario (sin contar esos que están en el conjunto - en ese caso de cero a muchos usuarios).
Ítems: La clase de algoritmos contiene una instancia de la clase de items.

[Atributo] Usuarios us

Tipo: protected
Breve descripción: Conjunto de usuarios que vamos a usar para inicializar nuestro algoritmo de collaborative filtering.
Valor por defecto: Clase de usuarios vacía (no hay usuarios)
Estático: No.

[Atributo] Usuario u

Tipo: protected
Breve descripción: Usuario desde el cual vamos a hacer las predicciones de los ítems.
Valor por defecto: Clase de usuario vacía (el usuario está vacío - no tiene id)
Estático: No.

[Atributo] Map <Integer, Double> predicciones

Tipo: protected
Breve descripción: Mapa de predicciones de los algoritmos
Valor por defecto: mapa vacío
Estático: No.

[Atributo] Items it

Tipo: protected
Breve descripción: Ítems que queremos usar para las predicciones
Valor por defecto: items vacío
Estático: No.

[Método] void processarItems()

Breve descripción: Procesado de los ítems para que estén acorde con lo que necesitamos en las predicciones
Parametros: N/A
Resultados: El conjunto de ítems que van a ser valorados están en la variable global it.

[Método] void collaborativeFiltering()

Breve descripción: Llamada a creación de recomendaciones usando los métodos de collaborative filtering (K-Means + Slope One).
Parametros: N/A
Resultados: Se ha llamado a las subclases K-Means y SlopeOne para crear las recomendaciones para un usuario u.

[Método] contentBasedFiltering()

Breve descripción: Llamada a creación de recomendaciones usando los métodos de content based filtering.

Parámetros: N/A

Resultados: Se ha llamado a la subclase NearestNeighbors para consultar los k ítems que le gustaran más al usuario.

[Método] hybrid()

Breve descripción: Llamada a la creación de recomendaciones utilizando un método híbrido (content based and collaborative).

Parámetros: N/A

Resultados: Se han ejecutado los algoritmos de content based y collaborative filtering para sacar unas recomendaciones híbridas

[Método] evaluarIDCG()

Breve descripción: Computa el ideal discounted cumulative value para una recomendación determinada

Parámetros:

- *Map<Integer, Integer> lt:* en formato Map<itemID, posiciónI> donde posiciónI es la posición que ocupa en la permutación ideal, cada elemento en lr

Resultados: devuelve el ideal discounted cumulative gain

[Método] evaluarRecomendaciones()

Breve descripción: Computa el Discounted cumulative value para una recomendación determinada.

Parámetros:

- *Map<Integer, Integer> lt:* donde el valor es la posición que ocupa en la permutación ideal cada ítem en lr, y la llave es el identificador del ítem.
- *ArrayList<Integer> lr:* contiene, ordenadas de mayor a menor relevancia, los identificadores de los items devueltos por el algoritmo recomendador.

Resultados: devuelve el Discounted Cumulative Gain para la recomendación hecha al usuario.

2.2.3 *[CLASE ABSTRACTA]* Atributo <T>

Descripción: La clase abstracta Atributo crea una clase base para inicializar los tipos de los atributos del ítem.

Cardinalidad:

- Ítem: un ítem tendrá cero o más atributos, de los cuales el valor estará implementado la clase atributo, instanciando la subclase que sea del mismo tipo que la clase.

[Método] <T> getValor()

Breve descripción: Getter del atributo con tipo genérico

Parámetros: N/A.

Resultados: Se ha devuelto la variable global de la subclase instanciada.

2.2.4 AtributoBoolean

Descripción: La clase AtributoBoolean es herencia de la clase atributo con tipo Booleano.

[Atributo] boolean AtributoBoolean

Tipo: privado
Breve descripción: El valor del atributo en formato boolean
Valor por defecto: No hay valor por defecto.
Estático: No.

[Método] getValor ()

Breve descripción: Getter del atributo en formato booleano.
Parámetros: N/A.
Resultados: Se ha devuelto el booleano de la clase (atributo AtributoBoolean).

2.2.5 AtributoCategorico

Descripción: La clase AtributoCategorico es herencia de la clase atributo con tipo String.

[Atributo] String valorString

Tipo: privado
Breve descripción: El valor del atributo en formato string
Valor por defecto: No hay valor por defecto.
Estático: No.

[Método] getValor()

Breve descripción: Getter del atributo en formato string.
Parámetros: N/A.
Resultados: Devuelve el string que tiene la clase como variable global.

2.2.6 AtributoCategoricoMultiple

Descripción: La clase AtributoCategoricoMultiple es herencia de la clase atributo con tipo Set<String>.

[Atributo] Set<String> valorArrayString

Tipo: privado
Breve descripción: El atributo en formato de set de strings (evita repeticiones).
Valor por defecto: No hay valor por defecto.
Estático: No.

[Método] getValor()

Breve descripción: Getter del atributo en formato de set de strings.
Parámetros: N/A.
Resultados: Se ha devuelto el set de strings de la clase.

2.2.7 AtributoFecha

Descripción: La clase AtributoFecha es herencia de la clase atributo con tipo long.

[Atributo] Long valorFecha

Tipo: privado
Breve descripción: El valor de la fecha en formato de long (número de segundos desde el 1970)

Valor por defecto: No hay valor por defecto.
Estático: No.

[Atributo] double valorNormalizado

Tipo: privado
Breve descripción: El valor de la fecha normalizado para el uso de Content Based Filtering (de 0 a 1).
Valor por defecto: No hay valor por defecto.
Estático: No.

[Método] getValor()

Breve descripción: Getter de la fecha, valor sin normalizar.
Parámetros: N/A.
Resultados: Se ha devuelto el valor en segundos de la fecha del atributo.

[Método] setValorNormalizado()

Breve descripción: Setter del valor normalizado de la fecha.
Parámetros:

- *Double valorNormalizado:* valor de la fecha normalizado.

Resultados: La variable global valorNormalizado de la fecha.

[Método] getValorNormalizado()

Breve descripción: Getter del valor normalizado de la fecha (0 a 1).
Parámetros: N/A.
Resultados: Se ha devuelto el valor del atributo valorNormalizado.

2.2.8 AtributoNumerico

Descripción: La clase AtributoNumerico es herencia de la clase atributo con tipo double.

[Atributo] Long valorFecha

Tipo: privado
Breve descripción: El valor del atributo en formato Double.
Valor por defecto: No hay valor por defecto.
Estático: No.

[Atributo] double valorNormalizado

Tipo: privado
Breve descripción: El valor del double normalizado para el uso de Content Based Filtering (de 0 a 1).
Valor por defecto: No hay valor por defecto.
Estático: No.

[Método] getValor()

Breve descripción: Getter del atributo en formato double, sin normalizar.
Parámetros: N/A.
Resultados: Se ha devuelto el valor en segundos de la númerico del atributo.

[Método] setValorNormalizado()

Breve descripción: Setter del valor normalizado del atributo númerico.
Parámetros:

- *Double valorNormalizado:* valor de la fecha normalizado.

Resultados: La variable global valorNormalizado del double.

[Método] getValorNormalizado()

Breve descripción: Getter del valor normalizado del atributo (0 a 1).
Parámetros: N/A.
Resultados: Se ha devuelto el valor del atributo valorNormalizado.

2.2.9 Centroide

Descripción: La clase Centroide contiene la implementación de un centroide, que, similarmente a un usuario contiene un identificador único y una serie de valoraciones a cada uno de los ítems presentes en el sistema.

Cardinalidad:

- *K-Means:* La clase de K-Means crea de cero (solo en el caso de el conjunto vacío de usuarios) a muchos (el valor k de la clase) centroides.

[Atributo] int centroID

Tipo: privado
Breve descripción: Identificador del Centroide
Valor por defecto: No hay valor por defecto.
Estático: No.

[Atributo] Map<Integer, Double> imaginaryUsers

Tipo: privado
Breve descripción: mapa de valoraciones por ítem del centroide
Valor por defecto: No hay valor por defecto.
Estático: No.

[Método] getCentroID()

Breve descripción: Getter del identificador del centroide.
Parámetros: N/A.
Resultados: Devuelve el identificador de este centroide.

[Método] setCentroID()

Breve descripción: Setter del identificador del centroide.
Parámetros:

- *int centroidID:* identificador del centroide de la clase

Resultados: la variable global que identifica la clase (*centroidID)* pasa a valer el parametro.

[Método] getImaginaryUsers()

Breve descripción: Getter del mapa de valoraciones del centroide.
Parámetros: N/A.
Resultados: Devuelve el mapa de valoraciones a los ítems de este centroide.

2.2.10 Item

Descripción: Implementación de los Ítems que leemos de los data sets.

Cardinalidad:

- *Atributo*: un ítem tiene de uno a muchos atributos que lo caracterizan.
- *Ítems*: la clase de items toma su base en un conjunto de ítems por lo tanto tendrá de cero a muchas instancias de la clase item. Como la clase items es estática, un ítem solo estará en una

[Atributo] int id

Tipo: privado
Breve descripción: El id del item instanciado en la clase, creado por el preproceso de datos.
Valor por defecto: No tiene valor por defecto.
Estático: No.

[Atributo] int originalId

Tipo: privado
Breve descripción: El id original del ítem, instanciado en los archivos.
Valor por defecto: No tiene valor por defecto.
Estático: No.

[Atributo] Map <String, Atributo> Atributos

Tipo: privado
Breve descripción: Variable atributos contiene todos los atributos de un ítem.
Valor por defecto: No tiene valor por defecto.
Estático: No.

[Atributo] double numValoraciones

Tipo: privado
Breve descripción: El número de valoraciones que se han hecho de este ítem.
Valor por defecto: cero
Estático: No.

[Atributo] double mediaValoraciones

Tipo: privado
Breve descripción: La media de las valoraciones que se han hecho de cada ítem.
Valor por defecto: cero
Estático: No.

[Método] equals()

Breve descripción: Comprueba la igualdad entre los dos objetos de la clase ítem.
Parámetros:

- *Objeto o*: otra instancia de la clase items que queremos comprobar que sea igual que el ítem que llama la función.

Resultados: Se ha devuelto true si los dos ítems tienen el mismo id y si no false.

[Método] getId()

Breve descripción: Getter del id del item.

Parámetros: N/A.
Resultados: Se ha devuelto el id del objeto de ítem.

[Método] getOriginalId()

Breve descripción: Getter del id original del ítem.
Parámetros: N/A.
Resultados: Se ha devuelto el id original del objeto de ítem.

[Método] getAtributos()

Breve descripción: Getter del mapa de atributos.
Parámetros: N/A.
Resultados: Se ha devuelto el mapa de atributos.

[Método] getAtributo()

Breve descripción: Coger el atributo del ítem con header "atr".
Parámetros: N/A.

- *String atr*: nombre del header del atributo al que queremos acceder

Resultados:

[Método] getNumValoraciones()

Breve descripción: Getter de la variable de clase numValoraciones.
Parámetros: N/A.
Resultados: Se ha devuelto el número de valoraciones que tiene el item objeto

[Método] getMediaValoraciones()

Breve descripción: Getter de la variable de clase mediaValoraciones.
Parámetros: N/A.
Resultados: Se ha devuelto la media de valoraciones que tiene el ítem objeto.

[Método] setOriginalId()

Breve descripción: Setter del originalId
Parámetros:

- *int originalId*: id del ítem original creado por los archivos (instanciado al leer datos)

Resultados: El originalId de ítem pasa a ser igual al parámetro que le pasamos al método.

[Método] setAtributos()

Breve descripción: Setter del mapa de atributos
Parámetros:

- *Map<String, Atributo> atributos*: mapa de atributos del ítem

Resultados: La variable global atributos que contiene el conjunto de atributos que ha hecho el usuario es el conjunto de atributos "atributos".

[Método] setAtributo()

Breve descripción: Añadir un atributo al mapa de atributos del usuario.
Parámetros:

- *String str*: el header del atributo que queremos añadir
- *String atr*: valor de ese atributo para añadir al mapa de atributos

Resultados: El mapa de atributos pasa a tener el entry {str, atr}

[Método] setNumValoraciones()

Breve descripción: Cambiar el número de valoraciones del ítem a "numValoraciones".

Parámetros:

- *double numValoraciones:* número de los usuarios que han valorado este ítem

Resultados: La variable local numValoraciones ahora es igual al parámetro de la función "numValoraciones"

[Método] setMediaValoraciones()

Breve descripción: Cambiar el número de valoraciones del ítem a "mediaValoraciones".

Parámetros:

- *double mediaValoraciones*: media de las valoraciones que han hecho los usuarios que han valorado este ítem

Resultados: La variable local mediaValoraciones ahora es igual al parámetro de la función "mediaValoraciones"

2.2.11 Items

Descripción: La clase de ítems contiene la implementación de un conjunto de ítems.

Cardinalidad:

- Item: la clase de items toma su base en un conjunto de ítems por lo tanto tendrá de cero a muchas instancias de la clase item.
- La clase de items es estática, dado esto todas las clases que le hagan referencia lo haran con cardinalidad uno. Las siguientes clases tienen esta referencia de cardinalidad:
    - Slope One
    - Nearest Neighbors
    - K Means
    - Parser

[Atributo] ArrayList<Item> items

Tipo: privado
Breve descripción: Conjunto de objetos de la clase de ítem que componen el conjunto de ítems.
Valor por defecto: Clase de usuarios vacía (no hay usuarios)
Estático: Si.

[Método] ArrayList<Item> getItems()

Breve descripción: Getter del ArrayList de ítems.
Parámetros: N/A.
Resultados: Se ha devuelto el ArrayList de ítems.

[Método] Item getItem()

Breve descripción: Getter del tamaño del ArrayList de ítems.
Parámetros:

- *int id*: id del item del que queremos conseguir el objeto (esté id tiene que ser el id generado por el preproceso).

Resultados: Devuelve el objeto de ítem con id "id"

[Método] int getSize()

Breve descripción: Getter del tamaño del ArrayList de ítems.
Parámetros: N/A.
Resultados: Se ha devuelto el tamaño del ArrayList de ítems.

[Método] void setItems()

Breve descripción: Setter del ArrayList de items
Parámetros:

- *ArrayList<Item> items*: ArrayList de los ítems del conjunto

Resultados: El atributo items de la instancia de la clase es una copia del parámetro items.

[Método] void setItem()

Breve descripción: Cambiar el objeto de un ítem en el ArrayList
Parámetros:

- *int id*: id de la posición del ArrayList que queremos cambiar, equivalente al id preprocesado del ítem.

- *Item item*: item al que queremos cambiar en el ArrayList

Resultados: El atributo items de la instancia de la clase es una copia del parámetro items.

[Método] void addItem()

Breve descripción: Cambiar el objeto de un ítem en el ArrayList

Parámetros:

- *Item item*: item al que queremos añadir en el ArrayList

Resultados: El ArrayList de ítems tiene un nuevo entry del item item.

[Método] boolean containsItem()

Breve descripción: Cambiar el objeto de un ítem en el ArrayList

Parámetros:

- *int id*: id de la posición del ArrayList que queremos ver si tiene item, equivalente al id preprocesado del ítem.

Resultados: El ArrayList de ítems tiene un nuevo entry del item item.

2.2.12 KMeans

Descripción: Implementación del algoritmo K-Means usado para clusterizar los usuarios en el collaborative filtering.

Cardinalidad:

- Es una clase estática y en herencia de la clase Algoritmos, que será la única que controle a esta clase.

[Atributo] double maxsc

Tipo: privado

Breve descripción: Valor máximo permitido para cada valoración

Valor por defecto: valor obtenido del preproceso (Parser.getMaxScore())

Estático: Si.

[Atributo] double minsc

Tipo: privado

Breve descripción: Valor mínimo permitido para cada valoración

Valor por defecto: valor obtenido del preproceso (Parser.getMinScore())

Estático: Si.

[Atributo] int k

Tipo: privado

Breve descripción: Número de clusters que se forman para el algoritmo

Valor por defecto: 10

Estático: Si.

[Atributo] int cota

Tipo: privado
Breve descripción: Máximo número de iteraciones permitidas
Valor por defecto: 30
Estático: Si.

[Atributo] Random random

Tipo: privado
Breve descripción: Objeto random para la creación de valoraciones aleatorias de items para cada centroides
Valor por defecto: random inicializado (new Random())
Estático: Si.

[Atributo] Map<Integer, ArrayList<Usuario>> clusterMap

Tipo: privado
Breve descripción: Map donde almacenamos cómo quedan los k clusters de usuarios formados
Valor por defecto: Map vacío con tamaño k (new HashMap<>(k))
Estático: Si.

[Atributo] ArrayList<Centroide> centroides

Tipo: privado
Breve descripción: ArrayList donde almacenamos los k centroides creados en el algoritmo
Valor por defecto: ArrayList vacío (new ArrayList<>())
Estático: Si.

[Método] double getMaxSc()

Breve descripción: getter de la variable maxsc
Parámetros: N/A.
Resultados: se ha devuelto el valor de maxsc

[Método] double getMinSc()

Breve descripción: getter de la variable minsc
Parámetros: N/A.
Resultados: se ha devuelto el valor de minsc

[Método] ArrayList<Centroide> getCentroides()

Breve descripción: getter de la ArrayList de los k centroides creados
Parámetros: N/A.
Resultados: se ha devuelto el Arraylist de centroides

[Método] int getK()

Breve descripción: getter del valor k usado para el algoritmo
Parámetros: N/A.
Resultados: se ha devuelto el valor de la k (número de clusters)

[Método] int getCota()

Breve descripción: getter de la cota usada para el algoritmo
Parámetros: N/A.
Resultados: se ha devuelto el valor de la cota (número máximo de iteraciones)

[Método] Map<Integer, ArrayList<Usuario>> getClusterMap()

Breve descripción: getter del map clusterMap donde se almacenan los clusters
Parámetros: N/A.
Resultados: se ha devuelto el mapa con con los k clusters y sus respectivos centroides identificados por un identificador único

[Método] Usuarios getUsersCluster()

Breve descripción: Encuentra a qué cluster pertenece un nuevo usuario u, para el cual estamos tratando de encontrar los usuarios más parecidos
Parámetros:

- *Usuario u:* nuevo usuario

Resultados: se ha encontrado cuál es el centroide más cercano al usuario u y por lo tanto a qué cluster pertenece.

[Método] void setCentroides()

Breve descripción: setter del atributo centroides
Parámetros:

- *Centroide centroides:* ArrayList de Centroides. Su identificador indica también su posición en la Array

Resultados: se ha asignado al atributo centroides la ArrayList pasada por parámetro

[Método] void setClusterMap()

Breve descripción: setter del atributo clusterMap
Parámetros:

- Map<Integer, ArrayList<Usuario>> clusterMap: mapa donde la llave es el identificador del centroide y el ArrayList de usuarios mapeados es el conjunto de usuarios que forman un cluster alrededor de dicho centtroide

Resultados: el clusterMap variable global de la clase pasa a ser clusterMpa pasado por parámetro

[Método] void setCota()

Breve descripción: setter del atributo cota que determina el número total de iteraciones
Parámetros:

- *int cota:* nuevo valor que pasa a tener el atributo cota. Determinará el número máximo de iteraciones permitidas

Resultados: se le ha asignado al atributo cota el valor pasado por parámetro

[Método] void setK()

Breve descripción: setter del atributo k, que determina el número de clusters
Parámetros:

- *int k:* nuevo valor que pasa a tener el atributo k. Determinará el número de clusters que se formen

Resultados: se le ha asignado al atributo k el valor pasado por parámetro

[Método] void setDefaults()

Breve descripción: devuelve a su estado inicial todos los atributos de la clase
Parámetros: N/A.
Resultados: los atributos k, centroides, cota y clusterMap pasan a tener sus valores iniciales.

[Método] void clusterFormation()

Breve descripción: Función principal del algoritmo kMeans. Después de como máximo cota iteraciones, ha separado los usuarios de entrada en k clusters
Parámetros:

- *Usuarios valoraciones:* contiene una lista de valoraciones no vacía, con las respectivas valoraciones de cada usuario a un conjunto de ítems.

Resultados: Se han formado k clusters formados por subconjuntos disjuntos de valoraciones.

[Método] double computeEuclideanDistance()

Breve descripción: Calcula la distancia euclídea entre un centroide y un usuario, a partir del conjunto de valoraciones que cada uno de estos ha dado a los distintos ítems.
Parámetros:

- *Map<Integer, Double> vals:* contiene las valoraciones que ha dado un usuario a una serie de ítems. La llave contiene el identificador del ítem, y el valor es la valoración dada.
- *Integer centroid:* contiene las valoraciones arbitrarias de un centroide a cada objeto del conjunto de ítems. La llave contiene el identificador del ítem y el valor es una valoración aleatoria entre maxsc y minsc.

Resultados: Se ha computado la distancia entre el centroide y el usuario al cual pertenezcan dichas valoraciones.

[Método] void randomCentroids()

Breve descripción: Forma 2k centroides para el algoritmo. Un centroide es el resultado de asignar, para cada uno de los items disponibles, una valoración aleatoria entre minsc y maxsc
Parámetros: N/A.
Resultados: centroides contiene k objetos de tipo Centroide, cada uno con sus respectivas valoraciones arbitrarias para todos los items.

[Método] void furthestCentroids()

Breve descripción: elimina los clusters más parecidos entre ellos, dejando sólo los k más alejados
Parámetros: N/A.
Resultados: los k centroides más cercanos entre sí han sido eliminados

[Método] int nearestCentroid()

Breve descripción: Determina cuál es el centroide más cercano a un usuario, calculando su distancia euclidea a cada centroide en centroides
Parámetros:

- *Map <Integer, Double> u:* Usuario cuyo centroide a distancia mínima queremos conocer.

Resultados: pos contiene la posición del centroide que se encuentra más cercano al usuario u.

[Método] void addToCluster()

Breve descripción: Añade el usuario f al cluster perteneciente al centroide c

Parámetros:

- *Usuario f:* usuario que deseamos añadir a un cluster
- *Integer c:* Centroide que del cluster al que queremos añadir al usuario f.

Resultados: el usuario f forma parte ahora del cluster c

[Método] void centroidRelocation()

Breve descripción: Recalcula las valoraciones de cada item para el centroide c, de manera que éste quede ahora en el centro de su cluster

Parámetros:

- *Integer id:* identificador del centroide al cual pertenece el cluster c
- *Map<Integer, Double> c:* valoraciones del centroide a todos los items
- *ArrayList<Usuario> cluster:* conjunto de usuarios del Centroide c.

Resultados: si el conjunto era vacía (centroide sin usuarios) nada ha cambiado, y si no lo era, las valoraciones del centroide han sido modificadas para posicionarlo en el centro del cluster de usuarios.

[Método] boolean shouldTerminate()

Breve descripción: Determina si el algoritmo de kMeans debería o no terminar antes de tiempo. Esto se debe dar solo en el caso de que nada haya cambiado en el estado de los clusters entre una iteración y la siguiente

Parámetros:

- *Map<Integer, ArrayList<Usuario>> b:* Mapa que representa el estado de los clusters en la última iteración de K-Means

Resultados: si a o b estaban vacíos o bien las valores varían para una o varias llaves determinadas, devuelve falso. Si a y b son idénticos devuelve cierto.

[Método] Double computeAverageWSS()

Breve descripción: Calcula el wss para todos los clusters de clusterMap y devuelve la media de estos

Parámetros: N/A.

Resultados: se ha calculado la distancia media usuario-centroide para cada cluster, y devuelto la media

2.2.12 NearestNeighbors

Descripción: Implementación del algoritmo nearest neighbors, usando content based filtering.

Cardinalidad:

- Es una clase estática y en herencia de la clase Algoritmos, que será la única que controle a esta clase.

[Atributo] int k

Tipo: privado

Breve descripción: Esta variable nos indica el número de vecinos más cercanos que tenemos que devolver.

Valor por defecto: 0

Estático: Si.

[Método] setNeighbours()

Breve descripción: Setter de la variable k.

Parámetros:

- *int k*: indica el número de vecinos mas cercanos que tenemos que devolver.

Resultados: Se ha cambiado el valor de la variable k al parámetro k.

[Método] distanceNums()

Breve descripción: Calcula la distancia entre dos atributos numéricos.

Parámetros:

- *String att:* Nombre del atributo numérico.
- *Item a:* Item del usuario.
- *Item b:* Item del cual se quiere saber la distancia.

Resultados: Se ha retornado la distancia entre el atributo att de los items a y b.

[Método] distanceBools()

Breve descripción: Calcula la distancia entre dos atributos booleanos.

Parámetros:

- *String att:* Nombre del atributo booleano.
- *Item a:* Item del usuario.
- *Item b:* Item del cual se quiere saber la distancia.

Resultados: Se ha retornado la distancia entre el atributo att de los items a y b.

[Método] distanceCategorico()

Breve descripción: Calcula la distancia entre dos atributos categóricos.

Parámetros:

- *String att:* Nombre del atributo categórico.
- *Item a:* Item del usuario.
- *Item b:* Item del cual se quiere saber la distancia.

Resultados: Se ha retornado la distancia entre el atributo att de los items a y b.

[Método] distanceCategoricoMultiple()

Breve descripción: Calcula la distancia entre dos atributos categóricos múltiples.

Parámetros:

- *String att:* Nombre del atributo categórico múltiple.
- *Item a:* Item del usuario.
- *Item b:* Item del cual se quiere saber la distancia.

Resultados: Se ha retornado la distancia entre el atributo att de los items a y b.

[Método] prediccionValoracionItem()

Breve descripción: Calcula las predicciones de valoraciones para un item que el usuario no ha valorado

Parámetros:

- *Usuario usuario:* Usuario al cual hay que recomendar items.
- *Item item:* Item del usuario.

Resultados: Se ha devuelto la predicción de la valoración para el ítem dado.

[Método] computeAllPredicciones()

Breve descripción: Calcula las distancias entre cada ítem que el usuario ha valorado y el resto de ítems.

Parámetros:

- *Usuario usuario*: Usuario al cual hay que recomendar items.

Resultados: Se ha devuelto un array con los k ítems más cercanos.

[Método] computeKNN()

Breve descripción: Calcula los k ítems más cercanos a los ítems valorados por el usuario.

Parámetros:

- *Usuario usuario*: Usuario al cual hay que recomendar items.
- *ArrayList<Item> items*: Los items los cuales va a valorar el algoritmo

Resultados: Se ha devuelto un array con los ítems más cercanos a los valorados por el usuario.

2.2.14 Parser

Descripción: Clase utilizada para la lectura y preprocesamiento de los datos.

Cardinalidad:

- El preproceso instancia los ítems del sistema, por lo tanto tiene cardinalidad de 1 con la clase de items.
- El preproceso creará tres sets de usuarios (ratings, known y unkown), por lo tanto tendrá un cardinal de tres con la clase de usuarios.

[Atributo] Map <Integer, Integer> itemsTable

Tipo: private

Breve descripción: Diccionario utilizado para traducir los ids de los ítems

Valor por defecto: Map vacio (new HashMap<>())

Estático: Si.

[Atributo] Map <Integer, Integer> usersTable

Tipo: private

Breve descripción: Diccionario utilizado para traducir los ids de los usuarios

Valor por defecto: Map vacio (new HashMap<>())

Estático: Si.

[Atributo] String pathItems

Tipo: Private.

Breve descripción: Variable utilizada para almacenar el path del archivo items.csv

Valor por defecto: "DATA/csv/queries/items.csv"

Estático: Si.

[Atributo] String pathRatings

Tipo: private

Breve descripción: Variable utilizada para almacenar el path del archivo ratings.db.csv

Valor por defecto: "DATA/csv/queries/ratings.db.csv"

Estático: Si.

[Atributo] String pathRatingsKnown

Tipo: private
Breve descripción: Variable utilizada para almacenar el path del archivo ratings.test.known.csv
Valor por defecto: "DATA/csv/queries/ratings.test.known.csv"
Estático: Si.

[Atributo] String pathRatingsUnkown

Tipo: private
Breve descripción: Variable utilizada para almacenar el path del archivo ratings.test.unknown.csv
Valor por defecto: "DATA/csv/queries/ratings.test.unknown.csv"
Estático: Si.

[Atributo] String pathTypes

Tipo: private
Breve descripción: Variable utilizada para almacenar el path del archivo types.csv
Valor por defecto: "DATA/csv/queries/types.csv"
Estático: Si.

[Atributo] String[] types

Tipo: private
Breve descripción: Array utilizado para almacenar los tipos de cada columna del archivo de ítems
Valor por defecto: No tiene valor por defecto.
Estático: Si.

[Atributo] String[] headerItems

Tipo: private
Breve descripción: Array utilizado para almacenar la cabecera de los ítems
Valor por defecto: No tiene valor por defecto.
Estático: Si.

[Atributo] String[] words

Tipo: private
Breve descripción: Array utilizado para almacenar los datos de cada fila de los archivos
Valor por defecto: No tiene valor por defecto.
Estático: Si.

[Atributo] Double[][] auxNormalizeNumeros

Tipo: private
Breve descripción: Array utilizado para almacenar los máximos y mínimos de cada dato numérico para posteriormente normalizar esos datos.
Valor por defecto: No tiene valor por defecto.
Estático: Si.

[Atributo] Long[][] auxNoramlizeFechas

Tipo: private

Breve descripción: Array utilizado para almacenar los máximos y mínimos de cada dato con formato fecha para posteriormente normalizar esos datos.
Valor por defecto: No tiene valor por defecto.
Estático: Si.

[Atributo] double maxValoraciones

Tipo: private
Breve descripción: Variable utilizada para almacenar el máximo número de valoraciones para posteriormente normalizar las valoraciones
Valor por defecto: 0
Estático: Si.

[Atributo] double minValoraciones

Tipo: private
Breve descripción: Variable utilizada para almacenar el máximo número de valoraciones para posteriormente normalizar las valoraciones.
Valor por defecto: Double.MAX_VALUE;
Estático: Si.

[Atributo] double maxScore

Tipo: private
Breve descripción: Variable utilizada para almacenar el máximo de los valores de las valoraciones para posteriormente normalizar las valoraciones.
Valor por defecto: 0
Estático: Si.

[Atributo] double minScore

Tipo: private
Breve descripción: Variable utilizada para almacenar el mínimo de los valores de las valoraciones para posteriormente normalizar las valoraciones.
Valor por defecto: Double.MAX_VALUE
Estático: Si.

[Método] double getMaxScore()

Breve descripción: Función utilizada en los tests para adquirir la variable maxScore.
Parámetros: N/A.
Resultados: Se ha devuelto maxScore

[Método] double getMinScore()

Breve descripción: Función utilizada en los tests para adquirir la variable minScore.
Parámetros: N/A.
Resultados: Se ha devuelto minScore

[Método] void setHeaders()

Breve descripción: Función utilizada en los tests para establecer los valores que contiene el array
Parámetros:

- *headers*: string de headers leídos en el archivo

Resultados: headersItems contiene los valores de headers

[Método] void setNormalize()

Breve descripción: Función utilizada en los test para establecer los valores que contienen los arrays utilizados para la posterior normalización de los datos

Parámetros:

- *Double[][] normNums:* Array de arrays que contiene los valores normalizados de los atributos numéricos
- *Long[][] normFechas*: Array de arrays que contiene los valores normalizados de los atributos de fecha

Resultados: headersItems contiene los valores de headers

[Método] void setMaxAndMins()

Breve descripción: Función utilizada en los test para establecer los valores máximos y mínimos del número de valoraciones y de sus valores

Parámetros:

- *int mV:* Número mínimo de valoraciones
- *double MV:* Número máximo de valoraciones
- *int mS:* Valor mínimo de valoraciones
- *double MS:* Valor máximo de valoraciones

Resultados: Las variables globales de máximos y mínimos de la clase han sido inicializadas con los valores pasados por parámetro.

[Método] String[] getInfo()

Breve descripción: Función auxiliar utilizada para almacenar la primera línea de las tablas, es decir, las cabeceras de un archivo csv.

Parámetros:

- *BufferReader csvReader*: lector de archivos csv

Resultados: csvReader apunta a la segunda línea del archivo que tratamos de leer

[Método] String applyRegex()

Breve descripción: Función auxiliar utilizada para dividir cada línea. Información obtenida de la página web:

Parámetros: N/A.

Resultados: Se ha devuelto un string de regex.

[Método] int translateItem()

Breve descripción: Función utilizada para obtener el identificador original del ítem

Parámetros:

- *int originalId*: id original del ítem

Resultados: Se ha devuelto el identificador del ítem con id = originalId en el archivo original

[Método] int translateUser()

Breve descripción: Función utilizada para obtener el identificador original del usuario

Parámetros:

- *int originalId:* id original del usuario

Resultados: Se ha devuelto el dentificador del usuario con userId = originalId en el archivo original

[Método] void normalizarItems()

Breve descripción: Función utilizada para normalizar los atributos númericos y de fecha de los ítems
Parámetros: N/A.
Resultados: Para cada ítem se han normalizado sus atributos numéricos y fechas

[Método] void normalizarValoraciones()

Breve descripción: Función utilizada para normalizar las valoraciones de los usuarios
Parámetros: N/A.
Resultados: Para cada ítem se han normalizado su valoración y el número de veces que ha sido valorado

[Método] Atributo transformarAtributo()

Breve descripción: Función utilizada para transformar el Atributo en el tipo que realmente sea
Parámetros:

- *int i:* Índice para el array "types" para obtener el tipo del elemento

Resultados: Se ha devuelto un atributo del tipo que sea requerido

[Método] void readItems()

Breve descripción: Función utilizada para leer los ítems de un archivo csv
Parámetros: N/A.
Resultados: Se ha leído el archivo y se han guardado sus datos sobre los ítems en la estructura de datos correspondiente

[Método] Usuarios readUsers()

Breve descripción: Función utilizada para leer los usuarios y sus valoraciones
Parámetros:

- *Path op*: Enum referenciando el path del archivo que queremos leer

Resultados: Se ha leído el archivo y se han guardado sus datos sobre los usuarios en la estructura de datos correspondiente

[Enum] Path

ratings: tiene el path del archivo csv de ratings
ratingsKnown: tiene el path del archivo csv de ratings known
ratingsUnkown: tiene el path del archivo csv de ratings unknown

2.2.15 SlopeOne

Descripción: Implementación del algoritmo slope one, usado en Collaborative Filtering después de KMens.

Cardinalidad: Es una clase estática y en herencia de la clase Algoritmos, que será la unica que controle a esta clase.

[Atributo] Usuarios us

Tipo: privado
Breve descripción: Usuarios que pertenecen al cluster más cercano a nuestro usuario
Valor por defecto: Clase usuarios vacía (new Usuarios())
Estático: Si.

[Atributo] Usuario u

Tipo: privado
Breve descripción: Usuario del cual vamos a predecir las valoraciones
Valor por defecto: Clase usuarios vacía (new Usuario())
Estático: Si.

[Atributo] Map <Integer, Map <Integer, Double> > diff

Tipo: privado
Breve descripción: Mapa de desviaciones para todos los usuarios formato Map <ItemId, Map <ItemId, Score>>
Valor por defecto: Mapa vacío (new HashMap<>())
Estático: Si.

[Atributo] Map <Integer, Map <Integer, Integer> > freq

Tipo: privado
Breve descripción: Cuantos usuarios han valorado los ítems
Valor por defecto: Mapa vacío (new HashMap<>())
Estático: Si.

[Atributo] Map <Integer, Double> pred

Tipo: privado
Breve descripción: Predicciones de la puntuación que le daría un usuario a un item
Valor por defecto: Mapa vacío (new HashMap<>())
Estático: Si.

[Atributo] double mediana

Tipo: privado
Breve descripción: Mediana de predicción de las valoraciones del usuario u
Valor por defecto: 0.0
Estático: Si.

[Método] setUsuarios()

Breve descripción: Setter de cluster de usuarios
Parámetros:

- *Usuarios users*: conjunto de usuarios desde el cual vamos a predecir los ratings del usuario.

Resultados: El atributo us que contiene el conjunto de usuarios del cual se basa el algoritmo es el conjunto de usuarios users.

[Método] setUser()

Breve descripción: Setter del usuario desde el cual ejecutaremos el Slope One
Parámetros:

- *Usuario use*r: usuario del cual queremos sacar las predicciones.

Resultados: El usuario de la clase Slope One es user y la variable mediana se ha inicializado con la mediana de las valoraciones del usuario.

[Método] getMediana()

Breve descripción: Getter de la mediana calculada del usuario.
Parámetros: N/A.
Resultados: Se ha devuelto la mediana.

[Método] getPredictionOnItem()

Breve descripción: Coger la predicción de un Item a partir de su id
Parámetros:

- *Integer item*: el id del item para el cual queremos una

Resultados: Si el usuario ya ha valorado el item se devolverá esta valoración, si no se devolver -1 si no se puede conseguir valoración de este item y la predicción si se puede.

[Método] getDiff()

Breve descripción: Getter de el mapa de desviaciones
Parámetros: N/A.
Resultados: Se ha devuelto el mapa de desviaciones.

[Método] getFreq()

Breve descripción: Getter de el mapa de frecuencias
Parámetros: N/A.
Resultados: Se ha devuelto el mapa de frecuencias.

[Método] getPred()

Breve descripción: Getter del mapa de predicciones
Parámetros: N/A.
Resultados: Se ha devuelto el mapa de predicciones.

[Método] calcularMediana()

Breve descripción: Calcular mediana del usuario, se calcula automáticamente cuando se introduce un usuario
Parámetros: N/A.
Resultados: Se ha inicializado el valor de la variable de clase mediana a l

[Método] computarDesviacion()

Breve descripción: Crear el mapa de desviaciones. Cogemos la diferencia entre cada usuario que tiene valoración para dos items i, j. Añadimos la desviación al mapa de diff y añadimos a la frecuencia de usuarios que han valorado i y j.
Parámetros: N/A.
Resultados: Se han calculado las desviaciones entre todos los items que un mismo usuario haya valorado.

[Método] computarPredicciones()

Breve descripción: Calcular las predicciones para el usuario u. A partir del mapa de desviación y frecuencia podemos calcular las valoraciones que hará el usuario. Cogeremos las diferencias de todo j e i donde i ha sido valorado por nuestro usuario, y la añadimos a la mediana de valoraciones de nuestro usuario la podemos predecir el valor que le dara nuestro usuario al ítem j.
Parámetros: N/A.

Resultados: Se ha inicializado el mapa de predicciones de manera que este contiene todas las predicciones que podemos hacer de un usuario base.

[Método] printDiff()

Breve descripción: Imprimir mapa de desviaciones
Parámetros: N/A.
Resultados:

[Método] printFreq()

Breve descripción: Imprimir mapa de frecuencias
Parámetros: N/A.
Resultados: El mapa de frecuencias ha sido imprimido por el output standard

[Método] printPred()

Breve descripción: Imprimir mapa de predicciones
Parámetros: N/A.
Resultados: El mapa de predicciones ha sido imprimido por el output estandard

2.2.16 Usuario

Descripción: La clase de usuario contiene la implementación de un usuario con su id y sus valoraciones.

Cardinalidad:

- Un usuario siempre estará como mínimo en una instancia de usuarios como máximo en muchas.

[Atributo] int userId

Tipo: privado
Breve descripción: El id del usuario instanciado en la clase.
Valor por defecto: No tiene valor por defecto.
Estático: No.

[Atributo] int origianlUserId

Tipo: privado
Breve descripción: El id del usuario instanciado en los archivos.
Valor por defecto: No tiene valor por defecto.
Estático: No.

[Atributo] Map <Integer, Double> valoraciones

Tipo: privado
Breve descripción: Variable valoraciones contiene todas las valoraciones que ha hecho el item
Valor por defecto: No tiene valor por defecto.
Estático: No.

[Método] boolean equals()

Breve descripción: Comprueba la igualdad entre los dos objetos de la clase Usuario.
Parámetros:

- Object o: Objeto de usuario con el que queremos comprobar igualdad

Resultados: Se ha devuelto true si los dos usuarios tienen el mismo id y si no false.

[Método] Map <Integer, Double> getValoraciones()

Breve descripción: Getter del mapa de valoraciones
Parámetros: N/A.
Resultados: Se ha devuelto el mapa de valoraciones.

[Método] Integer getUserId()

Breve descripción: Getter del id del usuario.
Parámetros: N/A.
Resultados: Se ha devuelto el id del objeto de usuario.

[Método] Integer getOriginalUserId()

Breve descripción: Getter del id original del usuario.
Parámetros: N/A.
Resultados: Se ha devuelto el id original del objeto de usuario.

[Método] void setValoraciones()

Breve descripción: Setter del mapa de valoraciones
Parámetros:

- *Map <Integer, Double> v*: mapa de valoraciones del usuario

Resultados: El atributo v que contiene el conjunto de valoraciones que ha hecho el usuario es el conjunto de valoraciones v.

[Método] void setOriginalUserId()

Breve descripción: Setter del originalUserId
Parámetros:

- *int originalUserId*: id del usuario original

Resultados: Usuario contiene originalUserId

[Método] void addValoracion()

Breve descripción: Añadir una valoración al mapa de valoraciones del usuario.
Parámetros:

- *Integer id*: id del item que el usuario ha valorado
- *Double v*: score de la

Resultados: El mapa de valoraciones pasa a tener el entry {id, v}

[Método] void tieneValoracion()

Breve descripción: Comprobar si el mapa de las valoraciones tiene el id "idItem"
Parámetros:

- *Integer idItem*: id del item del que queremos comprobar si el usuario ha hecho una valoración

Resultados: Ha devuelto true si el usuario ha valorado el item y si no false

[Método] void getValoracio()

Breve descripción: Coger la valoración del item con id "item"

Parámetros:

- *Integer item*: id del item del que queremos conseguir la valoración.

Resultados: Ha devuelto la valoración del item para el usuario instanciado

2.2.17 Usuarios

Descripción: La clase de usuario contiene la implementación de un conjunto de usuarios.

Cardinalidad:

- Usuario contiene de cero a muchos usuarios, así que instancia la clase usuario de cero a muchas veces.

[Atributo] ArrayList <Usuario>

Tipo: privado

Breve descripción: Conjunto de objetos de la clase de usuario que componen el conjunto de usuarios.

Valor por defecto: ArrayList vacío (new ArrayList<>())

Estático: No.

[Método] getUsers()

Breve descripción: Getter del ArrayList de usuarios

Parámetros: N/A.

Resultados: Se ha devuelto el ArrayList de usuarios.

[Método] getUser()

Breve descripción: Getter de un usuario con id "id"

Parámetros:

- *int id*: id del usuario del que queremos conseguir el objeto.

Resultados: Ha devuelto el usuario con id "id"

[Método] setUsers()

Breve descripción: Setter del ArrayList de usuarios

Parámetros:

- *ArrayList<Usuario> users:* ArrayList de los usuarios del conjunto

Resultados: El atributo users de la instancia de la clase es una copia del parámetro users.

[Método] setUser()

Breve descripción: Cambiar el objeto de un usuario en el ArrayList

Parámetros:

- *Integer id:* id de la posición del ArrayList que queremos cambiar.
- *Usuario user:* usuario que queremos cambiar en el ArrayList

Resultados: El ArrayList de usuarios tiene un entry {user id, user}

[Método] addUser()

Breve descripción: Añadir el objeto de usuario en el ArrayList

Parámetros:

- *Usuario user:* usuario que queremos meter en el ArrayList

Resultados: El ArrayList de usuarios tiene un nuevo entry del usuario user

[Método] containsUser()

Breve descripción: Comprobar si el ArrayList users tiene el usuario con id users

Parámetros:

- *int id*: id del usuario del que queremos comprobar si está en el conjunto de usuarios

Resultados: Ha devuelto true si el objeto tiene un usuario con id "id", falso sí no.

2.2 DESCRIPCIÓN DE ASOCIACIONES Y AGREGACIONES

2.2.1 Atributo – AtributoBoolean

AtributoBoolean es una subclase de la clase abstracta atributo. Cuando un atributo se instancia, se instancia con un tipo. En este caso el tipo es booleano. Por lo tanto todos los atributos de ítem que sean booleanos van a ser directamente parte de la clase AtributoBoolean y por lo tanto de Atributo.

2.2.2 Atributo – AtributoNumerico

AtributoNumerico es una subclase de la clase abstracta atributo. Cuando un atributo se instancia, se instancia con un tipo. En este caso el tipo es numérico (*double*). Por lo tanto todos los atributos de ítem que sean numéricos van a ser directamente parte de la clase AtributoNumerico y por lo tanto de Atributo.

2.2.3 Atributo – AtributoCategorico

AtributoCategorico es una subclase de la clase abstracta atributo. Cuando un atributo se instancia, se instancia con un tipo. En este caso el tipo es categórico (string). Por lo tanto todos los atributos de ítem que sean categóricos van a ser directamente parte de la clase AtributoCategorico y por lo tanto de Atributo.

2.2.4 Atributo – AtributoFecha

AtributoFecha es una subclase de la clase abstracta atributo. Cuando un atributo se instancia, se instancia con un tipo. En este caso el tipo es una fecha (que se guardará en modo long). Por lo tanto todos los atributos de ítem que sean fecha van a ser directamente parte de la clase AtributoFecha y por lo tanto de Atributo.

2.2.5 Atributo – AtributoCategoricoMultiple

AtributoCategoricoMultiple es una subclase de la clase abstracta atributo. Cuando un atributo se instancia, se instancia con un tipo. En este caso el tipo es categórico multiple (Set<String>). Por lo tanto todos los atributos de ítem que sean categórico multiples van a ser directamente parte de la clase AtributoCategoricoMultiple y por lo tanto de Atributo.

2.2.6 Atributo – Item

Un ítem tiene muchos atributos, que lo identifican y permiten que los algoritmos de recomendación basados en características similares de ítems se puedan ejecutar.

2.2.7 Items – Algoritmos

Los algoritmos utilizan ítems para realizar las comparaciones adecuadas y poder ejecutar su sistema de predicción de recomendaciones.

2.2.8 Item - NearestNeighbors

Además de la ejecución normal del algoritmo, nearest neighbor utiliza un ArrayList de ítems para saber en grupo de ítems basarse para sus predicciones.

2.2.9 Parser – Items

El preproceso creará nuestro conjunto inicial de ítems (y como la clase es estática) podemos decir que inicializa el conjunto sobre el que el sistema deberá funcionar.

2.2.10 Parser – Usuarios

El parser creará nuestro conjunto inicial de usuarios. Este creará inicialmente tres grupos, usuarios que vienen del fichero ratings, usuarios que vienen del fichero ratings unkown y usuarios que vienen del fichero ratings known.

2.2.11 Item – Items

Classe de agregación. Un conjunto de ítems es una lista de instancias de la clase item. Por lo tanto podemos afirmar que no existiría Items sin Ítem, por lo tanto vemos la clase de agregación que se forma entre estas dos.

2.2.12 Usuario – Usuarios

Classe de agregación. Un conjunto de usuarios es una lista de instancias de la clase usuario. Por lo tanto podemos afirmar que no existiría Usuarios sin Usuario, por lo tanto vemos la clase de agregación que se forma entre estas dos.

2.2.13 Usuarios - K-Means

El algoritmo de K-Means clusterizar los usuarios iniciales (de ratings, nuestra información de training) para implementar estos clusters usará conjuntos de usuarios.

2.2.14 Usuarios - Algoritmos

El algoritmo de collaborative filtering necesita inicialmente un grupo de usuarios sobre el que trabajar. Para nosotros este va a ser nuestros ratings (información de training de los algoritmos). De esta manera el k-means podrá clusterizar.

2.2.15 Usuario – Algoritmos

Todos los algoritmos que hemos implementado se basan sobre un usuario. Por lo tanto uno de los parámetros principales de la clase algoritmo es el usuario sobre el cual los algoritmos van a actuar.

2.2.16 K-Means – Centroides

K-Means clusterizar la información en centroides, durante la ejecución cada usuario en cada centroide. Por lo tanto esta conexión sirve para la reorganización de los usuarios dentro de los grupos.

2.2.17 Algoritmos – K-Means

K-Means es una subclase de la clase algoritmos. Esta es parte del algoritmo de recomendación del collaborative filtering.

2.2.17 Algoritmos – Slope One

Slope One es una subclase de la clase algoritmos. Esta es parte del algoritmo de recomendación del collaborative filtering.

2.2.18 Algoritmos – NearestNeighbors

Nearest Neighbors es una subclase de la clase algoritmos. Esta es parte del algoritmo de recomendación del content based filtering.

2.2.19 Algoritmos - Recomendación

La clase de recomendación se encargará de gestionar la llamada a los algoritmos e inicializar estos para que funcionen acorde con las recomendaciones que pida el usuario.

3 ARQUITECTURA TRES CAPAS

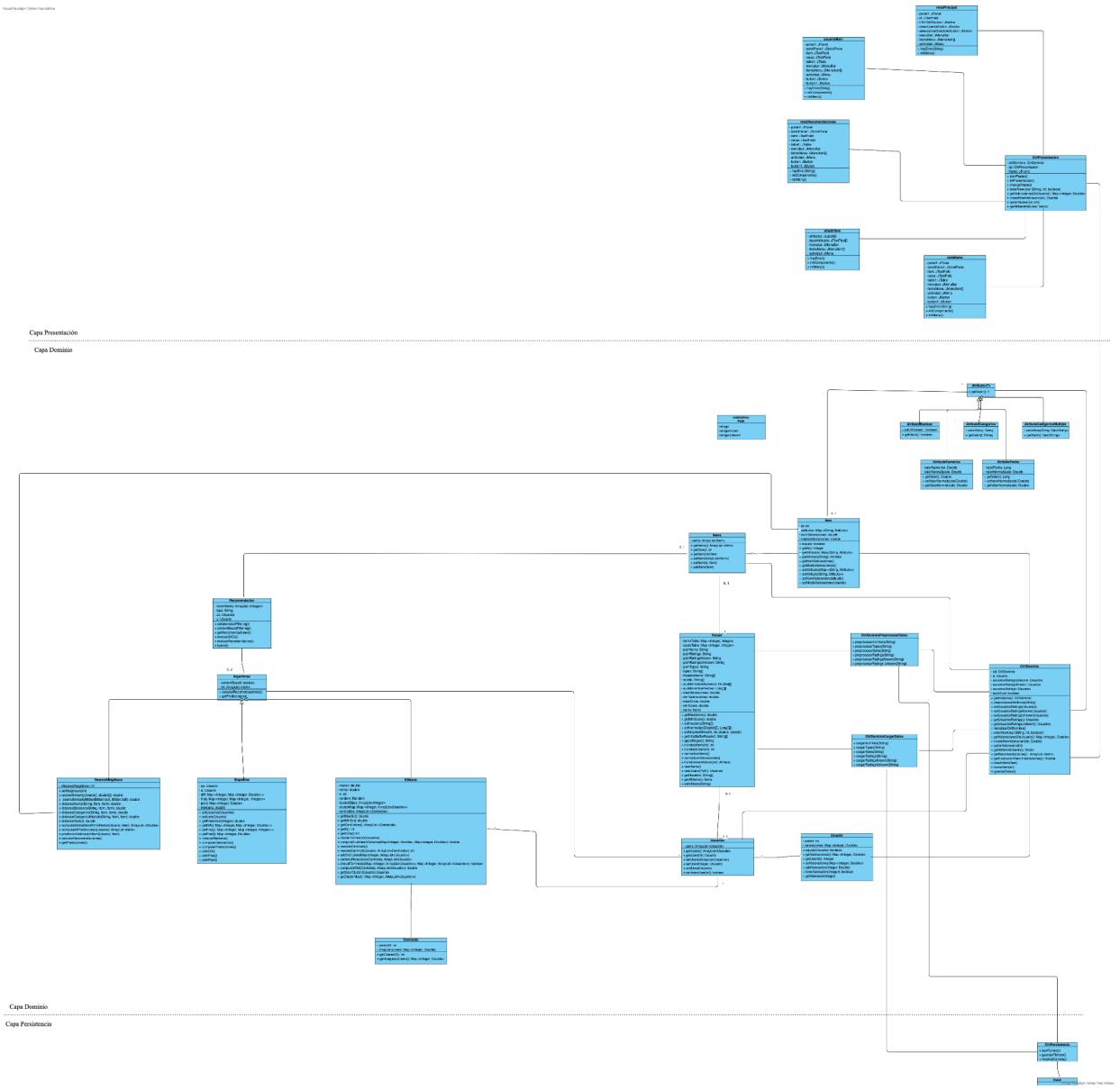
La arquitectura a tres capas es una metodología bien establecida en la ingeniería del software. Esta organiza la aplicación entre tres capas lógicas y físicas: el dominio, la presentación y la persistencia.

La presentación es la interfaz del usuario y por lo tanto la capa de comunicación con el usuario. Su propósito es enseñar la información al usuario y recoger más información de este. Para este proyecto la capa de presentación se ha implementado usando Java Swing.

El dominio representa el centro de nuestra aplicación, todos los datos que conseguimos de la capa de presentación es procesada en esta. Esta capa también tiene la potestad de modificar los datos de la capa de persistencia.

Finalmente, la capa de persistencia es donde la aplicación archiva y controla los datos necesarios para ejecutar nuestra aplicación. En este caso usaremos el formato *csv* para controlar los datos.

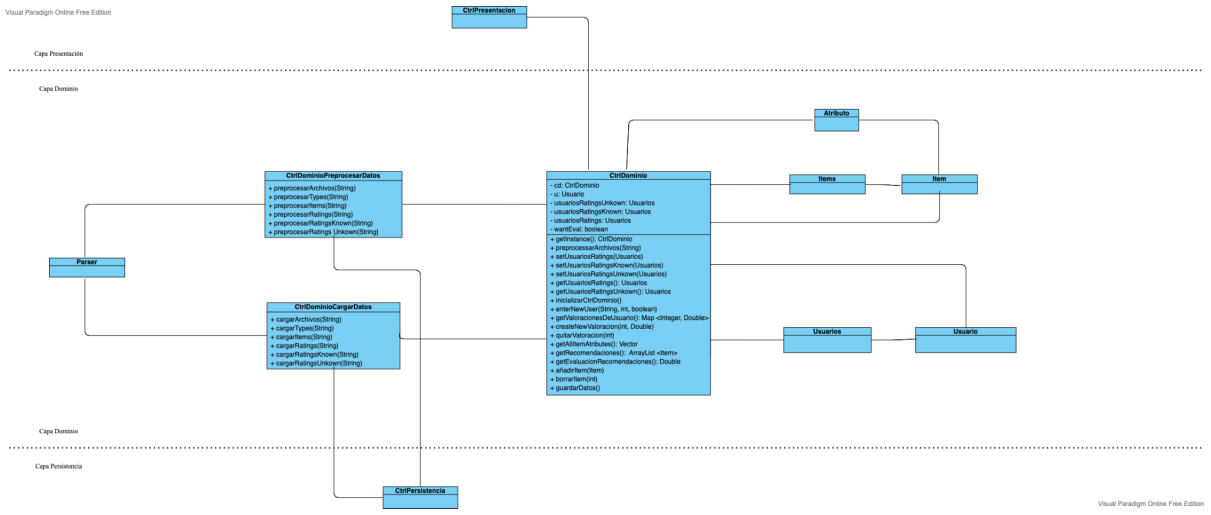
A continuación vemos el diagrama de la arquitectura de tres capas de nuestra aplicación:



3.1 Dominio

Muchas de las clases que forman parte de esta capa están explicadas en el apartado anterior (Modelo Conceptual de Datos) esta corresponde a la mayoría de funcionalidades de la aplicación. Dado la previa explicación de las clases miraremos más a fondo los diferentes controladores de esta capa.

Vemos un diagrama más completo de la implementación de estos controladores:



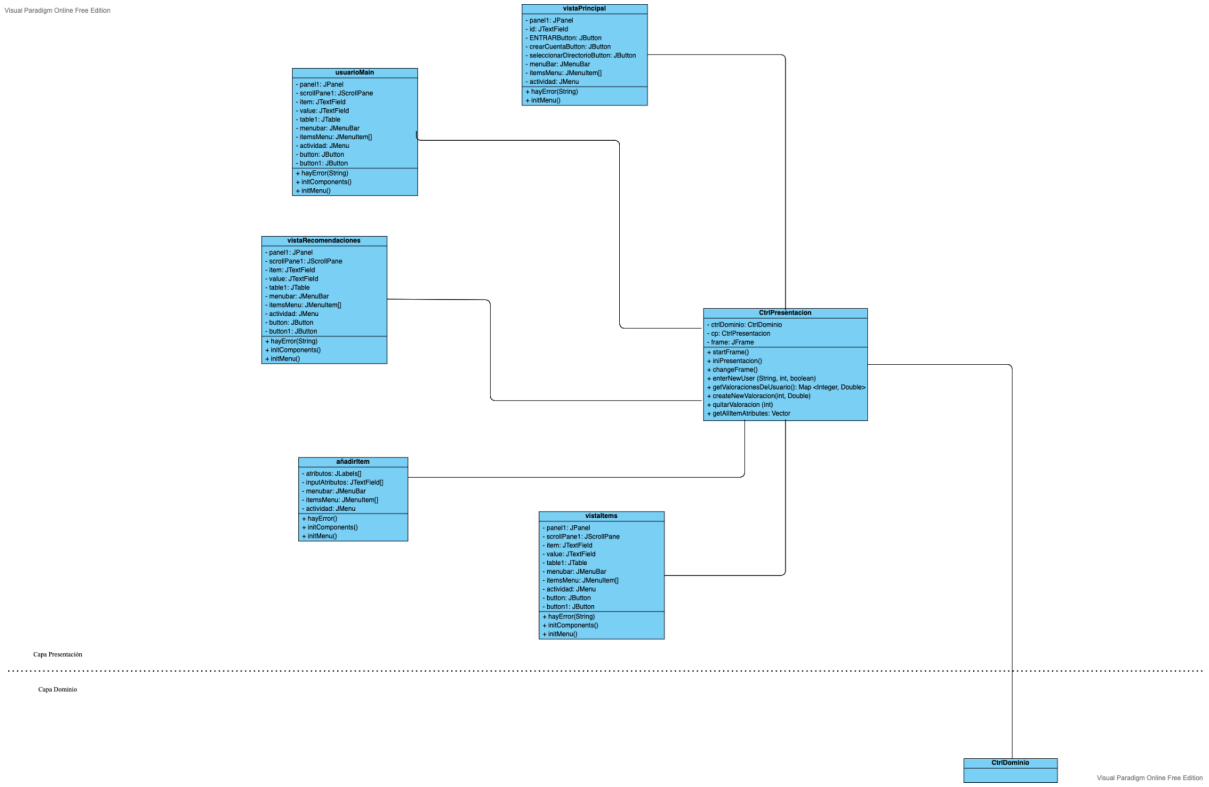
Tendremos tres controladores: un controlador general (*CtrlDominio*) este se encargará mayoritariamente de comunicar la capa de presentación con la capa de dominio y la capa de persistencia.

También tendremos controladores para gestionar la interacción entre los datos que requiere nuestro programa con la base de datos que tendremos. Nuestros controladores de Dominio *CtrlDominioCargarDatos* y *CtrlDominioPreprocesarDatos* se encargarán de leer los datos y procesarlos de forma que nuestro pueda tratar los datos de forma más sencilla. Estos controladores también estarán encargados de escribir los datos procesados cuando lo pida el usuario y leer los datos procesados en el caso que el usuario lo pida. Deberán diferenciar entre el tipo de datos que están leyendo y tratar estos de la forma adecuada, finalizando su ejecución con la producción de datos necesaria para nuestro sistema.

3.2 Presentación

La capa de presentación se encarga de comunicar las diferentes partes de nuestro sistema al usuario. Basadas en las diferentes funcionalidades que tendrá nuestro sistema, vistas en los casos de uso, deberemos implementar diferentes funcionalidades del sistema que permita a nuestros usuario acceder a estas. Para implementar esto deberemos tener varias vistas con diferentes funcionalidades y que estas puedan intercambiarse entre sí en cualquier momento dado del uso de la aplicación por parte del usuario.

A continuación vemos un esquema detallado de las diferentes vistas que implementará nuestro sistema:



3.2.1 Descripción de vistaPrincipal

Esta vista es la que se encargará de recibir al usuario tan pronto como se ejecute nuestro programa. Está formada por dos partes. Una sección que permite al usuario obtener sus recomendaciones, y una segunda, que las obtiene para un "nuevo" usuario que no cuenta con identificador.

Empezamos con la segunda sección puesto que es la más sencilla; solo consta de un botón para permitirle la entrada a los nuevos usuarios, que no estén registrados en el sistema.

Por otro lado, en la primera parte, tras un breve mensaje que le indica al usuario qué pasos debe seguir para obtener sus recoemndaciones, tenemos los siguientes elementos en pantalla. Tenemmos un campo para introducir el identificador del usuario en cuestión. Hay también un botón que permite escoger de qué directorio escoger los csv para cargar los datos en el sistema. además, el usuario también puede seleccionar si quiere evaluar o no las recomendaciones obtenidas. Finalmente, un segundo botón denominado "ENTRAR" que inicia el sistema y lleva al usuario a la siguiente vista.

Las siguientes vistas, permiten la navegación entre ellas a través de un menú en la parte superior izquierda de la ventana.

3.2.2 Descripción de usuarioMain

Esta vista se muestra cuando el usuario ha sido identificado y entra en el sistema. La vista permite al usuario añadir o quitar recomendaciones a ítems, así como ver la lista entera de valoraciones que ya ha dado. Cuenta además con un breve mensaje que indica al usuario qué pasos debe seguir para llevar a cabo la acción deseada.

Para escoger entre la agregación y la eliminación de vistas, el usuario debe seleccionar la opción adecuada (entre los dos radioButtons disponibles). Si lo que desea hacer es añadir una nueva valoración a un ítem, dos campos de texto se resaltará. El primero está pensado para escribir el identificador del ítem para el cual se quiere añadir una valoración, y en el segundo, la valoración en cuestión. Si, por otra parte, lo que el usuario desea hacer es eliminar una valoración que haya dado anterioremente, solamente el campo para introducir el identificador del ítem se verá resaltado.

Hay a continuación un botón que el usuario deberá pulsar una vez introducida toda la información necesaria acorde con la funcionalidad que el usuario está empeñando.

Y para terminar, la vista ofrece también la posibilidad de ver la lista con todas las valoraciones que han sido valoradas por el usuario.

3.2.3 Descripción de vistaRecomendaciones

Esta vista permite llevar a término varias funcionalidades de nuestro sistema. Por una parte, permite visualizar las recomendaciones para el usuario y valorar estos ítems desde la vista misma. Por la otra, si el usuario desea ver la valoración de las valoraciones, también podrá hacerlo en esta vista.

Por lo tanto, encontramos los siguientes elementos en esta vista. En primer lugar, una tabla con todas las recomendaciones que se le han hecho al usuario. Existe también un botón que permite al usuario valorar los ítems que le han sido recomendados, tal y como puede hacer desde la vista principal del usuario. Finalmente, un último botón mostrará al usuario, cuando lo pulse, la calidad de las recomendaciones que ha recibido. En otras palabras, la evaluación de las recomendaciones obtenidas a través de nuestros algoritmos.

3.2.4 Descripción de añadirItem

Esta vista implementa la funcionalidad que permite al usuario añadir nuevos ítems en el sistema. La vista es de lo más sencilla, y está constituida por una serie de campos en los que el usuario debe introducir la información adecuada para que el ítem pueda ser procesado y añadido al sistema.

En la parte superior de la vista habrá también un pequeño mensaje indicando al usuario qué pasos debe seguir.

Junto a cada campo de introducción de texto, hay una etiqueta que indica qué información relativa al futuro nuevo ítem, debe ser añadida. Algunos ejemplos de campos son el identificador, el género o clase al cual pertenece el ítem, una descripción de éste, etc. Dado que tratamos con ítems de distintos tipos de ítems (libros, películas, música, etc) ciertos campos pueden dejarse en blanco ya sea o bien

porque no tienen sentido para el ítem en cuestión, o bien porque el usuario no los conoce. Sin embargo hay algunos campos que sí son imprescindibles, como por ejemplo el identificador del ítem.

Cabe mencionar que, si el ítem que se está intentando añadir ya existe, los datos introducidos en los demás campos de datos, sustituirán a sus respectivos campos en la base de datos de nuestro sistema. Así que, en realidad, esta vista sirve tanto para añadir ítems como para modificar los ya existentes.

3.2.5 Descripción de vistaItems

En la última vista de nuestro sistema, se implementa la visualización de todos los ítems en el sistema. Es también desde esta vista que el usuario podrá eliminar ítems en el sistema, para complementar la adición y la modificación de ítems, ya implementadas en la vista anterior.

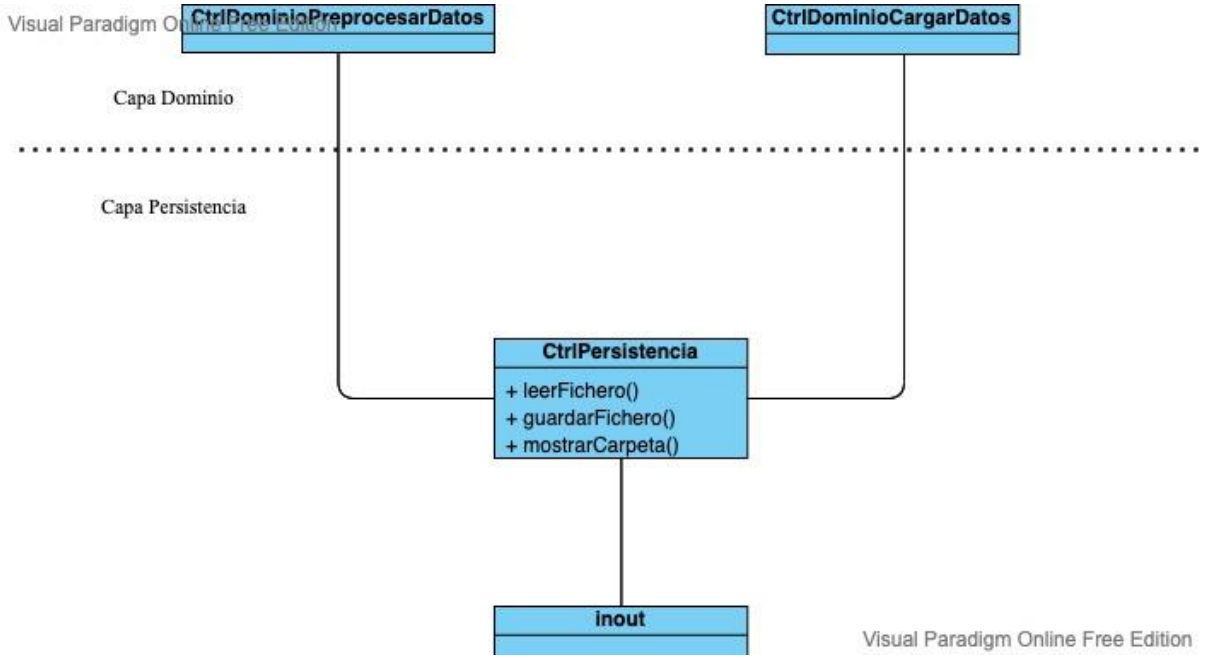
La vista contiene tres elementos básicos. El primero es un campo de texto donde el usuario puede introducir identificadores de ítems en la lista. A su lado, hay un botón que al ser pulsado actualiza la lista de ítems, habiendo eliminado tanto el ítem introducido como cualquier recomendación y valoración que se haya hecho de ése ítem en todo el sistema.

El último elemento es la tabla de ítems, que muestra los campos más relevantes de cada objeto. Por ejemplo, para películas, estos son el identificador, el título, el género y la popularidad de cada película.

3.3 Persistencia

La capa de persistencia se encarga de la conexión entre la base de datos, en nuestra caso los datos sin preprocesar estarán guardados en ficheros csv y los datos ya preprocesados estarán guardados con el formato txt.

A continuación vemos la esquemática perteneciente a la capa de persistencia de nuestro sistema:



El controlador de persistencia manejara la interacción entre los datos y la capa de dominio. Para todas las acciones de guardar datos, leer datos y manejar las carpetas que podemos acceder el controlador de persistencia se encargará. Cualquier tipo de input y output se maneja por el controlador de persistencia desde los controladores de dominio CtrlDominioProcesarDatos y CtrlDominioCargarDatos para preprocesar datos de nuestros csv o cargar datos de datos previamente procesados, respectivamente.

4 ESTRUCTURAS DE DADES

Nuestro sistema recomendador cuenta con diversas estructuras que nos permiten mantener un seguimiento de los elementos clave para la recomendación de contenido a los usuarios. Estos elementos, y por lo tanto sus respectivas estructuras de datos son explicados en los siguientes apartados.

4.1 Usuarios

A fin de poder proporcionar al usuario activo una recomendación, necesitamos una estructura que nos permita guardar información sobre los demás usuarios, en los que nos basaremos para escoger qué contenido recomendar y qué no. La clase *Usuarios* cumple éste cometido, y está implementada de un modo sencillo; una ArrayList de objetos del tipo *Usuario*. Esta estructura es ideal, pues nos permite mantener a todos nuestros usuarios ordenados según su identificador, y facilitar la consulta.

En resumen, la estructura principal relativa a los usuarios es la siguiente:

ArrayList<Usuario> usuarios (clase: Usuarios)

4.2 Items

Del mismo modo que los usuarios juegan un papel importante en la recomendación, es evidente que los ítems también lo tienen, pues ellos son la recomendación en sí. Es por esa razón que también nos conviene almacenar de qué ítems disponemos, para así poder hacer tanto consultas rápidas como seleccionar un subconjunto de ellos y formar una recomendación para el usuario activo.

Así que una vez más, la estructura que nos ha parecido más adecuada para guardar el conjunto de ítems del que disponemos es una ArrayList de objetos de tipo *Ítem*. Igualmente, los ordenamos en función de su identificador, que vuelve a ser un entero positivo y único para cada *ítem.*

Estructura principal para guardar ítems:

ArrayList<Item> *usuarios* (clase: Items)

4.3 Valoraciones

Finalmente, de cada objeto *Usuario* necesitamos guardar también cuáles son los ítems que ha valorado, y con qué nota. Es aquí donde nos aparece la necesidad de tener una tercera estructura de datos que almacenará información esencial para determinar la "similitud" o "cercanía" entre dos usuarios. Para hacerlo, mapeamos cada identificador de un ítem, a la valoración que *Usuario* ha dado a éste.

Por tanto, la tercera y última estructura principal del programa es el siguiente mapa:

Map<Integer, Double> *valoraciones* (clase: Usuario)

5 ALGORITMOS

Para conseguir determinar qué ítems recomendar a cada usuario activo, hemos implementado los siguientes algoritmos, que quedan descritos en los siguientes apartados. Los dos primeros son algoritmos de Collaborative Filtering, que recomiendan al usuario activo ítems que han gustado a usuarios similares a él. Por otra parte, el último algoritmo usa una estrategia propia del Content Based Filtering, que se basa en los ítems que han gustado al usuario para recomendarle otros similares.

5.1 K-Means

K-Means es un algoritmo de clusterización que forma k clusters alrededor de un elemento central denominado centroide. En nuestro caso, los clusters que deseamos formar son de usuarios, y la condición que determina si dos usuarios pertenecen al mismo cluster o no, es cómo de "parecidos" son éstos dos usuarios.

Basamos la similitud entre usuarios en su reacción a los distintos ítems. Cuando un par de usuarios consumen un ítem, y lo valoran, podemos averiguar si esos dos usuarios opinan lo mismo sobre dicho ítem o no. Cuando hacemos esto a gran escala (con muchos usuarios, y muchos ítems), lo que obtenemos es mucho más valioso; el cómo de parecidos son los gustos entre los usuarios observados. Por lo tanto, definiremos la semejanza entre usuarios como la distancia entre las valoraciones de sus ítems. Existen diversas formas de medir esta distancia, y nosotros hemos optado por la euclídea. Una vez aclarado esto, podemos pasar a la descripción del algoritmo.

K-Means agrupa los usuarios en k clusters y los reajusta, hasta encontrar una partición estabilizada. Distinguimos las siguientes partes en nuestro algoritmo: Creación de centroides arbitrarios, adición/relocalización de usuarios a clusters y reajuste de los centroides. El algoritmo hace una sola vez el primer paso y luego repite los dos últimos hasta que los clusters dejen de verse modificados.

En la primera parte del algoritmo, se crean k centroides, que en el fondo no son más que k usuarios imaginarios que han valorado todos los ítems. Para hacerlo, creamos una estructura de datos idéntica a la de valoraciones (mencionada en el apartado anterior) y asignamos un *score* aleatorio, dentro del rango permitido, a cada uno de los ítems. Para asegurarnos de que los k centroides con los que vamos a llevar a cabo el algoritmo sean suficientemente distintos y vayan a darnos mejores resultados, hacemos lo siguiente: creamos primero el doble de centroides a los verdaderamente necesarios, y luego pasamos la arraylist que los contiene por una función que se encarga de eliminar los k centroides que sean más cercanos entre ellos.

Cabe añadir que podríamos limitarnos a asignarle un valor aleatorio sin tener en cuenta el rango porque los centroides terminarían ajustándose igualmente. Sin embargo, es mala práctica, dado que partimos de un estado mucho más desequilibrado y forzamos al algoritmo a hacer más iteraciones que en el fondo son innecesarias.

Una vez creados los k centroides, el algoritmo repite los siguientes pasos hasta haber encontrado el equilibrio en todos los clusters. En primer lugar determina cuál es el centroide más cercano o parecido para cada usuario, y lo añade al cluster perteneciente a ese centroide. A continuación, calcula nuevamente las valoraciones de cada centroide, de manera que éste representa ahora el centro del cluster. Para hacerlo cambia cada valor (inicialmente arbitrario) por la valoración media que los usuarios en el cluster han dado a cada ítem. Finalmente repite el algoritmo. Vemos que al relocalizar

los centroides, es posible que algunos usuarios acaben teniendo más cerca un centroide distinto al que tenían en la iteración anterior, de modo que se van modificando los clusters.

Disponemos de un par de estructuras que nos facilitan la implementación del algoritmo. Por una parte un mapa que asocia cada centroide a su respectivo cluster, y por otra una ArrayList de tamaño k que almacena los centroides generados al inicio.

Las estructuras mencionadas, pues, tienen la siguiente forma:

Map<Integer, ArrayList<Usuario> > *clusterMap* (clase: KMeans)

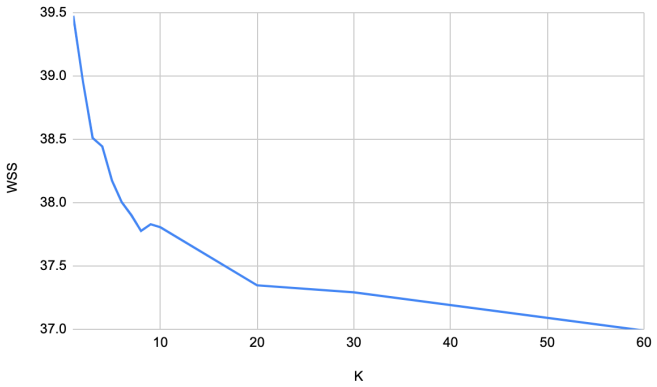
ArrayList<Centroide> *centroides* (clase: KMeans)

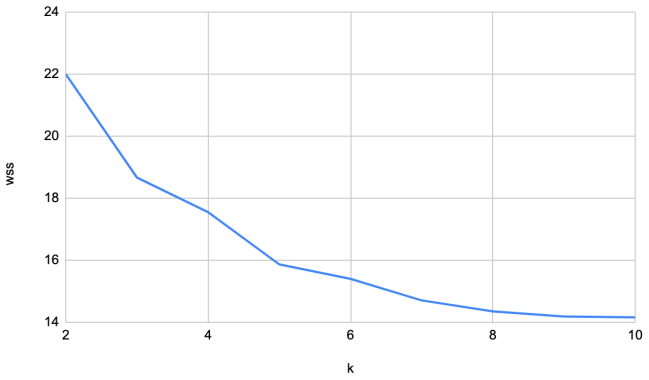
5.1.1 K óptima

Como podemos observar, el algoritmo se basa en la creación de k clusters de usuarios, en función de sus similitudes. Sin embargo, tanto un valor para k muy pequeño como uno muy grande (relativamente a la cantidad de datos con los que trabajamos), pueden ser contraproducentes para lo que pretendemos conseguir con éste algoritmo. En otras palabras, un sólo cluster con todos los usuarios nos es tan poco útil como Usuarios.size() clusters de un sólo usuario; y valores muy cercanos a éstos casos extremos, tampoco nos proporcionarán necesariamente los mejores resultados.

Existen diversos métodos para determinar cuál es la k óptima; nosotros hemos optado por el *elbow method*. Éste procedimiento consiste en calcular la distancia media usuario-centroide, así que sumamos la distancia euclídea de cada usuario en un clúster a su centroide y dividimos por el tamaño del clúster. Repetimos el proceso para diversas k, y escogemos aquella a partir de la cual el valor de wss empieza a disminuir; en el gráfico, esto se visualiza como una especie de "codo", y de allí proviene el nombre.

Hemos realizado pruebas para dos casos; uno con 2000 usuarios (usado para los tests) y otro con un conjunto de usuarios más pequeño (250 usuarios), para uno de los casos reales proporcionados. Después de ejecutar el algoritmo para diversos valores de k, hemos obtenido los dos gráficos siguientes:





El primer gráfico se corresponde con el caso de 2000 usuarios, mientras que el segundo es para 250. En el primero determinamos que la k óptima cae alrededor de los 20 clusters. Por otro lado, para 250 usuarios, una k de 5 o 6 clusters parece ser más adecuada.
Si bien es cierto que no son resultados completamente concluyentes, existen otros métodos donde se ve con más claridad cuál es el valor óptimo para la k. Está por ejemplo es el Silhouette Method ("método de las siluetas"), que implementaremos más adelante.

5.1.2 Coste

Ahora que hemos explicado el algoritmo, podemos analizar cuál es el coste de éste. Haremos un análisis de su eficiencia espacial, así como de la temporal. Como el algoritmo principal, implementado en la función ***clusterFormation,*** incluye una serie de funciones auxiliares, procederemos a analizar primero el coste de éstas para luego ver qué papel juegan una vez se aplican (una, o varias veces) en el algoritmo completo.

Algo importante que añadir ahora es que en java, las variables de tipos primitivos son pasadas por valor, y las de tipos no primitivos pasan la *referencia* al objeto por valor, haciendo que no sea relevante la complejidad del objeto referenciado. En consecuencia, consideramos el coste espacio temporal de todas las variables pasadas por parámetro (tanto de tipos primitivos como no primitivos) como constantes, y no lo tomaremos en consideración.

Para facilitar los análisis siguientes, vamos a denominar i a el número de ítems en el sistema, u al número de usuarios, y **k**, a el número de clusters que terminamos formando. Estas son las tres principales variables que afectan sobre todo al rendimiento de nuestro algoritmo, y por ello vale la pena denominarlas previamente.

En la función ***computeEuclideanDistance***, calculamos la distancia euclídea entre la serie de valoraciones de dos usuarios; más específicamente, entre un usuario y un centroide. No es difícil ver que el **coste espacial** de esta función es prácticamente **constante**, pues sólo hacemos uso de un par de variables para almacenar el resultado de los cálculos parciales. Por lo que al coste temporal respecta, estamos recorriendo el mapa de valoraciones de un centroide, que tiene por tamaño el número de ítems en el sistema. Como la función que usamos es *containsKey* empleada es constante, la **eficiencia temporal** de la función termina siendo de $O(i)$**.** Como veremos más adelante, hacemos uso de esta función en dos contextos distintos; en el primero, la estructura que almacena los centroides contiene dos veces el número de centroides necesarios, mientras que en el segundo contexto contiene exactamente k centroides. De todos modos, esto no afecta a ninguna de nuestras complejidades.

Para la generación de los centroides, hacemos uso de la función ***randomCentroids.*** La única estructura de datos que usamos aquí es un mapa, que llegará a contener una valoración por cada ítem en nuestro sistema, por lo que el **coste espacial**, se convierte en **lineal** respecto de i, esto es, $O(i)$. Por otro lado, la función consiste en dos bucles for anidados; el externo hace 2 veces k iteraciones para generar, como ya hemos mencionado, el doble de centroides a los necesitados. El bucle interno hace tantas iteraciones como ítems tenemos, para asignar una valoración aleatoria a cada uno de ésos ítems. Debido a que el bucle interno hace siempre un número constante de iteraciones, **el coste temporal** de esta función es de $O(ik)$.

La función ***furthestCentroids***, por su parte es la que se encarga de eliminar los k centroides sobrantes. Tiene un **coste de memoria lineal respecto a k** ya que hacemos una copia de la arraylist de centroides para poder comparar y eliminar los centroides más cercanos. El método consta de tres pasos: primero realiza la copia de la arraylist con 2k centroides, luego compara y elimina los más cercanos, y finalmente modifica los identificadores de los centroides supervivientes para asegurar que éstos vayan de 0 a k-1. La primera y la tercera parte tienen un claro coste lineal respecto a k, mientras que la segunda parte necesita un análisis más profundo. La segunda parte consiste en un bucle anidado que en el peor de los casos, tendrá un coste cuadrático. En el bucle más interno, hacemos llamadas a la función que computa las distancias euclídeas. Como hemos dicho, ésta función tiene coste lineal respecto a i, por lo que el **coste de temporal** de *furthestCentroids* es de $O(k + ik^2)$.

La función ***nearestCentroid*** tiene un claro **coste espacial constante**, puesto que no hay ninguna pesada estructura de datos que mantener. Respecto al **coste temporal**, es $O(ik)$, pues la función consiste básicamente en un bucle que recorre la arraylist de centroides, y que calcula la distancia euclídea para cada uno.

Después, tenemos ***addToCluster***. La función recorre una de las principales estructuras de datos de la clase, *clusterMap*. Este mapa almacena para cada llave identificadora de un centroide, una arraylist con los usuarios que pertenecen al cluster de dicho centroide. Como la estructura la guardamos como atributo de la clase, consideraremos que el **coste espacial** de esta función en concreto es **constante**, y al hacer el análisis final ya tomaremos en consideración el coste del mapa. La **complejidad temporal** de *addToCluster*, es **constante**, ya que aplicamos una lambda función de coste constante a uno de los centroides.

A continuación tenemos ***centroidRelocation,*** también tiene un coste **constante** en la **complejidad espacial,** ya que no hacemos uso de ninguna estructura auxiliar. Y el **coste temporal** es de $O(ui)$ en el peor de los casos, ya que aunque parezca constante respecto a la k, si depende del número de ítems y usuarios en el sistema. El peor de los casos se da tanto si ejecutamos el algoritmo con un solo cluster (k = 1) como si, por algún motivo, un cluster contuviera todos los usuarios en el sistema.

Finalmente, la función booleana ***shouldTerminate,*** tiene un coste **espacial constant**e, y el **temporal es** $O(uk^2)$ ya que en el peor caso itera tres bucles for anidados, donde los dos exteriores hacen k iteraciones, y el más interno depende del número de usuarios en el cluster. De nuevo el peor caso se da cuando k = 1 o cuando todos los usuarios forman parte de un cluster.

Ahora que hemos analizado de forma aislada el coste de cada función, podemos analizar el algoritmo de KMeans. El algoritmo aplica una sola vez las funciones *randomCentroids* y *furthestCentroids*, que como hemos visto, tienen un coste de $O(ik)$ y de $O(k + ik^2)$ en los peores casos. A continuación,

ejecuta un número de veces (nunca mayor a la cota establecida) la clusterización. Dentro del bucle tenemos un primer bucle for que itera sobre todos los usuarios, y llama a las funciones *nearestCentroid* y *addToCluster*, con costes $O(ki)$ y constante respectivamente. Tenemos un segundo bucle que se realiza en todas las iteraciones excepto la última. Antes de entrar en éste, llamamos a *shouldTerminate* (con coste $O(uk^2)$) y más adelante al método *clear* para HashMaps (coste lineal). Y una vez dentro, llama a tres funciones, dos de coste constante y una (*centroidRelocation*) tiene coste $O(ui)$. El coste del for loop es por tanto de $O(uik)$. El algoritmo termina haciendo una copia de un mapa, que nuevamente tiene coste O(k). La complejidad temporal total termina siendo entonces, de $O(k + i(k + k^2) + nk(1 + i + ui + ku))$. Donde k es el número de centroides, **u** el de usuarios, **i** el de ítems, y n es la cota.

Cabe mencionar que normalmente el algoritmo termina siempre antes de la cota, dado que comprobamos con ***shouldTerminate*** que, si el estado de los clusterMaps de dos iteraciones seguidas son iguales, que el algoritmo termine pronto. El único caso en el que se ejecutarán n iteraciones, sería si la cota fuera lo suficientemente pequeña como para que a los clusters no hubieran tenido suficientes iteraciones como para estabilizarse.

Para terminar, observamos la complejidad espacial de la clase. Ésta almacena dos estructuras de datos y una serie de atributos de tipos primitivos, que ignoraremos por ocupar memoria constante. Las estructuras, por otra parte, son un Map *clusterMap* y una ArrayList *centroides*. La primera, almacena para cada centroide los usuarios que forman parte de su cluster. Por lo tanto, el coste en memoria de guardar esta estructura nunca es mayor que $O(uk)$. La segunda estructura almacena tantos centroides como determine la k del algoritmo, es decir que tiene coste $O(k)$.

5.2 Slope One

Como bien se explica en la documentación de la práctica, el slope one forma parte del método de predicción con collaborative filtering. El collaborative filtering es un método de recomendación de ítems o información específicas a un usuario basado en un conjunto de usuarios. El collaborative filtering puede ser basado en usuarios o en ítems.

El Slope One se trata de un algoritmo de collaborative filtering basado en los ítems propuesto por Lemire y Maclachlan. El Slope One está basado en las desviaciones de valoraciones entre ítems de un mismo usuario. Por lo tanto recibe información de los usuarios que han valorado un mismo ítem y los diferentes ítems que ha valorado un mismo usuario.

Nos proponen varias formas de hacer este algoritmo. La primera forma es sumando la desviación entre dos ítems (uno valorado por el usuario y el otro no) y la valoración del usuario para el ítem si valorado. Sumando todas las posibles combinaciones entre un ítem valorado y un ítem no valorado por el usuario y dividiendo por el cardinal de estas podemos encontrar una predicción para el ítem no valorado.

La siguiente forma propone una simplificación de este algoritmo usando aproximaciones de los valores. De tal manera que vemos que, en el caso de tener un set suficientemente grande podemos intuir que cualquier par de ítems tendrá algún usuario que haya valorado esos dos ítems. Por lo tanto se podía aproximar que la predicción de un ítem era la media de valoraciones de un usuario más la suma de todas las desviaciones partidas por el cardinal del número total de desviaciones.

Aunque estas dos nos parecían muy buenas opciones, optamos por buscar más información sobre la implementación y los diferentes tipos que tenía el Slope One. Encontramos pues, el Weighted Slope One, que explicaremos a continuación, que nos proporcionará mejores resultados ya que tiene en cuenta el número de valoraciones que se han hecho por ítem para calcular la media ponderada entre estas, y así obtener mejores resultados

5.2.1 Estructuras de Datos

Las estructuras de datos de este algoritmo se pueden dividir en dos partes, el cálculo de las desviaciones y el cálculo de las predicciones.

Para el cálculo de las desviaciones tenemos las siguientes estructuras de datos:

Map <Integer, Map <Integer, Double> > devs

Map <Integer, Map <Integer, Integer> > freq

Nos guardaremos tanto las desviaciones que obtenemos como la frecuencia con la que dos ítems se han valorado por el mismo usuario. De esta manera simplificamos el coste temporal del algoritmo ya que no tenemos que recorrer los usuarios una segunda vez para mirar quien ha valorado los dos ítems con los que estamos trabajando en ese momento. Además, como hemos visto la frecuencia en la que valoramos los ítems también tiene que ver en las predicciones, no solo lo usamos para calcular las desviaciones. Por lo tanto estos dos mapas serán variables globales de la clase.

Estos dos mapas guardaran las desviaciones y frecuencias (respectivamente) por cualquier par de items i e j.

Para encontrar las predicciones por cada ítem que nuestro usuario aún no ha valorado pues, solo necesitaremos un mapa con las predicciones por ítem del usuario del cual queremos predecir. Si este sistema no fuese céntrico a un usuario, podríamos crear un mapa de mapas de doubles y así guardar para cada usuario las desviaciones. Como hemos optado por hacer el programa céntrico a un usuario (para futuras extensiones en la presentación). Tendemos la siguiente estructura de datos:

Map <Integer, Double> pred

De esta manera nos guardaremos para cada ítem que el usuario no ha valorado, la valoración esperada.

5.2.2 Weighted Slope One

Uno de los principales inconvenientes del algoritmo de predicciones es que la cantidad de ratings que observamos en cada una de las predicciones no se consideran en la ponderación de valoraciones. El Slope One Weighted se diferencia del algoritmo convencional de Slope One en que toma en consideración la cantidad de ítems que ha valorado nuestro usuario que también lo ha valorado el usuario con el que lo estamos comparando. De esta forma vamos a ponderamos los usuarios de forma que le daremos más importancia a las valoraciones de los usuarios que tengan ratings más similares al de nuestro usuario.

La siguiente es una fórmula representativa de cómo se harán las predicciones usando el Weighted Slope One.

$$P^{wS1}(u)_j = \frac{\sum_{i \in S(u) - \{j\}} (dev_{j,i} + u_i) * c_{j,i}}{\sum_{i \in S(u) - \{j\}} c_{j,i}}$$

5.2.3 Slope One con K-Means

El Slope One se basa en los usuarios que este recibe, por lo tanto debemos procurar que estos usuarios sean lo más próximos a nuestro usuario como sea posible. Por esta razón usaremos el K-Means, de manera que los ratings que estamos comparando entre sí serán lo más parecido posibles, y por lo tanto las predicciones que podemos hacer de nuestros usuarios estarán más ajustadas a la realidad de lo que predeciría nuestro usuario.

5.2.4 Coste

Este algoritmo se divide en dos partes: el cálculo de la matriz de desviaciones y el cálculo de las predicciones. Para la explicación del coste nos basaremos en las siguientes simplificaciones: n será el número de usuarios del sistema, m será el número de ítems a predecir y M el número de predicciones del usuario.

Inicialmente vamos a analizar la sección de calcular las desviaciones. En cuanto a coste temporal, inicialmente para cada usuario itera por todas sus valoraciones. En el peor caso donde haya una gran multitud de datos. Podemos decir que en el peor caso el usuario habrá valorado todos los posibles ítems, llegando a un coste de $O(n * m)$ para este primer procesado de datos. Posteriormente iteraremos para cada ítem, por todos los pares que algún usuario haya valorado. En el peor caso, esto nos daría una complejidad temporal $O(m^2)$. Por lo tanto para esta primera parte tenemos una complejidad temporal final de $O(n * m + m^2)$. En cuanto a coste del espacio nos guardamos dos matrices que en el peor caso tendrán un tamaño del número de ítems por el número de ítems. Por lo tanto tendremos una complejidad de espacio de $O(2 * m^2)$ que se puede simplificar a $O(m^2)$.

Para finalizar las predicciones, vamos a realizar las predicciones a partir de la matriz de desviaciones que hemos calculado. En cuanto a coste temporal, vamos a iterar por las valoraciones del usuario y para una de estas valoraciones vamos a iterar por todas las desviaciones calculadas para este ítem. Como se ha remarcado anteriormente este coste el siguiente $O(m * M)$ ya que en el peor caso para cada una de las valoraciones del usuario hay una desviación calculada con todos los items. Para el

espacio utilizamos tres vectores auxiliares, estos tendrán su tamaño limitado a la cardinalidad de los ítems. Por lo tanto $O(3 * m)$ que se puede simplificar a $O(m)$.

Por lo tanto, para coste temporal de la primera parte tenemos $O(n * m + m^2)$. Y de la segunda parte $O(m * M)$, en el caso que el usuario ya haya valorado todos los ítems sería $O(m^2)$. Por lo tanto el coste temporal sería $O(n * m + m^2)$. En cuanto a coste del espacio hemos visto que la primera parte nos daba la cardinalidad de los ítems al cuadrado, y la segunda la cardinalidad de los ítems. De esto deducimos el siguiente coste $O(m^2 + m)$ que se puede reducir a $O(m^2)$.

5.3 Nearest Neighbors

El algoritmo K-Nearest Neighbors es un método de clasificación supervisada. En nuestro caso, este algoritmo nos sirve para calcular los K elementos más cercanos a los elementos que ha valorado un usuario en concreto. Al ser un método de recomendación "content based" tendrá en cuenta los ítems y que tan parecidos son unos a otros, a diferencia del método "collaborative filtering" que tiene en cuenta a los usuarios.

Primero, hay que pensar en una forma correcta de calcular la distancia entre dos ítems. Para ello, dividimos el cálculo entre los diferentes atributos de los ítems y así, la distancia entre dos ítems será la media de las distancias de los atributos. Para los atributos numéricos (ya normalizados) la distancia será la diferencia en valor absoluto. Para los atributos booleanos y categóricos, será 0 cuando sean iguales y 1 en caso contrario. Finalmente, para los atributos categóricos múltiples, la distancia será la intersección entre la unión.

Una vez sabemos calcular la distancia entre atributos de ítems, definimos la siguiente fórmula:

$$valoración(u, i) = \sum_{\forall j \in Items} (valoración(u, j) * (1 - distancia(i, j)))$$

Donde *u* es el usuario al cual le queremos recomendar ítems, *i* es un item que el usuario no ha valorado y *j* es cada uno de los ítems que el usuario *u* ha valorado. Esta fórmula nos devuelve una predicción de la valoración que un usuario le daría a un ítem que no ha valorado previamente.

Después de calcular la predicción de la valoración para cada uno de los ítems que el usuario no ha valorado, se ordenan estas valoraciones en orden decreciente (ya que a más valoración, más le gustará ese ítem al usuario) para después proceder a quedarnos con los K primeros de esta lista. Finalmente, se devuelve una lista con estos K elementos, ordenados por la predicción de la valoración.

5.3.1 Coste

El coste de este algoritmo depende de varias variables. Para entenderlo mejor, vamos a empezar viendo cual es el coste de calcular la distancia entre 2 atributos. Para los atributos numéricos, booleanos y categóricos el coste es constante ya que solo hacemos una comparación o una resta entre ambos atributos. Después, la distancia entre atributos categóricos múltiples la calculamos iterando sobre todos los elementos de ese atributo. Por ello, el coste de este cálculo es $O(E)$ donde "E" es el número de elementos que tiene ese atributo. Cabe destacar que para calcular esta distancia usamos una estructura de datos muy eficiente, el BitSet, para hacer operaciones como AND. Finalmente, para calcular la distancia entre textos usamos una técnica llamada Tf-idf. El cuello de botella de este algoritmo es la función "cosineSimilarity" que itera sobre cada palabra de uno de los dos textos. Por

ello, la complejidad temporal de este cálculo es $O(P)$ donde "P" es el número de palabras en el texto. Podemos decir que el tiempo para calcular la distancia entre atributos es $O(P + E)$ ya que los otros términos son constantes.

Sabiendo el tiempo que se tarda en calcular las distancias entre atributos, pasemos a ver ahora las distancias entre ítems. Para calcular la distancia entre dos ítems, se recorren cada uno de los atributos y, para cada uno de ellos, se calcula la distancia entre atributos. Por ello, la complejidad temporal es $O(A * (P + E))$ donde "A" es el número de atributos que tienen los ítems. Después, para hacer la predicción de una valoración de un ítem, iteramos sobre todos los ítems que el usuario ha valorado, y para cada uno de ellos, hacemos el cálculo anterior (calcular la distancia entre dos ítems). Por ello, la complejidad temporal aumenta a $O(V * A * (P + E))$ donde "V" es el número de ítems que el usuario ha valorado. Finalmente, lo que hacemos desde un principio es, para cada ítem que el usuario no ha valorado, predecir su valoración. Por ello, la complejidad final del algoritmo KNN es $O(I * V * A * (P + E))$donde "I" es el número de ítems que el usuario no ha valorado.

Por otro lado, el coste espacial del algoritmo es $O(I + A)$ donde "I" es el número de ítems que el usuario no ha valorado y "A" es el número de atributos que tienen los ítems.

5.3.2 One Hot

Al inicio de esta práctica, utilizamos una solución muy sencilla para calcular la distancia entre atributos categóricos múltiples. La técnica que utilizamos fué dividir el número de elementos en común (la intersección) entre el número de elementos totales (la unión). Esto, como es de esperar, no es la solución más óptima. Por ello, hemos decidido implementar una nueva estrategia que mejoraría tanto la predicción de la distancia entre estos atributos como la complejidad temporal de este cálculo. Esta técnica se llama One Hot y para implementarla, hemos usado una estructura de datos llamada BitSet, que es un vector de bits (valores de 0 o 1 en cada posición). El algoritmo consiste en tener para cada atributo categórico múltiple, un BitSet del tamaño total de todos los elementos de ese atributo. Por ejemplo, si el atributo fueran géneros, tendríamos para cada ítem, un BitSet del tamaño del número de géneros que existen en todos los ítems de nuestros datos. Después, pondremos a 1 las posiciones que ese ítem tiene. Cabe destacar que, todo esto se realiza en el preproceso. En la clase NearestNeighbors, recibimos ya los ítems con los BitSets inicializados por lo que procedemos a realizar el cálculo. Éste consiste en la similitud de coseno entre ambos vectores. La fórmula es la siguiente:

$$similarity(A, B) = \frac{\sum_{i=1}^{n} A_i * B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} * \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Como hemos dicho antes, la estructura de datos BitSet nos permite realizar algunas operaciones muy eficientes. Primero, realizamos un "AND" entre ambos vectores que representa una multiplicación de sus elementos (el dividendo de la fórmula). Después, para calcular el divisor de la fórmula, aplicamos la función *cardinality* a cada uno de los BitSets, que devuelve el número de bits que valen 1, que representa cada uno de los sumatorios del divisor. Una vez tenemos estos valores, devolvemos el dividendo entre la multiplicación de las raíces cuadradas del resultado de aplicar la función *cardinality* a cada uno de los BitSets. Como hemos explicado anteriormente, este cálculo tiene coste $O(E)$ donde "E" es el número total de elementos que tiene ese atributo (la unión de todos los ítems).

5.3.3 Tf-idf

Para calcular la distancia entre dos atributos de tipo texto, hemos tenido que utilizar una técnica un poco más sofisticada. Esta técnica se llama Tf-idf. Por un lado, *Term Frequency* o frecuencia de término se refiere a la cantidad de veces que aparece un término en un documento (en nuestro caso, en un atributo de un ítem). Por otro lado, *Inverse document frequency* o frecuencia inversa de documento es el número de documentos en los que aparece ese término. La fórmula para cada una de estas partes es la siguiente:

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

donde

- t: término o palabra del atributo texto.
- d: documento o atributo texto de un ítem.
- $f(t,d)$: frecuencia del término *t* en el documento (atributo texto de un ítem) *d*.
- D: documentos o conjunto de atributos de texto del mismo tipo.
- $|W|$: número de palabras en el documento.
- $|D|$: cardinalidad de D o número de documentos de la colección (número de ítems).
- $|\{d \in D : t \in d\}|$: número de documentos donde aparece el término t.

Finalmente, tenemos que tf-idf se calcula como:

$$tfidf(t,d,D) = tf(t,d) \times idf(t,D)$$

Con este método, obtendremos un peso alto con una elevada frecuencia de término (en un atributo dado) y una pequeña frecuencia de ocurrencia del término en la colección completa de atributos de texto del mismo tipo. Con esto evitamos que palabras muy comunes tengan peso en la distancia, pero que sean palabras específicas las que determinen si dos textos son similares.

5.4 Híbrido

El algoritmo híbrido, como su nombre bien indica, es una combinación de los métodos de recomendación que hemos usado hasta ahora. Es decir, que combina la estrategia de collaborative filtering con la de content based filtering para conseguir unas predicciones de carácter mixto. El procedimiento es muy simple, consta tan sólo de dos partes; la obtención de las predicciones parciales y el cómputo de la media final.

En la primera parte, primero ejecutamos los algoritmos de KMeans y Slope one, tal y como ya hacíamos con collaborative filtering. Con el método *getPredicciones* de la clase Slope one conseguimos un mapa que contiene, para cada ítem, o bien una predicción de recomendación de ése item, o bien un -1.0, indicando que no hay predicción para ése ítem porque no ha podido computarse.

De un modo similar, llamamos también a la clase NearestNeighbours y tras ejecutar el algoritmo obtenemos también las predicciones para cada ítem en un segundo mapa. A diferencia del mapa de predicciones de Slope One, éste mapa contiene una valoración para todos y cada uno de los ítems, dado que el algoritmo permite encontrar predicciones para todos, y no sólamente cuando se cumplen una serie de características (explicadas en apartados anteriores), como ocurre en Slope One.

Una vez disponemos de las predicciones, obtenidas tanto por collaborative como por content based filtering, podemos proceder a la última parte del enfoque híbrido para la obtención de predicciones. Lo único que hacemos en esta parte es calcular la media de las predicciones para cada ítem, para así conseguir un resultado *mixto*, o *híbrido* que considera tanto la similitud entre usuarios y sus gustos, como la semejanza entre los ítems.

6 RELACIÓN DE LIBRERÍAS

En cuanto a las librerías, hemos utilizado solamente librerías para el testing. Las librerías que hemos utilizado son:

- apiguardian-api-1.1.0.jar
- hamcrest-all-1.3
- junit-jupiter-5.7.0.jar
- junit-jupiter-api-5.7.0.jar
- junit-jupiter-engine-5.7.0.jar
- junit-jupiter-params-5.7.0.jar
- junit-platform-commons-1.7.0.jar
- junit-platform-engine-1.7.0.jar
- junit-platform-console-standalone-1.7.0.jar

No hemos utilizado otras librerías que no hayan sido mencionadas en el enunciado.

7 POSIBLES EXTENSIONES DEL TRABAJO

Para un futuro cercano, contamos con la posibilidad de añadir el uso del algoritmo Slope One Weighted, para mejorar nuestras predicciones. Además hemos pensado en implementar el Hot One para poder implementar la similitud coseno: una medida de la similitud existente entre dos vectores en un espacio que posee un producto interior con el que se evalúa el valor del coseno del ángulo comprendido entre ellos. Otra extensión que nos planteamos es guardar los datos ya preprocesados, de esta manera no tendríamos que procesar los datos cada vez que el usuario empiece el sistema.

Lo primero que pensamos cuando iniciamos el curso del trabajo fue en añadir un sistema de login de los usuarios para que ellos tengan su propia cuenta y ahí puedan ver las recomendaciones que han ido recibiendo por el sistema.

Además, habíamos pensado en la implementación de un sistema de jerarquías de usuarios, en los que dividiremos entre Administradores y Usuarios comunes. Éstos primeros tendrían la posibilidad de añadir ítems (y eliminar) al conjunto generado inicialmente, mientras que los usuarios no podrían. En su lugar contarían con una opción llamada “solicitar” con la que podrían solicitar la adición de un ítem a un administrador. Para evitar confusiones, el sistema aportaría una plantilla mediante la cual el usuario se sentiría más cómodo y guiado durante el proceso.

La plantilla podría constar (de forma sencilla) de los campos:

- Nombre ítem
- Tipo de ítem (según los tipos que nosotros ofrecieramos, añadiremos diferentes opciones como películas, series, móviles…)
- Creador / Director / … (dependiendo del ítem)
- Año de publicación (en caso de ser un ítem publicado, como películas)
- Enlace a su página web (en caso que tenga)
- Valoración personal
- Comentarios extra

El administrador sería el encargado de leer esta plantilla y decidir si el ítem es real y si es válido para añadirlo. Más adelante, con el tiempo, podríamos añadir una inteligencia artificial que comprobase, mediante búsqueda, si los ítems son válidos y de esta forma no haría falta tener administradores sino simplemente usuarios comunes.

8 BIBLIOGRAFIA

- https://ieeexplore.ieee.org/document/8686857
- https://www.baeldung.com/java-hashmap-sort
- https://www.researchgate.net/publication/1960789_Slope_One_Predictors_for_Online_Rating-Based_Collaborative_Filtering
- https://www.cs.upc.edu/prop/data/uploads/enunciatt2122.pdf
- https://www.cs.upc.edu/prop/data/uploads/info-recomanadors.pdf
- https://link-springer-com.recursos.biblioteca.upc.edu/content/pdf/10.1007%2F978-3-319-47121-1.pdf
- https://www.it-swarm-es.com/es/java/java-dividir-una-cadena-separada-por-comas-pero-ignorar-comas-entre-comillas/969032997/