Recomendador

Estructuras de datos y algoritmos usados

Gerard Madrid Miró
Guillem Gràcia Andreu
Ismael Quiñones Gama
Pol Ken Galceran Kimura

Collaborative Filtering:

Estructuras de datos utilizadas:

- **ArrayList<ArrayList<Integer>> Kgroups:** Vector de vectores de UsersID. Cada posición del primer vector agrupa todos los UsersID de aquellos usuarios que comparten gustos similares.
- **HashMap<Integer, ArrayList<Rating>> ratingsMap:** Map en el que la clave es un cierto UserID y el valor es un vector de valoraciones (donde cada valoración consiste de un ItemID y un score). De esta forma podemos encontrar un cierto usuario con un coste muy bajo y poder acceder a todos sus ratings.
- **ArrayList<ArrayList<Rating>> centroids:** Vector de vectores de ratings. Cada posición *pos* del primer vector se refiere al centroide del clúster *pos* (Recordemos que en Kgroups hay K clusters distintos). Ese centroide está compuesto por un vector de ratings que representa la media de todos los ítems valorados por los usuarios de ese clúster.
- **HashMap<String, ArrayList<Rating>>**

Algoritmos utilizados:

- **Kmeans:** Este algoritmo trata de separar a los usuarios de ratings.csv en clusters, agrupando a todos los usuarios con gustos similares en un mismo cluster. Para ello se cogen K centroides aleatoriamente y se mira la distancia de todos los usuarios hacia cada uno de los centroides, de forma que podamos agrupar a los usuarios en el cluster de aquél centroide con menor distancia. Una vez hecho esto, se recalcula el centroide haciendo que ese centroide sea igual a un usuario ficticio el cual habrá valorado todas las películas que han valorado los usuarios que forman parte de ese cluster y, para cada una de ellas, tendrá una valoración igual a la media de las valoraciones de esos usuarios. Una vez hecho esto, se vuelve a mirar las distancias de todos los usuarios hacia los centroides volviéndolos a agrupar. Se repite todo lo comentado anteriormente hasta que llegamos a un punto en el que ningún usuario cambia de cluster. Ahí decimos que ha "convergido".
- **SlopeOne:** Este algoritmo trata de predecir la nota de un cierto usuario a todos los items que no ha valorado. Para ello primero cogemos la información de los usuarios similares y construimos una matriz con todas las valoraciones de esos usuarios sobre los items que SÍ ha valorado el usuario activo (cada fila contiene la información de un usuario y cada columna representa un ítem) y otra con todas las valoraciones de esos usuarios sobre los ítems que NO ha valorado el usuario activo. Para cada uno de los ítems que no ha valorado se calcula la desviación media de nota entre el ítem en cuestión y otro cierto ítem del sistema el cual sí había valorado el usuario activo. Entonces se predice la nota de un ítem a partir

de la suma de la nota del usuario activo sobre el ítem que sí había valorado y la desviación media que se ha obtenido con los datos de los usuarios similares entre el ítem a predecir y el otro ítem que sí había valorado el usuario activo. A partir de aquí tenemos una predicción hecha sobre un ítem basándonos únicamente en otro cierto ítem que sí había sido valorado por el usuario activo, por lo que finalmente hacemos la media entre todas esas predicciones parciales (ya que cada predicción se hizo en base a otro cierto ítem valorado por el usuario activo, por lo que tenemos tantas predicciones parciales como ítems hubiese valorado el usuario activo, y los ítems valorados por el usuario activo usados para cada predicción parcial son disjuntos entre ellos). Una vez hecha la media entre todas las predicciones parciales ya obtenemos la predicción final definitiva.

Content-based Filtering:

Estructuras de datos utilizadas:

- **ArrayList<Float> ponderations:** Array de valores en Float comprendidos entre 0 y 1. Cada posición hace referencia a un atributo de los ítems. A mayor valor tenga el atributo en su posición, mayor peso tendrá a la hora de calcular la similitud entre Ítems. Se llena mediante un algoritmo en ComputePonderations().
- **HashMap<Integer, ArrayList<Rating>> ratingsMap:** Map en el que la clave es un cierto UserID y el valor es un vector de valoraciones (donde cada valoración consiste de un ItemID y un score). De esta forma podemos encontrar un cierto usuario con un coste muy bajo y poder acceder a todos sus ratings.
- **ArrayList<Item> allItems:** Array de Items con todos los items existentes en el sistema. Se usa para poder comparar el ítem base a comparar con todos los demás y para calcular las ponderaciones.

Algoritmos utilizados:

K Nearest: Este algoritmo compara items hasta devolver los K con menor distancia (más grado de similitud) a un ítem base dado.

Jaccard: Este algoritmo compara dos strings dados hasta devolver un valor entre 0 y 1. Este valor resulta de la división de la intersección entre la unión de los caracteres de ambos strings.

ComputePonderations: Este algoritmo mide cuán útil es un atributo para comparar dos ítems. Si un atributo tiene el mismo valor para muchos de los ítems en el sistema, no es útil para comparar. De igual manera, si un atributo tiene siempre valor distinto tampoco es útil.

Para más información en cualquiera de los tres algoritmos, recomiendo ver el pseudocódigo presentado en el fichero correspondiente, dónde está argumentado cada paso.

Hybrid Filtering:

El funcionamiento de este algoritmo de recomendación se basa en mirar cuáles han sido los items que un cierto usuario ha valorado positivamente (>=4.0) y pedirle a Content-based filtering un cierto número de ítems con similitud mayor al 60% de esos items que le gustaron al user.
Una vez hecho eso, pedir una cierta cantidad de ítems recomendados al Collaborative Filtering para ese usuario y buscar ahí todos los elementos que nos había retornado el Content-based. Por tanto, nos quedaremos con los ítems parecidos a aquellos items que le habían gustado al usuario y que además el collaborative filtering ya nos quería recomendar, y en el mismo orden en el que el collaborative nos lo quería dar.
En caso de que esos ítems con los que nos hemos quedado sea menor al número de recomendaciones que quería el user, lo completamos con las primeras recomendaciones del collaborative filtering (a no ser que ya hubiésemos cogido esos ítems).

Evaluación de calidad recomendadores a través del DCG:

IDCG: Para el IDCG ordenamos los ítems de unknown en orden decreciente respecto a sus notas de unknown y aplicamos la siguiente fórmula:

$$\mathbf{DCG}_{i,P} = \sum_{d=1}^{q} \frac{2^{\mathbf{Unknown}[i][j_d]} - 1}{\log_2(d+1)}$$

DCG: Para el DCG ordenamos los ítems de unknown en orden decreciente respecto a las notas predichas por programa y aplicamos la fórmula anterior. Si hemos predicho el mismo orden que en el IDCG, tendremos que DCG = IDCG. Si no, tendremos que DCG < IDCG.

NDCG: Es la división entre DCG e IDCG. El valor está entre 0 y 1 y cuanto más cercano esté al 1 significa que hemos hecho una mejor recomendación.

Gestión de Datos

Estructuras de datos utilizadas:

- **DataItem items:** La clase DataItem define un ArrayList de ítems y en HashMaps para optimizar la búsqueda de éstos con nuevas funcionalidades. Por lo tanto almacena los datos de los Item guardados en disco.
- **DataRating ratingsX:** De igual manera, la clase DataRating define un ArrayList de ratings y diferentes HashMaps para optimizar la búsqueda de éstos con nuevas funcionalidades. Por lo tanto las declaraciones ratingsDB, ratingsTestKnown, ratingsTestUnknown, que guardan el contenido de los archivos.
- **DataUser systemUser:** Contiene todos los usuarios del sistema almacenados en un ArrayList y en diferentes HashMap para optimizar la búsqueda de éstos.
- **DataAlgorithm algorithmData:** Contiene todos los datos necesarios para almacenar la información generada por el algoritmo KMeans.

Interfície gráfica

Estructuras de datos utilizadas:

- **SceneManager**: Stack de Escenas y Stack de Ítems
    - Para gestionar las distintas ventanas a las que va accediendo el usuario y permitir volver hacia atrás, hemos implementado en la clase SceneManager dos estructuras de datos nuevas. Estas estructuras son pilas (Stacks) que gestionan por un lado las escenas que ha visitado el usuario, y por otro lado los Ítems que ha visitado. Esta última de Ítems permite ir hacia atrás en las ventanas de distintos ítems dónde la escena sería la misma.
    - La primera almacena el path relativo de la escena en cuestión para permitir un cambio de escena rápido cuándo se le da al botón de “Atrás”.
    - La segunda almacena el nombre del Item que puede ser enviado a la función de cambio de escena a ítem para permitir un cambio rápido al ítem anterior cuándo se le da al botón de “Atrás”.

- **GetStartedController**: Arrays de itemIDs y Valoraciones
    - Para gestionar los gustos del usuario en la ventana de GetStarted (donde se valoran 5 ítems para obtener sus gustos), se usa un Array de ints para guardar los identificadores de esos Ítems, y un Array de floats para guardar la valoración a los mismos. Se usan dos arrays distintos ya que se quieren los dos valores para funcionalidades separadas y no era necesario agruparlos.

- **RecommendMeController**: Array de itemIDs
    - En este caso, se deberán guardar todas las recomendaciones dadas al usuario en un array de itemIDs para poder guardarlas si así se pide o para no dar exactamente las mismas más adelante.