Universidad Autónoma de Madrid
Escuela politécnica superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

ESTUDIO Y APLICACIÓN DE ALGORITMOS Y ESTRUCTURAS DE DATOS A LOS SISTEMAS DE RECOMENDACIÓN

Autor: Pablo Sánchez Pérez
Tutor: Alejandro Bellogín Kouki
Ponente: Fernando Díez Rubio

Mayo 2016

ESTUDIO Y APLICACIÓN DE ALGORITMOS Y ESTRUCTURAS DE DATOS A LOS SISTEMAS DE RECOMENDACIÓN

Autor: Pablo Sánchez Pérez
Tutor: Alejandro Bellogín Kouki
Ponente: Fernando Díez Rubio

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo 2016

Resumen

Resumen

Los sistemas de recomendación se emplean en un gran número de aplicaciones debido al potencial que tienen para ayudar a las personas a elegir artículos en función de sus necesidades o gustos. En la era actual de las comunicaciones, son especialmente útiles en Internet, ya que es el principal medio empleado por los usuarios para intercambiar información. Además, los programas que trabajan con esta cantidad de datos, necesitan proporcionar los resultados en el menor tiempo posible, haciendo uso de un gran número de estructuras y algoritmos para operar de manera eficiente.

Este Trabajo de Fin de Grado consiste en un estudio y una investigación sobre dichos sistemas de recomendación, haciendo especial énfasis en los de filtrado colaborativo basados en vecinos. Se ha llevado a cabo la implementación de un sistema que permite ejecutar algunas medidas de similitud utilizadas en el campo, como la similitud coseno o la correlación de Pearson (con algunas variantes) y que además permite realizar ejecuciones con distintas estructuras de datos conocidas.

Por otro lado, se incluye un estudio sobre los distintos parámetros a configurar tanto en los algoritmos como en las estructuras, para evaluar qué combinación es la más beneficiosa. También, se ha llevado a cabo la adaptación e implementación al campo de la recomendación del algoritmo de la subcadena común más larga, para así comprobar su funcionamiento frente a las medidas de similitud anteriormente comentadas. Para comprobar el rendimiento de estos algoritmos con distintas configuraciones, se han efectuado pruebas con un conjunto de datos real, en concreto con Movielens (100.000 puntuaciones). Los resultados han sido obtenidos efectuando comparaciones mediante métricas de error (RMSE o MAE, centradas en comprobar la diferencia entre las puntuaciones reales y las predichas) y mediante métricas de ranking (precision, recall, NDCG y MRR, orientadas a identificar qué artículos son relevantes a partir de una lista de recomendados). En cuanto al análisis de las estructuras, se han realizado mediciones tanto de tiempo como de memoria consumidos en las ejecuciones.

Como resumen, hemos observado que las tablas hash son las estructuras que más memoria utilizan y no siempre son las que menos tiempo emplean, por lo que en ciertas partes de la aplicación son peores que otras estructuras más sencillas. Asimismo, se ha mostrado que en algunos casos (métricas de ranking) el algoritmo que se propone en este trabajo de la subcadena común más larga obtiene mejores resultados que el resto de medidas de similitud, aunque en el caso de las métricas de error, la correlación de Pearson logra un error menor. Este trabajo ha resultado en una publicación en un congreso nacional donde se reportan los resultados sobre el análisis de tiempo y memoria de las estructuras para recomendación [1].

Palabras Clave

Sistemas de recomendación, Filtrado colaborativo, Algoritmo, Estructura de datos, Métricas de similitud.

Abstract

Recommender systems are used in many applications due to the fact that they have potential to help people by selecting items according to their needs or tastes. In the current era of communications, they are especially useful on the Internet, which is the most important application used by users to exchange information. In addition, programs that work with this amount of data, need to provide the results in the shortest time possible, using a large number of structures and algorithms to operate efficiently .

This Bachelor's Thesis is a study and investigation on these recommendation systems, with particular emphasis on neighbourhood-based collaborative filtering ones. We have developed a system that is capable of running some similarity measures used in the field, such as cosine similarity or the Pearson correlation (with some variations), allowing different known data structures being plugged in the process.

On the other hand, a study of the various parameters is included to configure both algorithms and structures in order to achieve the most beneficial combination. Besides, a new algorithm in the field of recommendation has been implemented, the longest common subsequence algorithm, to test it against the other measures previously discussed. To check the performance of these algorithms with different configurations, some test with real data have been done, particularly with Movielens (100,000 ratings). The results have been obtained by making comparisons using error metrics (RMSE and MAE, focused on checking the difference between the actual and predicted scores) and using ranking metrics (precision, recall, NDCG, and MRR, oriented to identify which items are relevant to the user from a list of recommended items). As for the analysis of structures, measurements have been made in terms of time and memory used in the executions.

As a summary, we have observed that the hash tables are the structures that more memory uses and they do not always spend less time, so in certain parts of the application they are worse than other simpler structures. Furthermore, we show that in some cases (ranking metrics) the algorithm proposed herein of the longest common subsequence obtains better results than other common measures of similarity, although in the case of error metrics, the Pearson correlation, however, achieves the lowest error. This thesis has resulted in a publication in a national congress where results about time and memory analysis of data structures for recommendation are reported [1].

Key words

Recommender systems, Collaborative filtering, Algorithm, Data structures, Similarity measure.

Agradecimientos

En primer lugar quiero dar las gracias a Alejandro Bellogín, por ofrecerme hacer este trabajo y haberme animado a investigar acerca de los sistemas de recomendación. Su respaldo durante el desarrollo de este proyecto ha sido indispensable para poder llevarlo a cabo con éxito.

Gracias también a mis compañeros y amigos de la facultad con los que he compartido muchas horas tanto en clase como realizando las prácticas en los laboratorios, especialmente a Guillermo Sarasa, Alejandro Catalina y Sergio Sanz. Espero que a pesar de haber emprendido caminos distintos, sigamos manteniendo nuestra amistad en el futuro.

Por último, me gustaría agradecer a mi familia el apoyo que me ha brindado no sólo durante la carrera sino a lo largo de toda mi vida. Siempre me han ayudado, sobre todo en los momentos más difíciles, han respetado mis decisiones y las han apoyado.

Índice general

Índice de Figuras

Índice de Tablas

Glosario de acrónimos

- **IB**: Item based. Estrategia basada en artículos.
- **LCS**: Longest common subsequence. Algoritmo para obtener la subcadena común más larga de entre dos entradas.
- **MAE**: Mean absolute error. Métrica de evaluación.
- **MC**: Mean centering. Medida de normalización.
- **MRR**: Mean reciprocal rank. Métrica de evaluación.
- **NDCG**: Normalized discounted cumulative gain. Métrica de evaluación.
- **RMSE**: Root-mean-square error. Métrica de evaluación
- **SR**: Sistemas de recomendación. Técnicas que sugieren artículos a los usuarios.
- **Std**: Standard. Ejecución sin ninguna normalización.
- **UB**: User based. Estrategia basada en usuarios.

1 Introducción

Desde el surgimiento de Internet en 1969 se ha producido una auténtica revolución en el ámbito de las comunicaciones, sobre todo a partir de la década de 1990 en la que su uso comenzó a extenderse alrededor del mundo. Según [2], en 2014, el 40 % de la población mundial (es decir, más de 2.800.000.000 de personas) tiene conexión a la red y dicho número continúa aumentando (hoy la cifra supera los 3.000 millones). Ante esta situación, cada vez más empresas proporcionan sus servicios tanto en sus instalaciones físicas como en Internet e incluso algunas de ellas sólo disponen de versión online. En un gran número de estas compañías, debido a la cantidad de productos que pueden ofertar, es indispensable poder filtrar a los usuarios los artículos en función de lo que potencialmente puedan necesitar o demandar. La principal forma de detectar esos productos es mediante el uso de los sistemas de recomendación (SR).

Los sistemas de recomendación son herramientas software que *sugieren* ciertos artículos a los usuarios. Estas propuestas de los SR tienen como objetivo ayudar al usuario a decidir qué objeto elegir de entre un amplio número de posibilidades. Los objetos, conocidos como *items* (o artículos en castellano) dependen de la empresa o institución que los ofrezca, siendo los libros, la música y las películas algunos de los más conocidos e investigados por la comunidad científica.

Muchas compañías y aplicaciones mundialmente conocidas como YouTube[1], Amazon[2] o Netflix[3] emplean SR. Esta última alcanzó gran fama en 2009 con un premio en el que ofrecía un millón de dólares a aquel equipo que consiguiese mejorar sus algoritmos de predicción. "Pragmatic Chaos", de BellKor, se alzó con el premio con un algoritmo que mejoraba en un 10 % las predicciones de los algoritmos propios de la empresa [3]. Aunque finalmente el algoritmo ganador nunca llegó a implementarse debido al trabajo de ingeniería necesario para llevarlo a cabo [4], la competición fomentó la investigación en el campo de los sistemas de recomendación.

Estos sistemas requieren de estructuras y algoritmos para el manejo eficiente de sus datos. Netflix en 2015 tenía unos 60 millones de suscriptores [5] y Amazon, en el mismo año quintuplicaba esa cifra [6]. Por esta razón, el almacenamiento de estos usuarios y de los productos tiene que ser de la manera más eficiente posible para calcular aquellos ítems que pueden resultar atractivos para el conjunto diverso de usuarios que confían en dichas empresas. Por otro lado, cuanto mejor y más útiles sean esas predicciones, más usuarios podrán convertirse en clientes de la empresa. Como aseguró Jeff Bezos, director ejecutivo de Amazon.com "If I have 3 million customers on the Web, I should have 3 million stores on the Web." [7].

[1] `http://www.youtube.com` [2] `http://www.amazon.com` [3] `https://www.netflix.com`

1.1. Motivación del proyecto

La motivación del proyecto, en este caso, es doble. Por un lado se busca un acercamiento al estudio de los SR (tipos de SR, métricas empleadas, aproximaciones, ...) y por otro se analiza el empleo de estructuras y algoritmos generales conocidos para poder aplicarlos en los SR.

Por ello, la bibliografía fundamental del trabajo son los libros "Introduction to Algorithms" [8] y "Recommender Systems Handbook" [9]. En el primero, se detalla un gran número de algoritmos y estructuras de datos que se emplean en general en toda la rama de la ingeniería informática. En cuanto a las estructuras, se incluyen tanto estructuras simples (árboles binarios de búsqueda, heaps o listas enlazadas) como estructuras más complejas (árboles estadísticos, heaps de Fibonnacci o estructuras de conjuntos disjuntos). También se relatan algunos de los problemas más conocidos y algunas soluciones y aproximaciones (en forma de algoritmos) para resolverlos.

El segundo libro, en cambio, es específico para los SR. En él, se explican los distintos tipos de SR que hay (colaborativos, basados en contenido, demográficos...), y los mecanismos más habituales que se emplean para poder efectuar recomendaciones a un usuario determinado, así como sus posibles evoluciones para seguir mejorando futuras recomendaciones (generalmente orientadas a mejorar el acierto de las preferencias del usuario, aunque también interesa mejorar otros aspectos como la diversidad de las sugerencias o el tiempo empleado en generarlas).

1.2. Objetivos y enfoque

El objetivo principal del TFG es encontrar el nexo de unión entre estructuras y algoritmos obtenidos del libro "Introduction to Algorithms" [8] y los empleados en una rama específica de la informática como son los sistemas de recomendación, detallados en el libro "Recommenders Systems Handbook" [9]. Por ello, es necesario efectuar un estudio de ambos libros para detectar qué elementos de [8] pueden ser aplicados en los SR, con un enfoque orientado a la novedad (¿qué estructuras o algoritmos no se han usado aún?) y horizontal (aplicable a cualquier aspecto de los SR: almacenamiento de datos, predicción, evaluación, etc.).

1.3. Estructura de la memoria

El presente documento está dividido en cuatro partes fundamentales. En el estado del arte, se hará un resumen tanto de las estructuras de datos y algoritmos más importantes como de la información que hay actualmente acerca de los sistemas de recomendación. En el capítulo de sistema desarrollado, se explica la aplicación que se ha realizado para el estudio de los sistemas de recomendación haciendo uso de la teoría, las estructuras y los algoritmos enunciados en el estado del arte. Hay también una sección de experimentos en la que se incluye tanto el resultado obtenido al realizar pruebas con conjuntos de datos reales como el análisis del rendimiento de las distintas ejecuciones del sistema para varias configuraciones. Para finalizar el cuerpo del TFG, se muestran las conclusiones obtenidas y se comentan los trabajos futuros a llevar a cabo.

En última instancia, los anexos proporcionan más información acerca de algunas de las secciones del documento, como imágenes de las estructuras desarrolladas, pseudocódigos de los algoritmos y algunos resultados específicos.

2

Estado del arte

2.1. Estructuras de datos

En cualquier tipo de aplicación que opera con datos es necesario diseñar estructuras que sean capaces de trabajar eficientemente con ellos. Algunas clásicas como las listas o las colas pueden ser suficientes para aplicaciones específicas, pero para el almacenamiento o la búsqueda rápida de una gran cantidad de elementos, son claramente inadecuadas. Debido a la importancia de un tratamiento eficaz de los datos, además de emplear ordenadores potentes, es necesario utilizar estructuras más elaboradas. A continuación se muestran algunas de las descritas en el libro "Introduction to Algorithms" ([8]) (tanto básicas como complejas).

2.1.1. Pilas

Una pila es una estructura de datos en la que se accede a los elementos siguiendo una política LIFO (*last in first out*), es decir, que el primer elemento a extraer será siempre el último elemento insertado en la estructura. Las operaciones fundamentales de una pila son la de apilar (insertar) y desapilar (extraer), que manipulan el puntero que indica el último elemento de la pila (en cada inserción se desplazará el puntero hacia la siguiente posición y en caso de eliminación, se desplazará al lugar inmediatamente anterior).

2.1.2. Colas

Si las pilas seguían una política LIFO, las colas siguen una política FIFO (*first in first out*), es decir, el primer elemento que entra en la cola será el primer elemento en salir. En esta estructura hay dos punteros, uno que hace referencia al lugar donde insertar (llamado *tail*) y otro que indica cuál es el primer elemento que se insertó (llamado *head*). Por tanto, cuando se inserta un nuevo elemento, se inserta en la posición de *tail* y se actualiza el puntero a la siguiente posición. En el caso de la extracción, se elimina el elemento head y se actualiza dicho puntero de la misma manera. Cuando tanto *head* como *tail* coinciden, la cola está vacía.

Existe un tipo especial de colas, llamadas *colas de prioridad*. En ellas, los elementos se atienden en función de una determinada prioridad asociada a cada uno de ellos. Suelen ser implementadas a partir de un heap, por lo tanto serán explicadas en el apartado 2.1.9.

2.1.3. Listas enlazadas

Una lista enlazada es una estructura de datos compuesta por nodos. Cada nodo dispone de una clave (elemento que va a guardar) y de un puntero al siguiente nodo de la lista. La lista suele tener dos punteros, uno al inicio del primer nodo y otro al final de la lista, para poder insertar los elementos tanto al principio como al final de la lista con un tiempo constante ($O(1)$). Aunque en las listas enlazadas simples sólo es posible desplazarse hacia delante (de un nodo al siguiente), los nodos pueden incorporar un puntero hacia su antecesor, permitiendo que la lista sea recorrida en ambas direcciones (en este caso, se les denomina listas doblemente enlazadas).

2.1.4. Árboles binarios de búsqueda

Un árbol binario de búsqueda (*BST, Binary Search Tree* en inglés) es un árbol binario en el que el subárbol situado a la izquierda de cualquiera de sus nodos se compone obligatoriamente de elementos menores que el elemento de dicho nodo y el subárbol situado a su derecha estará conformado por los elementos mayores. El coste para las búsquedas, inserción de elementos y eliminación de elementos depende de la altura del árbol, por lo que cuanto menor sea dicha altura, más rápido se realizarán estas operaciones. Normalmente, en un árbol binario balanceado (árbol en el que las alturas de los subárboles de cada nodo se diferencian como máximo en 1), la altura será $O(\log(n))$ donde n indica el número de nodos. Para las búsquedas, supone una notable mejora respecto a las estructuras básicas anteriormente comentadas, en donde el coste era lineal ($O(N)$, donde N es el número de elementos almacenados en la estructura). No obstante, en su caso peor, la altura del árbol coincidirá con el número de elementos y la búsqueda tendrá un coste lineal, igual que el de las listas enlazadas (las inserciones ordenadas provocarán que los elementos se inserten siempre en el mismo subárbol).

2.1.5. Árboles binarios rojo-negros

Un árbol rojo-negro es un árbol binario de búsqueda auto-balanceado. Es decir, un árbol que intenta mantener su altura lo más baja posible en cada momento (después de cada inserción/eliminación de un nodo). Un árbol binario rojo-negro tiene la misma estructura que un árbol binario común (elementos menores a la izquierda y elementos mayores a la derecha) y además cumplen las siguientes restricciones:

- Todos los nodos son rojos y/o negros.
- La raíz siempre es un nodo negro.
- Todas las hojas son negras.
- Si un nodo es rojo, sus dos hijos son negros.
- Para cada nodo, todos los caminos hasta sus hojas tienen el mismo número de nodos negros.

Estos árboles son más complejos que los árboles binarios de búsqueda normales, ya que las inserciones y eliminaciones intentan mantener la condición de árbol balanceado. No obstante, permiten que tanto la inserción, la eliminación y la búsqueda tengan *siempre* un coste $O(\log(n))$, donde n indica el número de nodos. Esto agiliza en gran medida las búsquedas, a cambio de que las inserciones y las eliminaciones sean más complejas debido a las operaciones necesarias para intentar mantener dicho balanceo.

Al insertar un nodo en un árbol rojo-negro se realiza la misma acción que en un árbol binario de búsqueda (moverse entre los subárboles izquierdo y derecho hasta encontrar una hoja libre), poniendo el nuevo nodo insertado como rojo. En el caso de las eliminaciones, también se opera de la misma forma que en los árboles binarios. No obstante, ambas operaciones pueden violar las reglas comentadas al principio, por lo que después de insertar o eliminar un nodo, es necesario comprobar que el árbol sigue teniendo la estructura correcta y en caso de no ser así, modificar el árbol hasta conseguirlo. Los procedimientos para mantener las propiedades explicadas anteriormente hace uso de *rotaciones* para lograr este objetivo. La complejidad de dichas rotaciones no depende del tamaño del árbol, solamente se cambian punteros, por lo que el coste es $O(1)$. Esto, unido a que las operaciones de balanceo sólo se realizan como máximo $O(log(N))$ veces, hace que no afecte al coste total de las inserciones/eliminaciones del árbol.

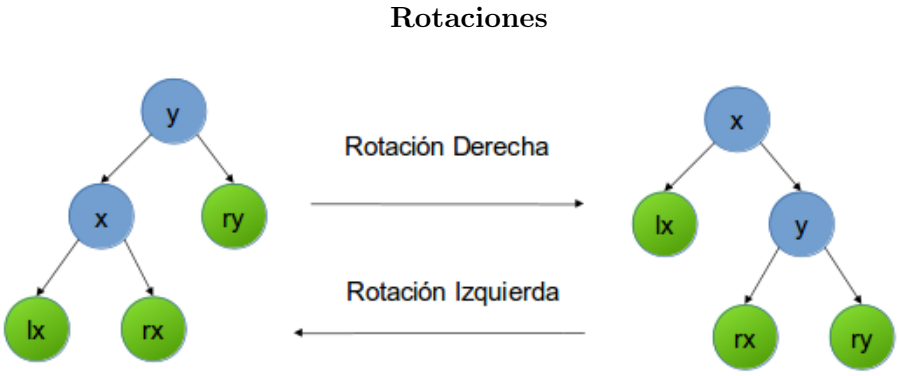

Figura 2.1: Rotaciones a izquierda y derecha.

En la figura A.1 se puede ver una figura de un árbol rojo-negro. En algunas aplicaciones, una estructura simple es suficiente para trabajar de manera eficiente con los datos. En pocas ocasiones se requiere una innovación total en lo que estructuras de datos se refiere, pero sí es más común realizar una ampliación de las ya existentes para permitir operaciones que con la estructura original no son posibles. En el caso de los árboles rojo-negros, hay dos ejemplos importantes de estructuras "aumentadas".

2.1.5.1. Árboles binarios rojo-negros estadísticos

Se basa en la implementación del árbol rojo-negro, es decir, es autobalanceable. Además de la información de los nodos del árbol rojo-negro, los nodos de esta estructura disponen de un campo más, el tamaño, que es la suma del tamaño del subárbol izquierdo más el tamaño del subárbol derecho más 1 (por él mismo), teniendo en cuenta que los nodos hojas se consideran de magnitud 1. Así pues, por ejemplo, para un árbol con tres nodos A, B y C siendo B la raíz, el nodo B tendrá dimensión 3 y cada hijo (A y C) tendrá dimensión 1.

Estos árboles permiten una acción que el árbol rojo-negro no permitía: la obtención del nodo i-ésimo del árbol (el i-ésimo elemento es aquel que tiene la clave iésima más baja en una ordenación 1,2 ... n), en un tiempo $O(\log(n))$. Esta información es útil si se quiere obtener un elemento específico a partir de su posición.

2.1.5.2. Árboles de intervalos

En estos árboles (también extensión de los árboles rojo-negros), cada nodo almacena un elemento que contienen un intervalo (dos valores t_1, t_2 con $t_1 \leq t_2$). Las operaciones soportadas por esta estructura son las mismas que las comentadas en el árbol binario rojo-negro:

- Insertar (elemento): inserta un elemento que contiene un intervalo en el árbol.
- Eliminar (elemento): borra un elemento del árbol de intervalos.
- Buscar (intervalo): busca el elemento cuyo intervalo se superponga con el intervalo que se está buscando (no busca el elemento que tenga el mismo intervalo). Dos intervalos se superponen si la intersección de ambos intervalos no producen el conjunto vacío.

La clave de cada elemento es el valor mínimo del intervalo, y junto al intervalo, también se almacena una variable que es el valor máximo que puede tener el extremo de un intervalo de cualquiera de sus hijos. Los intervalos resultan útiles para representar eventos que se producen en un intervalo continuo de tiempo. Por ejemplo, se pueden almacenar en esta estructura todos los eventos que se producen en unas horas determinadas (eventos ocurridos de 13:00 h a 13:05 h, de 13:05 h a 13:10 h ...) y posteriormente obtener todos aquellos que hayan ocurrido entre franjas más largas.

2.1.6. Árboles B

Los árboles B son árboles de búsqueda balanceados en los que todos los nodos hoja deben estar en la misma altura. Al contrario que los otros árboles comentados, los nodos no tienen por qué ser necesariamente binarios, sino que pueden tener un gran número de hijos. Las características más importantes de este tipo de árboles son las siguientes:

- Cada nodo posee un número n que indica el número de claves que tiene, un conjunto de claves ordenado de forma ascendente (*keys*) y una variable que indica si es una hoja o no.
- Todos los nodos, excepto las hojas, contiene también $n + 1$ punteros a sus hijos.
- Todas las hojas deben tener la misma profundidad.
- Los nodos tienen un límite inferior y superior sobre el número de claves que pueden almacenar. Estos límites determinan el grado mínimo del árbol (t), obligando a los hijos a tener como mínimo $t - 1$ claves y como máximo $2 \cdot t - 1$.

Debido a estas restricciones, la operación de inserción puede dar lugar a la separación de un nodo en dos, mientras que la operación de eliminación puede unir varios nodos en uno. Un ejemplo de árbol B se puede encontrar en la imagen A.2. Este tipo de árboles tienen aplicaciones en la creación de índices para las bases de datos, ya que permite unas búsquedas eficientes.

2.1.7. Estructura de conjuntos disjuntos

Una estructura de conjuntos disjuntos mantiene una colección de conjuntos dinámicos. Cada uno de ellos se identifica mediante un representante (un elemento que forma parte de dicho conjunto). Esta estructura soporta las siguientes operaciones:

- Crear conjunto (x): crea un nuevo conjunto con un elemento poniendo a dicho elemento como representante (requiere que el elemento x no esté en ningún conjunto).
- Unión (x,y): crea un nuevo conjunto a partir de dos conjuntos disjuntos.
- Encontrar conjunto (x): devuelve un puntero al representante del conjunto que contiene al elemento x.

La forma de almacenar los conjuntos distintos se puede realizar de varias maneras, aunque las dos más comunes son mediante listas y mediante árboles, siendo esta última más eficiente que la primera. En el caso de implementación mediante listas, el representante suele ser el primer elemento de la lista. Tanto la creación de un conjunto como la búsqueda (si cada elemento además de tener un puntero a su siguiente, tiene un puntero al primer elemento de la lista) son $O(1)$, pero en el caso de unión (una lista se concatena a otra), el coste sería $O(n)$. Dado que en la estructura de conjuntos se crearán n conjuntos y luego se efectuarán n - 1 uniones, se llegaría a una complejidad total de $O(n^2)$.

En el caso de los árboles, cada conjunto se puede considerar como un árbol en el que cada nodo dispone de un puntero hasta su padre, siendo el representante la raíz del árbol. No obstante, para buscar un elemento habrá que recorrer todo el árbol, haciendo que el coste dependa de la altura de éste, pudiendo tener el mismo coste que la versión basada en listas. Para reducir tanto el coste de las búsquedas como de las uniones, se emplean la unión por rango (para ello, cada nodo mantiene un rango que inicialmente es 0 y se va modificando a medida que se van uniendo) y la compresión de caminos (como se pude ver en la ilustración A.4), consiguiendo en el peor caso una complejidad prácticamente lineal ([8, pp. 571-574]).

2.1.8. Tabla hash

Las tablas hash son estructuras de datos que permiten obtener la búsqueda, eliminación e inserción de elementos en $O(1)$ para un caso medio. El eje vertebrador de las tablas hash se basa en la aplicación de una función cuyo valor devuelto sea equiprobable para todas las posibles entradas. Aunque las funciones hash deberían devolver un resultado distinto para cada entrada, normalmente se emplea un tamaño de hash menor, a cambio de producir colisiones, que pueden ser resueltas mediante mecanismos como direccionamiento abierto o encadenamiento. A pesar de sus ventajas en búsqueda, inserción y eliminación, las tablas hash tienen dos inconvenientes importantes. El primero es la dificultad si se quiere acceder a todos los elementos para recorrerlos y el segundo es que consumen gran cantidad de memoria si se reserva para la totalidad de los posibles elementos. Normalmente, para aplicaciones que emplean gran cantidad de datos, se suele emplear esta estructura para acceder de forma rápida a los datos necesarios.

2.1.9. Montículos binarios (heap)

Los montículos binarios son otra forma de almacenar los datos de un árbol binario, pero en este caso los elementos de los nodos son siempre mayores que sus dos hijos (en caso de ser un max-heap) o menor que sus dos hijos (en caso de ser un min-heap). Esto se puede conseguir insertando todos los elementos y después efectuando el procedimiento de *heapify* para la mitad de los elementos del heap. En lugar de almacenar los elementos en forma de árbol, los heaps permiten ser guardados en un array, de forma que el nodo raíz siempre se almacena en la posición 0 del array y los hijos de una posición k se almacenan en las posiciones $2k+1$ y $2k+2$ respectivamente. Esta disposición del array permite además efectuar un algoritmo de ordenación llamado *heapsort* que ordena dicho array en un tiempo de $O(n \log(n))$. Otra aplicación importante de los heaps es la posibilidad de crear una cola de prioridad. Usando un min-heap se crearían colas de prioridad mínima (el próximo elemento a extraer será el que tiene menor valor), mientras que un max-heap define colas de prioridad máxima (el próximo elemento a extraer será el que tenga mayor valor).

Debido a que las operaciones en una cola de prioridad máxima y mínima son simétricas, se detallará a continuación la implementación de una cola de prioridad máxima a partir de un max-heap.

2.1.9.1. Cola de prioridad máxima

Dado que un heap se puede almacenar como un array, la inserción de un elemento se realiza en la siguiente posición a la última ocupada del array y cuando todos los elementos han sido insertados, se realiza el procedimiento de heapify (en este caso, de max-heapify). La inserción en una cola de prioridad cambia, ya que es necesario que el array mantenga siempre la condición de max-heap. Por ello, antes de finalizar la inserción, es necesario llamar a un procedimiento de incrementar la clave (compara el elemento con su padre, cambiando la clave con éste cuantas veces sea necesario mientras la clave del elemento es mayor), para mantener dicha condición. El algoritmo se puede ver en A.3.

2.1.10. Heaps de Fibonacci

Los heaps de Fibonacci son heaps "mergeables", es decir, que además de soportar las operaciones básicas (crear e insertar), disponen de tres nuevas operaciones:

- Obtener mínimo: obtienen el mínimo elemento de la estructura.
- Unir: obtiene un nuevo heap a partir de dos.
- Extraer-mínimo: permite obtener el elemento cuyo valor es mínimo en el heap.

Además, el heap de Fibonacci implementa también otro método, el de decrementar la clave, que cambia la clave de un elemento por otra si y sólo si la nueva no es mayor que la actual.

Un heap de Fibonacci se define como un conjunto de raíces de árboles que cumplen la propiedad de min-heap. Es decir, que en todos los árboles, las claves de los nodos son menores que las claves de cada uno de los hijos. Además, se mantiene un puntero a la clave mínima del heap de Fibonacci. Todos los hijos de un nodo, además de disponer de un puntero a su padre y sus respectivos hijos, disponen de un puntero a los elementos de su izquierda y de su derecha (lista doblemente enlazada circular). En A.5 se puede encontrar un heap de Fibonacci.

2.2. Algoritmos

Junto con el almacenamiento de datos, el empleo de algoritmos en el ámbito tecnológico es de suma importancia. No se podría entender, por ejemplo, el enrutamiento de los paquetes en internet sin algoritmos que calculan las distancias mínimas entre los nodos de la red (algoritmos de distancias mínimas en grafos), o sería imposible el comercio electrónico sin los algoritmos criptográficos para proteger las transacciones bancarias. Aunque hay una cantidad enorme de algoritmos, hay algunos que, aunque son básicos, se emplean en cantidad de aplicaciones en algún paso intermedio. Algunos ejemplos de este tipo de algoritmos son los algoritmos de ordenación o de factorización de números. Dado que parte de este TFG está en encontrar algoritmos que se puedan utilizar en los sistemas de recomendación, se comentan de forma resumida algunos algoritmos interesantes del libro "Introduction to Algorithms" [8].

2.2.1. Algoritmos de ordenación

Estos algoritmos reciben una lista o un vector de elementos y devuelven una permutación de dicha entrada, cuyos elementos están ordenados según alguna relación de orden (estas relaciones suelen ser comúnmente el orden lexicográfico y el orden númerico). Los algoritmos de ordenación más comunes están explicados en la siguiente tabla.

Tabla 2.1: Tabla comparativa de algoritmos de ordenación.

Algoritmo	**Detalles**	**Coste**
Quicksort	Recursivo. Se basa en la elección de un pivote para ordenar los elementos en función de dicho pivote.	$O(n^2)$ en el peor caso, $O(n \log(n))$ en el caso medio
Mergesort	Recursivo. Se basa en la copia de sublistas a partir de la lista original, ordenando dichas sublistas para después volverlas a combinar.	$O(n \log(n))$
Heapsort	Ordena el conjunto creando un max-heap. A la hora de ejecutar el algoritmo, se irán extrayendo los elementos manteniendo la condición siempre de max-heap.	$O(n \log(n))$

2.2.2. Longitud de la subcadena común más larga

Este algoritmo se basa en encontrar la cadena más larga w tal que los caracteres de esta cadena se encuentren en el mismo orden en las cadenas X e Y, *no necesariamente consecutivos*. Dicho algoritmo genera una matriz bidimensional que tiene tantas filas como longitud de una de las cadenas más uno y tantas columnas como la longitud de la otra más uno. La forma de rellenar dicha matriz es la siguiente:

$$C[i,j] = \begin{cases} 0 & \text{si i=0 o j=0} \\ C[i-1,j-1]+1 & \text{si } i,j>0 \text{ y } X_i = Y_j \\ \text{máx}(C[i,j-1], C[i-1,j]) & \text{si } i,j>0 \text{ y } X_i \neq Y_j \end{cases} \tag{2.1}$$

En la figura A.6 se puede encontrar dicha fórmula en pseudocódigo.

Por ejemplo, para las cadenas de "radio" y "noria", la matriz sería la siguiente:

Tabla 2.2: Ejemplo matriz LCS.

	0	n	o	r	i	a
0	0	0	0	0	0	0
r	0	0	0	1	1	1
a	0	0	0	1	1	2
d	0	0	0	1	1	2
i	0	0	0	1	2	2
o	0	0	1	1	2	2

El último valor de la matriz (el de más a la derecha y abajo) indica la longitud de la subcadena común más larga (en este caso 2, debido a "ra"). Este algoritmo se emplea para medir la similitud entre dos entradas. Una de sus diversas aplicaciones es la comparación entre secuencias de ADN. Para obtener la cadena resultante, hay que recorrer la matriz hacia atrás, como se explica en la ilustración A.7.

2.2.3. Multiplicación de matrices en cadena

Si se dispone de un conjunto de N matrices, el orden en el que se realizan las multiplicaciones no afecta al resultado total (propiedad asociativa), pero afecta en gran medida al coste computacional de dichas operaciones. Supongamos que tenemos tres matrices de 10x100 (A_1),

100x5 (A_2) y 5x50 (A_3). Si las operaciones se realizan en el siguiente orden $(A_1A_2)A_3$, el número de multiplicaciones escalares será de 10x100x5 + 10x5x50 =7.500. En cambio, si el orden es $A_1(A_2A_3)$, el número de operaciones será de 100x5x50+10x100x50=75.000. Este algoritmo, al igual que el anterior, emplea la programación dinámica para obtener el orden óptimo de multiplicación de dichas matrices. El algoritmo asume que en la secuencia de matrices $A_1, A_2...A_n$, cada matriz A_i tiene dimensiones de p_{i-1}xp_i. Recibe como argumento una secuencia $p = p_0, p_1...p_n$. Por lo tanto, la longitud de dicha secuencia será siempre una unidad mayor al conjunto de matrices a multiplicar. El algoritmo, además, emplea dos tablas bidimensionales. La tabla m que almacena los costes intermedios (costes en multiplicaciones escalares) y la tabla s, en donde en cada s[i,j] se almacena el valor k para indicar el lugar óptimo de colocación del paréntesis a la hora de efectuar la multiplicación.

La multiplicación de matrices se realiza en un gran número de operaciones, como las ecuaciones matriciales. Un algoritmo como éste, que sea capaz de agilizar la multiplicación estableciendo un orden óptimo, permitirá un ahorro computacional importante. En la imagen A.8 se puede encontrar el pseudocódigo del algoritmo.

2.2.4. Algoritmos de grafos

Los grafos son unas estructuras que, si bien conceptualmente tienen casi 300 años, adquieren una gran importancia en la vida contemporánea. Ejemplo de ello lo podemos encontrar en el diseño de carreteras o de redes de comunicaciones. Un grafo se define como un conjunto de objetos, llamados vértices (V), conectados mediante aristas (E), siendo una arista un par (u, v), con $u, v \in V$. A pesar de esta simple explicación, es posible extraer más definiciones relativas a los grafos, como la ordenación topológica (ordenación de vértices en un grafo acíclico tal que si el grafo dispone de la arista $(u, v) \in E$, u siempre aparecerá antes que v), o las componentes fuertemente conexas (subgrafos a partir de un grafo dirigido en los que cada vértice del subgrafo es accesible a partir de los demás). Hay un gran número de algoritmos relativos a estas estructuras, pero los más importantes son los de búsqueda y distancias mínimas desde un origen.

2.2.4.1. Búsqueda

Hay dos algoritmos fundamentales de búsqueda en grafos. Búsqueda primero en anchura y búsqueda primero en profundidad. El primero, es un algoritmo que va explorando los vértices en el mismo orden en el que han sido descubiertos. Se comienza en la raíz y se exploran todos los vecinos adyacentes a dicho nodo (se insertan en una cola). Por cada uno de dichos nodos descubiertos, se exploran cada uno de sus vecinos, hasta haber explorado todo el árbol. Búsqueda en profundidad, en cambio, no emplea una cola y es recursivo. Expande los nodos que va descubriendo de forma recurrente (de ahí el nombre de búsqueda en profundidad). El pseudocódigo de ambos algoritmos se pueden encontrar en las ilustraciones A.9 y A.10.

2.2.4.2. Algoritmos de distancias mínimas desde un origen

Estos algoritmos permiten obtener la distancia mínima de un vértice al resto de los vértices de grafo. Uno de los algoritmos más conocidos de este ámbito es el de Dijkstra, que resuelve el problema para un grafo dirigido cuyo peso en las aristas sea siempre mayor o igual que 0 ($w(u, v) \geq 0$ para cada arista $(u, v) \in E$). La idea fundamental del algoritmo es la obtención de los caminos mínimos que parten desde el vértice de origen al resto de nodos del grafo. Cuando se obtienen todos los caminos mínimos, el algoritmo acaba. Dijkstra tiene un coste de $O(|E|log|V|)$ si emplea una cola de prioridad. El algoritmo se puede encontrar en la figura A.11.

2.3. Sistemas de Recomendación (SR)

Los sistemas de recomendación (*Recommender Systems* en inglés) son herramientas que tienen como objetivo realizar sugerencias acerca de distintos artículos (*items*) para un determinado usuario. En función de cómo se realizan dichas recomendaciones, se pueden distinguir varios tipos de sistemas de recomendación, entre los que destacan los siguientes ([9], [10]):

- Basados en contenido *(Content-based)*: las recomendaciones son realizadas basándose en los artículos que votó positivamente el usuario en el pasado. La similitud entre los artículos suele ser calculada buscando las características comunes entre ellos. Por ejemplo, en el caso de las películas, se suele estudiar los directores o los géneros favoritos del usuario y obtener otros filmes del mismo género y/o director.
- Filtrado colaborativo *(Collaborative filtering)*: las sugerencias son obtenidas a partir de la similitud del usuario o del artículo con el resto de usuarios/artículos. Estas similitudes se suelen calcular analizando el historial de los artículos/usuarios y comparándolos entre ellos.
- Demográficos *(Demographic)*: las recomendaciones realizadas para el usuario están determinadas por el perfil demográfico del usuario. Asume que usuarios que conviven en un núcleo determinado de población consultarán artículos similares.
- Basados en conocimiento *(Knowledge-based)*: se basan en realizar inferencia entre las preferencias y necesidades de los usuarios para sugerir las recomendaciones. Recomiendan artículos al usuario en función de un problema determinado.
- Basados en comunidad *(Community-based)*: las recomendaciones se obtienen a partir de las preferencias de amigos del usuario. Se basan en las evidencias de que las personas tienden a confiar más en las recomendaciones de los amigos y conocidos que otros usuarios desconocidos. Gozan de gran popularidad en las redes sociales.
- Híbridos *(Hybrid)*: los sistemas anteriormente mencionados tienen sus ventajas y sus inconvenientes (véase la tabla 2.3) y normalmente se suelen emplear ideas de todos ellos obteniendo sistemas híbridos, al igual que en algunos sistemas de aprendizaje automático, emplean la combinación de distintos algoritmos para obtener una mejora global.

A continuación se muestran los principales problemas y las ventajas que en general pueden tener estos sistemas según [11] y [12]. En la tabla 2.3 se resumen estas características, relacionando las ventajas y los inconvenientes de cada sistema de recomendación.

Problemas comunes:

- PS: Sobrespecialización. Equivalente al sobreaprendizaje en aprendizaje automático. Es un problema que provoca que el sistema recomiende elementos que son únicamente muy similares a los que ya han sido votados (es complicado que tenga lugar la serendipia o descubrimiento por puro azar).
- PAF: Arranque en frío: problema con nuevos usuarios o con nuevos ítems. En el caso de un nuevo usuario, es difícil recomendar artículos si no se ha registrado ninguna actividad suya previa. En el caso de nuevos ítems, éstos no disponen de ninguna valoración para considerar o aquellas puntuaciones de las que disponen, no son maduras.
- PON: Oveja-negra. Sujetos que tienen unos gustos bastante diferenciados respecto a la mayoría de los otros usuarios, siendo difícil encontrar un conjunto de recomendaciones adecuadas.

- PVD: Volumen de datos. Para obtener buenas recomendaciones, requiere una cantidad de datos grande.
- PD: Poseer información demográfica.
- PIC: Ingeniería del conocimiento. Se necesita de algún conocimiento especial ya sea por parte del producto o por parte del usuario.
- PPI: Poca investigación. No se ha realizado un estudio en profundidad acerca de dicho sistema.

Ventajas generales:

- VDD: No es necesario un conocimiento acerca del dominio de los datos.
- VGC: Capacidad para proporcionar a los usuarios artículos fuera de lo familiar.
- VA: Aprende a lo largo del tiempo. Esto depende del algoritmo que se emplee. En la tabla se indicará si esta característica está disponible para los agoritmos más populares de cada sistema.
- VDI: Buen comportamiento con datos implícitos (datos que no ha introducido el usuario directamente, sino que se han obtenido analizando su actividad).
- VSP: Sensible a cambios acerca de las preferencias del usuario.
- VDA: Puede trabajar con más datos auxiliares, no sólo los datos específicos de los ítems (términos de garantía, fechas de distribución).

Tabla 2.3: Principales ventajas e inconvenientes de los distintos sistemas de recomendación.

Sistema	**Ventajas**	**Inconvenientes**
Basados en contenido	VA VDI	PS PAF PVD
Filtrado colaborativo	VDD VA VDI VGC	PAF PON PVD
Basados en conocimiento	VSP VDA	PIC
Demográfico	VDD VGC	PON PD PPI PVD
Comunidad	VDD VA	PAF PON

2.3.1. Definición del problema y notación

Consideremos el conjunto de usuarios como $\mathcal{U}$ y el conjunto de artículos (ítems) como $\mathcal{I}$. Consideremos también $\mathcal{R}$ como el conjunto de ratings que han sido almacenados en el sistema. Las letras u y v harán referencia a cualquier usuario del sistema, es decir, $u, v \in \mathcal{U}$. Un rating es un valor dado por un usuario a un ítem. Estos valores pueden ser continuos o nominales. Normalmente, los ratings suelen estar acotados entre los números del 1 al 5 (siendo el 1 la peor nota y el 5 la nota más alta) o también es posible verlos como números reales del 1 al 10 o como valores binarios, por ejemplo: Me gusta / No me gusta. El conjunto de valores que pueden tomar los ratings los consideraremos como $\mathcal{V}$ ($\mathcal{V} = [1, 5]$ o $\mathcal{V} = \{\text{Me gusta/No me gusta}\}$). Cuando el conjunto de $\mathcal{V}$ es conocido, a dichos datos se les denominan *datos explícitos*, en otro caso, se les conoce como *datos implícitos*.

2.3.2. Recomendación basada en vecindarios

En este trabajo hemos decidido centrarnos en los SR de filtrado colaborativo basados en vecinos. Los sistemas de filtrado colaborativo se pueden dividir en dos subclases: los *basados en vecinos* y los *basados en modelos*. Los primeros, a su vez, pueden ser basados en usuario (calcular los usuarios más parecidos al que se está evaluando según una medida de similitud) o basados en artículos (calcular la ponderación de dicho ítem basándose en las ponderaciones de otros ítems similares). A diferencia de las aproximaciones por vecinos, los SR de filtrado colaborativo basados en modelos no emplean dichos ratings directamente, sino que se utilizan para construir un modelo predictivo (en la etapa de entrenamiento se construye el modelo) y en la etapa de test se procede a su evaluación. Ejemplos de este tipo de modelos predictivos son los clasificadores bayesianos, cálculo por máxima entropía, o máquinas de vectores de soporte.

En los SR de tipo filtrado colaborativo, los métodos basados en vecinos próximos son ampliamente utilizados debido a su:

- Facilidad de implementación: estos métodos, en general, son sencillos a la hora de implementarlos en cualquier lenguaje de programación. Normalmente, la única variable que requiere de un ajuste es la que indica el número de vecinos cercanos a emplear. En ocasiones, en lugar un valor para el número de vecinos, se suelen coger aquellos que superen una similitud mínima [9, pp. 130].

- Estabilidad: estos sistemas no se ven demasiado afectados por la constante inserción de nuevos elementos o usuarios.

- Eficiencia: la fase de entrenamiento no debe ser actualizada constantemente. Esto hace que para aplicaciones empresariales, en donde la cantidad de usuarios y artículos están en constante actualización, sea una alternativa a tener en cuenta.

A pesar de estas ventajas, sus resultados no suelen ser tan buenos como los métodos basados en modelos. No obstante, según [13], ninguna de estas dos aproximaciones es óptima por separado. En cambio, si se realiza un combinación de ambas, los resultados al efectuar las predicciones de rating mejoran.

Una de las formas de evaluar este tipo de aproximaciones es mediante la *predicción de rating*. De esta forma, el conjunto de ratings $\mathcal{R}$ se divide en un conjunto de entrenamiento (*train*) $\mathcal{R}_{train}$, empleado para obtener los vecinos, según si su aproximación es de usuarios o de artículos y un conjunto de test $\mathcal{R}_{test}$, empleado para evaluar los resultados obtenidos.

El rating predicho se puede obtener de la siguiente manera:

$$\hat{r}_{ui} = \frac{1}{|\mathcal{N}_i(u)|} \sum_{v \in \mathcal{N}_i(u)} r_{vi} \tag{2.2}$$

En donde $\hat{r}_{ui}$ hace referencia al rating que vamos a predecir según nuestro algoritmo y $\mathcal{N}_i(u)$ hace referencia a los k vecinos más próximos al usuario u (k por definir) que han votado el item i (cuya nota se está intentando calcular).

Esta aproximación tiene el inconveniente de que no toma en consideración el grado de similitud entre los usuarios. Es decir, que independientemente de su parecido, todos los k vecinos son considerados iguales. Esto no siempre es cierto ya que hay usuarios que se parecen más a unos que a otros, siendo todos ellos relativamente similares.

Por ello, a veces se emplea la siguiente métrica:

$$\hat{r}_{ui} = \frac{\sum_{v \in \mathcal{N}_i(u)} r_{vi} w_{uv}}{\sum_{v \in \mathcal{N}_i(u)} |w_{uv}|} \tag{2.3}$$

La variable w_{uv} es el valor de similitud entre el usuario u y el usuario v. En esta fórmula, los usuarios con una similitud mayor tendrán más peso en la predicción de valor del rating. Por otro lado, para la recomendación basada en artículos la fórmula es parecida:

$$\hat{r}_{ui} = \frac{\sum_{j \in \mathcal{N}_u(i)} r_{uj} w_{ij}}{\sum_{j \in \mathcal{N}_u(i)} |w_{ij}|} \tag{2.4}$$

En donde w_{ij} es el valor de similitud entre un ítem i y un ítem j. Siendo $N_u(i)$ los ítems votados por el usuario u más similares al ítem i.

Estas definiciones también tiene un problema y es que los usuarios pueden tener distinto margen a la hora de puntuar un mismo ítem. Por ejemplo, un usuario puede dar la nota más alta solamente a un número muy determinado de ítems, mientras que otro menos exigente reparte el máximo valor de forma menos estricta. Por esta razón, los ratings predichos en las fórmulas anteriores suelen pasar por una normalización. Dos técnicas conocidas son *mean centering* y *z-score*, mostradas en [9, pp. 120-122].

La idea del mecanismo de *mean centering* es la de determinar si un rating es bueno o malo en función de la media de los ratings. De esta forma, en el caso de un sistema de vecinos basado en usuario, la fórmula para determinar el rating predicho para un usuario u a un ítem i sería:

$$\text{Mean centering: } \hat{r}_{ui} = \overline{r}_u + \frac{\sum_{v \in \mathcal{N}_i(u)} (r_{vi} - \overline{r}_v) w_{uv}}{\sum_{v \in \mathcal{N}_i(u)} |w_{uv}|} \tag{2.5}$$

En donde $\overline{r}_u$ es la media del rating del usuario u y $\overline{r}_v$ es la media del usuario v (siendo v un vecino de u). También es aplicable a los sistemas basados en artículos:

$$\text{Mean centering: } \hat{r}_{ui} = \overline{r}_i + \frac{\sum_{j \in \mathcal{N}_u(i)} (r_{uj} - \overline{r}_j) w_{ij}}{\sum_{j \in \mathcal{N}_u(i)} |w_{ij}|} \tag{2.6}$$

En donde $\overline{r}_i$ es la media del ítem i y $\overline{r}_j$ es la media del ítem j siendo j un ítem similar a i.

Z-score además de considerar la media de los artículos y de los usuarios, también tiene en cuenta la desviación estándar, es decir, la desviación de dichos ratings respecto a su media. Al igual que en mean centering, z-score es aplicable tanto a los sistemas basados en usuarios como en los basados en artículos:

Sistemas de recomendación colaborativos basados en usuarios:

$$\text{Z-score: } \hat{r}_{ui} = \overline{r}_u + \sigma_u \frac{\sum_{v \in \mathcal{N}_i(u)} (r_{vi} - \overline{r}_v) w_{uv} / \sigma_v}{\sum_{v \in \mathcal{N}_i(u)} |w_{uv}|} \tag{2.7}$$

Y basados en artículos:

$$\text{Z-score: } \hat{r}_{ui} = \overline{r}_i + \sigma_i \frac{\sum_{j \in \mathcal{N}_u(i)} (r_{uj} - \overline{r}_j) w_{ij} / \sigma_j}{\sum_{j \in \mathcal{N}_u(i)} |w_{ij}|} \tag{2.8}$$

Un método para calcular la similitud entre dos objetos a y b es efectuar una transformación de dichos objetos en dos vectores y efectuar el cálculo de la *similitud coseno*:

$$\text{Cosine vector}(a, b) = \cos(a, b) = \frac{a \cdot b}{|a||b|} \tag{2.9}$$

Esta fórmula se puede emplear para calcular la similitud entre usuarios y entre ítems. En el caso de los usuarios, habrá que representar a cada usuario u como el conjunto de items que ha puntuado. Y en el caso de los ítems, se puede considerar a cada ítem como el conjunto de usuarios que lo ha ponderado. En ambos casos, la similitud, cuanto más cercana a 1, mejor. De esta forma, el coseno entre dos usuarios es:

$$\text{Cosine vector } (u,\ v) = \cos(u, v) = \frac{\sum_{i \in \mathcal{I}_{uv}} r_{ui} r_{vi}}{\sqrt{\sum_{i \in \mathcal{I}_u} r_{ui}^2 \sum_{j \in \mathcal{I}_v} r_{vj}^2}} \tag{2.10}$$

Y la similitud coseno entre dos ítems:

$$\text{Cosine vector } (i,\ j) = \cos(i, j) = \frac{\sum_{u \in \mathcal{U}_{ij}} r_{ui} r_{uj}}{\sqrt{\sum_{u \in \mathcal{U}_i} r_{ui}^2 \sum_{u \in \mathcal{U}_j} r_{uj}^2}} \tag{2.11}$$

Donde r_{ui} y r_{vi} hacen referencia a los ratings que da el usuario u y v al ítem i, $\mathcal{I}_{uv}$ hace referencia a los artículos ponderados tanto por u como por v y r_{ui}, r_{uj} y $\mathcal{U}_{ij}$ el equivalente para los artículos. La similitud coseno, aunque es una aproximación sencilla, tiene un problema. No considera las diferencias entre la media y la varianza de los usuarios u y v. (Podría ocurrir que la media de valoraciones del usuario u sea de un 4.5 y la media de v sea de un 3). Por esta razón, junto a la similitud coseno, se emplea la correlación de Pearson (PC):

$$\text{PC } (u,\ v) = \frac{\sum_{i \in \mathcal{I}_{uv}} (r_{ui} - \overline{r}_u)(r_{vi} - \overline{r}_v)}{\sqrt{\sum_{i \in \mathcal{I}_{uv}} (r_{ui} - \overline{r}_u)^2 \sum_{i \in \mathcal{I}_{uv}} (r_{vi} - \overline{r}_v)^2}} \tag{2.12}$$

En donde $\overline{r}_u$ representa la media del rating del usuario u y $\overline{r}_v$ indica la media del rating del usuario v. Los artículos deben ser ponderados tanto por el usuario u como por el v (conjunto de intersección), al igual que en el numerador de la similitud coseno. Aunque esta sea la fórmula originaria, se pueden efectuar modificaciones para considerar los conjuntos de unión poniendo una nota por defecto en aquellos artículos que no hayan sido puntuados por ambos usuarios.

También es posible calcular la correlación de Pearson entre los ítems i y j:

$$\text{PC } (i,\ j) = \frac{\sum_{u \in \mathcal{U}_{ij}} (r_{ui} - \overline{r}_i)(r_{uj} - \overline{r}_j)}{\sqrt{\sum_{u \in \mathcal{U}_{ij}} (r_{ui} - \overline{r}_i)^2 \sum_{u \in \mathcal{U}_{ij}} (r_{uj} - \overline{r}_j)^2}} \tag{2.13}$$

Por último, un ejemplo de sistemas básicos de predicción son los predictores base [9, pp. 147-148]. Uno de los más conocidos es el siguiente:

$$b_{ui} = \mu + b_u + b_i \tag{2.14}$$

Donde μ hace referencia a la media de los ratings totales del sistema, b_u hace referencia al sesgo del usuario (desviación de la nota del usuario con respecto a la media del sistema) y b_i es el sesgo del artículo. La suma de la media y los sesgos da como resultado el rating predicho b_{ui}. A pesar de su simpleza, no es el más básico (se puede devolver siempre la media del total de los artículos, o la media del usuario). Estos predictores base sirven para poder establecer comparaciones con otros predictores más complejos y determinar la mejora obtenida.

2.3.3. Evaluación de las recomendaciones

Por otro lado, es necesario evaluar los métodos propuestos y explicados previamente entre sí. Para ello, empezaremos con dos métricas que directamente comparan el valor predicho y el valor real dado por el usuario en el conjunto de test:

$$\text{Mean Squared Error MAE}=\frac{1}{|\mathcal{R}_{test}|}\sum_{r_{ui}\in\mathcal{R}_{test}}|f(u,i)-r_{ui}| \tag{2.15}$$

$$\text{Root Mean Squared Error RMSE}=\sqrt{\frac{1}{|\mathcal{R}_{test}|}\sum_{r_{ui}\in\mathcal{R}_{test}}(f(u,i)-r_{ui})^2} \tag{2.16}$$

Donde f(u,i) es la función que devuelte el rating predicho de ese usuario para ese ítem.

En ocasiones, los ratings no están disponibles, sino únicamente los ítems que un usuario ha consumido. En estos casos, ya no se efectúa una predicción de rating (ya que este no existe), sino que se compara la lista de ítems consumidos con una lista de ítems recomendados. En [9] se describen varias métricas para evaluar este problema. La lista de ítems recomendados puede obtenerse prediciendo la nota que le daría el usuario a un conjunto de ítems y ordenando de mayor a menor dicha lista, o también devolviendo los ítems más populares en ese momento que se adecúen al gusto del usuario. Opcionalmente, estas métricas se pueden calcular sólo sobre los p primeros ítems devueltos (denotado como '@p').

$$\text{Precision(L)} = \frac{1}{|\mathcal{U}|}\sum_{u\in\mathcal{U}}\frac{|L(u)\cap T(u)|}{|L(u)|} \tag{2.17}$$

$$\text{Recall(L)} = \frac{1}{|\mathcal{U}|}\sum_{u\in\mathcal{U}}\frac{|L(u)\cap T(u)|}{|T(u)|} \tag{2.18}$$

La métrica de *precisión* se basa en obtener el conjunto de intersección entre los documentos relevantes ($T(U)$) y los recomendados ($L(U)$) dividiendo dicha intersección entre el total de los recomendados. *Recall* se basa en la división de la intersección entre los relevantes. Dichos documentos, en este caso, serán los artículos que ha votado realmente el usuario y los recomendados, aquellos ítems que ha proporcionado el recomendador para un determinado usuario. No obstante, hay otras métricas interesantes, como son MRR (*Mean Reciprocal Rank*) y NDCG (*Normalized Discounted Cumulative Gain*).

$$\text{MRR} = \frac{1}{|\mathcal{I}|}\sum_{i=1}^{|\mathcal{I}|}\frac{1}{rank_i} \tag{2.19}$$

MRR se basa en comparar la posición de un artículo recomendado con su posición en el conjunto de relevantes. Es decir, aquellos artículos recomendados (con una nota alta en test) en primeras posiciones, se valorarán mejor.

$$\text{NDCG} = \frac{DCG@p}{IDCG@p} \tag{2.20}$$

$$\text{DCG@}p = rel_1 + \sum_{i=2}^{p}\frac{rel_i}{log_2(i)} \tag{2.21}$$

NDCG se basa en la relevancia de los artículos recomendados. Para ello, se calcula primero la fórmula DCG, opcionalmente sobre un máximo de p elementos (DCG@p). Este valor obtenido debe compararse con el ideal (IDCG o IDCG@p), que se obtiene de la misma forma que DCG pero ordenando los elementos relevantes de mayor a menor preferencia.

3

Sistema, diseño y desarrollo

3.1. Objetivo y descripción del sistema

En función de lo explicado en la sección anterior, se ha creado un sistema que efectúe tanto evaluaciones de rating (*rating prediction*) como de ranking (*ranking evaluation*), permitiendo el uso de distintas estructuras de datos y algoritmos. De esta forma, se permitirá comparar el rendimiento de las estructuras y el resultado en los algoritmos. Es decir, por un lado, es un framework que permite el tratamiento de datos para realizar medidas de predicción y evaluación de error y por otro permite la medición de tiempos y memoria de distintas estructuras de datos programadas.

Para dar solución de una manera ordenada a estas funcionalidades, el sistema global se encuentra dividido en distintos subsistemas:

- Algoritmos: subsistema que contiene la implementación de los algoritmos que serán utilizados en recomendación (correlación de Pearson y similitud coseno, con sus variantes explicadas en 3.4).
- Estructuras: implementa las estructuras de datos útiles que serán empleadas en la aplicación para los experimentos. Como en el libro de "Introduction to Algorithms" [8] se detallan tanto estructuras como algoritmos, se ha visto relevante hacer esta división entre los subsistemas.
- Transformación: permite transformar ficheros de datos que sean de tipo implícitos a ficheros de datos de tipo explícitos. De igual manera, permite obtener a partir de un fichero de datos un subconjunto de entrenamiento y un subconjunto de test a partir del conjunto de datos total.
- Evaluación: contiene los procedimientos necesarios para efectuar las recomendaciones basadas en artículos o en usuarios. Admite algunos parámetros de configuración como el empleo de algoritmos de normalización. Este subsistema es el más grande, ya que se subdivide en más módulos (métricas de ranking, métricas de rating, lectura y carga de datos de los distintos datasets).

3.2. Requisitos del sistema

En vista de las estructuras de datos que se debían considerar en el proyecto obtenidas de [8] y los algoritmos a emplear, se vio necesario emplear un lenguaje de programación que fuese de alto nivel (para facilitar la creación de código mediante el empleo de funciones útiles de librerías), pero además eficiente. Se eligió Java por estos motivos:

- Lenguaje de alto nivel, permitiendo hacer uso de funciones y estructuras de sus bibliotecas estándar agilizando la programación.
- Lenguaje familiar. Al haber trabajado con él a lo largo de la carrera, no era necesario emplear tiempo en aprender un nuevo lenguaje
- Es multiplataforma.
- Soporta la programación orientada a objetos.

3.2.1. Requisitos funcionales

1. El sistema debe permitir el trabajo con las siguientes estructuras de datos: árboles binarios de búsqueda, árboles rojo-negros, árboles estadísticos, tablas hash, listas doblemente enlazadas y árboles B.
2. Es necesario permitir tanto predicciones de rating basadas en usuarios como basadas en artículos y evaluaciones de ranking.
3. Debe ser posible añadir nuevas estructuras o algoritmos al sistema.
4. El sistema admitirá ejecuciones con distintas estructuras de datos, así como mediciones de espacio (memoria) y tiempo de las ejecuciones.
5. En caso de encontrar algún dato anómalo (usuario sin vecinos, artículo en el conjunto de test pero no en el de entrenamiento, etc.), el programa no puede finalizar de manera abrupta.
6. El programa no dispondrá de interfaz gráfica.
7. El sistema no generará gráficas a partir de los datos obtenidos, solamente ficheros de texto.
8. Deberá ser posible pasar de datos implícitos a datos explícitos.

3.2.2. Requisitos no funcionales

1. Toda la aplicación deberá estar documentada en inglés.
2. El sistema será accesible desde un repositorio GIT (como github[1] o bitbucket[2]).
3. El programa debe funcionar en cualquier sistema operativo que tenga una versión Java 1.7 o superior.

[1] https://github.com/ [2] https://bitbucket.org/

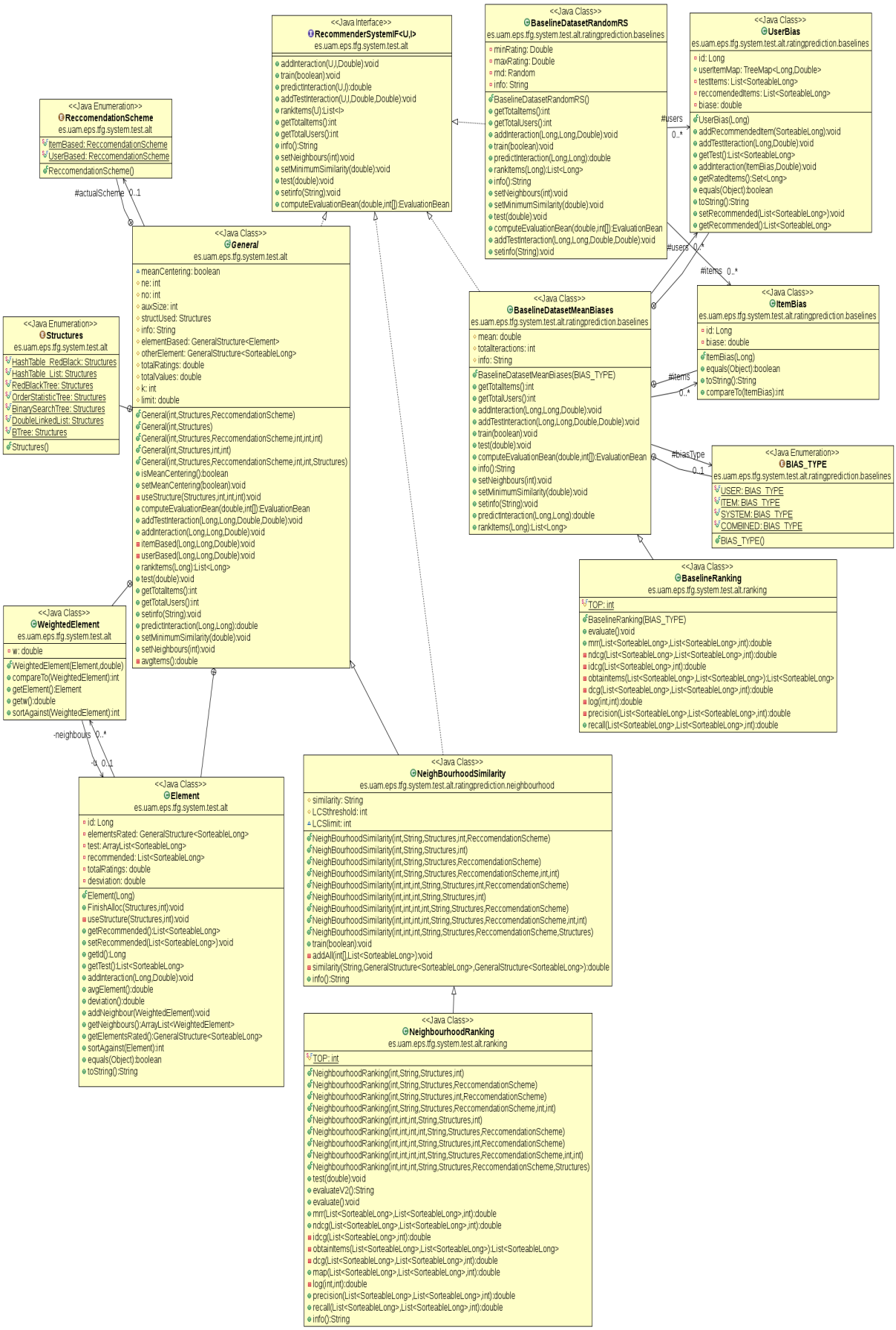

Figura 3.1: Diagrama del subsistema de evaluación.

3.3. Diseño

La Figura 3.1 muestra el diagrama de clases del subsistema de evaluación, siendo este el subsistema más importante de la aplicación. Además, en el directorio *documentation/diagrams* del repositorio se pueden encontrar el resto de diagramas (estructuras y nodos) para obtener más información acerca del diseño completo del sistema.

Primeramente, se realizó un diseño preliminar que cumpliese con los requisitos fundamentales. Por un lado, era necesario que todas las estructuras (árboles, heaps, listas, colas de prioridad, etc.) trabajasen con clases generales y por otro lado, que los objetos de dichas clases pudieran compararse entre ellos, es decir, que fuesen ordenables para que pudiesen realizarse búsquedas eficientes. Java dispone de una interfaz llamada Comparable que permite determinar para dos objetos, cuál es mayor, menor o igual que otro. Aunque Comparable era una buena candidata, se optó por crear una interfaz llamada *Sorteable* que tuviese un método para ordenar (a efectos prácticos era equivalente a Comparable, pero al crear otra interfaz, se podía utilizar Comparable para otras necesidades). De esta forma, se podían crear estructuras genéricas siempre y cuando los elementos almacenados en ellas implementasen el método de ordenación. Para añadir un nivel de abstracción mayor, se decidió crear otra interfaz llamada *GeneralStructure* que admitiese un tipo parámetrico (que extendiese de *Sorteable*). Cualquier estructura de datos creada debía implementar los métodos de la interfaz *GeneralStructure*. Los más relevantes son:

- search: método para buscar un elemento en la estructura. Recibe un elemento y devuelve el elemento equivalente guardado o null si el elemento no existe. Es importante señalar que no devuelve un booleano indicando si está dicho elemento o no, sino que devuelve un objeto.
- insert: inserción de un elemento en la estructura. De tipo void (sin retorno).
- getStructure: método para obtener una cadena de caracteres que indique el tipo de estructura que es (similar al método toString(), pero sin pasar toda la clase a una cadena sino sólo información importante).
- getTotalElements: obtiene un entero que indica el número de elementos de la estructura.
- toList: permite obtener los elementos de la estructura en un List de Java del mismo tipo de elemento que los almacenados en la estructura.

Como se ha mostrado en el apartado 3.1, los paquetes de estructuras y de algoritmos están separados. El primero, además, se divide en varios subpaquetes, los básicos (listas doblemente enlazadas y tablas hash), los nodos (empleados por algunas estructuras como los árboles binarios), y los árboles. Estos últimos, se dividen en árboles de búsqueda y heaps. En los árboles de búsqueda hay una clase abstracta, llamada *BinaryTree*, que contiene los métodos generales para todos los árboles binarios (método toList y los métodos para recorrer los árboles). Así pues, el árbol binario de búsqueda extiende de la clase anterior, el árbol rojo-negro extiende a su vez del árbol binario y el árbol estadístico hereda del árbol rojo-negro. Gracias al diseño de programación de objetos, se ha podido evitar redundancia de código en estos casos. De forma similar están organizados los heaps. Por un lado hay una clase abstracta de heap binario (*BinaryHeap*) y dos clases (*MaxHeap* y *MinHeap*) que heredan de él. Finalmente las colas de prioridad, dado que pueden ser construidas a partir de un heap, extienden o bien de MaxHeap (en caso de una cola de prioridad máxima) o de MinHeap (en caso de prioridad mínima). Además, las tablas hash, resolverán las posibles colisiones mediante encadenamiento, aunque en lugar de ser mediante listas, pueden ser configuradas para que las resuelva además, mediante árboles.

En cuanto al diseño de las clases relativas al subsistema de recomendación, hay una interfaz notable a tener en cuenta: *RecommenderSystemIF*. Esta interfaz proporciona métodos importantes a la hora de efectuar tanto implementaciones basadas en predicción de rating como en obtención de listas de ítems para recomendar, teniendo en cuenta que nos restringimos a los algoritmos basados en vecinos (ver apartado 2.3). A partir de ahora, cuando se hable de rating, se referirá a la predicción de rating y ranking se referirá a la obtención de un listado de ítems a recomendar al usuario.

Los métodos más importantes de dicha interfaz son:

- addInteraction: recibe un usuario, un ítem y el valor de dicha ponderación y lo almacena en la estructura correspondiente.
- addTestInteraction: recibe un usuario, un ítem, el valor de dicha ponderación y una cota. Este método permite añadir en la lista de elementos de test un ítem determinado si su ponderación es mayor o igual que la cota.
- train: método de entrenamiento. Recibe un booleano que permite indicar si sólo se ejecuta el entrenamiento sobre los usuarios de test.
- predictInteraction: recibe un usuario, un ítem y obtiene un valor que determina el valor predicho a dicho ítem.
- test: método de test. Recibe un valor por defecto a devolver en caso de no poder predecir.
- setNeighbours: método para actualizar el valor de los vecinos.
- info: devuelve la información asociada a dicha implementación (las clases que implementen dicha interfaz deberían mostrar datos importantes como estructuras y métricas empleadas).

Una clase que implementa gran parte de estos métodos es la clase *General*, del paquete *test.alt* (dentro del paquete de *system*). Esta clase general implementa los métodos de *addInteraction*, *addTestInteraction*, *predictInteraction* y *setNeighbours*. Además, dispone de dos estructuras generales tanto para usuarios como para ítems (en el método constructor se puede especificar la estructura con la que se va a trabajar y además se puede especificar si se va a realizar un sistema basado en usuario o en artículos). Por otro lado, tiene otras dos clases auxiliares que son *Element* y *WeightedElement*, en el mismo fichero. La clase de *Element* hace referencia al elemento sobre el que está basado el sistema, siendo dicho elemento un usuario en el caso de estar basado en usuarios y un artículo en caso contrario. Además de un identificador, dicha clase también tendrá una lista de vecinos (objetos de la clase de *WeightedElement*, que almacena además de un elemento, un peso de similitud con respecto al mismo), una estructura de elementos puntuados y listas de elementos recomendados y de elementos de test.

La ejecución total del sistema completo para la predicción de rating se puede dividir en tres fases importantes. La carga de datos, en donde simplemente se almacenan los datos leídos de un fichero, la etapa de entrenamiento (*train*), en donde se calculan los vecinos de los usuarios/ítems (dependiendo del esquema elegido) y la etapa de test, en la que por cada línea leída del archivo de test, se predice la puntuación de un determinado usuario para un determinado artículo. El resultado de error se hace empleando las métricas de MAE (2.15) y RMSE (2.16), para el caso de rating y de Precision (2.17), Recall (2.18), MRR (2.19) y NDCG (2.20) en el caso de ranking. El seguimiento de la ejecución es, por lo tanto, el siguiente:

1. Se lee el fichero de datos de entrenamiento. Por cada puntuación dada por un usuario, se añade el usuario y el ítem si no están añadidos en las estructuras específicas para ellos (las

estructuras generales globales que almacenan los items del sistema y los usuarios). En caso de ser basado en usuario, se almacena dicho artículo con su nota dentro del usuario. En caso de basarse en artículos, para dicho artículo se guarda el usuario y su nota. La forma de almacenar estas interacciones es mediante otra estructura general.

2. Una vez finalizada la etapa de carga de datos, por cada elemento, se obtiene una lista de todos los demás elementos ordenados por su similitud (sea cual sea la métrica empleada). Actualmente esto se realiza manteniendo una cola de prioridad máxima, de forma que las extracciones de la cola serán siempre en orden de similitud. Se emplea una cola en lugar de una lista debido a que en caso de indicar un tamaño máximo de la cola, ésta seguirá ordenada en todo momento.

3. En la etapa de test, se cogen los k vecinos más próximos que hayan votado al ítem i (basados en usuario) o que tengan al usuario u (basados en artículos) y se aplica la fórmula de 2.3 o 2.5 según la variante escogida (o 2.4 o 2.6 para los algoritmos basados en ítems). Se devuelve entonces el rating predicho.

En el caso del ranking, los dos primeros puntos anteriores se mantienen, pero la parte del test cambia. En entrenamiento se calculan los vecinos ordenados por similitud y en test se obtiene una lista de artículos recomendados ordenados de mayor a menor, empleando el método de *predictInteraction* como se hacía para predicción de rating. Posteriormente, se podrá llamar a cualquier métrica de ranking con las listas de los elementos de test y los elementos recomendados. Para automatizar ambos tipos de métricas, se encuentra la clase *EvaluatorUtils.java*, en el paquete system/eval. Esta clase permitirá obtener el listado de artículos recomendados por cada usuario, así como una bean que contendrá el resultado de todas las métricas implementadas.

Por último, a la hora de transformar los datos implícitos a explícitos, se ha creado una clase que por cada usuario, obtiene el artículo con más visionados/consumiciones. Una vez obtenido dicho valor, por cada artículo que ha consumido, se calcula la nota transformada multiplicando el número de visionados de dicho artículo por cinco (si la escala es 1-5) dividiéndolo por el número de visionados del ítem más consumido. Dado que dicha división puede no ser exacta, se cogerá el techo de dicho resultado. Este método está basado en la transformación explicada en [14].

3.4. Desarrollo y codificación

El desarrollo del proyecto comenzó con la implementación de las estructuras mencionadas en el apartado anterior. Se comenzó con las básicas, las tablas hash y las listas doblemente enlazadas. Posteriormente se creó la estructura de árboles, desde las clases más generales hasta las más específicas. Una vez creadas las estructuras más importantes, se creó el paquete con los algoritmos de similitud. Todos estos algoritmos se encuentran en la clase *Similarity*. En concreto se incluyen:

- Correlación de Pearson estándar: versión de correlación de Pearson explicada en ecuación (2.12). Recibe dos estructuras de tipo general que almacenan los items o usuarios de dos elementos distintos.

- Correlación de Pearson union: versión modificada de la anterior en la que consideramos el conjunto de unión de los dos elementos. Es decir, en caso de estar basado en usuario, en lugar de considerarse el conjunto de ítems de la intersección, se consideran los artículos que forman la unión entre los votados por los dos usuarios. Aquellos items de un usuario que no hayan sido ponderados por el otro y viceversa, se ponderan como el valor por defecto. En

el caso de estar basado en artículos, funciona de igual manera, pero obteniendo el conjunto de unión entre artículos.

- Similitud coseno estándar: versión implementada según lo explicado en ecuación (2.10). Recibe los conjuntos de artículos de dos usuarios o el conjunto de usuarios de dos artículos.
- Similitud coseno union: al igual que la versión de Pearson union, esta segunda implementación opera con el conjunto de unión, no el de intersección.

Se había pensado en un principio el considerar en lugar del conjunto de intersección o de unión entre dos elementos, considerar el total de los elementos del sistema. No obstante, esta alternativa fue rechazada debido al coste de memoria y ejecución que suponía. Igualmente, su resultado en algunos experimentos preliminares no difería a los métodos de unión.

Además, también se ha llevado a cabo la programación de un predictor de referencia (explicado en la fórmula 2.14) que permite operar de cuatro maneras diferentes. La primera de ellas devuelve siempre la media del total de artículos. La segunda, devuelve la media total de artículos del sistema sumándole el sesgo del usuario. Una tercera que calcula la media total de los artículos, sumándole el sesgo del artículo a predecir y una conjunta que para un determinado artículo y un usuario, devuelve la media total del sistema sumándole ambos sesgos.

Por último, se ha probado un nuevo algoritmo para calcular la similitud entre dos usuarios que no es una métrica estándar de la literatura (es, por tanto, hasta donde sabemos, algo novedoso en el área). Este algoritmo es la longitud de la subcadena común más larga *Longest Common Subsequence (LCS)* en inglés, explicado en 2.2.2. Este algoritmo generaba una matriz bidimensional con los caracteres de las dos cadenas. La función empleada para rellenar dicha matriz es la siguiente:

$$C[i,j] = \begin{cases} 0 & \text{si i=0 o j=0} \\ C[i-1,j-1]+1 & \text{si } i,j>0 \text{ y } X_i = Y_j \\ \max(C[i,j-1], C[i-1,j]) & \text{si } i,j>0 \text{ y } X_i \neq Y_j \end{cases} \tag{3.1}$$

Para aplicar este algoritmo a recomendación, es necesario efectuar una transformación en los datos del sistema. Se realizará la explicación para un sistema basado en usuarios, pero en caso de ser un sistema basado en artículos, la aproximación es simétrica.

Una primera alternativa es considerar al usuario u como una cadena de caracteres con lo items que ha puntuado y aquellos ítems que no haya puntuado, aparezcan con un carácter especial. Si un usuario ha puntuado los ítems 1,2,4 y 7 de un total de 7 ítems en la base de datos, la cadena resultante sería: 12-4–7. Esta representación tiene varios problemas:

- La mayor parte de los usuarios no habrán ponderado casi todos los artículos, por lo que la mayoría de los caracteres de las cadenas serán caracteres comodines, obteniéndose similitudes entre usuarios que han votado pocos artículos.
- No se hace referencia a los ratings de los items. Dos usuarios pueden haber consumido los mismos ítems pero los pueden haber ponderado de forma muy distinta.
- Desperdicio de memoria almacenando caracteres comodines.

Los dos primeros problemas se pueden solucionar si se ignoran los caracteres comodines (efectuando una modificación en el algoritmo LCS) y concatenando al identificador del ítem su nota. Es decir, para el usuario anterior, si puntuó el item 1 con un 5, el 2 con un 3, el 4 y el 7 con un 2, la cadena sería: 1523-42–72. No obstante sigue habiendo varios problemas:

- Mayor complejidad modificando el algoritmo debido a los caracteres que deben ser ignorados para crear correctamente la matriz.
- De nuevo, desperdicio de memoria almacenando caracteres comodines.

Por esta razón, se optó por la siguiente alternativa: cada usuario tiene una lista de enteros que corresponden con cada ítem que ha puntuado. Cada elemento de dicha lista es el ID del ítem multiplicado por un valor sumado a la nota ponderada (se ha empleado una multiplicación por el valor 100), de manera que se obtenga una biyección que nos permita obtener la representación del ítem y su rating a partir de estos datos, o viceversa a partir de la representación. Si se mantiene la cadena de caracteres, sería muy costoso modificar el algoritmo para operar bien con esa representación de datos. De esta forma, adaptar el algoritmo de la subcadena común más larga es sencillo (una cadena de caracteres se puede interpretar como una lista, por lo que de igual manera se puede trabajar con una cadena de enteros). Se ahorra en espacio (no es necesario añadir caracteres extra) y se ahorra en tiempo de computación (trabajar con enteros es menos costoso que hacerlo con cadenas).

A la hora de comparar las métricas estándar con la aplicación del algoritmo de la subcadena común mas larga, se efectúan los mismos pasos, menos en la etapa de train. Esto es debido a que las demás métricas (Pearson, coseno y sus variantes) realizan la misma transformación de los datos, pero en LCS esta transformación es diferente. En LCS, cada ítem debe ser transformado a un entero junto con su nota para permitir el cálculo de LCS, como se ha explicado previamente.

Por otro lado, al igual que en Pearson se pueden eliminar los vecinos que tengan una similitud negativa [9], en LCS no puede haber tal restricción ya que los valores devueltos están entre 0 y la longitud máxima. No obstante, sí se pueden eliminar aquellos vecinos que tengan una longitud de subcadena corta en función de algún parámetro (por ejemplo, no considerar similares aquellos que no tengan mínimo una coincidencia de 10 elementos). Además es posible ser menos restrictivo con el algoritmo LCS a la hora de considerar cuándo dos caracteres son iguales. Debido a la transformación que se ha realizado a arrays de enteros, se puede considerar que dos ratings son iguales si, por ejemplo, en lugar de estar ponderados con la misma puntuación, están ponderados con una diferencia de 1 (si un usuario ha votado un ítem con un 4 y otro con un 5, se podría considerar que dichos usuarios han votado el ítem de la misma manera); para esto, basta con redefinir lo que significa la igualdad en el algoritmo (A.6) o en las ecuaciones que lo definen (2.1 y 3.1). En los experimentos probaremos estas variantes para comprobar su impacto en el algoritmo. No obstante, no es la única forma de operar con LCS. Esta aproximación, dependiendo del orden de los datos, puede resultar equivalente a la obtención del conjunto de artículos que han sido puntuados por ambos usuarios de la misma manera (conjunto de intersección).

Una posible alternativa es, tener en cuenta la fecha del visionado del artículo (*timestamp*). El orden de las cadenas a comparar, podría ser en función de la nota y en caso de empate, dicho valor de tiempo. Por ejemplo, si un usuario ha votado el artículo 1 y 2 con un 5, el artículo 3 con un 3 y el 4 con un 4 y ha consumido los artículos en ese orden, la cadena resultante, sería (aplicando el multiplicador) 105,205,404 y 303. Por otro lado, si un segundo usuario ha votado únicamente el artículo 1 y el artículo 2, pero el segundo lo consumió antes que el primero, el orden quedaría 205 y 105. Obtiendo el LCS de ambos usuarios utilizando la fecha, el resultado sería 1 y no 2, como sería si se ordenara por el ID del ítem. En lugar del tiempo de consumición del artículo, podría emplearse también la media del usuario al género de dicho artículo.

4

Experimentos realizados y resultados

4.1. Pruebas Unitarias

Para comprobar el correcto funcionamiento de los algoritmos implementados, se han creado clases de test JUnit tanto para los algoritmos de similitud (LCS, Pearson, coseno ...) como para las estructuras implementadas y las métricas de ranking. Todas las clases de test se encuentran en el directorio de tfg/test. Por otro lado, también se han realizado pruebas sobre el rendimiento de algunas estructuras con parámetros variables (ver anexo C).

4.2. Datasets

En cuanto a las pruebas del sistema completo, se ha empleado el conjunto de datos de Movielens. En un principio se quería emplear también el dataset de Lastfm (para el cual se implementaron métodos de transformación de datos implícitos a explícitos), pero debido a problemas para ejecutar los resultados, sólo se ha podido trabajar de manera completa con el de Movielens. Este dataset se compone de 100.000 registros (943 usuarios y 1682 artículos únicos) cada uno de ellos con cuatro campos, el id del usuario, el id del ítem, el valor dado por ese usuario a ese ítem entre 1 y 5 (variable discreta) y el instante de tiempo en que el se produjo esa votación; este último campo no se considerará en los experimentos.

4.3. Descripción y resultado de las pruebas

Aunque en un principio pueda parecer que los SR basados en vecinos sólo tengan el parámetro de vecinos para configurar, esto no es así. Hay otros aspectos a tener en cuenta, entre los que destacan:

- Forma de ponderar los ratings. Para un determinado número de vecinos, se puede considerar que todos ellos son vecinos iguales (aplicando una media ponderada). Pero otra alternativa es multiplicar el rating del vecino por su similitud, según la fórmula 2.3.

- Algunos vecinos pueden tener similitudes negativas o demasiado bajas en algunos algoritmos (como la correlación de Pearson). Aunque las fórmulas comentadas en la sección anterior operan bien con dichos valores, puede ser útil eliminar dichos vecinos ya que no proporcionan información valiosa para los cálculos y sólo generan ruido.

- En algunas ocasiones no es posible mantener en entrenamiento todos los vecinos del usuario, y por lo tanto se hace un número distinto de vecinos para entrenamiento que para test.

- En [9], la media en la fórmula de la Correlación de Pearson se calcula con el total de los artículos puntuados por el usuario, en cambio, también sería posible obtener la media únicamente con aquellos artículos que han sido puntuados por ambos usuarios, como en el ejemplo mostrado en [15]. Algunas herramientas, como mahout también trabajan de esta manera [1].

- Aplicación de alguna medida de normalización. Aunque no sea estrictamente obligatorio, en el libro estudiado [9] se recomienda aplicar alguna medida como mean centering o z-score (2.5 y 2.7) para mejorar los resultados.

- Valor por defecto en caso de no encontrar vecinos. Pudiera ocurrir que en conjunto de test apareciesen usuarios o artículos que no existiesen en el conjunto de train. No hay una forma general de tratar estos casos. Algunos recomendadores devuelven la media total de los items y otros una variable de error, ignorando dicho elemento de test.

- Predicciones fuera de rango. Es posible que en función de los algoritmos empleados y del método de normalización, algunos ratings predichos estén fuera del rango del sistema. Si por ejemplo, las notas de los artículos están en el intervalo [1,5], es posible que la nota predicha sea superior a 5 o inferior a 1. En ese caso, se pueden acotar los resultados devueltos.

En el caso del sistema desarrollado los ratings son ponderados. Es decir, los vecinos con una similitud mayor serán más importantes que otros cuya similitud sea más baja. Se harán pruebas tanto con similitudes negativas como cuando sólo se consideren las positivas, para evaluar su impacto. En cuanto a la normalización, la aplicación soporta ejecuciones tanto con el mecanismo de mean centering la forma estándar. Las diferencias entre dichas alternativas serán comparadas en esta sección. En caso de que no se pueda devolver un rating predicho (se devuelva NaN), se admitirá poner tanto un valor por defecto como ignorar dicho ítem del conjunto de test. Por último, los ratings devueltos no estarán acotados en su rango original ni tampoco se hará un filtrado de vecinos en entrenamiento.

En cuanto a las pruebas, se van a efectuar comparaciones entre la correlación de Pearson, similitud coseno (tanto la normal como las del conjunto de unión), predictores base y el algoritmo de la longitud de la subcadena común más larga, tanto para predicciones de rating como de ranking. También se analizará tanto la memoria como los tiempos de ejecución de las pruebas a la hora de almacenar los datos en cada estructura.

4.3.1. Resultados Movielens

Se dejará un 80 % de los datos para entrenamiento (*train*) y un 20 % para test, y este proceso se repetirá 5 veces (trabajaremos, por tanto, con 5 *folds*). Asimismo, los resultados de las métricas de evaluación serán una media de los resultados entre los 5 ficheros de test.

[1] `https://github.com/apache/mahout/blob/master/mr/src/main/java/org/apache/mahout/cf/taste/impl/similarity/AbstractSimilarity.java`

Para comparar el rendimiento de las estructuras de datos para Movielens, se han realizado pruebas empleando la misma estructura en todas las etapas de la aplicación, es decir, si se empleaban tablas hash en una estrategia basada en usuarios, tanto los artículos como los usuarios eran almacenados en estas estructuras y cada usuario, además mantendría una tabla hash del tamaño especificado para almacenar los artículos ponderados (en caso de ser basado en artículos, serían los artículos los que mantendrían una tabla hash de usuarios). Esto se ha hecho con todas las estructuras implementadas empleando todas las combinaciones de Pearson y coseno.

Structure	Load training memory	Load training time	Training memory	Training time
Binary search tree	71,82	1,96	214,00	64,74
Red-black tree	71,82	0,55	312,64	53,23
Order statistic tree	71,82	0,57	304,63	54,92
Double linked list	71,82	1,94	353,20	69,49
Hash table (list)	2772,26	9,40	Nan	324,88
Hash table (red-black tree)	2761,07	12,54	Nan	399,82
Btree	92,34	0,72	214,27	65,02

Figura 4.1: Comparativa entre las estructuras desarrolladas ejecutando la obtención de listas de recomendación por cada usuario. Tiempo en segundos y memoria en megabytes. Sin lazy instantiation. BTree con T = 3.

En esta primera aproximación (4.1), se vio que las tablas hash consumían mucha más memoria que el resto de las estructuras y que además, eran las que tardaban más tiempo. Esto se debía a la reserva de memoria de cada tabla de cada usuario para la totalidad de artículos. El tiempo que tardaban en las etapas de train y de test era mayor que en otras estructuras debido a que recorría todas las posiciones de la tabla hash aunque no tuvieran datos. Además, en este tipo de aplicaciones, por lo general, los usuarios no han votado todos los artículos que hay en el sistema, por lo que es un desperdicio de recursos emplear esta configuración de estructuras.

Por otro lado, a la hora de leer los usuarios del fichero de entrenamiento, se procedía a crear un nuevo elemento y a insertarlo en caso de que no estuviese ya en el sistema. Esto puede resultar ineficiente si se crea el objeto del elemento completo. Debido a esto, se cambió este mecanismo empleando la estrategia conocida como lazy instantiation. Esta estrategia crea la información justamente necesaria del objeto y después termina su creación en caso de ser necesario. Como para comprobar si un usuario está en el sistema, sólo necesitamos su identificador, basta con guardar el id y en caso de que no esté, terminar la reserva de los datos.

La segunda prueba se realizó implementando esta alternativa (imagen 4.2) y se obtuvo una mejora de memoria del 14 % en todas las estructuras leyendo el fichero de entrenamiento (menos en las tablas hash que el porcentaje llega casi al 90 %). Además, viendo la problemática de reservar para todos los artículos, se decidió que cada usuario tuviese una tabla hash de 23 posiciones (un array de 23 listas o árboles para resolver las colisiones). En la figura 4.2 se pueden ver los resultados tras aplicar "lazy instantiation" para todas las estructuras y cuando para las tablas hash se hace la distinción del tamaño 23 y del tamaño total de artículos. Las tablas que emplean un tamaño de 23 consumen bastante menos memoria (en la parte de lectura entrenamiento, hasta un 70 % y en el entrenamiento un 65 %, para las tablas basadas en listas). El tiempo, aunque en menor medida, también consigue una mejora importante.

No obstante, las tablas hash implementadas no dejan de ser una implementación muy básica. En algunos lenguajes de programación, las tablas hash son implementadas con un tamaño pequeño fijo y a medida que se va llenando, se crean otras tablas hash más grandes y se le añaden los datos de las tablas anteriores (rehashing). Aunque pueden ser costoso las sucesivas creaciones, compensaría ya que hay muchos usuarios con pocos datos y no siempre sería necesario el rehashing. Un estudio más completo sobre estas estructuras se puede encontrar en el anexo C.

Structure	Load training memory	Load training time	Training memory	Training time
Binary search tree	61,56	1,82	191,99	64,27
Red-black tree	61,56	0,45	216,57	53,16
Order statistic tree	61,56	0,48	242,49	54,40
Double linked list	61,56	1,85	296,36	69,08
Btree	61,56	0,58	146,12	64,14
Hash table (23 lists)	87,21	2,07	672,44	87,52
Hash table (23 red-black trees)	92,34	0,60	511,78	68,15
Hash table (list)	292,42	0,80	1957,53	331,26
Hash table (red-black tree)	202,52	1,26	2070,80	411,04

Figura 4.2: Comparativa entre las estructuras desarrolladas ejecutando la obtención de listas de recomendación por cada usuario. Tiempo en segundos y memoria en megabytes. Con lazy instantiation.

En la imagen superior, el árbol B siempre se inicializa con un T=3 (5 claves, 6 hijos como máximo). La etiqueta hash table (list) hace referencia al empleo de listas en la tabla hash y hash table (tree) a árboles. El 23 indica el tamaño de la tabla que almacena los artículos de los usuarios.

A continuación, mostramos las predicciones de rating usando métricas de error (valores más bajos son mejores). En todas las figuras, en caso de que la nota de algún ítem del conjunto de test no pueda ser predicha, será descartado del conjunto de test, esto nos permite comparar la efectividad real de los recomendadores utilizados, en vez de utilizar cualquier otro valor por defecto. Primero, comparamos los cuatro predictores base.

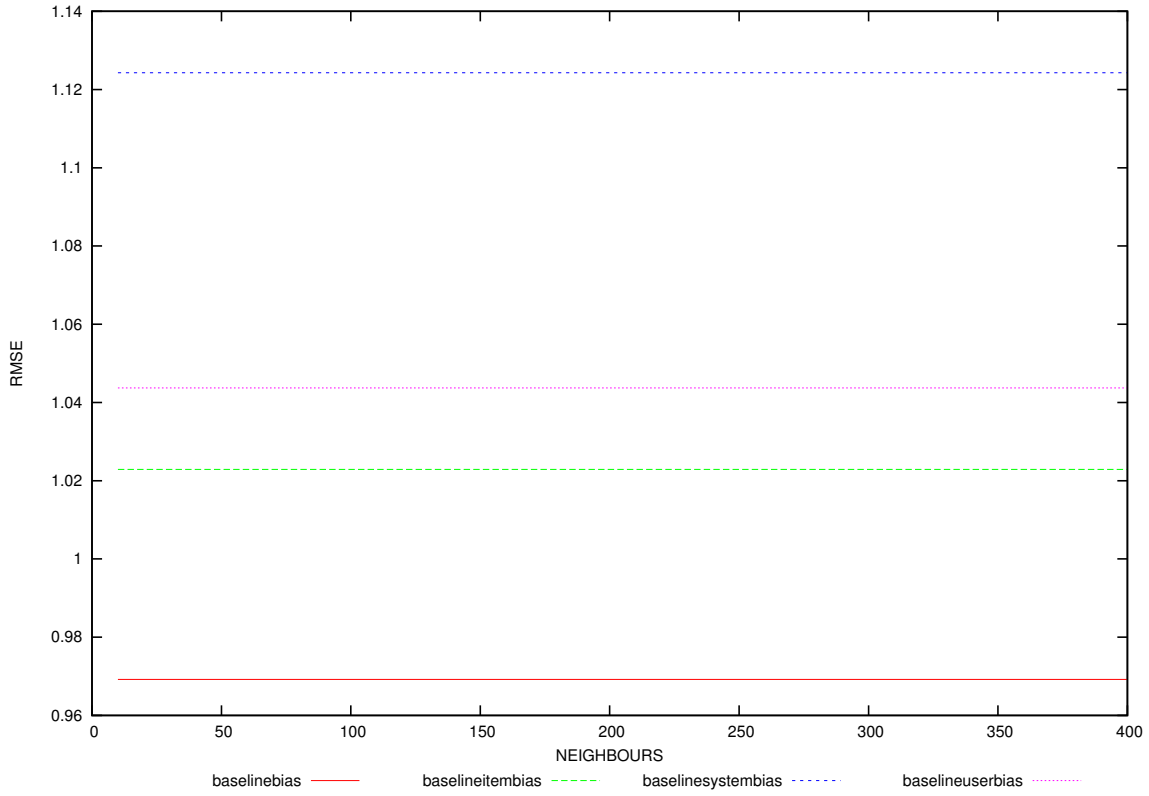

Figura 4.3: Comparativa de los cuatro predictores base.

De entre los cuatro predictores base, el etiquetado como baselinebias es el que obtiene mejores resultados. Dicho predictor es el que, recibiendo un usuario y un artículo, devuelve la media de los artículos totales, sumándole tanto el sesgo de dicho usuario como el sesgo de dicho ítem. El predictor baselineitembias es el que le suma el sesgo del artículo a predecir con el total del sistema, el baselinesystembias es el que devuelve únicamente la media de los artículos del sistema y el baselineuserbias devuelve la media de los artículos sumándole el sesgo del usuario. Dado que el predictor baselinebias es el mejor, será el empleado para compararlo con la correlación de Pearson, la similitud coseno y el algoritmo de la longitud de la subcadena común más larga.

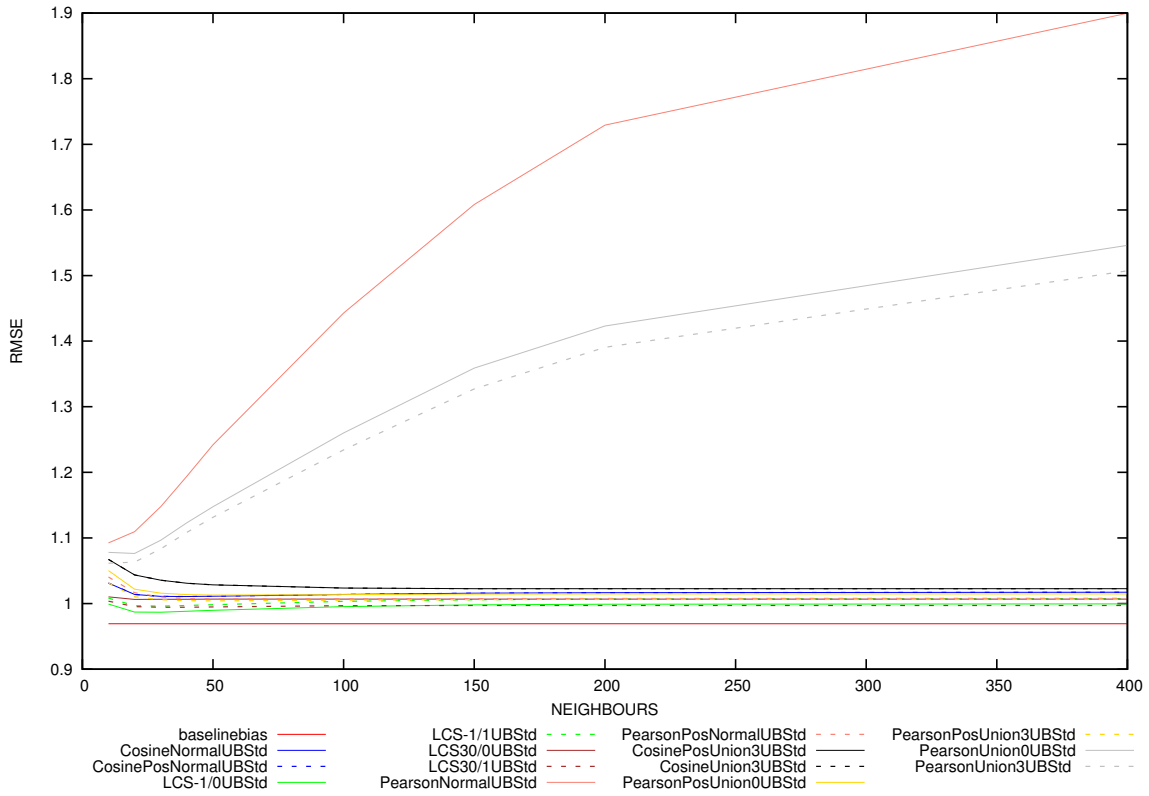

Figura 4.4: RMSE para distintos vecinos con la estrategia basada en usuarios.

En la imagen superior se comparan los resultados de los algoritmos implementados basados en usuario y sin ninguna normalización. Las tres versiones de Pearson hacen referencia a: la aplicación de Pearson con el conjunto de unión usando como valor por defecto 0 (PearsonUnion0) o valor 3 (PearsonUnion3), el algoritmo normal (PearsonNormal); además, como ya vimos se puede aplicar la variación en la que al algoritmo normal se eliminan los vecinos negativos (PearsonPosNormal), junto con sus distintas combinaciones (por ejemplo PearsonPosUnion0 indica que coge el conjunto de unión con valores 0 y además sólo los vecinos positivos). El algoritmo del coseno es equivalente, siendo las versiones el coseno normal y el coseno union con un valor por defecto de 3. Por último, en el algoritmo LCS tenemos también varias implementaciones. En todas ellas el primer número hace referencia al número mínimo de caracteres necesarios para considerar un vecino como próximo y el segundo número indica la desviación aceptada entre distintos ratings para un mismo ítem. De esta forma, el algoritmo etiquetado como LCS30/0 considera que en la subcadena común no puede haber diferencias de ratings y además cualquier vecino que tenga 30 o más elementos en común pueda considerarse válido.

En cuanto a los resultados mostrados, se puede ver cómo esta primera comparación de algoritmos no es capaz de batir al predictor base, aunque el algoritmo de la longitud de la subcadena común más larga es el que más se aproxima. Por otro lado, resalta que la correlación de Pearson con muchos vecinos tenga el peor comportamiento. De hecho, cuantos más vecinos empiezan a considerarse, el error continúa aumentando. Esto es debido a que toma en consideración todos los vecinos, incluso aquellos con similitudes negativas y termina siendo contraproducente. Se puede observar que el error de la versión de Pearson que sólo considera vecinos positivos es mucho menor. Aun así, esta comparación ha sido realizada basada en usuarios y sin aplicar ningún algoritmo de normalización. La siguiente figura muestra los mismos algoritmos, aplicando sobre ellos la normalización de mean centering.

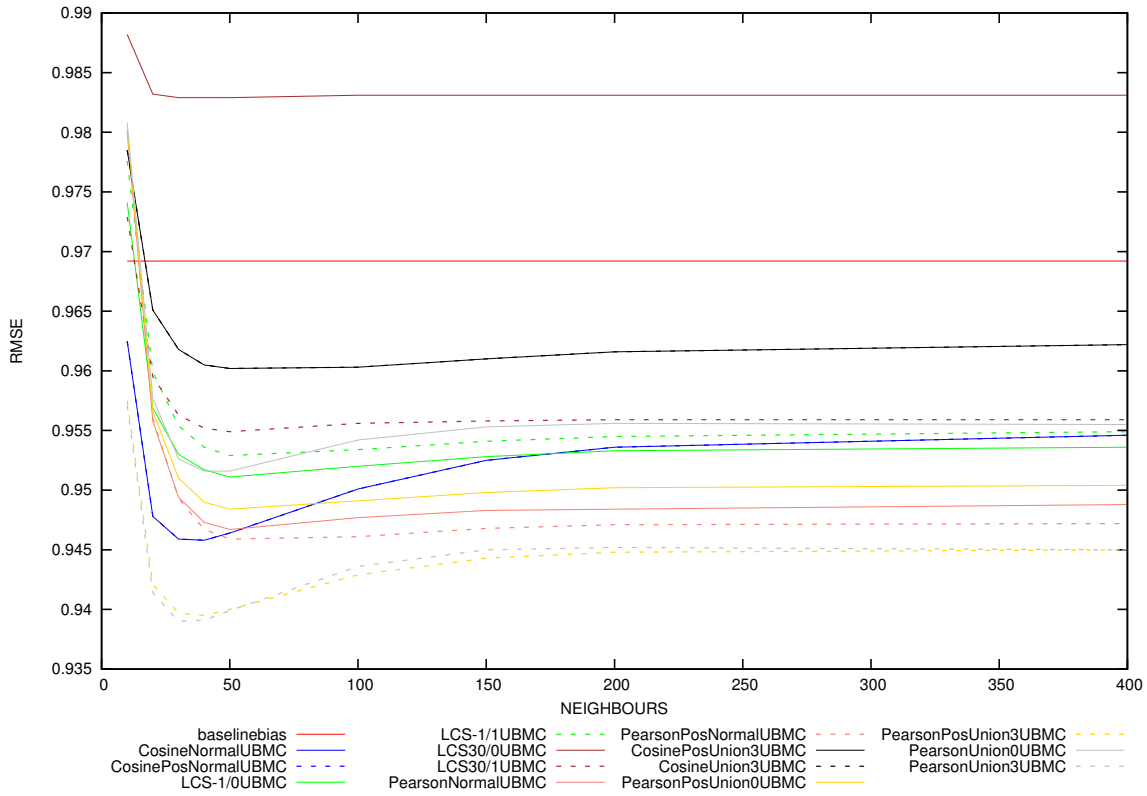

Figura 4.5: RMSE para distintos vecinos con la estrategia basada en usuarios con mean centering.

Hay tres cuestiones a tener en cuenta en esta gráfica. La primera es que la mayoría de los algoritmos superan al predictor base salvo una combinación de LCS. El mejor en este caso es el algoritmo de Pearson con Union de 3. El segundo aspecto es que parece ser que algunos de ellos encuentran un pico de vecinos (sobre 30) como valor óptimo de error. La última cuestión es que la normalización mediante mean centering produce resultados mucho mejores que la estrategia basada en usuarios estándar.

En las métricas de error, cuanto más cerca esté la nota predicha de la real, menor penalización habrá (es peor predecir un 5 cuando en realidad la nota de usuario es un 3, que predecir un 4). Por esta razón, si las predicciones se "centran" en la media del usuario, será más difícil obtener predicciones alejadas al historial del usuario, reduciendo en parte el error.

Comprobaremos ahora si dicho comportamiento se mantiene con las estrategias basada en artículos y, si además, la estrategia basada en artículos obtiene mejores resultados que la estrategia basada en usuarios. Como antes, primero se mostrarán los algoritmos estándar y después con la mejora de mean centering.

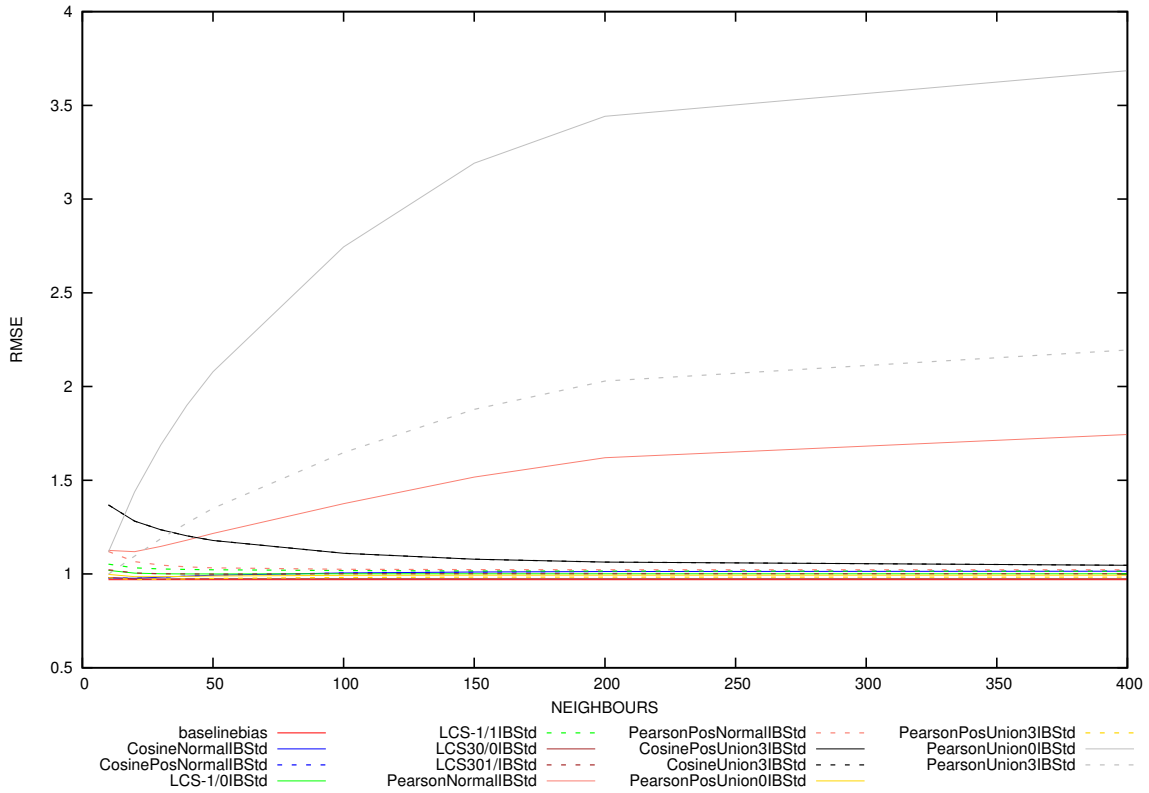

Figura 4.6: RMSE para distintos vecinos con la estrategia basada en artículos.

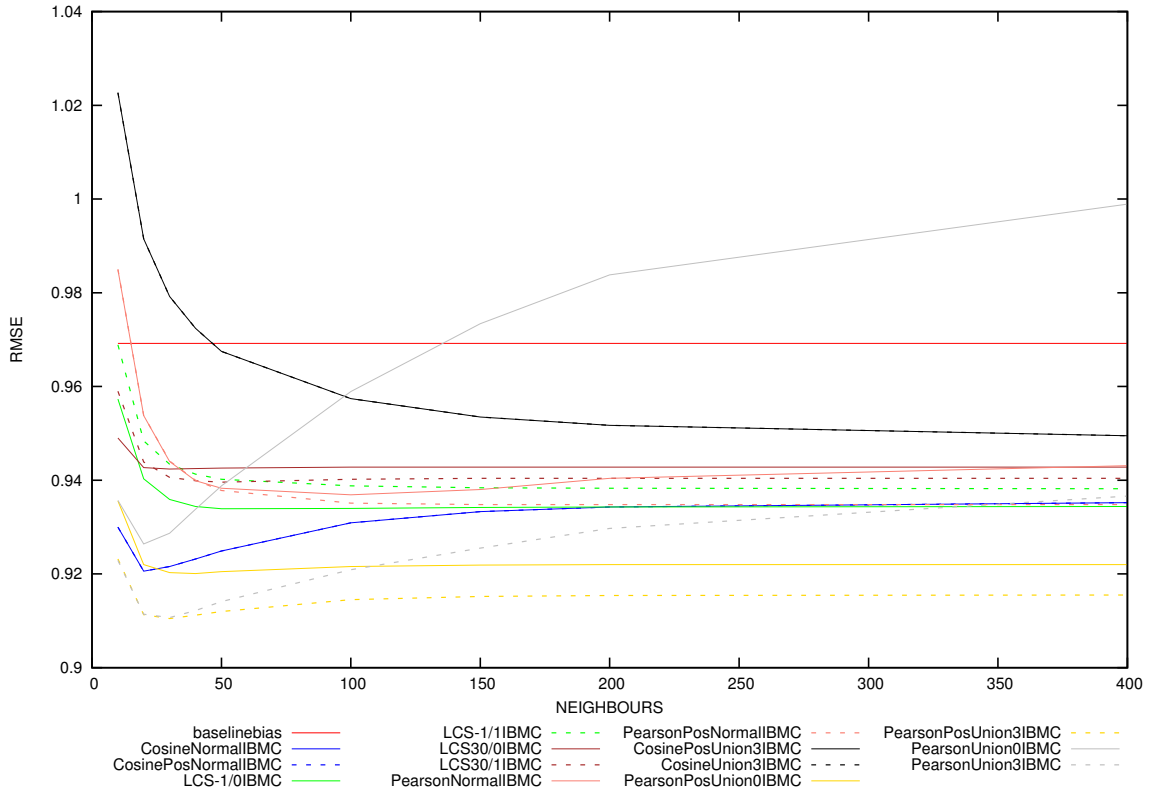

Figura 4.7: RMSE para distintos vecinos entre con la estrategia basada en artículos mean centering.

En el caso de la imagen 4.6, al igual que ocurría para el caso basado en usuarios, ninguno es capaz de ganar al predictor base que suma los sesgos, aunque muchos de ellos están muy cercanos, más que en la figura 4.4 anterior. No obstante, se pueden observar similitudes ya que los algoritmos de Pearson (Union0, Union3 y normal, con vecinos negativos) son los peores.

En el caso de la estrategia basada en artículos y con mean centering (figura 4.7), la mayoría de los algoritmos superan al predictor base, siendo el algoritmo de Pearson sin vecinos negativos con el conjunto de union 3, el mejor de todos ellos. Además, comparando con la gráfica 4.5, se puede ver cómo la estrategia basada en artículos obtiene mejores resultados de RMSE que la estrategia basada en usuarios. En general, los resultados son acordes con lo comentado en el estado del arte. Por un lado, el empleo de la estrategia basada en artículos produce unos mejores resultados que la basada en usuarios y por el otro, el empleo de algún mecanismo de normalización reduce en parte el error de las predicciones sin normalizar.

A continuación, comprobamos el comportamiento de estas medidas de similitud en las métricas de ranking (en este caso, valores más altos son mejores). Para comparar estas métricas, hemos eliminado del conjunto de test aquellos ítems que no han sido puntuados por el usuario con más de un 4. Esto se ha hecho así ya que un artículo que haya sido puntuado con un 1 o un 2 realmente no es un artículo relevante para el usuario. Además, si un artículo no podía predecirse, no se añadía al conjunto de los recomendados. Debido al tamaño de las imágenes, se han colocado en el Anexo B. Aunque todas los resultados están entre rangos válidos (las métricas de ranking deben estar siempre entre 0 y 1), son unos resultados bastante bajos. Según lo explicado en el apartado 3.3, los elementos devueltos para estas métricas son los ítems que no ha consumido el usuario en train ordenados de manera decreciente (los que potencialmente puedan ser puntuados mejor por el usuario van primero). No obstante, el usuario no ha ponderado tantos ítems como en un principio pueda parecer y muchos de los artículos recomendados, a pesar de que puedan tener buena nota por parte del usuario, por lo general no estarán en el conjunto de test, dando lugar a unos resultados muy bajos. Para resaltar esto, se ha añadido un baseline aleatorio (para cada artículo a predecir, se devuelve un valor aleatorio entre 1 y 5), que pone de evidencia lo difícil que resulta predecir en este contexto.

Los predictores base que devuelven la media total de los artículos junto con la que devuelve la media de los artículos más la media del usuario (baselinesystembias y baselineuserbias), devuelven la misma nota para todos los artículos (dependiendo de cada usuario en el caso del baselineuserbias). Esto hace que no sean recomendadores adecuados para estas métricas. No obstante, hay un algoritmo que consigue resultados bastante buenos: el LCS30/0IBMC; a pesar de que otras configuraciones del algoritmo LCS obtienen resultados bastante malos (incluso peores que el aleatorio), por lo que se puede observar que los distintos parámetros son muy importantes para conseguir resultados positivos. Dentro de las métricas estándar, PearsonPosNormal también obtiene buenos resultados. Por otro lado, se puede observar cómo en la imagen relativa a Recall@1000 (B.6), los resultados son los más altos. Es fácil ver según 2.18 que en caso de devolver el total de artículos del sistema, se obtendrá un valor de 1. En este caso, el valor más alto (0,5668) no está cerca del 1 debido a que no estamos devolviendo todos los artículos (1000 de los 1682 artículos que hay) y además, que aquellos cuya nota no se puede predecir, no se añaden a la lista de recomendados. Por otro lado, como se ha comentado, resulta sorprendente que el algoritmo de la subcadena común más larga, con la estrategia basada en artículos, obtenga los mejores resultados en casi todas las métricas.

En las imágenes B.8 y B.9 se pueden ver los datos mostrados en la imagen B.1 en forma de curvas. En concreto, esta métrica que tiene en cuenta el orden de preferencia del usuario (cuanto más coincida el orden de los artículos recomendados con el orden de los artículos reales en función de su nota, mejor), unido a que los artículos recomendados que no están en test serán considerados con relevancia 0, hace que en el momento en el que haya artículos recomendados que no han sido puntuados, el resultado baje abultadamente.

5

Conclusiones y trabajo futuro

Disponer de un sistema de recomendación útil es algo indispensable para las empresas que venden sus productos a través de Internet. Cada vez más tiendas online incorporan dichos programas para facilitar al usuario la elección de los productos en función de sus intereses [16]. Como se comentó anteriormente, buenas recomendaciones se traducen fácilmente tanto en el mantenimiento como en la aparición de nuevos clientes. Por esta razón, es conveniente continuar innovando para ofrecer a los clientes el mejor servicio posible.

A lo largo de este trabajo se han ido desarrollando las ideas fundamentales de los sistemas de recomendación. Además de haber programado un recomendador desde el principio (junto con algunas estructuras de datos), se ha probado un nuevo algoritmo que nunca antes se había aplicado en recomendación, el algoritmo de la subcadena común más larga, junto con parámetros configurables para poder realizar distintas predicciones. Se ha demostrado, en vista de los resultados obtenidos, que es posible aplicar algoritmos generales para un problema específico como es el de recomendación. No obstante, habrá que hacer modificaciones puntuales a dichos algoritmos para adaptarlos satisfactoriamente (como el caso de transformación de los datos de cadenas a enteros en LCS). Como trabajo futuro se ha pensado en continuar con esa idea y aplicar otros algoritmos y estructuras conocidos a los sistemas de recomendación. Además, se han hecho pruebas de las distintas estructuras de datos estudiadas y cómo afectan en el rendimiento (en términos de tiempo de ejecución y memoria) de los sistemas de recomendación. Como resultado de esto, se ha publicado un artículo en una conferencia [1] resumiendo los experimentos que aquí se han presentado.

Un trabajo futuro que puede resultar interesante es el problema del conjunto de cobertura, explicado en ([8, pp. 1117-1122]). Dado un conjunto de elementos (universo) y n conjuntos, el problema consiste en detectar el menor número de conjuntos cuya unión contenga todos los elementos del universo dado. No obstante, este problema es NP-Completo, por lo que no se ha encontrado el algoritmo óptimo que sea capaz de resolver este problema. A pesar de esto, existen aproximaciones que son capaces de devolver resultados cercanos a los óptimos. Aplicado a los sistemas de recomendación, sería posible almacenar a los ítems como conjuntos y a los usuarios como elementos de dichos conjuntos. Al ejecutar el algoritmo, podríamos determinar las películas más populares entre el conjunto de películas totales obteniendo el mínimo número de películas que hayan sido votadas por la mayoría de los usuarios.

Otro algoritmo potencialmente útil es el de OS-Select, que permite en un árbol rojo-negro estadístico obtener el i-ésimo elemento en un recorrido en orden. Aplicado a sistemas de recomendación, a la hora de recomendar a un usuario un cierto ítem, se puede tener almacenado los artículos ponderados por el usuario en forma de árbol, elegir mediante OS-Select los k elementos mejor ponderados y obtener los artículos más similares a dichos elementos de manera rápida.

Una estructura de datos eficiente, junto con una representación acertada de los datos a calcular, también se traduce en una mejora de rendimiento en el procesamiento de los datos, por esta razón, sería conveniente diseñar una estructura específica para estos sistemas. Por ejemplo, en el sistema desarrollado, cuando se calculaba la similitud entre dos usuarios (u y v), ésta se calculaba dos veces, una cuando se calculaba u con v y otra cuando era v con u. Si hubiese alguna forma de almacenar los datos de forma distinta, no sería necesario volver a calcular dicho valor. Otra opción, sería una estructura probabilística. En muchos casos, los usuarios de un sistema tienen comportamientos muy dispares. Unos acceden para recibir recomendaciones de manera muy frecuente y otros en intervalos muy puntuales. En lugar de cargar todas las posibilidades en memoria, sería interesante mantener únicamente aquellos que tengan más probabilidad de entrar en el sistema y de esta manera agilizar las búsquedas.

Asimismo, este trabajo sólo ha sido una introducción a estos sistemas. A pesar de haber comentado de manera general las características de los distintos tipos de sistemas de recomendación, solo se ha efectuado una investigación de técnicas de filtrado colaborativo basadas en vecinos. Sería conveniente hacer una investigación en otras aproximaciones, como los métodos basados en factorización de matrices o directamente ampliar los conocimientos de los sistemas recomendación investigando sobre los basados en contenido o basados en comunidad.

Por último, la URL del repositorio donde se encuentra el código fuente del sistema desarrollado es la siguiente: `https://bitbucket.org/PabloSanchezP/dt4recsys`

Bibliografía

[1] Pablo Sánchez, Alejandro Bellogín, and Iván Cantador. Studying the effect of data structures on the efficiency of collaborative filtering systems. In *CERI '16, June 14 - 16, 2016, Granada, Spain*, 2016.

[2] Internet users. `http://www.internetlivestats.com/internet-users/`. Accessed: 6/10/2015.

[3] Premio netflix. `http://www.netflixprize.com/`. Accessed: 6/10/2015.

[4] Netflix no implementó el algoritmo ganador. `http://www.wired.com/2012/04/netflix-prize-costs/`. Accessed: 2/27/2016.

[5] Subcripciones netflix. `https://es.wikipedia.org/wiki/Netflix`. Accessed: 6/10/2015.

[6] Cuentas de usuario de amazon. `http://www.statista.com/statistics/237810/number-of-active-amazon-customer-accounts-worldwide/`. Accessed: 6/10/2015.

[7] How to connect the worlds of content and commerce. `http://www.hinttech.com/article/how-to-connect-the-worlds-of-content-and-commerce/`. Accessed: 19/3/2016.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[9] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors. *Recommender Systems Handbook*. Springer, 2011.

[10] Francesco Ricci. Recommender systems: Models and techniques. In *Encyclopedia of Social Network Analysis and Mining*, pages 1511–1522. 2014.

[11] Robin D. Burke. Hybrid web recommender systems. In *The Adaptive Web, Methods and Strategies of Web Personalization*, pages 377–408, 2007.

[12] Robin D. Burke. Hybrid recommender systems: Survey and experiments. *User Model. User-Adapt. Interact.*, 12(4):331–370, 2002.

[13] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pages 426–434, 2008.

[14] O. Celma. *Music Recommendation and Discovery in the Long Tail*. PhD thesis, Universitat Pompeu Fabra, Barcelona, Spain, 2008.

[15] Dietmar Jannach and Gerhard Friedrich. Tutorial: Recommender systems. `http://ijcai13.org/files/tutorial_slides/td3.pdf`, 2013. International Joint Conference on Artificial Intelligence Beijing, August 4, 2013.

[16] J. Ben Schafer, Joseph A. Konstan, and John Riedl. E-commerce recommendation applications. *Data Min. Knowl. Discov.*, 5(1/2):115–153, 2001.

A

Estructuras y pseudocódigos

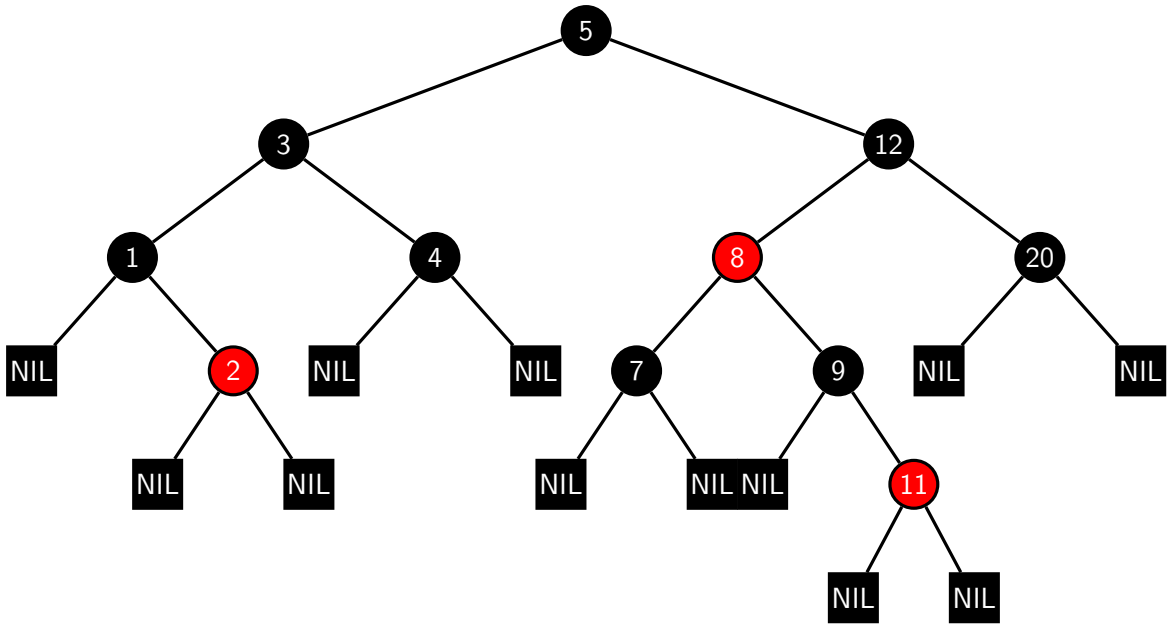

Figura A.1: Ejemplo de árbol rojo-negro.

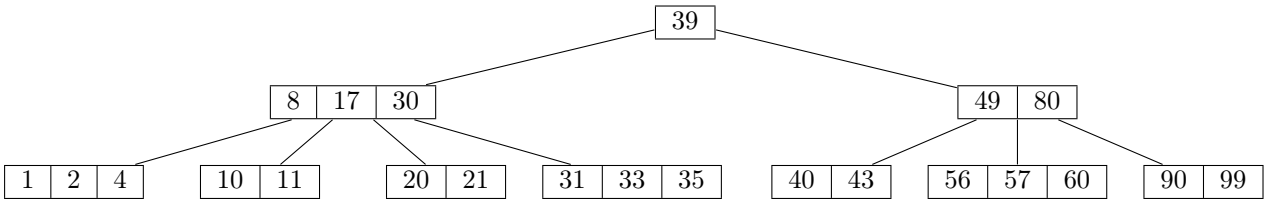

Figura A.2: Ejemplo de árbol B con t = 3.

function INCREASE KEY(Heap, i,key)
  if $key < Heap[i]$ **then**
    return Error
  end if
  $Heap[i] \leftarrow key$
  while $i > 1$ and $Heap[parent(i)] < Heap[i]$ **do**
    swap Heap[i] with Heap[parent(i)]
    $i \leftarrow parent(i)$
  end while
end function

Figura A.3: Incrementar clave en una cola de prioridad máxima.

```
function MakeSet(x)
    x.p ← x
    x.rank = 0
end function
function Union(x)
    Link(findSet(x), findSet(y))
end function
function Link(x,y)
    if x.rank > y.rank then
        y.p ← x
    else
        x.p ← y
        if x.rank == y.rank then
            y.rank ← y.rank + 1
        end if
    end if
end function
function findSet(x)
    if x ≠ x.p then
        x.p ← findSet(x.p)
    end if
    return x.p
end function
```

Figura A.4: Principales operaciones de la estructura de conjuntos disjuntos con compresión de caminos y unión por rango.

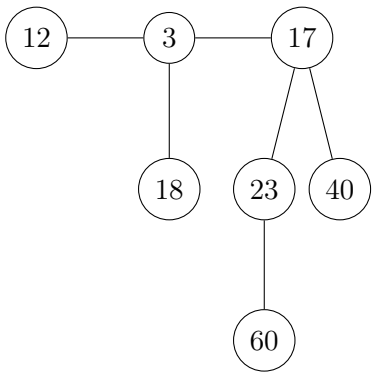

Figura A.5: Heap de Fibonacci. La raíz sería el elemento 3.

```
function LCS(string1, string2)
    m ← length(string1)
    n ← length(string2)
    b ← [1..m, 1..n]
    c ← [0..m, 0..n]
    c ← Initialize0()
    for i = 1 to m do
        for j = 1 to n do
            if string1[i] == string2[j] then
                c[i, j] ← c[i − 1, j − 1] + 1
                b[i, j] ← LeftUp
            else if c[i − 1, j] ≥ c[i, j − 1] then
                c[i, j] ← c[i − 1, j]
                b[i, j] ← Up
            else
                c[i, j] ← c[i, j − 1]
                b[i, j] ← Left
            end if
        end for
    end for
    return c and b
end function
```

Figura A.6: Longitud de la subcadena común más larga.

```
function PrintLCS(b, string1, i,j)
    if i == 0 or j == 0 then
        return
    end if
    if b[i, j] == LeftUp then
        PrintLCS(b, string1, i − 1, j − 1)
        print string[i]
    else if b[i, j] == Up then
        PrintLCS(b, string1, i − 1, j)
    else
        PrintLCS(b, string1, i, j − 1)
    end if
end function
```

Figura A.7: Obtener la subcadena común más larga. En la primera llamada i y j deben ser la longitud de la primera y segunda cadena, respectivamente.

```
function MatrixOptimalOrder(p)
    n ← p.length − 1
    m ← [1..n, 1..n]
    s ← [1..n − 1, 2..n]
    for i = 1 to n do
        m[i, i] ← 0
    end for
    for l = 2 to n do
        for i = 1 to n − l + 1 do
            j ← i + l − 1
            m[i, j] ← ∞
            for k = i to j − 1 do
                q ← m[i, k] + m[k + 1, j] + p_{i−1} * p_k * p_j
                if q < m[i, j] then
                    m[i, j] ← q
                    s[i, j] ← k
                end if
            end for
        end for
    end for
    return m and s
end function
```

Figura A.8: Obtener el orden óptimo del producto de matrices.

```
function BFS(Graph,Node)
    for each u in Graph.V do
        u.v ← notVisited , u.d ← ∞ , u.p ← null
    end for
    Node.v ← visiting
    Node.d ← 0
    Node.p ← null
    initQueue(Q)
    AddQueue(Q, Node)
    while EmptyQueue(Q) == false do
        u ← Extract(Q)
        for each v adyacent to u do
            if v.v == notVisited then
                v.v = visiting
                v.d = u.d + 1
                v.p = u
                AddQueue(Q, v)
            end if
        end for
        u.color = visited
    end while
end function
```

Figura A.9: Búsqueda primero en anchura.

```
function DFS(Graph)
    for each u ∈ Graph.V do
        u.v ← notVisited , u.p ← null
    end for
    t ← 0
    for each u ∈ Graph.V do
        if u.v == notVisited then
            DFSVISIT(u, Graph)
        end if
    end for
end function
function DFSVISIT(Node, Graph)
    Node.v ← visiting
    t ← t + 1
    Node.d ← t
    for each v adyacent to Node do
        if v.v == notVisited then
            v.p ← Node
            DFSVISIT(v, Graph)
        end if
    end for
    Node.v ← visited
    t ← t + 1
    Node.f ← t
end function
```

Figura A.10: Búsqueda primero en profundidad.

```
function DIJKSTRA(Graph, Node)
    for each u in Graph.V do
        u.v ← notVisited , u.d ← ∞ , u.p ← null
    end for
    initQueue(Q)
    Node.d ← 0
    AddPQueue(Q, Node)
    while EmptyQueue(Q) == false do
        w ← ExtractPQueue(Q)
        if w.v == false then
            w.v ← visited
            for each z adyacent to w do
                if z.d > w.d + c(w, z) then
                    z.d ← w.d + c(w, z)
                    z.p ← w
                    AddPQueue(Q, z)
                end if
            end for
        end if
    end while
end function
```

Figura A.11: Algoritmo de Dijkstra.

B

Resultados Métricas de Ranking

A continuación se muestran los resultados de las métricas de Ranking para el dataset de Movielens. Los resultados han sido obtenidos siguiendo el mismo procedimiento que en rating prediction (5 folds en cada uno de ellos, con un 80 % en el conjunto de train y un 20 % en el conjunto de test).

Algorithm	10 Neighbours	20 Neighbours	30 Neighbours	40 Neighbours	50 Neighbours	100 Neighbours	150 Neighbours	200 Neighbours	400 Neighbours
baselinebias	0,0005								
baselinerandom	0,0101								
CosineNormalIBMC	0,0086	0,0038	0,0024	0,0018	0,0014	0,0008	0,0007	0,0007	0,0006
CosineNormalIBStd	0,0445	0,0379	0,0319	0,0281	0,0223	0,0091	0,0031	0,0010	0,0002
CosineNormalUBMC	0,0107	0,0043	0,0025	0,0016	0,0013	0,0005	0,0003	0,0003	0,0003
CosineNormalUBStd	0,0047	0,0014	0,0009	0,0007	0,0006	0,0005	0,0004	0,0004	0,0004
CosinePosNormalIBMC	0,0086	0,0038	0,0024	0,0018	0,0014	0,0008	0,0007	0,0007	0,0006
CosinePosNormalIBStd	0,0445	0,0379	0,0319	0,0281	0,0223	0,0091	0,0031	0,0010	0,0002
CosinePosNormalUBMC	0,0107	0,0043	0,0025	0,0016	0,0013	0,0005	0,0003	0,0003	0,0003
CosinePosNormalUBStd	0,0047	0,0014	0,0009	0,0007	0,0006	0,0005	0,0004	0,0004	0,0004
LCS-1/0IBMC	0,0038	0,0020	0,0016	0,0012	0,0012	0,0009	0,0009	0,0009	0,0008
LCS-1/0IBStd	0,0247	0,0165	0,0130	0,0098	0,0078	0,0024	0,0007	0,0003	0,0002
LCS-1/0UBMC	0,0082	0,0026	0,0013	0,0010	0,0006	0,0004	0,0003	0,0003	0,0003
LCS-1/0UBStd	0,0094	0,0011	0,0006	0,0005	0,0004	0,0004	0,0004	0,0004	0,0004
LCS-1/1IBMC	0,0032	0,0021	0,0013	0,0011	0,0010	0,0009	0,0009	0,0009	0,0009
LCS-1/1IBStd	0,0222	0,0182	0,0153	0,0129	0,0109	0,0039	0,0014	0,0004	0,0001
LCS-1/1UBMC	0,0097	0,0033	0,0014	0,0011	0,0009	0,0004	0,0004	0,0004	0,0004
LCS-1/1UBStd	0,0052	0,0009	0,0006	0,0006	0,0005	0,0004	0,0004	0,0004	0,0004
LCS30/0IBMC	0,0800	0,0766	0,0754	0,0746	0,0744	0,0742	0,0742	0,0742	0,0742
LCS30/0IBStd	0,0624	0,0573	0,0566	0,0565	0,0564	0,0564	0,0564	0,0564	0,0564
LCS30/0UBMC	0,0134	0,0125	0,0123	0,0122	0,0122	0,0122	0,0122	0,0122	0,0122
LCS30/0UBStd	0,0151	0,0139	0,0139	0,0139	0,0139	0,0139	0,0139	0,0139	0,0139
LCS30/1IBMC	0,0531	0,0474	0,0455	0,0442	0,0432	0,0427	0,0426	0,0425	0,0425
LCS30/1IBStd	0,0276	0,0227	0,0221	0,0220	0,0219	0,0218	0,0218	0,0218	0,0218
LCS30/1UBMC	0,0090	0,0066	0,0061	0,0060	0,0059	0,0058	0,0058	0,0058	0,0058
LCS30/1UBStd	0,0084	0,0069	0,0068	0,0068	0,0067	0,0067	0,0067	0,0067	0,0067
PearsonNormalIBMC	0,0383	0,0369	0,0342	0,0325	0,0315	0,0287	0,0280	0,0277	0,0278
PearsonNormalIBStd	0,0098	0,0077	0,0061	0,0053	0,0043	0,0027	0,0020	0,0016	0,0013
PearsonNormalUBMC	0,0008	0,0003	0,0003	0,0003	0,0002	0,0002	0,0002	0,0002	0,0002
PearsonNormalUBStd	0,0089	0,0073	0,0071	0,0070	0,0072	0,0078	0,0078	0,0068	0,0008
PearsonPosNormalIBMC	0,0383	0,0364	0,0335	0,0317	0,0308	0,0276	0,0270	0,0268	0,0267
PearsonPosNormalIBStd	0,0096	0,0072	0,0053	0,0044	0,0032	0,0017	0,0014	0,0012	0,0012
PearsonPosNormalUBMC	0,0019	0,0009	0,0006	0,0005	0,0004	0,0004	0,0004	0,0004	0,0004
PearsonPosNormalUBStd	0,0025	0,0008	0,0007	0,0006	0,0006	0,0005	0,0005	0,0005	0,0005
CosinePosUnion3IBMC	0,0098	0,0051	0,0033	0,0024	0,0017	0,0009	0,0009	0,0006	0,0006
CosinePosUnion3IBStd	0,0155	0,0140	0,0137	0,0118	0,0103	0,0075	0,0072	0,0064	0,0061
CosinePosUnion3UBMC	0,0008	0,0003	0,0003	0,0003	0,0003	0,0003	0,0003	0,0003	0,0003
CosinePosUnion3UBStd	0,0012	0,0005	0,0004	0,0005	0,0004	0,0004	0,0003	0,0003	0,0003
CosineUnion3IBMC	0,0098	0,0051	0,0033	0,0024	0,0017	0,0009	0,0009	0,0006	0,0006
CosineUnion3IBStd	0,0155	0,0140	0,0137	0,0118	0,0103	0,0075	0,0072	0,0064	0,0061
CosineUnion3UBMC	0,0008	0,0003	0,0003	0,0003	0,0003	0,0003	0,0003	0,0003	0,0003
CosineUnion3UBStd	0,0012	0,0005	0,0004	0,0005	0,0004	0,0004	0,0003	0,0003	0,0003
PearsonPosUnion0IBMC	0,0063	0,0034	0,0023	0,0020	0,0019	0,0017	0,0017	0,0017	0,0017
PearsonPosUnion0IBStd	0,0130	0,0059	0,0035	0,0023	0,0018	0,0014	0,0013	0,0013	0,0013
PearsonPosUnion0UBMC	0,0210	0,0076	0,0039	0,0025	0,0016	0,0006	0,0005	0,0005	0,0005
PearsonPosUnion0UBStd	0,0031	0,0009	0,0007	0,0006	0,0007	0,0006	0,0006	0,0006	0,0006
PearsonPosUnion3IBMC	0,0067	0,0032	0,0023	0,0019	0,0017	0,0015	0,0015	0,0015	0,0015
PearsonPosUnion3IBStd	0,0243	0,0137	0,0088	0,0062	0,0046	0,0016	0,0012	0,0011	0,0011
PearsonPosUnion3UBMC	0,0051	0,0023	0,0013	0,0008	0,0007	0,0005	0,0005	0,0005	0,0005
PearsonPosUnion3UBStd	0,0039	0,0010	0,0006	0,0006	0,0006	0,0004	0,0003	0,0003	0,0003
PearsonUnion0IBMC	0,0062	0,0036	0,0026	0,0026	0,0027	0,0026	0,0026	0,0027	0,0028
PearsonUnion0IBStd	0,0124	0,0056	0,0035	0,0027	0,0026	0,0040	0,0053	0,0062	0,0071
PearsonUnion0UBMC	0,0123	0,0049	0,0027	0,0015	0,0009	0,0003	0,0002	0,0002	0,0002
PearsonUnion0UBStd	0,0106	0,0083	0,0075	0,0072	0,0069	0,0072	0,0074	0,0062	0,0013
PearsonUnion3IBMC	0,0069	0,0036	0,0030	0,0029	0,0029	0,0030	0,0032	0,0033	0,0035
PearsonUnion3IBStd	0,0240	0,0137	0,0095	0,0074	0,0067	0,0070	0,0092	0,0106	0,0116
PearsonUnion3UBMC	0,0028	0,0011	0,0006	0,0004	0,0002	0,0001	0,0001	0,0001	0,0001
PearsonUnion3UBStd	0,0112	0,0085	0,0081	0,0075	0,0072	0,0068	0,0065	0,0058	0,0012

Figura B.1: Resultados para NDCG@10.

Algorithm	10 Neighbours	20 Neighbours	30 Neighbours	40 Neighbours	50 Neighbours	100 Neighbours	150 Neighbours	200 Neighbours	400 Neighbours
baselinebias	0,2444								
baselinerandom	0,1539								
CosineNormalIBMC	0,2503	0,2498	0,2490	0,2484	0,2479	0,2465	0,2460	0,2458	0,2457
CosineNormalIBStd	0,2334	0,2308	0,2257	0,2215	0,2164	0,2024	0,1953	0,1922	0,1899
CosineNormalUBMC	0,2465	0,2448	0,2439	0,2430	0,2418	0,2386	0,2365	0,2352	0,2341
CosineNormalUBStd	0,2526	0,2532	0,2531	0,2523	0,2516	0,2492	0,2476	0,2467	0,2459
CosinePosNormalIBMC	0,2503	0,2498	0,2490	0,2484	0,2479	0,2465	0,2460	0,2458	0,2457
CosinePosNormalIBStd	0,2334	0,2308	0,2257	0,2215	0,2164	0,2024	0,1953	0,1922	0,1899
CosinePosNormalUBMC	0,2465	0,2448	0,2439	0,2430	0,2418	0,2386	0,2365	0,2352	0,2341
CosinePosNormalUBStd	0,2526	0,2532	0,2531	0,2523	0,2516	0,2492	0,2476	0,2467	0,2459
LCS-1/0IBMC	0,2470	0,2480	0,2477	0,2473	0,2469	0,2462	0,2460	0,2459	0,2458
LCS-1/0IBStd	0,2260	0,2270	0,2256	0,2235	0,2216	0,2153	0,2122	0,2109	0,2100
LCS-1/0UBMC	0,2467	0,2449	0,2431	0,2418	0,2406	0,2373	0,2356	0,2348	0,2342
LCS-1/0UBStd	0,2562	0,2538	0,2531	0,2522	0,2513	0,2486	0,2473	0,2466	0,2461
LCS-1/1IBMC	0,2464	0,2477	0,2476	0,2473	0,2470	0,2464	0,2462	0,2461	0,2461
LCS-1/1IBStd	0,2222	0,2253	0,2240	0,2223	0,2207	0,2145	0,2110	0,2093	0,2082
LCS-1/1UBMC	0,2481	0,2463	0,2443	0,2429	0,2418	0,2378	0,2357	0,2346	0,2337
LCS-1/1UBStd	0,2528	0,2525	0,2519	0,2513	0,2507	0,2484	0,2472	0,2466	0,2461
LCS30/0IBMC	0,2415	0,2392	0,2387	0,2384	0,2382	0,2381	0,2381	0,2381	0,2381
LCS30/0IBStd	0,2319	0,2283	0,2273	0,2268	0,2266	0,2265	0,2265	0,2265	0,2265
LCS30/0UBMC	0,0948	0,0941	0,0938	0,0937	0,0937	0,0936	0,0936	0,0936	0,0936
LCS30/0UBStd	0,0965	0,0959	0,0957	0,0957	0,0956	0,0956	0,0956	0,0956	0,0956
LCS30/1IBMC	0,2722	0,2696	0,2682	0,2674	0,2668	0,2662	0,2660	0,2660	0,2660
LCS30/1IBStd	0,2515	0,2458	0,2435	0,2417	0,2405	0,2384	0,2381	0,2381	0,2381
LCS30/1UBMC	0,1615	0,1599	0,1588	0,1581	0,1576	0,1565	0,1563	0,1562	0,1562
LCS30/1UBStd	0,1646	0,1640	0,1635	0,1630	0,1627	0,1620	0,1619	0,1619	0,1619
PearsonNormalIBMC	0,2671	0,2695	0,2695	0,2690	0,2686	0,2675	0,2672	0,2671	0,2671
PearsonNormalIBStd	0,1793	0,1930	0,2017	0,2072	0,2111	0,2223	0,2247	0,2243	0,2223
PearsonNormalUBMC	0,2132	0,2154	0,2164	0,2171	0,2175	0,2192	0,2194	0,2186	0,2152
PearsonNormalUBStd	0,2584	0,2647	0,2674	0,2691	0,2705	0,2694	0,2592	0,2445	0,2086
PearsonPosNormalIBMC	0,2669	0,2689	0,2685	0,2679	0,2676	0,2663	0,2660	0,2659	0,2659
PearsonPosNormalIBStd	0,1769	0,1820	0,1846	0,1862	0,1871	0,1895	0,1903	0,1905	0,1906
PearsonPosNormalUBMC	0,2258	0,2298	0,2313	0,2322	0,2325	0,2329	0,2329	0,2328	0,2328
PearsonPosNormalUBStd	0,2418	0,2437	0,2443	0,2443	0,2442	0,2436	0,2432	0,2431	0,2430
CosinePosUnion3IBMC	0,2434	0,2439	0,2440	0,2437	0,2437	0,2439	0,2442	0,2441	0,2443
CosinePosUnion3IBStd	0,0708	0,0647	0,0614	0,0570	0,0537	0,0475	0,0454	0,0445	0,0434
CosinePosUnion3UBMC	0,2289	0,2314	0,2313	0,2312	0,2307	0,2305	0,2304	0,2307	0,2311
CosinePosUnion3UBStd	0,2403	0,2447	0,2455	0,2460	0,2459	0,2461	0,2460	0,2456	0,2444
CosineUnion3IBMC	0,2434	0,2439	0,2440	0,2437	0,2437	0,2439	0,2442	0,2441	0,2443
CosineUnion3IBStd	0,0708	0,0647	0,0614	0,0570	0,0537	0,0475	0,0454	0,0445	0,0434
CosineUnion3UBMC	0,2289	0,2314	0,2313	0,2312	0,2307	0,2305	0,2304	0,2307	0,2311
CosineUnion3UBStd	0,2403	0,2447	0,2455	0,2460	0,2459	0,2461	0,2460	0,2456	0,2444
PearsonPosUnion0IBMC	0,2454	0,2452	0,2448	0,2446	0,2443	0,2440	0,2439	0,2439	0,2439
PearsonPosUnion0IBStd	0,1937	0,1889	0,1867	0,1857	0,1850	0,1839	0,1838	0,1837	0,1837
PearsonPosUnion0UBMC	0,2571	0,2529	0,2504	0,2488	0,2476	0,2443	0,2429	0,2423	0,2420
PearsonPosUnion0UBStd	0,2486	0,2511	0,2516	0,2518	0,2518	0,2513	0,2508	0,2505	0,2503
PearsonPosUnion3IBMC	0,2480	0,2483	0,2478	0,2474	0,2470	0,2464	0,2463	0,2462	0,2462
PearsonPosUnion3IBStd	0,2086	0,2040	0,2016	0,1998	0,1988	0,1964	0,1958	0,1957	0,1956
PearsonPosUnion3UBMC	0,2424	0,2432	0,2431	0,2428	0,2423	0,2411	0,2403	0,2399	0,2397
PearsonPosUnion3UBStd	0,2511	0,2524	0,2524	0,2521	0,2517	0,2500	0,2490	0,2486	0,2483
PearsonUnion0IBMC	0,2456	0,2469	0,2474	0,2477	0,2479	0,2481	0,2485	0,2486	0,2488
PearsonUnion0IBStd	0,1968	0,1995	0,2023	0,2046	0,2063	0,2106	0,2145	0,2171	0,2206
PearsonUnion0UBMC	0,2473	0,2442	0,2423	0,2409	0,2399	0,2386	0,2383	0,2382	0,2355
PearsonUnion0UBStd	0,2627	0,2677	0,2696	0,2711	0,2722	0,2736	0,2697	0,2627	0,2359
PearsonUnion3IBMC	0,2482	0,2499	0,2505	0,2509	0,2511	0,2517	0,2521	0,2523	0,2524
PearsonUnion3IBStd	0,2113	0,2147	0,2183	0,2205	0,2220	0,2252	0,2255	0,2259	0,2265
PearsonUnion3UBMC	0,2339	0,2341	0,2339	0,2338	0,2336	0,2335	0,2337	0,2334	0,2311
PearsonUnion3UBStd	0,2642	0,2687	0,2703	0,2714	0,2724	0,2728	0,2677	0,2602	0,2338

Figura B.2: Resultados para NDCG@1000.

Algorithm	10 Neighbours	20 Neighbours	30 Neighbours	40 Neighbours	50 Neighbours	100 Neighbours	150 Neighbours	200 Neighbours	400 Neighbours
baselinebias	0,0006								
baselinerandom	0,0103								
CosineNormalIBMC	0,0115	0,0052	0,0032	0,0024	0,0018	0,0008	0,0007	0,0007	0,0006
CosineNormalIBStd	0,0451	0,0395	0,0333	0,0299	0,0251	0,0105	0,0036	0,0012	0,0002
CosineNormalUBMC	0,0140	0,0062	0,0036	0,0023	0,0018	0,0007	0,0004	0,0004	0,0004
CosineNormalUBStd	0,0052	0,0022	0,0014	0,0010	0,0008	0,0007	0,0006	0,0006	0,0006
CosinePosNormalIBMC	0,0115	0,0052	0,0032	0,0024	0,0018	0,0008	0,0007	0,0007	0,0006
CosinePosNormalIBStd	0,0451	0,0395	0,0333	0,0299	0,0251	0,0105	0,0036	0,0012	0,0002
CosinePosNormalUBMC	0,0140	0,0062	0,0036	0,0023	0,0018	0,0007	0,0004	0,0004	0,0004
CosinePosNormalUBStd	0,0052	0,0022	0,0014	0,0010	0,0008	0,0007	0,0006	0,0006	0,0006
LCS-1/0IBMC	0,0055	0,0028	0,0022	0,0014	0,0014	0,0010	0,0010	0,0009	0,0009
LCS-1/0IBStd	0,0270	0,0199	0,0160	0,0127	0,0104	0,0034	0,0011	0,0004	0,0003
LCS-1/0UBMC	0,0110	0,0037	0,0018	0,0014	0,0009	0,0004	0,0004	0,0004	0,0004
LCS-1/0UBStd	0,0060	0,0017	0,0009	0,0008	0,0006	0,0006	0,0005	0,0005	0,0005
LCS-1/1IBMC	0,0043	0,0030	0,0017	0,0014	0,0013	0,0011	0,0011	0,0011	0,0011
LCS-1/1IBStd	0,0240	0,0206	0,0187	0,0154	0,0136	0,0054	0,0019	0,0006	0,0002
LCS-1/1UBMC	0,0128	0,0045	0,0019	0,0015	0,0012	0,0005	0,0004	0,0004	0,0004
LCS-1/1UBStd	0,0039	0,0014	0,0009	0,0008	0,0006	0,0006	0,0005	0,0005	0,0005
LCS30/0IBMC	0,0805	0,0773	0,0756	0,0749	0,0748	0,0744	0,0744	0,0744	0,0744
LCS30/0IBStd	0,0604	0,0544	0,0536	0,0534	0,0533	0,0533	0,0533	0,0533	0,0533
LCS30/0UBMC	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
LCS30/0UBStd	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
LCS30/1IBMC	0,0572	0,0505	0,0486	0,0474	0,0462	0,0456	0,0455	0,0454	0,0454
LCS30/1IBStd	0,0289	0,0230	0,0221	0,0220	0,0218	0,0217	0,0217	0,0217	0,0217
LCS30/1UBMC	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
LCS30/1UBStd	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
PearsonNormalIBMC	0,0394	0,0400	0,0377	0,0372	0,0362	0,0334	0,0330	0,0324	0,0325
PearsonNormalIBStd	0,0103	0,0087	0,0070	0,0065	0,0054	0,0035	0,0026	0,0020	0,0016
PearsonNormalUBMC	0,0012	0,0004	0,0004	0,0003	0,0002	0,0002	0,0002	0,0002	0,0002
PearsonNormalUBStd	0,0120	0,0110	0,0106	0,0102	0,0106	0,0116	0,0112	0,0096	0,0010
PearsonPosNormalIBMC	0,0393	0,0396	0,0372	0,0367	0,0359	0,0331	0,0325	0,0324	0,0323
PearsonPosNormalIBStd	0,0101	0,0081	0,0063	0,0054	0,0040	0,0023	0,0019	0,0017	0,0017
PearsonPosNormalUBMC	0,0028	0,0012	0,0009	0,0006	0,0005	0,0005	0,0005	0,0005	0,0005
PearsonPosNormalUBStd	0,0021	0,0010	0,0008	0,0007	0,0007	0,0006	0,0006	0,0006	0,0006
CosinePosUnion3IBMC	0,0100	0,0055	0,0035	0,0025	0,0017	0,0008	0,0007	0,0006	0,0006
CosinePosUnion3IBStd	0,0126	0,0119	0,0117	0,0105	0,0087	0,0063	0,0059	0,0052	0,0048
CosinePosUnion3UBMC	0,0012	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005
CosinePosUnion3UBStd	0,0016	0,0009	0,0007	0,0007	0,0007	0,0006	0,0005	0,0005	0,0005
CosineUnion3IBMC	0,0100	0,0055	0,0035	0,0025	0,0017	0,0008	0,0007	0,0006	0,0006
CosineUnion3IBStd	0,0126	0,0119	0,0117	0,0105	0,0087	0,0063	0,0059	0,0052	0,0048
CosineUnion3UBMC	0,0012	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005
CosineUnion3UBStd	0,0016	0,0009	0,0007	0,0007	0,0007	0,0006	0,0005	0,0005	0,0005
PearsonPosUnion0IBMC	0,0084	0,0045	0,0029	0,0024	0,0022	0,0020	0,0019	0,0019	0,0019
PearsonPosUnion0IBStd	0,0141	0,0071	0,0040	0,0026	0,0019	0,0013	0,0012	0,0012	0,0012
PearsonPosUnion0UBMC	0,0259	0,0107	0,0058	0,0037	0,0023	0,0007	0,0006	0,0006	0,0006
PearsonPosUnion0UBStd	0,0031	0,0011	0,0009	0,0007	0,0008	0,0006	0,0006	0,0006	0,0006
PearsonPosUnion3IBMC	0,0090	0,0044	0,0030	0,0024	0,0021	0,0019	0,0019	0,0018	0,0018
PearsonPosUnion3IBStd	0,0249	0,0156	0,0105	0,0072	0,0056	0,0017	0,0012	0,0010	0,0010
PearsonPosUnion3UBMC	0,0070	0,0031	0,0018	0,0010	0,0007	0,0005	0,0005	0,0005	0,0005
PearsonPosUnion3UBStd	0,0033	0,0013	0,0009	0,0008	0,0008	0,0005	0,0004	0,0004	0,0004
PearsonUnion0IBMC	0,0085	0,0050	0,0035	0,0033	0,0034	0,0032	0,0031	0,0035	0,0034
PearsonUnion0IBStd	0,0139	0,0070	0,0043	0,0033	0,0032	0,0050	0,0067	0,0079	0,0091
PearsonUnion0UBMC	0,0174	0,0072	0,0042	0,0023	0,0014	0,0004	0,0003	0,0003	0,0003
PearsonUnion0UBStd	0,0133	0,0116	0,0104	0,0097	0,0092	0,0096	0,0095	0,0084	0,0019
PearsonUnion3IBMC	0,0092	0,0048	0,0040	0,0038	0,0039	0,0040	0,0042	0,0044	0,0047
PearsonUnion3IBStd	0,0249	0,0158	0,0114	0,0087	0,0080	0,0081	0,0106	0,0122	0,0136
PearsonUnion3UBMC	0,0041	0,0018	0,0009	0,0006	0,0004	0,0002	0,0002	0,0001	0,0001
PearsonUnion3UBStd	0,0140	0,0124	0,0117	0,0109	0,0101	0,0092	0,0089	0,0079	0,0016

Figura B.3: Resultados para Precision@10.

Algorithm	10 Neighbours	20 Neighbours	30 Neighbours	40 Neighbours	50 Neighbours	100 Neighbours	150 Neighbours	200 Neighbours	400 Neighbours
baselinebias	0,0149								
baselinerandom	0,0103								
CosineNormalIBMC	0,0149	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150
CosineNormalIBStd	0,0137	0,0138	0,0139	0,0139	0,0139	0,0138	0,0138	0,0138	0,0138
CosineNormalUBMC	0,0146	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147
CosineNormalUBStd	0,0148	0,0149	0,0149	0,0149	0,0149	0,0150	0,0150	0,0150	0,0150
CosinePosNormalIBMC	0,0149	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150
CosinePosNormalIBStd	0,0137	0,0138	0,0139	0,0139	0,0139	0,0138	0,0138	0,0138	0,0138
CosinePosNormalUBMC	0,0146	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147
CosinePosNormalUBStd	0,0148	0,0149	0,0149	0,0149	0,0149	0,0150	0,0150	0,0150	0,0150
LCS-1/0IBMC	0,0149	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150
LCS-1/0IBStd	0,0137	0,0141	0,0143	0,0143	0,0144	0,0144	0,0144	0,0144	0,0144
LCS-1/0UBMC	0,0146	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147
LCS-1/0UBStd	0,0149	0,0149	0,0149	0,0149	0,0149	0,0150	0,0150	0,0150	0,0150
LCS-1/1IBMC	0,0149	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150
LCS-1/1IBStd	0,0138	0,0142	0,0143	0,0144	0,0144	0,0144	0,0144	0,0144	0,0144
LCS-1/1UBMC	0,0146	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147	0,0147
LCS-1/1UBStd	0,0148	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149
LCS30/0IBMC	0,0670	0,0670	0,0670	0,0670	0,0670	0,0670	0,0670	0,0670	0,0670
LCS30/0IBStd	0,0670	0,0670	0,0670	0,0670	0,0670	0,0670	0,0670	0,0670	0,0670
LCS30/0UBMC	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
LCS30/0UBStd	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
LCS30/1IBMC	0,0359	0,0359	0,0359	0,0359	0,0359	0,0359	0,0359	0,0359	0,0359
LCS30/1IBStd	0,0359	0,0359	0,0359	0,0359	0,0359	0,0359	0,0359	0,0359	0,0359
LCS30/1UBMC	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
LCS30/1UBStd	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
PearsonNormalIBMC	0,0150	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151
PearsonNormalIBStd	0,0116	0,0123	0,0127	0,0130	0,0133	0,0141	0,0143	0,0143	0,0143
PearsonNormalUBMC	0,0141	0,0142	0,0142	0,0142	0,0142	0,0143	0,0144	0,0145	0,0145
PearsonNormalUBStd	0,0152	0,0153	0,0152	0,0151	0,0151	0,0149	0,0147	0,0146	0,0144
PearsonPosNormalIBMC	0,0150	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151
PearsonPosNormalIBStd	0,0116	0,0120	0,0123	0,0125	0,0126	0,0130	0,0131	0,0131	0,0131
PearsonPosNormalUBMC	0,0145	0,0147	0,0148	0,0148	0,0148	0,0148	0,0148	0,0148	0,0148
PearsonPosNormalUBStd	0,0148	0,0149	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150
CosinePosUnion3IBMC	0,0147	0,0148	0,0148	0,0148	0,0149	0,0149	0,0149	0,0149	0,0149
CosinePosUnion3IBStd	0,0038	0,0035	0,0034	0,0032	0,0030	0,0028	0,0026	0,0026	0,0025
CosinePosUnion3UBMC	0,0144	0,0146	0,0146	0,0146	0,0146	0,0146	0,0146	0,0146	0,0146
CosinePosUnion3UBStd	0,0147	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149
CosineUnion3IBMC	0,0147	0,0148	0,0148	0,0148	0,0149	0,0149	0,0149	0,0149	0,0149
CosineUnion3IBStd	0,0038	0,0035	0,0034	0,0032	0,0030	0,0028	0,0026	0,0026	0,0025
CosineUnion3UBMC	0,0144	0,0146	0,0146	0,0146	0,0146	0,0146	0,0146	0,0146	0,0146
CosineUnion3UBStd	0,0147	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149
PearsonPosUnion0IBMC	0,0149	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150
PearsonPosUnion0IBStd	0,0126	0,0127	0,0127	0,0128	0,0128	0,0129	0,0129	0,0129	0,0129
PearsonPosUnion0UBMC	0,0147	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149
PearsonPosUnion0UBStd	0,0149	0,0150	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151
PearsonPosUnion3IBMC	0,0149	0,0150	0,0151	0,0150	0,0150	0,0151	0,0151	0,0151	0,0151
PearsonPosUnion3IBStd	0,0129	0,0131	0,0132	0,0132	0,0132	0,0133	0,0134	0,0134	0,0134
PearsonPosUnion3UBMC	0,0147	0,0148	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149	0,0149
PearsonPosUnion3UBStd	0,0150	0,0150	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151
PearsonUnion0IBMC	0,0149	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150	0,0150
PearsonUnion0IBStd	0,0127	0,0130	0,0132	0,0133	0,0134	0,0135	0,0137	0,0138	0,0140
PearsonUnion0UBMC	0,0146	0,0146	0,0147	0,0147	0,0147	0,0147	0,0148	0,0149	0,0149
PearsonUnion0UBStd	0,0151	0,0153	0,0153	0,0152	0,0152	0,0151	0,0150	0,0149	0,0147
PearsonUnion3IBMC	0,0149	0,0150	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151	0,0151
PearsonUnion3IBStd	0,0130	0,0133	0,0136	0,0137	0,0138	0,0140	0,0139	0,0139	0,0140
PearsonUnion3UBMC	0,0145	0,0146	0,0146	0,0146	0,0146	0,0147	0,0147	0,0148	0,0148
PearsonUnion3UBStd	0,0152	0,0153	0,0153	0,0152	0,0152	0,0151	0,0150	0,0150	0,0148

Figura B.4: Resultados para Precision@1000.

Algorithm	10 Neighbours	20 Neighbours	30 Neighbours	40 Neighbours	50 Neighbours	100 Neighbours	150 Neighbours	200 Neighbours	400 Neighbours
baselinebias	0,0002								
baselinerandom	0,0037								
CosineNormalIBMC	0,0029	0,0011	0,0007	0,0005	0,0003	0,0002	0,0002	0,0002	0,0002
CosineNormalIBStd	0,0107	0,0085	0,0071	0,0062	0,0049	0,0020	0,0007	0,0003	0,0001
CosineNormalUBMC	0,0047	0,0024	0,0014	0,0009	0,0008	0,0003	0,0002	0,0002	0,0002
CosineNormalUBStd	0,0014	0,0006	0,0005	0,0004	0,0003	0,0002	0,0002	0,0002	0,0002
CosinePosNormalIBMC	0,0029	0,0011	0,0007	0,0005	0,0003	0,0002	0,0002	0,0002	0,0002
CosinePosNormalIBStd	0,0107	0,0085	0,0071	0,0062	0,0049	0,0020	0,0007	0,0003	0,0001
CosinePosNormalUBMC	0,0047	0,0024	0,0014	0,0009	0,0008	0,0003	0,0002	0,0002	0,0002
CosinePosNormalUBStd	0,0014	0,0006	0,0005	0,0004	0,0003	0,0002	0,0002	0,0002	0,0002
LCS-1/0IBMC	0,0013	0,0006	0,0005	0,0004	0,0004	0,0003	0,0003	0,0003	0,0003
LCS-1/0IBStd	0,0061	0,0039	0,0032	0,0024	0,0018	0,0006	0,0002	0,0001	0,0001
LCS-1/0UBMC	0,0042	0,0012	0,0006	0,0005	0,0003	0,0002	0,0002	0,0002	0,0002
LCS-1/0UBStd	0,0022	0,0004	0,0003	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002
LCS-1/1IBMC	0,0012	0,0008	0,0005	0,0004	0,0004	0,0004	0,0003	0,0003	0,0003
LCS-1/1IBStd	0,0057	0,0043	0,0041	0,0031	0,0025	0,0011	0,0003	0,0001	0,0000
LCS-1/1UBMC	0,0054	0,0023	0,0009	0,0007	0,0006	0,0002	0,0002	0,0002	0,0002
LCS-1/1UBStd	0,0018	0,0005	0,0003	0,0003	0,0002	0,0002	0,0002	0,0002	0,0002
LCS30/0IBMC	0,0306	0,0296	0,0290	0,0289	0,0289	0,0288	0,0288	0,0288	0,0288
LCS30/0IBStd	0,0236	0,0225	0,0223	0,0223	0,0223	0,0222	0,0222	0,0222	0,0222
LCS30/0UBMC	0,0033	0,0031	0,0030	0,0030	0,0030	0,0030	0,0030	0,0030	0,0030
LCS30/0UBStd	0,0034	0,0032	0,0032	0,0032	0,0032	0,0032	0,0032	0,0032	0,0032
LCS30/1IBMC	0,0200	0,0189	0,0186	0,0182	0,0178	0,0177	0,0177	0,0177	0,0177
LCS30/1IBStd	0,0103	0,0090	0,0089	0,0089	0,0088	0,0088	0,0088	0,0088	0,0088
LCS30/1UBMC	0,0037	0,0029	0,0027	0,0027	0,0027	0,0026	0,0026	0,0026	0,0026
LCS30/1UBStd	0,0032	0,0029	0,0029	0,0029	0,0029	0,0029	0,0029	0,0029	0,0029
PearsonNormalIBMC	0,0152	0,0147	0,0136	0,0135	0,0132	0,0128	0,0126	0,0125	0,0125
PearsonNormalIBStd	0,0036	0,0028	0,0023	0,0024	0,0022	0,0017	0,0013	0,0012	0,0011
PearsonNormalUBMC	0,0002	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001
PearsonNormalUBStd	0,0038	0,0035	0,0036	0,0038	0,0038	0,0042	0,0047	0,0043	0,0003
PearsonPosNormalIBMC	0,0151	0,0142	0,0130	0,0128	0,0125	0,0118	0,0116	0,0116	0,0116
PearsonPosNormalIBStd	0,0032	0,0025	0,0018	0,0016	0,0012	0,0007	0,0006	0,0005	0,0005
PearsonPosNormalUBMC	0,0008	0,0003	0,0003	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002
PearsonPosNormalUBStd	0,0006	0,0003	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002
CosinePosUnion3IBMC	0,0038	0,0015	0,0008	0,0006	0,0004	0,0002	0,0002	0,0002	0,0002
CosinePosUnion3IBStd	0,0071	0,0071	0,0065	0,0057	0,0052	0,0044	0,0043	0,0042	0,0041
CosinePosUnion3UBMC	0,0004	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002
CosinePosUnion3UBStd	0,0006	0,0002	0,0002	0,0002	0,0002	0,0001	0,0001	0,0001	0,0001
CosineUnion3IBMC	0,0038	0,0015	0,0008	0,0006	0,0004	0,0002	0,0002	0,0002	0,0002
CosineUnion3IBStd	0,0071	0,0071	0,0065	0,0057	0,0052	0,0044	0,0043	0,0042	0,0041
CosineUnion3UBMC	0,0004	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002
CosineUnion3UBStd	0,0006	0,0002	0,0002	0,0002	0,0002	0,0001	0,0001	0,0001	0,0001
PearsonPosUnion0IBMC	0,0022	0,0012	0,0009	0,0008	0,0008	0,0007	0,0007	0,0007	0,0007
PearsonPosUnion0IBStd	0,0036	0,0020	0,0013	0,0009	0,0008	0,0007	0,0007	0,0007	0,0007
PearsonPosUnion0UBMC	0,0093	0,0035	0,0018	0,0013	0,0009	0,0003	0,0002	0,0002	0,0002
PearsonPosUnion0UBStd	0,0010	0,0004	0,0003	0,0003	0,0003	0,0003	0,0003	0,0003	0,0003
PearsonPosUnion3IBMC	0,0024	0,0012	0,0009	0,0007	0,0007	0,0006	0,0006	0,0006	0,0006
PearsonPosUnion3IBStd	0,0065	0,0039	0,0024	0,0017	0,0014	0,0005	0,0004	0,0004	0,0004
PearsonPosUnion3UBMC	0,0026	0,0011	0,0006	0,0004	0,0003	0,0002	0,0002	0,0002	0,0002
PearsonPosUnion3UBStd	0,0009	0,0003	0,0002	0,0002	0,0002	0,0001	0,0001	0,0001	0,0001
PearsonUnion0IBMC	0,0022	0,0017	0,0014	0,0015	0,0015	0,0016	0,0016	0,0017	0,0016
PearsonUnion0IBStd	0,0033	0,0021	0,0022	0,0020	0,0022	0,0031	0,0038	0,0041	0,0043
PearsonUnion0UBMC	0,0049	0,0021	0,0010	0,0006	0,0004	0,0002	0,0001	0,0001	0,0001
PearsonUnion0UBStd	0,0058	0,0051	0,0046	0,0047	0,0045	0,0049	0,0053	0,0043	0,0005
PearsonUnion3IBMC	0,0025	0,0015	0,0016	0,0016	0,0017	0,0018	0,0019	0,0020	0,0020
PearsonUnion3IBStd	0,0065	0,0042	0,0035	0,0033	0,0036	0,0043	0,0052	0,0057	0,0059
PearsonUnion3UBMC	0,0013	0,0005	0,0002	0,0002	0,0001	0,0001	0,0001	0,0000	0,0000
PearsonUnion3UBStd	0,0049	0,0045	0,0046	0,0044	0,0045	0,0043	0,0043	0,0040	0,0005

Figura B.5: Resultados para Recall@10.

Algorithm	10 Neighbours	20 Neighbours	30 Neighbours	40 Neighbours	50 Neighbours	100 Neighbours	150 Neighbours	200 Neighbours	400 Neighbours
baselinebias	0,5471								
baselinerandom	0,3699								
CosineNormalIBMC	0,5504	0,5528	0,5540	0,5537	0,5536	0,5533	0,5532	0,5532	0,5532
CosineNormalIBStd	0,4779	0,4854	0,4864	0,4858	0,4859	0,4851	0,4848	0,4847	0,4846
CosineNormalUBMC	0,5385	0,5414	0,5433	0,5444	0,5439	0,5436	0,5430	0,5428	0,5430
CosineNormalUBStd	0,5428	0,5481	0,5501	0,5510	0,5514	0,5514	0,5511	0,5512	0,5514
CosinePosNormalIBMC	0,5504	0,5528	0,5540	0,5537	0,5536	0,5533	0,5532	0,5532	0,5532
CosinePosNormalIBStd	0,4779	0,4854	0,4864	0,4858	0,4859	0,4851	0,4848	0,4847	0,4846
CosinePosNormalUBMC	0,5385	0,5414	0,5433	0,5444	0,5439	0,5436	0,5430	0,5428	0,5430
CosinePosNormalUBStd	0,5428	0,5481	0,5501	0,5510	0,5514	0,5514	0,5511	0,5512	0,5514
LCS-1/0IBMC	0,5505	0,5537	0,5538	0,5539	0,5536	0,5535	0,5535	0,5535	0,5535
LCS-1/0IBStd	0,4942	0,5091	0,5142	0,5156	0,5162	0,5177	0,5179	0,5179	0,5179
LCS-1/0UBMC	0,5361	0,5403	0,5413	0,5415	0,5417	0,5417	0,5419	0,5422	0,5424
LCS-1/0UBStd	0,5445	0,5491	0,5503	0,5504	0,5508	0,5517	0,5520	0,5521	0,5521
LCS-1/1IBMC	0,5500	0,5529	0,5534	0,5534	0,5532	0,5530	0,5529	0,5529	0,5529
LCS-1/1IBStd	0,4907	0,5074	0,5107	0,5119	0,5123	0,5129	0,5125	0,5125	0,5124
LCS-1/1UBMC	0,5349	0,5393	0,5401	0,5405	0,5406	0,5403	0,5399	0,5398	0,5402
LCS-1/1UBStd	0,5427	0,5471	0,5482	0,5489	0,5496	0,5502	0,5505	0,5506	0,5507
LCS30/0IBMC	0,3469	0,3469	0,3469	0,3469	0,3469	0,3469	0,3469	0,3469	0,3469
LCS30/0IBStd	0,3469	0,3469	0,3469	0,3469	0,3469	0,3469	0,3469	0,3469	0,3469
LCS30/0UBMC	0,1340	0,1340	0,1340	0,1340	0,1340	0,1340	0,1340	0,1340	0,1340
LCS30/0UBStd	0,1341	0,1341	0,1341	0,1341	0,1341	0,1341	0,1341	0,1341	0,1341
LCS30/1IBMC	0,4878	0,4878	0,4878	0,4878	0,4878	0,4878	0,4878	0,4878	0,4878
LCS30/1IBStd	0,4878	0,4878	0,4878	0,4878	0,4878	0,4878	0,4878	0,4878	0,4878
LCS30/1UBMC	0,2712	0,2717	0,2718	0,2719	0,2720	0,2720	0,2720	0,2720	0,2720
LCS30/1UBStd	0,2725	0,2728	0,2730	0,2731	0,2731	0,2731	0,2731	0,2731	0,2731
PearsonNormalIBMC	0,5552	0,5584	0,5585	0,5592	0,5590	0,5588	0,5589	0,5589	0,5589
PearsonNormalIBStd	0,4429	0,4790	0,4956	0,5041	0,5108	0,5267	0,5305	0,5306	0,5296
PearsonNormalUBMC	0,5055	0,5090	0,5106	0,5108	0,5107	0,5161	0,5198	0,5230	0,5254
PearsonNormalUBStd	0,5618	0,5651	0,5610	0,5577	0,5552	0,5428	0,5289	0,5212	0,5072
PearsonPosNormalIBMC	0,5548	0,5573	0,5576	0,5581	0,5582	0,5581	0,5581	0,5581	0,5581
PearsonPosNormalIBStd	0,4371	0,4526	0,4614	0,4664	0,4693	0,4766	0,4788	0,4793	0,4793
PearsonPosNormalUBMC	0,5281	0,5383	0,5416	0,5437	0,5439	0,5449	0,5455	0,5454	0,5454
PearsonPosNormalUBStd	0,5447	0,5505	0,5519	0,5523	0,5527	0,5535	0,5536	0,5537	0,5537
CosinePosUnion3IBMC	0,5413	0,5458	0,5472	0,5468	0,5469	0,5469	0,5468	0,5468	0,5468
CosinePosUnion3IBStd	0,1567	0,1496	0,1403	0,1336	0,1275	0,1183	0,1149	0,1139	0,1126
CosinePosUnion3UBMC	0,5281	0,5360	0,5357	0,5357	0,5346	0,5347	0,5341	0,5355	0,5362
CosinePosUnion3UBStd	0,5424	0,5515	0,5512	0,5514	0,5506	0,5498	0,5490	0,5480	0,5473
CosineUnion3IBMC	0,5413	0,5458	0,5472	0,5468	0,5469	0,5469	0,5468	0,5468	0,5468
CosineUnion3IBStd	0,1567	0,1496	0,1403	0,1336	0,1275	0,1183	0,1149	0,1139	0,1126
CosineUnion3UBMC	0,5281	0,5360	0,5357	0,5357	0,5346	0,5347	0,5341	0,5355	0,5362
CosineUnion3UBStd	0,5424	0,5515	0,5512	0,5514	0,5506	0,5498	0,5490	0,5480	0,5473
PearsonPosUnion0IBMC	0,5461	0,5478	0,5481	0,5483	0,5483	0,5483	0,5483	0,5483	0,5483
PearsonPosUnion0IBStd	0,4437	0,4471	0,4488	0,4505	0,4510	0,4518	0,4520	0,4520	0,4520
PearsonPosUnion0UBMC	0,5421	0,5490	0,5501	0,5508	0,5510	0,5516	0,5516	0,5517	0,5517
PearsonPosUnion0UBStd	0,5440	0,5521	0,5543	0,5559	0,5566	0,5571	0,5572	0,5572	0,5572
PearsonPosUnion3IBMC	0,5511	0,5543	0,5550	0,5550	0,5549	0,5550	0,5550	0,5550	0,5550
PearsonPosUnion3IBStd	0,4592	0,4663	0,4688	0,4702	0,4712	0,4729	0,4733	0,4733	0,4733
PearsonPosUnion3UBMC	0,5400	0,5467	0,5489	0,5498	0,5497	0,5506	0,5507	0,5508	0,5509
PearsonPosUnion3UBStd	0,5467	0,5524	0,5538	0,5545	0,5547	0,5554	0,5556	0,5556	0,5556
PearsonUnion0IBMC	0,5459	0,5480	0,5493	0,5491	0,5493	0,5492	0,5494	0,5493	0,5492
PearsonUnion0IBStd	0,4543	0,4757	0,4865	0,4925	0,4958	0,5014	0,5055	0,5077	0,5102
PearsonUnion0UBMC	0,5304	0,5334	0,5340	0,5339	0,5330	0,5377	0,5425	0,5464	0,5484
PearsonUnion0UBStd	0,5607	0,5668	0,5652	0,5641	0,5628	0,5553	0,5491	0,5452	0,5303
PearsonUnion3IBMC	0,5511	0,5557	0,5559	0,5565	0,5565	0,5568	0,5568	0,5567	0,5567
PearsonUnion3IBStd	0,4674	0,4932	0,5060	0,5119	0,5157	0,5207	0,5203	0,5207	0,5218
PearsonUnion3UBMC	0,5272	0,5294	0,5306	0,5307	0,5305	0,5332	0,5377	0,5402	0,5424
PearsonUnion3UBStd	0,5616	0,5666	0,5646	0,5631	0,5615	0,5559	0,5487	0,5449	0,5342

Figura B.6: Resultados para Recall@1000.

Algorithm	10 Neighbours	20 Neighbours	30 Neighbours	40 Neighbours	50 Neighbours	100 Neighbours	150 Neighbours	200 Neighbours	400 Neighbours
baselinebias	0,0280								
baselinerandom	0,0420								
CosineNormalIBMC	0,0399	0,0332	0,0313	0,0302	0,0298	0,0286	0,0283	0,0282	0,0280
CosineNormalIBStd	0,1104	0,0878	0,0748	0,0642	0,0513	0,0242	0,0129	0,0090	0,0066
CosineNormalUBMC	0,0443	0,0335	0,0302	0,0283	0,0270	0,0238	0,0220	0,0208	0,0195
CosineNormalUBStd	0,0441	0,0357	0,0343	0,0337	0,0329	0,0311	0,0296	0,0286	0,0275
CosinePosNormalIBMC	0,0399	0,0332	0,0313	0,0302	0,0298	0,0286	0,0283	0,0282	0,0280
CosinePosNormalIBStd	0,1104	0,0878	0,0748	0,0642	0,0513	0,0242	0,0129	0,0090	0,0066
CosinePosNormalUBMC	0,0443	0,0335	0,0302	0,0283	0,0270	0,0238	0,0220	0,0208	0,0195
CosinePosNormalUBStd	0,0441	0,0357	0,0343	0,0337	0,0329	0,0311	0,0296	0,0286	0,0275
LCS-1/0IBMC	0,0329	0,0306	0,0295	0,0292	0,0289	0,0283	0,0282	0,0281	0,0280
LCS-1/0IBStd	0,0702	0,0452	0,0355	0,0291	0,0244	0,0142	0,0110	0,0101	0,0096
LCS-1/0UBMC	0,0413	0,0321	0,0285	0,0266	0,0252	0,0222	0,0209	0,0201	0,0194
LCS-1/0UBStd	0,0669	0,0338	0,0323	0,0315	0,0308	0,0286	0,0274	0,0267	0,0260
LCS-1/1IBMC	0,0325	0,0305	0,0298	0,0295	0,0289	0,0286	0,0285	0,0284	0,0284
LCS-1/1IBStd	0,0652	0,0529	0,0426	0,0364	0,0316	0,0176	0,0125	0,0103	0,0094
LCS-1/1UBMC	0,0453	0,0347	0,0308	0,0286	0,0272	0,0234	0,0218	0,0208	0,0198
LCS-1/1UBStd	0,0496	0,0337	0,0326	0,0319	0,0313	0,0292	0,0281	0,0274	0,0267
LCS30/0IBMC	0,1708	0,1665	0,1650	0,1643	0,1640	0,1640	0,1640	0,1640	0,1640
LCS30/0IBStd	0,1539	0,1476	0,1465	0,1464	0,1463	0,1463	0,1463	0,1463	0,1463
LCS30/0UBMC	0,0378	0,0363	0,0358	0,0357	0,0356	0,0355	0,0355	0,0355	0,0355
LCS30/0UBStd	0,0451	0,0409	0,0406	0,0405	0,0405	0,0405	0,0405	0,0405	0,0405
LCS30/1IBMC	0,1238	0,1137	0,1102	0,1084	0,1081	0,1070	0,1068	0,1068	0,1068
LCS30/1IBStd	0,0845	0,0745	0,0735	0,0730	0,0727	0,0725	0,0725	0,0725	0,0725
LCS30/1UBMC	0,0345	0,0299	0,0280	0,0270	0,0264	0,0251	0,0249	0,0249	0,0249
LCS30/1UBStd	0,0382	0,0323	0,0318	0,0315	0,0312	0,0306	0,0305	0,0304	0,0304
PearsonNormalIBMC	0,1002	0,0930	0,0869	0,0825	0,0801	0,0754	0,0737	0,0735	0,0736
PearsonNormalIBStd	0,0412	0,0339	0,0307	0,0290	0,0268	0,0230	0,0208	0,0199	0,0182
PearsonNormalUBMC	0,0181	0,0159	0,0155	0,0152	0,0151	0,0147	0,0142	0,0134	0,0109
PearsonNormalUBStd	0,0477	0,0424	0,0431	0,0434	0,0444	0,0469	0,0474	0,0437	0,0147
PearsonPosNormalIBMC	0,0999	0,0918	0,0849	0,0805	0,0781	0,0720	0,0705	0,0701	0,0700
PearsonPosNormalIBStd	0,0406	0,0309	0,0261	0,0232	0,0205	0,0157	0,0147	0,0145	0,0144
PearsonPosNormalUBMC	0,0227	0,0194	0,0187	0,0183	0,0180	0,0178	0,0176	0,0175	0,0174
PearsonPosNormalUBStd	0,0336	0,0260	0,0255	0,0250	0,0249	0,0240	0,0236	0,0234	0,0233
CosinePosUnion3IBMC	0,0435	0,0366	0,0344	0,0325	0,0311	0,0291	0,0297	0,0285	0,0286
CosinePosUnion3IBStd	0,0503	0,0430	0,0424	0,0361	0,0311	0,0221	0,0205	0,0175	0,0160
CosinePosUnion3UBMC	0,0237	0,0208	0,0201	0,0199	0,0198	0,0200	0,0200	0,0199	0,0194
CosinePosUnion3UBStd	0,0300	0,0290	0,0291	0,0289	0,0288	0,0289	0,0293	0,0294	0,0278
CosineUnion3IBMC	0,0435	0,0366	0,0344	0,0325	0,0311	0,0291	0,0297	0,0285	0,0286
CosineUnion3IBStd	0,0503	0,0430	0,0424	0,0361	0,0311	0,0221	0,0205	0,0175	0,0160
CosineUnion3UBMC	0,0237	0,0208	0,0201	0,0199	0,0198	0,0200	0,0200	0,0199	0,0194
CosineUnion3UBStd	0,0300	0,0290	0,0291	0,0289	0,0288	0,0289	0,0293	0,0294	0,0278
PearsonPosUnion0IBMC	0,0367	0,0307	0,0291	0,0284	0,0281	0,0275	0,0274	0,0274	0,0274
PearsonPosUnion0IBStd	0,0450	0,0252	0,0197	0,0169	0,0154	0,0135	0,0132	0,0132	0,0132
PearsonPosUnion0UBMC	0,0632	0,0402	0,0339	0,0305	0,0284	0,0240	0,0225	0,0218	0,0212
PearsonPosUnion0UBStd	0,0389	0,0316	0,0308	0,0302	0,0297	0,0286	0,0279	0,0275	0,0271
PearsonPosUnion3IBMC	0,0371	0,0312	0,0294	0,0285	0,0281	0,0272	0,0270	0,0270	0,0270
PearsonPosUnion3IBStd	0,0660	0,0414	0,0310	0,0258	0,0221	0,0157	0,0146	0,0143	0,0142
PearsonPosUnion3UBMC	0,0340	0,0282	0,0261	0,0248	0,0240	0,0221	0,0214	0,0209	0,0205
PearsonPosUnion3UBStd	0,0424	0,0318	0,0305	0,0299	0,0296	0,0279	0,0270	0,0265	0,0261
PearsonUnion0IBMC	0,0365	0,0321	0,0310	0,0312	0,0313	0,0319	0,0325	0,0328	0,0334
PearsonUnion0IBStd	0,0445	0,0260	0,0215	0,0204	0,0203	0,0237	0,0270	0,0297	0,0332
PearsonUnion0UBMC	0,0467	0,0342	0,0297	0,0271	0,0254	0,0217	0,0202	0,0191	0,0166
PearsonUnion0UBStd	0,0516	0,0457	0,0455	0,0455	0,0455	0,0469	0,0474	0,0446	0,0250
PearsonUnion3IBMC	0,0373	0,0327	0,0318	0,0317	0,0318	0,0323	0,0330	0,0332	0,0337
PearsonUnion3IBStd	0,0663	0,0434	0,0349	0,0320	0,0304	0,0307	0,0344	0,0366	0,0385
PearsonUnion3UBMC	0,0282	0,0239	0,0223	0,0213	0,0207	0,0191	0,0183	0,0175	0,0153
PearsonUnion3UBStd	0,0552	0,0460	0,0455	0,0454	0,0456	0,0462	0,0460	0,0438	0,0239

Figura B.7: Resultados para MRR.

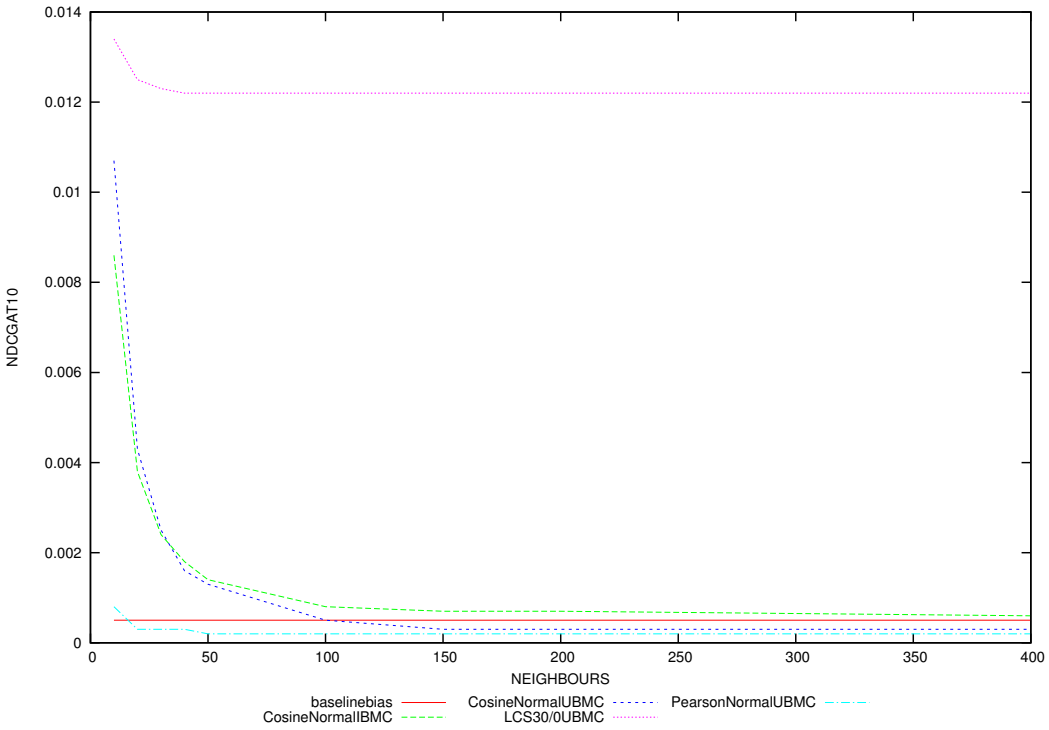

Figura B.8: NDCG@10 para algunos algoritmos.

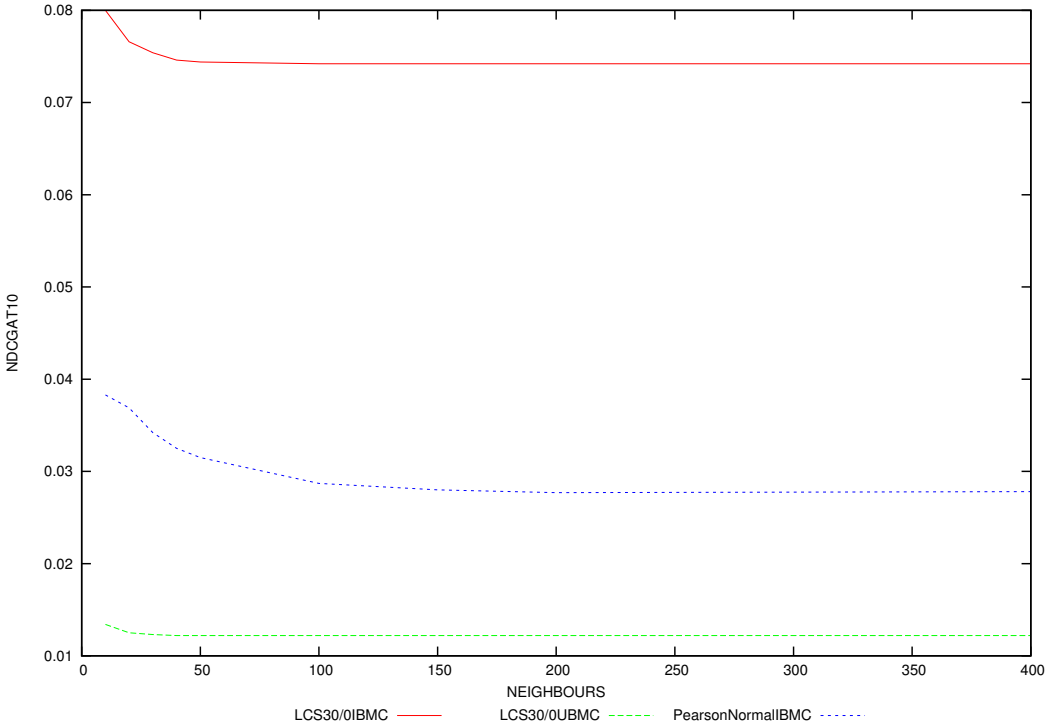

Figura B.9: NDCG@10 para los mejores algoritmos.

C

Comparación entre distintos parámetros en estructuras

Algunas estructuras implementadas en la aplicación pueden ser configuradas mediante distintos parámetros. La primera es la tabla hash, que puede resolver las colisiones de muchas maneras. Una de ellas es empleando listas, de forma que cada celda de la tabla hash disponga de una lista enlazada con los elementos que tengan el mismo valor hash. En las tablas hash desarrolladas, admitimos el empleo de cualquier tipo de estructura que implemente la interfaz de GeneralStructure, aunque sólo se ha habilitado el uso de listas o de árboles rojo-negros. No obstante, al considerarse las tablas hash como un array de listas o de árboles, el tamaño de dicho array afecta al rendimiento de la estructura. Por otro lado, está el árbol B. Cuantas más claves sea capaz de almacenar cada nodo, menor será la altura del árbol. No obstante, cuantas más claves sea capaz de almacenar, más tiempo se empleará buscando secuencialmente en el nodo.

En este anexo se muestran los resultados obtenidos al efectuar distintas ejecuciones tanto de árboles B como de tablas Hash. Para estas últimas, mostramos seis configuraciones diferentes empleando tres primos de distinto tamaño tanto con listas como con árboles rojo-negros. Los resultados se han obtenido efectuando listas de recomendación (evaluación de ranking). Así pues, primeramente se ha leído el fichero de train (load training memory y load training time), después se ha efectuado el entrenamiento (training memory y training time), posteriormente se ha leído el fichero de test (load test memory y load test time) y por último se han obtenido las listas de recomendación por cada usuario (test memory y test time). Es importante señalar que los resultados son orientativos, ya que la máquina virtual de Java (JVM) puede llamar al recolector de basura en cualquier momento a lo largo de la ejecución. Todos los datos han sido obtenidos a partir del conjunto de datos de Movielens.

En la imagen C.1 pueden observarse algunos resultados interesantes. Primeramente, vemos cómo en general para primos pequeños se obtienen mejores resultados que para primos grandes. Esto es extraño, ya que teóricamente, si hay un primo mayor, habrá menos posibilidades de obtener colisiones y por lo tanto las búsquedas serán más rápidas. No obstante, hay que tener en cuenta que no sólo se crean tablas hash para los usuarios/artículos globales, sino que cada usuario tiene una tabla hash con el conjunto de artículos que ha puntuado. Al crear tantas tablas tan grandes, se pierde más tiempo en la creación de dichas tablas que en las búsquedas en sí. Se puede ver cómo aumentando el tamaño del primo, se incrementa el tiempo en todas las operaciones. Por otro lado, en la sección de test para primos pequeños, cabe destacar que las

configuraciones que emplean árboles rojo-negros en lugar de listas, obtienen mejores resultados que aquellas que emplean listas.

Algorithm	Structure	Load training memory	Load training time	Training memory	Training time	Load test memory	Load test time	Test memory	Test time
Pearson Correlation	Hash table (list) small	63,76	0,41	119,71	68,7	18,09	0,1	141,52	799,87
	Hash table (list) medium	62,22	0,74	142,89	192,1	18,25	0,05	257,72	856,79
	Hash table (list) big prime	91,08	1,16	183,09	365,62	18,97	0,05	321,85	868,22
	Hash table (red-black tree) small	65,69	0,7	58,51	69,42	18,5	0,05	184,31	607,88
	Hash table (red-black tree) medium	76,15	1,4	243,96	200,95	19,15	0,08	278,95	615,65
	Hash table (red-black tree) big	105,72	2,48	115,56	387,92	20,15	0,05	297,58	568,58
Cosine Normal	Hash table (list) small	63,1	0,65	167,98	68,66	18,06	0,05	187,83	809,82
	Hash table (list) medium	62,53	0,74	198,98	146,81	18,38	0,09	374,48	804,35
	Hash table (list) big	90,85	1,27	281,3	270,06	18,93	0,09	275,91	809,46
	Hash table (red-black tree) small	65,75	0,72	152,33	77,14	18,34	0,08	227,72	544,55
	Hash table (red-black tree) medium	76,47	1,44	227,37	155,67	18,51	0,07	267,04	549,45
	Hash table (red-black tree) big	105,63	2,34	307,59	294,06	19,57	0,08	307,4	582,86

Figura C.1: Comparativa entre la memoria y el tiempo para las tablas hash empleando la correlación de Pearson y la similitud coseno. Los primos han sido 139 (small), 827 (medium) y 2143 (big). Memoria medida en megabytes y tiempo en segundos. Los resultados son una media entre tres ficheros de train y tres ficheros de test (3 folds).

En vista de los resultados mostrados, puede concluirse que aunque las tablas hash sean unas estructuras potentes, puede resultar contraproducente su empleo en todos los ámbitos de la aplicación. Tener una tabla hash de un tamaño muy grande para cada usuario resulta desfavorable, ya que la mayoría de usuarios ha votado muy pocos artículos. En estos casos, con alguna estructura más liviana como un árbol autobalanceado, sería suficiente.

Structure	Load training memory	Load training time	Training memory	Training time	Load test memory	Load test time	Test memory	Test time
Btree (3)	36,28	1,66	64,86	51,6	13,37	0,06	164,53	735
Btree (7)	31,62	2,07	77,57	45,6	14,46	0,18	150,37	702,62
Btree (11)	17,08	1,78	190,81	44,13	12,38	0,17	98,08	852,07
Btree (20)	17,83	3,41	47,81	19,99	13,14	0,07	90,28	964,37
Btree (36)	30,98	2,27	115,89	33,93	14,13	0,07	155,5	1208,62
Btree (60)	33,14	2,03	116,1	56,55	14,13	0,17	155,19	1488,93
Btree (70)	32,42	2,04	233,94	56,75	15,4	0,19	140,23	1559,59

Figura C.2: Comparativa entre la memoria y el tiempo para árboles B empleando la similitud coseno con distintos valores del parámetro t. El número de claves soportadas por nodo sería 2t-1. Memoria medida en megabytes y tiempo en segundos.

En esta imagen se analiza el rendimiento (tanto en memoria como en tiempo) del árbol B en función del tamaño del nodo. En principio, cuanto mayor sea el número de claves a guardar en el árbol, menor profundidad tendrá el árbol, pero las búsquedas dentro del nodo tardarán más tiempo (serían lineales). Se puede ver que para tamaños de nodo grandes, el tiempo acaba siendo mayor que para claves más pequeñas (acaban consiguiendo el mejor rendimiento). No obstante, el parámetro de T, al igual que el primo en las hash, dependerá del conjunto de datos que se esté analizando.