Análisis y Diseño de Software

Practica 4



Tabla de contenido

[Apartado 1. Modelo de datos 3](#_Toc5551603)

[Apartado 2. Recomendador de vecinos 3](#_Toc5551604)

[Apartado 3. Otros recomendadores 3](#_Toc5551605)

[Apartado 4. Evaluar recomendaciones 4](#_Toc5551606)

[Apartado 5. Comprobar que todo funciona correctamente 4](#_Toc5551607)

[Diagrama de clases 5](#_Toc5551608)

[Nota 5](#_Toc5551609)

# Apartado 1. Modelo de datos

Para implementar la interfaz ModeloDatos hemos creado la clase ModeloDatosRecomendacion, en ella para crear el método leeFicheroPreferencias() hemos modificado el método de lectura de ficheros de la practica anterior corrigiendo el erros de uso de números mágicos. Para guardar los usuarios con la score que le dan a cada ítem y los ítems con la score que le dan cada usuario hemos utilizado dos mapas, preferenciasAllUsuarios y preferenciasAllItems respectivamente, simplificando así los métodos getPreferenciasUsuario(), getPreferenciasItem(), getUsuariosUnicos(), getItemsUnicos().

# Apartado 2. Recomendador de vecinos

Para implementar la interfaz Recomendador hemos creado la clase RecomendadorGeneral que contiene el método protegido itemsInUsuario() y el modelo de datos que se utiliza para los cálculos de la recomendación, de la que hereda RecomendadorVecinos. Este crea un método privado userGetItemScore() para facilitar la obtención de los scores de cada ítem en el método recomienda(). Este utiliza la implementación de Similitud dada por SimilitudCoseno que sigue la formula en el enunciado para calcular cuanto se parecen dos usuarios. Se almacenan los cálculos juntos con los usuarios con los que se han calculado gracias a la clase Tupla y se ordenan de mayor a menor gracias a que hemos sobrescrito el método compareTo() comparando las scores de la similitud. Luego se calcula para cada objeto cual es el score que se debería de dar tras calcularlo con simulación y se vuelven a guardar en una segunda lista de tuplas que se ordenan también de mayor a menor. Por último, se van extrayendo tantos ítems como los indicados en el argumento de la función recomienda de mayor a menor y se almacenan en la clase Recomendación, que guarda una lista de estos pares ítem y su score y el id del usuario del cual es la recomendación y se devuelve.

A la hora de comparar las tuplas y ordenar de mayor a menor la lista de tuplas solo comparamos el score de estas, sin tener en cuenta la id y de ocurrir empates se mantendrían como empate, pues nos da igual que valor es mayor o menor de los dos puesto que son iguales. Además como se nos indica si longitudRecomendacion es menor igual que 0, hace un throws de la señal que hemos creado RecomendacionInvalida con el mensaje de error "La longitud de la recomendacion tiene que ser mayor que 0 (longitud = "+longitud+")" y si el usuario no existe en el modelo de datos, se hace un throws también de RecomendaciónInvalida con el mensaje "El usuario "+user+" no existe".

# Apartado 3. Otros recomendadores

RecomendadorAleatorio y RecomendadorPopularidad como RecomendadorVecinos son 2 clases que heredan de RecomendadorGeneral e implementan el método de recomienda() devolviendo ellas también una Recomendación, que guarda una lista de estos pares ítem y su score y el id del usuario del cual es la recomendación. La score en RecomendadorAleatorio se calcula aleatoriamente utilizando la función Math.random() y en RecomendadorPopularidad la score es igual al número de usuarios que han consumido ese ítem. Además como en recomienda() de RecomendadorVecinos también lanzan la excepción RecomendacionInvalida en los mismos casos de error.

# Apartado 4. Evaluar recomendaciones

Para implementar la interfaz Metrica hemos creado la clase MetricaGeneral que contiene el método publico getItemsRelevantes() y el modelo de datos que se utiliza para los cálculos de la métrica, de la que hereda MetricaPrecision y MetricaRecall. Estas dos clases contienen cada una el método evalua() que mediante la operación en el enunciado comprueba si las recomendaciones son buenas o no devolviendo un valor entre 0 y 1.

En evalua() para obtener los ítems relevante utilizamos el método getItemsRelevantes() que devuelve un Set con la lista de ítems relevantes para ese usuario y la lista de tuplas de la recomendación a evaluar. De estar el Set de ítems relevantes vacio se lanza la excepción UsuarioNoRelevante con el mensaje "El conjunto de items relevantes ha de ser mayor que 0 (n = "+n+")". Tras eso comprobará con contains si itemsRelevantes tiene alguno de los Id de recomendaciones y dependiendo del tipo de metrica lo dividirá entre una cosa size de itemsRelevantes o n que son los primeros n articulos de la lista obtenida del recomendador.

# Apartado 5. Comprobar que todo funciona correctamente

Para comprobar que todo funcionase correctamente, hemos creado un main en Apartado5.java que crea y evalúa todas las recomendaciones con los datos que se nos pone en el enunciado, (número de vecinos = 100, nota mínima para considerar un ítem relevante = 3, tamaño de las recomendaciones = 5) que vienen definidos como atributos finales y estáticos fuera del main. También se nos pedía modificar alguno de los parámetros y ver qué efectos tiene en las métricas hemos decidido crear otras dos métricas para los recomendadores teniendo una nota mínima de 4 (NOTA\_MIN + 1) en vez de 3. Y tras cada ejecución se imprimiría por la terminar el tipo de recomendación, el valor de la métrica de precisión y el valor de la métrica recall.

# Diagrama de clases

El .png de la imagen se encuentra en el .zip si se quiere ver más grande y a mejor calidad.

# Nota

Documentación en ./doc, código fuente en ./src y ficheros utilizados en Moodle.

Compilar y ejecutar desde el directorio ./src