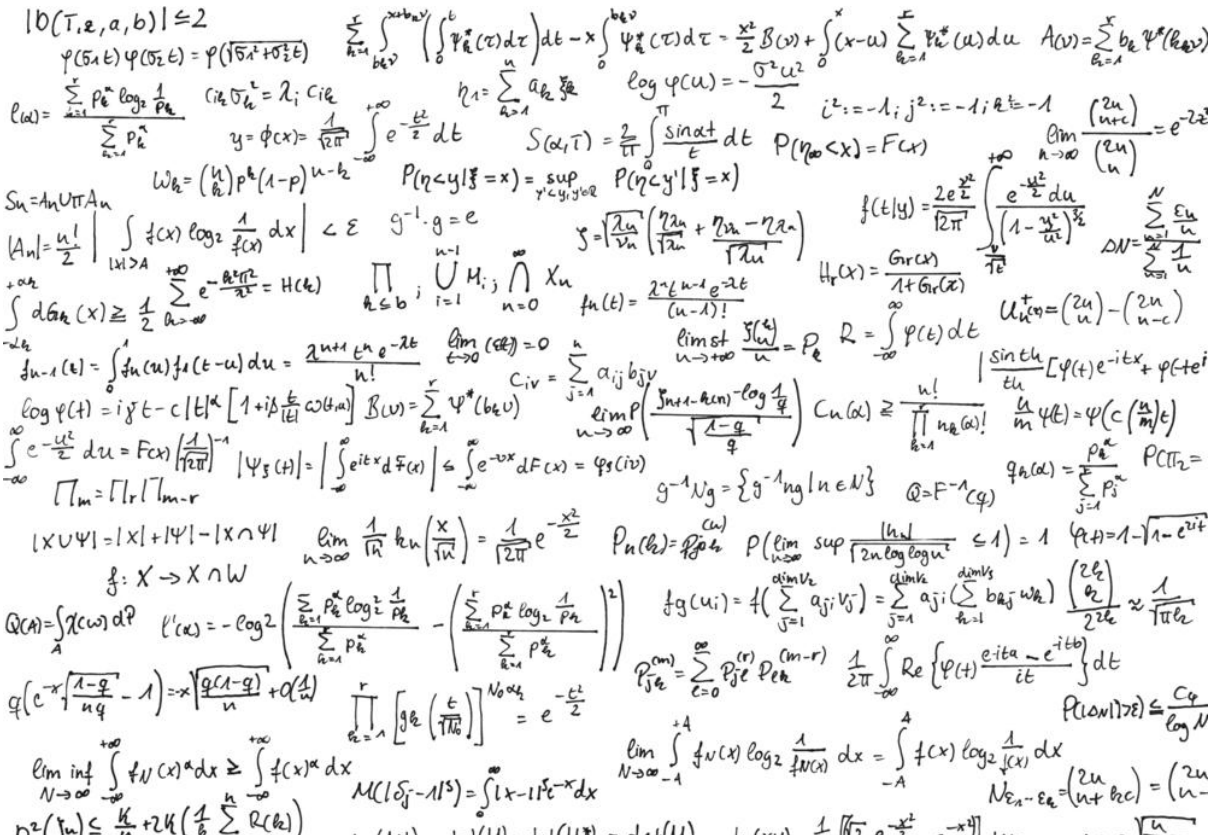
SISTEMA RECOMENDADOR



2ª Entrega

Proyectos de Programación (PROP)

15 de Diciembre de 2021

Grupo 2.3

Delgado Sanchez, Marc - marc.delgado.sanchez@estudiantat.upc.edu

Gutiérrez Jariod, Miguel - miguel.gutierrez.jariod@estudiantat.upc.edu

Martí Jariod, Oriol - oriol.marti.jariod@estudiantat.upc.edu

Piera Garrigosa, Manel - manel.piera@estudiantat.upc.edu

ÍNDICE

Clases del modelo conceptual de datos

Descripción de clases

Anotación: En el doxygen tenemos una descripción detallada de los métodos de cada clase. Hemos marcado con este color las secciones de la documentación que se han actualizado de la primera entrega a la segunda.

Persistencia:

CSVParserItem

La clase CSVParserItem se encarga de realizar la lectura de los ficheros csv que tienen un formato como los ejemplos items.csv y procesa los datos con la finalidad de facilitar las operaciones de los datos en los algoritmos.

Básicamente se utiliza la librería Scanner para poder leer el documento y mediante una expresión regex nos separa el documento de forma que nos proporciona una lectura cómoda de los datos para así procesarlos convenientemente para el uso de los algoritmos. Esta lectura la realizamos mediante una *List<List<String>>* llamada *content*. También contamos con una *List<String> header* que contiene la cabecera del documento, es necesaria para identificar los identificadores y para evitar que un elemento tenga dos filas.

Con respecto al preproceso de los datos hemos planteado el siguiente escenario: con tal de facilitar las operaciones de los algoritmos, hemos tenido en cuenta los tipos booleanos, enteros, doubles y los elementos categóricos para hacer más eficiente el trato de datos, además de los strings, aunque sean bastante ineficientes. Hemos creado un *Map<Integer, List<Content>> mapRateData* donde los datos están indicados en base la categoría del elemento comentado anteriormente. A continuación usamos la clase Content para poder identificarlos y almacenarlos correctamente en nuestra estructura.

Finalmente contamos con una *List<String> id_Items* que almacena todos los ids de los ítems teniendo en cuenta su posición en la lectura y procesado del documento csv.

CSVParserRate

La clase CSVParserRate se encarga de realizar la lectura de los ficheros csv que tienen un formato como los ejemplos ratings.csv y procesa los datos con la finalidad de facilitar las operaciones de los datos en los algoritmos

En esta usamos la librería Scanner para poder leer el documento y mediante una expresión regex, menos complicada que la de la anterior clase, nos permite separar el documento en un formato que nos otorga una mayor facilidad para la manipulación de los datos. Esta lectura la realizamos mediante una *List<List<String>>* llamada *content*. También contamos con una *List<String> header* que contiene la cabecera del documento, es necesaria para identificar el orden en los que entran los elementos para introducirlos correctamente en la estructura diseñada.

Para el preproceso de los datos hemos usado un *Map<Integer, Map<Integer,Float>> mapRate* para almacenar los de forma que por cada id de un usuario se tengan los id de los ítems que lo relacionan con su valoración correspondiente.

Dominio:

Content

En la clase content se dotan una serie de elementos para hacer posible el preproceso de datos de la clase CSVParseritem. En esta podemos encontrar un string, que sirve de identificador o de elemento para procesar el contenido. Un entero el cual es usado para determinar si el objeto anterior se puede convertir en este tipo y si es así atribuirlo. Un double donde se ejerce la misma funcionalidad que con el entero. Por último cuenta con una *List<String> categorics* donde se almacenan el conjunto de elementos que forman parte de este grupo.

K-NN

La clase K-NN implementa el algoritmo de k-nearest-neighbours para calcular recomendaciones usando la filosofía del content based filtering. El principal atributo de esta clase es la llamada similarityTable, una matriz de doubles que almacena las similitudes entre todos los ítems del sistema. Usando esta matriz, la clase hace uso del método k-nn para encontrar los k ítems más parecidos a un ítem dado (no valorados por el usuario).

También almacenamos el mapa que nos informa de los ítems que ha valorado cada usuario, así como de la valoración que le ha dado a cada ítem, y una lista de enteros, que almacena los id's reales de los ítems. Esta lista es necesaria porque, para hacer el cómputo del algoritmo, el método usa la posición de los ítems en la tabla de entrada del programa como id's, por una cuestión de comodidad y sinergia con la matriz de similitudes.
Para hacer una recomendación para un usuario dado, usamos el método k-nn sobre todos los ítems que el usuario ha valorado, y obtenemos los k ítems más parecidos a todos ellos. Una explicación más detallada se encuentra en el desarrollo del algoritmo de K-Nearest-Neighbours (sección 3.3).

K_Means

La clase K_means implementa el algoritmo *K means* (K medias) para organizar los usuarios en K grupos diferentes, estos grupos se constituyen por usuarios con opiniones parecidas. El único atributo de esta clase es *opinions*, el cual representa las valoraciones de los usuarios sobre los ítems. Esta representación se hace mediante un *Map<Integer, Map<Integer, Float>>.* En la sección 3.1 hay la explicación del algoritmo *K Means* para encontrar los susodichos K grupos.

Slope One

La clase Slope One implementa el algoritmo Slope One, el cual estima una valoración para un ítem no valorado de un usuario. Consta de dos métodos para calcular el algoritmo y un tercero para la llamada de estas y limpiar los datos. La clase consta de cuatro atributos, los cuales tres son matrices de maps y un map.
Hay el map de datos *map_data* que contiene todos los datos necesarios para calcular la predicción. Está representado por *Map<Integer, Map<Integer,Float>>*, que representan *UserID<ItemID,Valoración>*. Hay el map *map_des* que contiene las diferencias entre dos ítems tal que *Map<Integer,Map<Integer,Float>>* que se traduce a *ItemID_1<ItemID_2,Diferencia>*. Hay el map de frecuencias *map_freq* que se guardan las veces que hemos computado un par de items, se ve representado *Map<Integer,Map<Integer,Float>>* que toman el significado de *ItemID_1<ItemID_2,Veces_computados>*. Y finalmente el map de predicción *map_pred* el cual guarda la predicción hecha para el usuario que queremos. Está representado por *Map<Integer,Float>* que tienen el valor de *<ItemID,Valoracion_estimada>*. En el apartado 3.2 hay una explicación más extensa y que profundiza más en el funcionamiento y cálculo del algoritmo.

CollaborativeFiltering

La clase CollaborativeFiltering implementa la combinación de los algoritmos *K Means* y *Slope One* para crear una recomendación para un usuario al que llamaremos A. En esta se usa el primer algoritmo para agrupar los usuarios parecidos en los mismos grupos. Posteriormente localizamos el conjunto de usuarios al que pertenece A y usamos este grupo de usuarios como input para el algoritmo *Slope One*. *Slope One* nos calcula entonces unas predicción de las valoraciones que haría A de los ítems que no ha valorado, de las cuáles elegimos las 10 más altas para nuestra recomendación.

Evaluation

La clase Evaluation implementa el cálculo de la evaluación de las recomendaciones. A esta clase se le introduce la recomendación y un conjunto de valoraciones del usuario al que estamos haciendo la recomendación que inicialmente eran desconocidas para hacer la recomendación(unknown). La evaluación es el cálculo del Discounted Cumulative Gain, cuya fórmula es:

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

En la que se tiene una recomendación de *p* ítems ordenada de forma decreciente. La *i* representa la posición del ítem en la recomendación. El valor *reli* es la valoración que el usuario ha dado al ítem en la posición *i* de la recomendación. Cuanto mayor sea el DCG mejor es la recomendación.

Rating

La clase *Rating* representa una valoración de un usuario de un ítem determinado, tiene dos atributos, un *int* que es el ID del ítem valorado y un *float* que es el valor de la valoración.

Recommendation

La clase *Recommendation* implementa la representación de una recomendación a un usuario. Sus atributos son una lista de *Rating* y un *int* que representa el ID del usuario al que se le hace la recomendación. Los algoritmos de las clases *CollaborativeFiltering, K_NN,* y *Hybrid* devuelven instancias de esta clase.

Hybrid

La clase *Hybrid* implementa la combinación de la recomendación mediante el *CollaborativeFIltering* y *K_NN*. Esta clase tiene dos atributos, una instancia de la clase *CollaborativeFiltering* y una instancia de la clase *K_NN*. La función principal de *Hybrid* es *recommend()* la cual devuelve una *Recommendation* que es la mezcla entre los dos tipos de recomendaciones anteriores, explicada su implementación en el apartado **Hybrid Algorithm**.

Descripciones de los algoritmos

K-Means

La aplicación del algoritmo de K means se hace en la clase *K-Means*. En primer lugar se usa la creadora de la clase pasando como parámetro un Map<Integer,Map<Integer, Float>> que representa las valoraciones de los usuarios de los ítems y se lo atribuimos al atributo *opinions* de la clase. Después de esta inicialización ejecutamos el método *k_means* para encontrar los los k clústers de usuarios, para ello usamos el algoritmo más común que usa una técnica iterativa.

Consideramos que los usuarios forman un vector en el cual cada ítem representa una componente. Las componentes no nulas para un usuario son las valoraciones de los ítems que ha valorado. Las medias representan "usuarios medios" de cada conjunto de usuarios y por lo tanto también se entienden como vectores.

En primera instancia, inicializamos los k conjuntos cuyos índices van de 0 a k-1. Para ello recorremos el Map opinions atribuyendo a cada usuario un conjunto de forma que el usuario que aparece en la i-ésima posición, empezando desde i = 0, pertenece al (i%k)-ésimo conjunto. Una vez ya tenemos los k conjuntos inicializados, debemos calcular sus medias de forma que

$$M_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j$$

dónde S_i es el conjunto i-ésimo y M_i su media. No hay que olvidar que esto no son valores escalares sino que son vectores y realmente estamos sumando sus componentes.

Ahora entramos en la parte iterativa del algoritmo. Esta parte tiene dos fases:

1. Fase de asignación.
2. Fase de actualización.

Fase de asignación: Calculamos la distancia entre los usuarios y las medias y colocamos cada usuario en el conjunto que tiene la media más cercana (menor distancia). En nuestro caso no hemos usado una función de distancia sino que hemos usado una función de similitud, el coseno cuadrado, que calcula el cuadrado del coseno del ángulo que forman los vectores del usuario y la media. Consecuentemente cada usuario es asignado al conjunto con el que tiene mayor similitud con la media.

Fase de actualización: Volvemos a calcular la media de los conjuntos con la fórmula que hemos usado anteriormente.

Estos dos procesos se repiten indefinidamente hasta que en la fase de asignación los conjuntos creados son los mismos que en la iteración anterior.

Slope One

La aplicación del algoritmo de Slope One se hace en la clase *SlopeOne*. El algoritmo consiste en estimar/predecir una valoración para un ítem concreto que le daría un usuario, a partir de las valoraciones de ese ítem de los otros usuarios.

El cálculo del algoritmo empieza con la función *slopeone* que le llega por parámetros una matriz *Map<Integer,Map<Integer, Float>>* que representa las valoraciones de los usuarios de los ítems y se lo atribuimos al atributo *map_data*, que será nuestra base de datos para el cálculo. Y un *Map<Integer,Float>* denominado *user* que representa las valoraciones de los ítems del usuario al que queremos estimar las valoraciones que aún no ha realizado en ciertos ítems.

Este método básicamente centraliza la ejecución del algoritmo, haciendo llamadas a los dos métodos principales del cálculo.

Primero llama a la función *desviacio_mitjana()*. Aquí nos calculará con nuestra matriz *map_data* la desviación de las valoraciones de cada ítem con los otros ítems, por usuario. Lo que nos construirá una matriz *map_des* simétrica con la diagonal con valor 0.

Seguidamente llama a la función *prediccio(user)* que se le pasa por parámetro el map del usuario que estimaremos sus valoraciones. Y nos llenará el map *map_pred* con las estimaciones calculadas por cada ítem, los ítems que sí que había evaluado el usuario y los que no, solamente nos interesan estos últimos.

Así que el último procedimiento del método *slopeone* es un bucle que itera a través del map *user* y elimina los items que sean compartidos entre este map y *map_pred* dejando solamente los ítems estimados. Finalmente retorna *map_pred*.

Desviacio_mitjana:
En este método nos computa la matriz de desviaciones de cada ítem respecto los otros ítems. El resultado será el de una matriz simétrica con la diagonal a 0. La fórmula para el cálculo es la siguiente.

$$dev_{j,i} = \sum_{u \in S_{j,i}(X)} \frac{u_j - u_i}{card(S_{j,i}(X))}$$

Siendo u_j la valoración del usuario u al ítem j, menos u_i siendo la valoración del usuario u al ítem i, donde u es el vector de las valoraciones del usuario u.

De aquí sacamos la diferencia, y lo dividimos por $card(S_{j,i}(X))$ el conjunto de usuarios que han valorado j e i. Obtenemos la desviación media.

Aplicada esta fórmula al código, tenemos que iterar primeramente por todos los usuarios de la matriz de datos, y a través de ella por todos los datos de cada usuario. Añadimos el ítem que tratamos, si no estaban, a las matrices de desviación y frecuencia, esta última nos servirá para calcular las cardinalidades de cada conjunto de usuarios. Y tenemos que volver a iterar a través de los datos de un segundo usuario. Entonces calculamos la diferencia entre las valoraciones de los dos usuarios, y lo guardamos en la matriz de desviaciones, pero no sin antes haber comprobado y sumado si es necesario con las diferencias para ese ítem acumuladas de otros usuarios anteriores.

Vamos sumando las veces que hemos tratado aquel ítem para un par de usuarios y se guarda en la matriz de frecuencia, teniendo en cuenta la acumulación de este valor para casos anteriores. Para finalizar después de haber recorrido todos los usuarios. Recorremos una última vez la matriz de desviaciones, y con la diferencia la dividimos con la cardinalidad del mapa de frecuencias y obtenemos la desviación media de cada ítem.

Predicció:
En este segundo método calculamos la segunda parte del algoritmo, la parte de la predicción de evaluación de ítem. El algoritmo con pesos, nos propone la siguiente fórmula para hacer la predicción.

$$P_{wso} = \frac{\Sigma_{i \in (I(u) - [j])}(Dev_{i,j} + R_{ui}) * Num(U_{i,j})}{\Sigma_{i \in (I(u) - [j])} Num(U_{i,j})}$$

Tenemos que calcular un sumatorio por todos los ítems valorados por el usuario, sumando las desviaciones del ítem más su valoración y multiplicando por el peso de este, que simplemente es la frecuencia del ítem.Y dividimos entre la suma de estas frecuencias. Y este cálculo nos da la predicción con pesos del ítem.

K-Nearest-Neighbours

La aplicación del algoritmo de K-Nearest-Neighbours se hace en la clase *K-NN*, usando la matriz de similitudes entre todos los ítems del sistema.
La clase contiene un método de inicialización de la matriz de similitudes, que se ejecuta cuando leemos el conjunto de ítems de entrada. La inicialización consiste en calcular las similitudes entre todos los ítems dados y almacenarlas en la matriz.

El cálculo de las similitudes entre ítems se hace comparando sus tags, que se reciben como instancias de la clase Content. La similitud entre dos ítems será la suma de las similitudes entre todos sus tags. Para normalizar este valor y mantenerlo en un rango entre 0 y 1, la similitud entre dos ítems distintos se divide entre la similitud entre un ítem y él mismo (que siempre será máxima en una misma fila de la matriz).

En función del tipo del tag (booleano, entero, double, categórico o string), el valor de la similitud es calculado de manera distinta y tiene una importancia diferente.
En nuestro caso, se valoran como más importantes las coincidencias en los tipos booleanos, seguido por los tipos categóricos, estos seguidos por los enteros y los doubles, y finalmente los strings.

Las coincidencias entre dos valores booleanos y dos valores categóricos aportan una similitud que se calcula de forma absoluta; la similitud prende valor máximo si los valores coinciden y, en cambio, prende valor mínimo (0) si no coinciden.
Las coincidencias entre dos valores enteros y dos valores double, por otra parte, aportan una similitud que se calcula de forma relativa a la variancia entre los valores dados, es decir, la similitud puede tomar cualquier valor entre el valor máximo (que se tomará si los números son iguales) y el valor mínimo (0). La similitud será igual a:

$$(1 - x) * X$$

siendo x la varianza entre los dos números dados, X el valor máximo que puede tomar la similitud entre enteros/doubles.

Una vez calculadas las similitudes entre todos los ítems del sistema, cada vez que queramos encontrar los k elementos más similares a un ítem dado, únicamente tendremos que recorrer la fila de la matriz correspondiente al ítem dado, y quedarnos con los k elementos con mayor similitud.

Este preciso procedimiento se aplica en el método kNN, el cual dado un id de un ítem x y un k, retorna los k elementos más parecidos a x. Esto lo conseguimos haciendo uso de una cola de prioridad que almacena parejas de id's y similitudes. Para implementar esta cola de prioridad de parejas, se ha implementado una clase ordenable Pair, que almacena un entero (el id del item con el que se compara) y un double (la similitud entre los dos ítems). La cola de prioridad se ordena en función de la similitud, dejando los Pairs con menor similitud como los primeros que saldrán en caso de llegar un Pair con mayor similitud. Al terminar la iteración sobre la fila del ítem x, el método kNN retorna una lista con los k ítems más cercanos a x.

Además, este método podemos aplicarlo para calcular una recomendación para un usuario dado, llamémosle T. El método *recommend*, que recibe como parámetros el id del usuario que quiere una recomendación (T en este caso), y cuantos ítems quiere en la recomendación (k) se encarga de calcular una recomendación de k elementos para el usuario T. Tomando todos los ítems que ha valorado T (a los que llamaremos x_i), calculamos los k elementos más cercanos a cada uno de ellos (a los que llamaremos x_ij). Para cada elemento presente en las listas de los k-NN, computamos la similitud global entre ese ítem y el usuario T. Esta similitud global se computa multiplicando la similitud entre un x_ij y x_i por la valoración que el usuario T le ha dado al ítem x_i, para todos los x_i.

SECCIÓN NUEVA:

Para normalizar el valor de la similitud global, lo dividimos por la suma de las similitudes entre el ítem x_ij y todos los ítems valorados por el usuario T en los que ha aparecido como uno de los *k* ítems más cercanos. De esta manera, nos aseguramos que el resultado del cálculo es un número que se encuentra entre los límites de las valoraciones de los usuarios. Por lo que podemos considerar esta similitud global como una "predicción" de la valoración que daría el usuario para el que estamos calculando la recomendación, lo que implica que es comparable con el resultado del otro algoritmo de recomendación, el collaborative-filtering.

Es decir, la similitud global entre x_ij (ítem no valorado por T) y T será:

$$Sim(x_{ij}, T) = \frac{\sum_{i=0}^{n} (sim(x_{ij}, x_i) * val(x_i))}{\sum_{i=0}^{n} sim(x_{ij}, x_i)}$$

Calculado este valor para todos los x_ij presentes en todos los vectores k-NN (donde puede haber x_ij repetidos, no causa ningún problema), lo único que nos falta es tomar los k ítems con mayor similitud global entre ellos y el usuario T.

El cálculo del algoritmo se realiza usando la posición de los ítems en la tabla de entrada como identificador de los ítems, por una cuestión de comodidad y sinergia con la matriz de similitudes. Es por ello que, para retornar el id real de un ítem, necesitamos una lista que nos traduzca la posición de la tabla que ocupa un ítem a su identificador real. Esta lista es el otro parámetro global de la clase, que debe pasarse como parámetro al instanciar un objeto de la clase.

Hybrid Algorithm

La implementación del *Hybrid Algorithm* se hace en la clase *Hybrid*. Para ello necesitamos dos recomendaciones. En primer lugar, ejecutamos la recomendación del *Collaborative Filtering* que implementa el algoritmo *Slope-one* con los clústers hechos por el *K-Means* y posteriormente calculamos la recomendación con el algoritmo *K-NN* (Content Based Filtering). Para la nueva recomendación, los primeros ítems serán los que sean comunes a las dos recomendaciones, con los que realizaremos la multiplicación de sus dos *expected ratings* y ordenados de forma decreciente según este resultado. Si con esto no llegamos a suficientes ítems, que será el caso en la mayoría de ocasiones, elegiremos los ítems que tengan el mayor *expected rating* de las dos recomendaciones hasta llegar al número deseado.

Estructuras de datos y costes

Decisión estructuras de datos

ArrayList

ArrayList es una clase propia de Java muy similar a los arrays de los lenguajes de programación, pero esta nos permite añadir y quitar elementos sin tener en cuenta el tamaño del mismo. La hemos tenido en cuenta de cara a la agregación o eliminación de elementos de las listas que hemos desarrollado.

List

Para la lectura de los documentos csv hemos optado por el desarrollo de las listas. Más concretamente hemos creado una List<List<Strings>> de forma que las listas de Strings contienen los elementos del csv, separados correspondientemente, y situados de la misma forma tal y como dispone el documento. Finalmente la lista que engloba el conjunto de las anteriores nos permite hacer referencia a las filas, como dispone el formato del documento csv. A priori hemos pensado que esta era una de las mejores formas de poder realizar la lectura. Una vez finalizado dicho proceso hemos elegido una nueva estructura con tal de tener la información de una forma más eficiente, ya que las consultas de las listas pueden resultar costosas.

TreeMap

La estructura que hemos decidido utilizar es el TreeMap. Principalmente porque hemos visto que la entrada de elementos es un tanto significativa y al no saber si este número se podía expandir, optar por esta opción que nos ofrece coste en tiempo log(n) nos parecía conveniente. Es cierto que por el momento la ordenación podría no tenerse mucho en cuenta, pero al tratar con una cantidad de datos significativa también nos ha hecho remarcar el uso de esta opción.

Esta clase está implementada a partir de un árbol binario de búsqueda equilibrada, esto significa que el equilibrio depende de las etiquetas, al agregar o eliminar los elementos. Por lo tanto el equilibrio es importante para mantener un buen rendimiento, que corresponderá directamente con la altura del árbol. Hemos de evitar que se desequilibre el árbol con tal de dar una altura mayor y que empiece así a afectar al rendimiento. Nos garantiza entonces que el tiempo de los métodos contains, get o put serán 0(log(n)).

El principal uso ha sido para la organización de la información de los documentos csv. Para el csv de los ítems hemos desarrollado un Map<Integer, List<Content>> el cual la clave hace referencia a la fila con la que se ha obtenido dichos elementos y la lista contiene el conjunto de valores leídos previamente pero modificados con tal de proporcionar una mejor resolución a los algoritmos. Para el caso de los otros documentos, al tener definidos el conjunto de valores, hemos implementado un Map<Integer, Map<Integer, Float>> atribuyendo la clave del primer map el id del usuario, en la segunda clave el id del ítem y el valor del último la valoración correspondiente.

Hashmap

Esta clase no la hemos tenido en cuenta de cara a la implementación pero creemos que es una buena idea resaltar sus características para hacer una mejor comparación y ver porque la opción que hemos escogido resulta más adecuada. El HashMap ofrece ventajas en términos de rendimiento, que es constante para operaciones como get y put, pero ante un crecimiento de la estructura a través del tiempo se debería de tener en cuenta ya que podría cambiar su coste.

Los dos factores a tener en cuenta son la carga y la capacidad. La primera se refiere a la amplitud de los “buckets” y la segunda al número de “buckets” que se han creado. A medida que crece el número de estos se ha de rehashear para crear más “buckets”, lo que puede ser una operación costosa dependiendo del número de entradas. Si el factor de carga se mantiene por debajo del 75% la implementación en java funciona mejor, lo que quiere decir es que para mantener el rendimiento siempre se asigna más memoria de la que se necesita para el almacenaje de las entradas, por eso debemos usarlo exclusivamente si se cumplen los siguientes aspectos: queremos un coste en tiempo constante, tenemos una idea de la dimensión de la colección y no añadiremos o eliminaremos los elementos con regularidad.

Costes de las diferentes funciones

CSVparserItem

Estas son las funciones con un coste más relevante:

- **obtain_id_header(List<String> header) :** Esta función que obtiene la posición del elemento donde se encuentra el id, tiene un coste lineal O(n), siendo n el tamaño de header, ya que debe recorrer la lista.
- **readLoadItem():** Esta función tiene un coste cuadrático ya que al encargarse de leer el documento csv necesariamente necesita recorrer todos los elementos.
- **MapItemData(List<List<String>> rate_content):** Esta función también tiene un coste cuadrático ya que tiene que recorrer cada elemento del rate_content para procesarlo como dato para los algoritmos.

CSVparserRate

Estas son las funciones con un coste más relevante:

- **obtain_id_header(List<String> header) :** Esta función que obtiene la posición del elemento donde se encuentra el id, tiene un coste lineal O(n), siendo n el tamaño de header, ya que debe recorrer la lista.
- **readLoadRate():** Esta función tiene un coste cuadrático ya que al encargarse de leer el documento csv necesariamente necesita recorrer todos los elementos.
- **LoadRate(List<List<String>> rate_content):** Esta función también tiene un coste cuadrático ya que tiene que recorrer cada elemento del rate_content para procesarlo como dato para los algoritmos.

Content

No tiene funciones con costes relevantes, se usa como soporte para el desarrollo de la clase CSVparserItem.

K_NN

Estas son las funciones con un coste más relevante:

- **initSimilarityTable(Map<Integer, List<Content>> map):** La inicialización de la tabla de similitudes entre ítems tiene coste pseudo-cuadrático, ya que se aprovecha el hecho de que la matriz es simétrica (antes de aplicar la normalización) para computar la (n^2)/2 operaciones de cálculo de similitud. Por lo que podemos decir que este método tiene coste O(n^2), lo cual es asumible ya que solo se ejecuta una vez, cuando inicializamos el programa.
- **calculate_similarity(List<Content> list1, List<Content> list2):** Dadas dos listas de tamaño n, el cálculo de la similitud es lineal con n, ya que la comparación de tags se hace uno a uno y las dos listas se recorren simultáneamente. En caso de recibir un tag multivaluado, la comparación entre estos también se hace de manera lineal con el número de evaluaciones del tag, ya que se emplea un algoritmo de comparación basado en el cálculo de la intersección de las dos listas en tiempo lineal. Por lo que podemos concluir que este método tiene coste O(n).
- **kNN(int id_item, int k, int id_usuari):** El método para el cálculo de los k vecinos más cercanos a un ítem dado es lineal con el número de ítems, ya que recorremos la fila de la matriz de similitudes del ítem dado. Considerando las comparaciones, las asignaciones y la gestión de la cola de prioridad como O(1), podemos concluir que este método tiene coste O(n).
- **Map<Integer,Float> recommend(int id_usuari, int k):** El método para hacer una recomendación basada en el content based filtering a un usuario dado tiene coste pseudo-cuadrático (en la versión actual), ya que se calculan los k vecinos más cercanos de todos los ítems que el usuario ha valorado. En el caso peor, en el que el usuario ha valorado todos los ítems del sistema (menos un conjunto pequeño, para poder recibir una recomendación), el método kNN (de coste O(n)) se ejecutaría n veces, dando lugar a un coste de O(n^2). Ahora bien, este coste se puede reducir fácilmente a un coste lineal si, en vez de calcular los k ítems más parecidos de todos los que ha valorado el usuario, solo lo hiciéramos de los t ítems con mejor valoración.
  Aunque es sabido que esta implementación reduciría el coste de ejecución drásticamente en casos desfavorables, se ha valorado que, para esta primera entrega, las entradas no tendrán casos muy desfavorables y la precisión en la recomendación se valora más positivamente que el tiempo de ejecución.
  En caso de cambiar de opinión en el futuro, la implementación de esta modificación es sencilla y se aplicará sin problema.

Pair

No tiene funciones con costes relevantes, se usa como soporte para el desarrollo de la clase K_NN, concretamente para la implementación de una cola de prioridad que almacene dos valores.

K_Means

Estas son las funciones de coste más relevante:

- **cosineSquaredSimil(Map<Integer, Float> u1, Map<Integer, Float> u2):** Si consideramos que los dos usuarios tienen *m* valoraciones esta función tiene un coste O(m*logm).

- **equalClusters(Vector<Vector<Integer>> nuevoClusters, Vector<Vector<Integer>> Clusters):** En el peor caso hay que recorrer todos los usuarios y por lo tanto el coste es lineal, O(n) siendo n el número de usuarios.
- **k_means(Integer k):** En esta función tenemos el bucle *while* que se ejecuta un número indefinido de veces. Como el coste depende de muchos factores consideraremos que tenemos *n* usuarios con cuyo número medio de valoraciones es *m,* para el cálculo consideraremos que todos los usuarios tienen el mismo número de valoraciones. Dentro de este bucle, el proceso más costoso es calcular a qué cluster pertenecen los usuarios. Todos los usuarios deben encontrar la media más cercana ejecutando la función cosineSquaredSimil así que tenemos un coste O(m*logm)*n = O(n*m*logm). Suponemos que el bucle *while* no se ejecutará más de *n* veces ya que como mínimo un usuario llegará a su cluster correcto en cada iteración. Tenemos entonces un coste de $O(n^2*m*logm)$.

CollaborativeFiltering

Sólo ejecuta funciones de las clases *K_Means* y Slope One. La creadora, con parámetros, tiene el mismo coste que *k_means*, $O(n^2*m*logm)$. La función *recommend* tiene el mismo coste que *Slope One.*

Hybrid

La clase *Hybrid* sólo es una mezcla de *CollaborativeFiltering* y *K_NN,* por lo tanto el coste de esta clase es el mismo que el mayor de estas dos clases. El coste es entonces el máximo entre $O(n^2*m*logm)$ dónde n es el número de usuarios y m el número medio de valoraciones por usuario y $O(n^2)$ dónde n es el número de ítems en el sistema.

Evaluation

Funciones de coste más relevante:

- **Evaluation(Map<Integer, Float> unknown, Map<Integer, Float> recommendation):** El coste de esta función es ordenar la recomendación, así que tenemos un coste O(n*logn), siendo *n* el número de ítems recomendados.

- **DCG():** Esta función sólo itera a través de toda la recomendación, tiene un coste lineal, O(n).

SlopeOne

Estas son las funciones de coste relevante en la clase:

- **slopeone(Map<Int,Map<Int,Float>> data, Map<Int,Float> user):**
  En esta función solamente tenemos un bucle de tamaño n por lo tanto el coste es lineal O(n).

- **desviacio_mitjana():** En esta función nos encontramos con un triple bucle, cada bucle recorre el map entero, uno de usuarios y los otros dos de sus datos. En el caso peor sería que el número de usuarios y de ítems valorados por los usuarios fuese el mismo, es decir n. Al recorrer los maps al completo, tenemos que el coste de esta función es O(n^3).

- **prediccio(Map<Int,Float> u_data):** En esta función tenemos un doble bucle al calcular la mediana de desviaciones del ítem respecto al que estimaremos. En el caso peor es que el tamaño de los conjuntos de ítems a calcular sea el mismo, es decir n. Al recorrer los dos bucles al completo nos sale que el coste de esta función es O(n^2).

En resumen, el coste del algoritmo sería el de O(n^3).

Relaciones entre clases y miembros del equipo

Miguel:
CSVparserItem
CSVparserRate
Content
ControladorPersistencia

Marc:
K-NN
Pair
ControladorDominio
Hybrid
Recommendation

Manel:
K-Means
CollaborativeFiltering
Evaluation
ControladorDominio
Rating
Recommendation
Hybrid

Oriol:
SlopeOne
ControladorPresentación