Memoria práctica 4

Alejandro Pascual Pozo y Víctor Yrazusta Ibarra

Apartado 1:

Una vez desarrollada la interfaz hemos implementado un modelo de datos basado en conjuntos hash, ya que no vemos particularmente interesante mantener un orden en los datos y preferimos optimizar los accesos a la información.

También hemos decidido implementar el método getCorrecto() que permite conocer si el modelo de datos ha podido completar correctamente su última carga desde fichero.

Apartado 2:

En este apartado hemos seguido el algoritmo propuesto sin mayores complicaciones.

Como optimización en las llamadas a recomienda() hemos decidido guardar la lista de "vecinos" de cada usuario la primera vez que se pide una recomendación para él. De esta manera nos ahorramos el cálculo dependiente de constantes en futuras llamadas.

Hemos pensado que podría ser interesante, de ser necesario optimizar más el tiempo de ejecución, almacenar directamente las recomendaciones. Esto podría venir acompañado de un sistema similar al que gestiona el crecimiento de las ArrayList generando recomendaciones con cierto margen cada vez que nos pidan más de las que tenemos almacenadas para un determinado usuario.

Apartado 3:

Hemos implementado ambos recomendadores siguiendo las reglas descritas. Nos hemos asegurado de que no puedan recomendar al usuario un ítem que ya haya comprado.

Apartado 4:

Hemos definido la interfaz e implementado ambas versiones tal y como vienen descritas en el enunciado.

Apartado 5:

Hemos implementado el test y hemos podido observar que ambas métricas dan el mismo resultado: el sistema de recomendación por "vecinos" es el más acertado, seguido del sistema basado en la popularidad global. El modelo aleatorio es, como cabía esperar, el que peor rinde de los tres.

Hemos diseñado el test de tal manera que hacemos dos llamadas distintas a la función recomienda() para cada usuario y cada recomendador. Hemos podido comprobar que aunque la optimización que hemos hecho acelera algo la segunda ronda de llamadas, haciendo que tarde aproximadamente la mitad, sigue suponiendo un cálculo costoso.

Apartado opcional:

Para implementar el sistema de serialización hemos desarrollado una clase abstracta (RecomendadorSerializable) que extiende Recomendador y es implementada por las diferentes versiones de este. Esta clase crea un fichero en la ruta especificada con las recomendaciones generadas para cada usuario, cada uno de estos conjuntos de recomendaciones está ordenado en base a la puntuación de dicha recomendación.

Para implementar el sistema de carga hemos desarrollado una clase abstracta (MetricaDeFichero) que extiende Metrica y es implementada por las diferentes versiones de esta. Esta clase carga la información del fichero especificado y asume que todas las recomendaciones de un mismo usuario se encuentran ordenadas de mayor a menor en base a la puntuación, además de ser consecutivas.

Para comprobar el correcto funcionamiento de la serialización y carga de las recomendaciones hemos desarrollado una prueba similar a la del apartado 5, aunque solo del recomendador por “vecinos”, que permite especificar si se quiere primero generar las recomendaciones y después cargarlas o si se desean cargar directamente de un fichero. Los resultados obtenidos han sido los mismo que en el apartado 5.

Diagrama de clases: