View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Academica-e

E.T.S. de Ingeniería Industrial,

Informática y de Telecomunicación

Sistemas de Recomendación en Apache Spark

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autora: Elena Izaguirre Equiza

Director: José Ramón González de Mendívil Moreno

Pamplona, junio de 2015

upna Universidad Pública de Navarra Nafarroako Unibertsitate Publikoa

Índice

Resumen....4

Palabras clave....5

Justificación y objetivos....6

1. Sistemas de Recomendación....8

1.1. ¿Qué es un Sistema de Recomendación? ¿Qué retos tiene?....8

1.2. ¿Qué información utiliza?....9

1.3. Estructura....9

1.4. Métricas de evaluación de las recomendaciones....10

1.4.1. Métricas de precisión en la calificación....10

1.4.2. Correlaciones de las calificaciones y clasificaciones....10

1.4.3. Métricas de precisión en la clasificación....11

1.4.4. Índices de ponderación en la clasificación....11

1.4.5. Diversidad....12

1.4.6. Innovación....13

1.4.7. Cobertura....13

1.4.8. Resumen....14

1.5. Tipos de Sistemas de Recomendación....14

1.6. Amazon: un ejemplo real de un sistema de recomendación....15

2. Algoritmos de Recomendación Colaborativos....16

2.1. El algoritmo de los K vecinos....16

2.1.1. Algoritmo de los K vecinos “usuario a usuario”....16

2.1.2. Algoritmo de los K vecinos “ítem a ítem”....16

2.1.3. Medidas de similitud....17

2.1.4. Funciones de agregación....18

2.1.5. Ejemplo....18

2.2. Algoritmo de filtrado colaborativo basado en tendencias....19

2.2.1. Cálculo de tendencias....19

2.2.2. Análisis de los casos....19

2.2.3. Ejemplo....20

3. Sistemas de Recomendación para aplicaciones de juegos 22
3.1. MARS 22
3.1.1. Estimación del éxito y progreso 22
3.1.2. Estimación de la motivación 22
3.1.3. Algoritmos de recomendación 23
3.2. Juegos adaptativos basados en recomendaciones 24
3.2.1. Ámbito externo 24
3.2.2. Ámbito interno 24
4. Tecnologías empleadas 26
4.1. MapReduce 26
4.1.1. Arquitectura de MapReduce 26
4.1.2. Ejemplo de MapReduce 27
4.2. Apache Hadoop 28
4.2.1. HDFS: Hadoop Distributed File System 28
4.3. Apache Spark 29
4.3.1. Principales módulos de Spark 29
4.3.2. RDD: Resilient Distributed Dataset 30
4.4. Apache Hadoop NextGen MapReduce (YARN) 30
5. Implementación de los algoritmos de recomendación en Spark 32
5.1. Algoritmo basado en similitudes (vecinos) 32
5.1.1. Clase Neighbours 32
5.1.2. Clase CosineSimilarity 33
5.1.3. Clase PearsonSimilarity 33
5.1.4. Clase Metrics 33
5.2. Algoritmo basado en tendencias 33
5.2.1. Clase Trends 33
5.2.2. Clase RatingsPredictor 34
5.2.3. Clase Metrics 34
5.3. Resultados obtenidos 34
5.3.1. Algoritmo basado en similitudes 35

5.3.2. Algoritmo basado en tendencias 37
Conclusiones y trabajos futuros 39
5.4. Los algoritmos 39
5.5. ZombieLib 39
Bibliografía 41
Anexo 1: código generado para el algoritmo basado en similitudes 43
Clase Neighbours 43
Clase CosineSimilarity 46
Clase PearsonSimilarity 47
Clase PearsonAccumulator 48
Clase Average 50
Clase AverageMapper 51
Clase JoinedMapper 51
Clase AccumulatorMapper 51
Clase Metrics 52
Anexo 2: código generado para el algoritmo basado en tendencias 54
Clase Trends 54
Clase RatingsPredictor 56
Clase Average 58
Clase AverageMapper 59
Clase ReorganizerMapper 60
Clase Metrics 60

Resumen

Actualmente, los sistemas de recomendación juegan un papel fundamental en el comercio electrónico. Estos sistemas proponen sugerencias de productos a los usuarios, basadas en diferentes fuentes de información, con el objetivo de que los usuarios continúen visitando el sitio y aumente la probabilidad de que efectúen compras.

Para conseguir estas recomendaciones se utilizan algoritmos para el cálculo de predicciones. Estas predicciones representan la posibilidad de que un cierto usuario vaya a acceder o valorar un determinado ítem. Dado el gran volumen de usuarios e ítems en un sistema, hacen falta herramientas adecuadas que permitan obtener las recomendaciones en un tiempo razonable, además de garantizar que el sistema de recomendación esté siempre disponible.

En este trabajo se estudian las distintas técnicas para proporcionar recomendaciones y se programan dos algoritmos sobre el sistema Hadoop, empleando el mecanismo de MapReduce que proporciona el subsistema Spark. Por una parte, la técnica de MapReduce permite ofrecer, con ciertas limitaciones, una forma de ejecutar en paralelo ciertas tareas sobre el gran volumen de datos a procesar. Por otro lado, al ejecutar las tareas de MapReduce sobre el sistema ofrecido por Hadoop se consigue el requisito de tolerancia a fallos que es necesario en un sistema que debe mantenerse disponible en todo momento.

Palabras clave

Inicio en frío / cold start: situación existente en los sistemas de recomendación con escasez de datos, en los que existe la posibilidad de no poder ofrecer una recomendación precisa al usuario por no tener suficientes datos en el sistema con los que poder calcular las predicciones.

Información implícita: información recopilada por el sistema a partir del comportamiento de los usuarios, sin que éstos sean conscientes o tengan la intención de aportar dicha información. Por ejemplo, un caso de información implícita sería que el sistema contemplase los clicks de ratón que ejerce el usuario sobre la representación de los ítems.

Información explícita: información aportada por el usuario al sistema de forma voluntaria y consciente, como las calificaciones sobre los ítems.

Predicción: calificación que el sistema estima que realizará un usuario sobre un ítem, calculada a partir de la aplicación de algoritmos sobre los datos almacenados.

Calificación / valoración / rating: opinión que un usuario expresa sobre un ítem, encerrada en un formato y dominio establecidos por el sistema.

Clasificación / ranking: lista ordenada de ítems recomendados a un usuario concreto por el sistema.

Escasez de datos: situación dada en un sistema de recomendación relativamente nuevo, en el que hay pocos datos almacenados acerca de usuarios, ítems o calificaciones, lo que puede derivar en recomendaciones imprecisas.

Escalabilidad: propiedad deseada en un sistema de recomendación, referente a la capacidad de éste para realizar su función a medida que aumentan los datos a procesar, sin ralentizar en exceso su ejecución.

Diversidad de datos: situación dada en un sistema de recomendación en la que las calificaciones de usuarios a ítems están distribuidas de forma desigual, es decir, unos usuarios realizan más calificaciones que otros o unos ítems se califican más que otros.

Usuario: persona, generalmente física, que utiliza un sistema de recomendación, aportando algunos datos personales (ocasionalmente) y posteriormente realizando calificaciones sobre ciertos ítems, recibiendo a cambio recomendaciones sobre ítems que podrían interesarle.

Ítem: objeto que recibe las calificaciones de los usuarios y es recomendado a los mismos.

RDD: acrónimo de Resilient Distributed Dataset, abstracción de datos de Spark que se asemeja a una tabla con varias tuplas.

HDFS: acrónimo de Hadoop Distributed File System, el sistema de ficheros distribuido de Hadoop, que almacena cada archivo partido en pedazos que distribuye por las diferentes máquinas del clúster.

Justificación y objetivos

A lo largo de este último semestre, el Grupo de Sistemas Distribuidos de la Universidad Pública de Navarra ha venido colaborando con la empresa Biko2 en el desarrollo de un sistema de información para los desarrolladores de videojuegos. La plataforma que se está desarrollando, denominada ZombiApp (http://www.biko2.com/zombiapps/) trata de dar información útil a los desarrolladores sobre el estado de uso de su app, además de ofrecer a los jugadores recomendaciones sobre nuevas apps que pueden ser de su interés. Para ofrecer estas recomendaciones se debe estudiar tanto la información que se va a disponer de usuarios y juegos, como las técnicas más adecuadas para construir una plataforma de recomendación. Los requisitos de la plataforma son bastante exigentes porque, para que pueda resultar en un futuro rentable, debe ofrecer una tasa de recomendaciones alta. Se estima que se deben realizar sobre unas 150 recomendaciones por segundo para un volumen acumulado de aproximadamente 6 millones de usuarios.

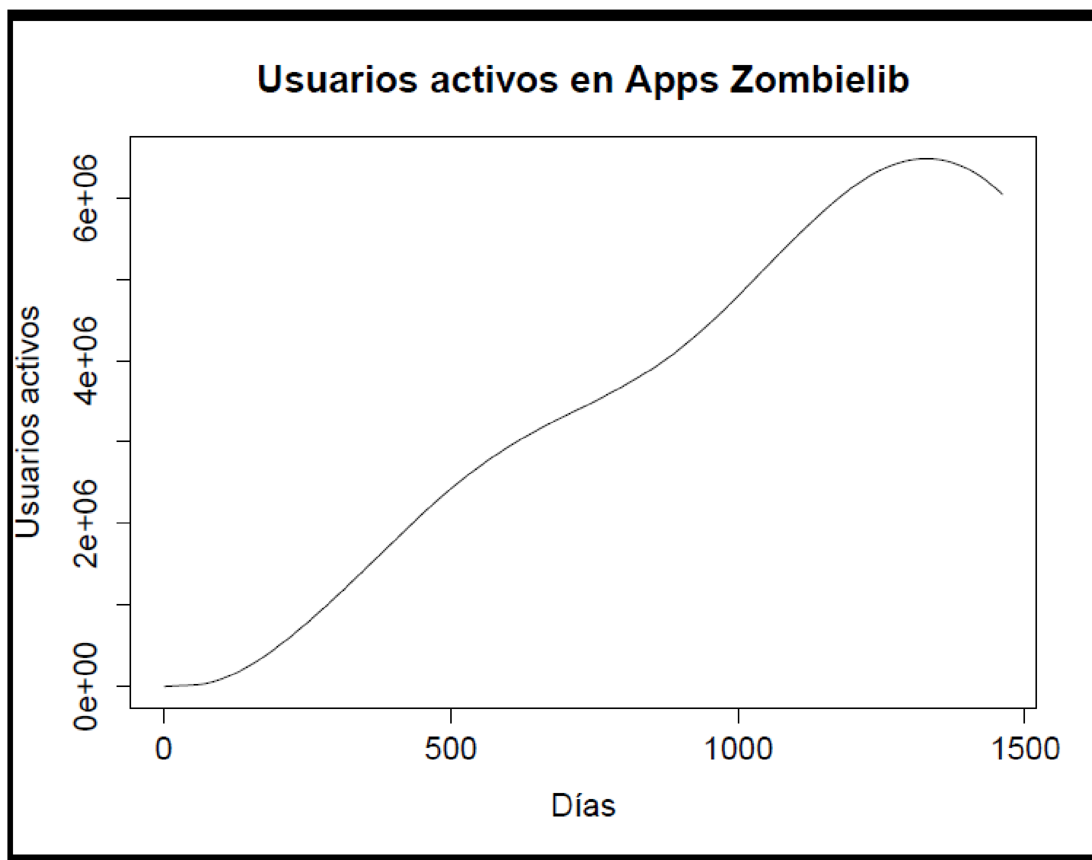

Ilustración 1: expectativa de usuarios activos en la plataforma Zombielib.

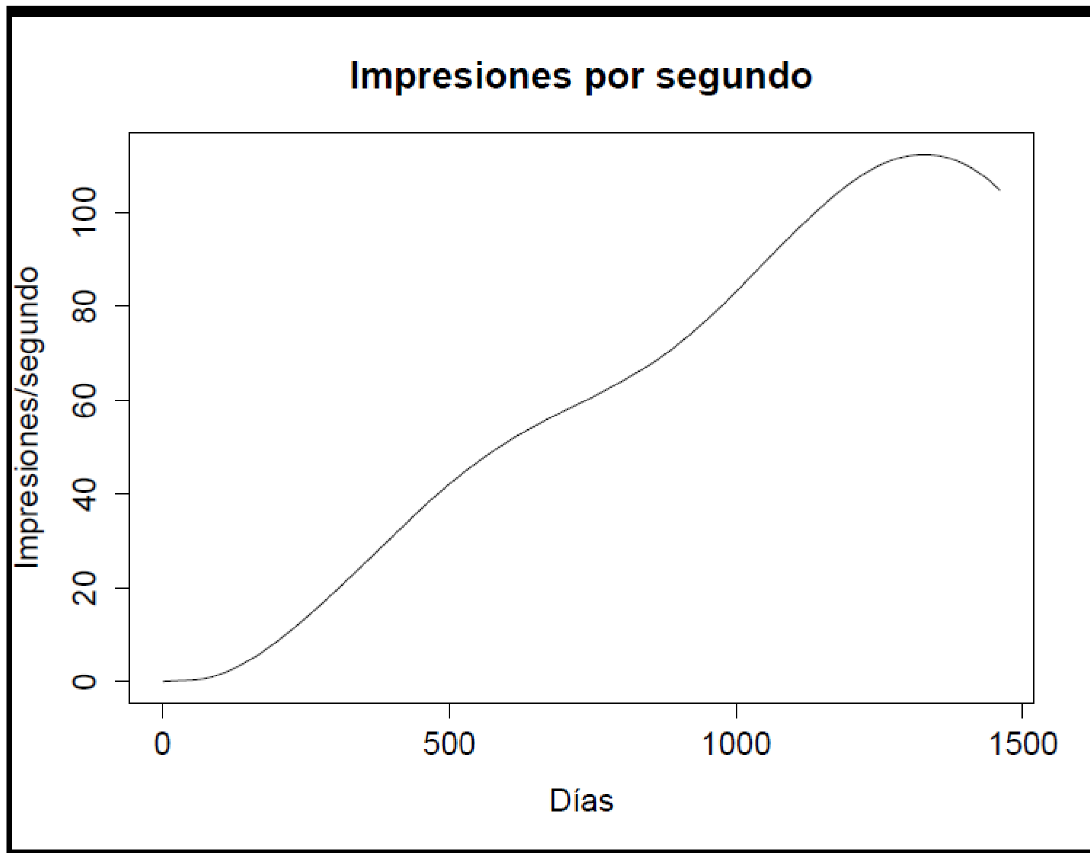

Ilustración 2: expectativa del número de recomendaciones ofrecidas por la plataforma Zombielib.

En la actualidad la manera de construir un sistema escalable, capaz de tratar con el volumen de datos considerado, y desarrollado con software de libre distribución, es basar la implementación en el ecosistema Hadoop. Por este motivo, los objetivos que se plantean en este proyecto son los siguientes:

1. Estudiar diferentes trabajos resumen publicados sobre sistemas de recomendación.
2. Analizar los sistemas de recomendación colaborativos y elegir algunos que proporcionen buen rendimiento y fiabilidad.
3. Estudiar algún sistema de recomendación concreto para recomendaciones sobre juegos.
4. Implementar algoritmos de recomendación sobre Hadoop utilizando la librería ofrecida por Spark y medir su rendimiento.

La memoria se ha organizado siguiendo los objetivos anteriores: En el capítulo uno se presenta un resumen de los objetivos de un sistema de recomendación y las métricas que se emplean a la hora de evaluar dichos sistemas. En el capítulo 2 se estudian dos algoritmos de recomendación del tipo colaborativo basados en puntuaciones: el algoritmo de los K vecinos, y el algoritmo de las tendencias. En el capítulo 3 se presentan las ideas principales del algoritmo MARS, un algoritmo para realizar recomendaciones sobre juegos. En el capítulo 4 se presentan las tecnologías empleadas para la implementación. En el capítulo 5 se presenta la implementación de los algoritmos en Spark y los resultados obtenidos. Finalmente, la memoria termina con una sección de conclusiones y trabajos futuros. En los apéndices se encuentran los códigos de los algoritmos que se han implementado.

1. Sistemas de Recomendación

1.1. ¿Qué es un Sistema de Recomendación? ¿Qué retos tiene?

[1] [2] Un sistema de recomendación es un sistema inteligente que proporciona a los usuarios una serie de sugerencias personalizadas sobre un determinado tipo de elementos. La tarea de un sistema de recomendación es transformar los datos y preferencias de los usuarios en predicciones de posibles gustos e intereses futuros de los usuarios. Las recomendaciones personalizadas deben ayudan a decidir el contenido correcto para la persona adecuada.

A continuación se explican brevemente los principales retos de un sistema de recomendación.

Diversidad de datos. Cuando el número medio de evaluaciones por usuario o ítem es alto, éstas pueden estar distribuidas desigualmente y por lo tanto existir usuarios o ítems con muchas valoraciones y otros con pocas.

Escalabilidad. A medida que crece el número de usuarios e ítems, es esencial considerar los problemas de coste computacional y buscar algoritmos de recomendación poco exigentes o fáciles de paralelizar (o ambas). Otra posible solución es utilizar versiones incrementales de los algoritmos, donde, a medida que la información crece, las recomendaciones no se recalculan globalmente, pero sí incrementalmente.

Inicio en frío. Cuando un usuario nuevo entra en el sistema, no hay suficiente información para producir una recomendación para él. Un buen sistema de recomendación debe poder hacer frente a esta situación.

Diversidad versus precisión. Cuando la tarea se centra en recomendar ítems susceptibles de ser apreciados por un usuario particular, suele ser más efectivo recomendar ítems con valoraciones altas y populares. Una buena lista de ítems recomendados debe contener también ítems menos obvios a los que es más difícil que el usuario llegue por sí mismo.

Vulnerabilidad a ataques. Dada su importancia en aplicaciones de comercio electrónico, los sistemas de recomendación son objetivo de ataques para promover o reprimir ciertos ítems (bastaría con crear perfiles de usuario falsos con los que realizar calificaciones positivas para aquellos ítems que quisieran promover y negativas para los que quisieran reprimir). Existe un amplio surtido de herramientas para evitar estos ataques.

El valor del tiempo. Hay ítems que únicamente interesan durante un período de tiempo determinado. Interesa saber cuándo se deben declinar las opiniones antiguas y cuáles son los patrones de tiempo en las evaluaciones de los usuarios y la relevancia de los ítems.

Evaluación de las recomendaciones. Tenemos a nuestra disposición distintas métricas, pero elegir la mejor para la situación dada es todavía una cuestión abierta. La comparación entre diferentes algoritmos de recomendación es problemática porque resuelven diferentes tareas.

Interfaz de usuario. Las recomendaciones deben ser transparentes para que el usuario sepa por qué se le han recomendado esos ítems. Debe mostrarse al usuario una larga lista pero presentada de forma simple y que sea fácil de navegar.

1.2. ¿Qué información utiliza?

Los sistemas de recomendación tratan dos tipos de informaciones: calificaciones explícitas e implícitas.

Las *calificaciones explícitas* son aquellas que los usuarios han expresado consciente y voluntariamente acerca de los ítems (como la conocida clasificación por estrellas).

Las *calificaciones implícitas* se obtienen a partir de la observación del comportamiento del usuario; que un usuario haga click sobre un ítem refleja cierto interés en él. El número de horas que está utilizando una aplicación pone de manifiesto si ésta es de su agrado…

1.3. Estructura

La estructura de un sistema de recomendación tiene forma de grafo bipartido, en el que en uno de los subgrafos se representan los usuarios y en el otro los ítems. De esta forma, los arcos del grafo que unen los usuarios con los ítems representan las calificaciones entre ambos.

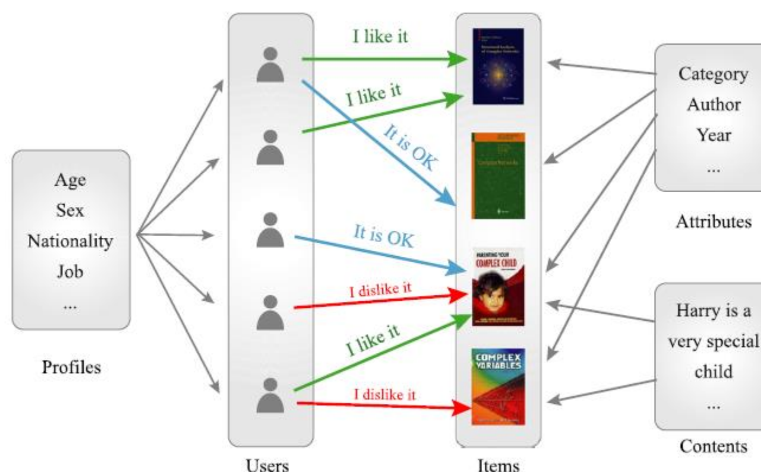

Ilustración 3: Representación de la información de un sistema de recomendación mediante un grafo bipartito. Fuente: [1].

Adicionalmente, se pueden añadir al grafo datos demográficos para realizar comparaciones entre los usuarios y características de los ítems para realizar recomendaciones más específicas.

1.4. Métricas de evaluación de las recomendaciones

[1] Normalmente se divide el conjunto de calificaciones de las que se dispone en dos subconjuntos: E^T es el subconjunto de las calificaciones que se utilizarán como conjunto de entrenamiento, es decir, para realizar las predicciones; por otro lado, E^P será el subconjunto que se comparará con las correspondientes predicciones para evaluar la eficacia de las recomendaciones.

1.4.1. Métricas de precisión en la calificación

Se utilizan dos métricas, *Mean Absolute Error (MAE)* y *Root Mean Squared Error (RMSE)*, para medir la proximidad de las predicciones a las calificaciones reales. Si $r_{i\alpha}$ es el valor de la calificación real del usuario i sobre el ítem α y $\hat{r}_{i\alpha}$ es el valor predicho por el sistema para esa calificación, MAE y RMSE se definen como:

$$MAE = \frac{1}{|E^P|} \sum_{(i,\alpha)\in E^P} |r_{i\alpha} - \hat{r}_{i\alpha}|$$

$$RMSE = \sqrt{\frac{1}{|E^P|} \sum_{(i,\alpha)\in E^P} (r_{i\alpha} - \hat{r}_{i\alpha})^2}$$

Cuanto menores sean estos valores, mayor será la precisión de las predicciones.

1.4.2. Correlaciones de las calificaciones y clasificaciones

Otra forma de evaluar las predicciones es calcular la correlación entre éstas y las calificaciones reales.

La *correlación de Pearson (PCC)* mide el alcance con el que una relación lineal se presenta entre dos conjuntos de valoraciones. Se define como:

$$PCC = \frac{\sum_\alpha(\hat{r}_\alpha - \bar{\hat{r}})(r_\alpha - \bar{r})}{\sqrt{\sum_\alpha(\hat{r}_\alpha - \bar{\hat{r}})^2}\ \sqrt{\sum_\alpha(r_\alpha - \bar{r})^2}}$$

donde r_α y $\hat{r}_\alpha$ son, respectivamente, los valores de las calificaciones real y predicha para el ítem α mientras que $\bar{\hat{r}}$ es la media de todas las calificaciones predichas y $\bar{r}$ es la media de todas las calificaciones reales.

El *coeficiente de correlación de Spearman* ρ se define de la misma manera que la correlación de Pearson, exceptuando que r_α y $\hat{r}_\alpha$ se reemplazan por la posición en el ranking de los ítems correspondientes.

De forma similar a la correlación de Spearman, la *Tau de Kendall* también mide el alcance con el cual las dos clasificaciones coinciden con los valores exactos de las calificaciones. Se define como:

$$\tau = \frac{C - D}{C + D}$$

donde C es el número de parejas de ítems que el sistema ha predicho en el orden correcto de clasificación y D es el número de parejas de ítems que el sistema ha predicho en un orden erróneo. $\tau = 1$ cuando las clasificaciones real y predicha son idénticas y $\tau = -1$ cuando son exactamente opuestas.

Para el caso de objetos con las mismas calificaciones real y predicha, se propone una *variación de la Tau de Kendall*:

$$\tau \approx \frac{C - D}{\sqrt{(C + D + S_T)(C + D + S_P)}}$$

donde S_T es el número de parejas de ítems para las cuales las calificaciones reales son iguales y S_P es el número de parejas de ítems para las cuales las calificaciones predichas coinciden.

1.4.3. Métricas de precisión en la clasificación

Como a los usuarios reales sólo les interesa el principio de la lista de recomendación, una aproximación práctica es considerar el número de objetos relevantes (apreciados por el usuario) en los primeros L puestos. *Precision (P)* y *Recall (R)* son las métricas más populares basadas en la idea anterior. Para el usuario actual i, se definen como:

$$P_i(L) = \frac{d_i(L)}{L}$$

$$R_i(L) = \frac{d_i(L)}{D_i}$$

donde $d_i(L)$ indica el número de objetos relevantes entre los L primeros de la lista de recomendación y D_i es el número total de objetos relevantes para i.

Promediando los valores de Precision y Recall para todos los usuarios obtenemos los valores medios $P(L)$ y $R(L)$. Se pueden combinar ambos en una métrica menos dependiente de L, F_1*-score*:

$$F_1(L) = \frac{2PR}{P + R}$$

1.4.4. Índices de ponderación en la clasificación

Half-life Utility. La métrica Half-life Utility tiene por objeto evaluar la utilidad de una lista de recomendación para un usuario. Se basa en la suposición de que la probabilidad de que un usuario examine un ítem recomendado decrece exponencialmente con la clasificación de los ítems. La utilidad esperada para las recomendaciones dadas a un usuario i se convierte en:

$$HL_i = \sum_{\alpha=1}^{N} \frac{\max(r_{i\alpha} - d, 0)}{2^{(o_{i\alpha}-1)/(h-1)}}$$

donde los objetos se ordenan por su predicción $\hat{r}_{i\alpha}$ en orden descendente, $o_{i\alpha}$ representa la posición en la clasificación predicha para el objeto α en la lista de recomendación del usuario i, d es la calificación por defecto (por ejemplo se puede situar en la calificación media de entre las posibles) y la "half-life" h es la posición del objeto de la lista con un 50% de probabilidades de que el usuario finalmente lo examine. Cuando HL_i se promedia para todos los usuarios, obtenemos una utilidad para todo el sistema.

Discounted Cumulative Gain. Para una lista de recomendación de tamaño L, DCG se define como:

$$DCG(b) = \sum_{n=1}^{b} r_n + \sum_{n=b+1}^{L} \frac{r_n}{\log_b n}$$

donde r_n indica la relevancia del objeto de la n posición de la clasificación ($r_n = 1$ para un objeto relevante y $r_n = 0$ en otro caso) y b es un parámetro que normalmente vale 2, por lo que simplificando, DCG se calcularía como:

$$DCG = r_1 + r_2 + \sum_{n=3}^{L} \frac{r_n}{\log_2 n}$$

La intención de DCG es que los objetos del principio de la clasificación den más satisfacción y utilidad que los del final.

Rank-biased Precision. Esta métrica se basa en la presunción de que los usuarios siempre miran el primer objeto de la lista y progresan de un objeto al siguiente con cierta probabilidad p (con la probabilidad complementaria $1-p$ terminan de mirar la lista de recomendación). Para una lista de longitud L, esta métrica se define como:

$$RBP = (1-p) \sum_{n=1}^{L} r_n p^{n-1}$$

donde r_n se define de la misma forma que en el DCG. La diferencia entre esta métrica y DCG es que ésta descuenta la relevancia mediante una secuencia geométrica, mientras que DCG utiliza una forma logarítmica.

1.4.5. Diversidad

La diversidad en sistemas de recomendación se refiere a cómo de diferentes son los objetos recomendados unos de otros. Existen dos niveles para interpretar la diversidad: la diversidad inter-usuario (se refiere a la habilidad de un algoritmo para devolver diferentes resultados a diferentes usuarios) y la diversidad intra-usuario (mide la extensión con la cual un algoritmo puede ofrecer diversos objetos a cada usuario individual).

Diversidad inter-usuario. Se define considerando la variedad de las listas de recomendación de los usuarios. Dados los usuarios i y j, la diferencia entre los primeros L puestos de sus listas de recomendación se puede medir con la distancia de Hamming:

$$H_{ij}(L) = 1 - \frac{Q_{ij}(L)}{L}$$

donde $Q_{ij}(L)$ es el número de objetos comunes en los L primeros puestos de las listas de los usuarios i y j.

Diversidad intra-usuario. Denotando los ítems recomendados para el usuario i como $\{o_1, o_2, \dots, o_L\}$, se puede utilizar la similitud de esos ítems, $s(o_\alpha, o_\beta)$, para medir la diversidad intra-usuario. La similitud media de los ítems recomendados al usuario i,

$$I_i(L) = \frac{1}{L(L-1)} \sum_{\alpha \neq \beta} s(o_\alpha, o_\beta)$$

se puede promediar entre todos los usuarios para obtener la intra-similaridad media de las listas de recomendación, $I(L)$.

1.4.6. Innovación

La innovación en sistemas de recomendación se refiere a cómo de diferentes son los ítems recomendados con respecto a los que el usuario ya ha visto antes. La forma más fácil de cuantificar la habilidad de un algoritmo para generar resultados nuevos y no esperados es medir la *popularidad media* de los ítems recomendados

$$N(L) = \frac{1}{ML} \sum_{i=1}^{M} \sum_{\alpha \in O_R^i} k_\alpha$$

donde O_R^i es la lista de recomendación del usuario i, k_α denota el grado (popularidad) del objeto α y M es el número de usuarios. Un bajo índice de popularidad indica una alta novedad en los resultados.

Otra posibilidad para medir la innovación es utilizar la *información propia* de los ítems. Dado un objeto α, la probabilidad de que un usuario seleccionado aleatoriamente lo haya visto es k_α/M y por lo tanto su información propia es:

$$U_\alpha = \log_2\left(\frac{M}{k_\alpha}\right)$$

1.4.7. Cobertura

La cobertura mide el porcentaje de objetos que un algoritmo es capaz de recomendar (respecto del conjunto total de objetos) a los usuarios del sistema. Denotando el número total de distintos objetos

en las primeras L posiciones de todas las listas de recomendación como N_d y el total de objetos del sistema como N, la cobertura L-dependiente se define como:

$$COV(L) = \frac{N_d}{N}$$

Una baja cobertura indica que el algoritmo puede acceder y recomendar solo un pequeño número de objetos distintos (normalmente los más populares), los cuales a menudo resultan en recomendaciones poco diversas. Por lo contrario, los algoritmos con una alta cobertura son más susceptibles de proporcionar recomendaciones diversas.

1.4.8. Resumen

En la tabla siguiente se recoge un resumen de las diferentes métricas de evaluación de las recomendaciones. Las primeras dos columnas hacen referencia al nombre de la métrica y el símbolo utilizado habitualmente en la bibliografía. La tercera columna indica si es preferible obtener un valor pequeño o grande en la métrica. En la cuarta se especifica qué es lo que mide la métrica. En la columna de clasificación se indica si la evaluación de la métrica depende de la clasificación, es decir, del orden de preferencia que el sistema ha predicho sobre los ítems para el usuario. La última columna especifica si la métrica es dependiente del tamaño de la lista de recomendaciones.

Nombre	*Símbolo*	*Preferencia*	*Ámbito*	*Clasificación*	L
MAE	MAE	Pequeño	Precisión de la calificación	No	No
RMSE	RMSE	Pequeño	Precisión de la calificación	No	No
Pearson	PCC	Grande	Correlación de la calificación	No	No
Spearman	ρ	Grande	Correlación de la calificación	Sí	No
Kendall	τ	Grande	Correlación de la calificación	Sí	No
Precision	$P(L)$	Grande	Precisión de la clasificación	No	Sí
Recall	$R(L)$	Grande	Precisión de la clasificación	No	Sí
F_1*-score*	$F_1(L)$	Grande	Precisión de la clasificación	No	Sí
Half-life utility	$HL(L)$	Grande	Satisfacción	Sí	Sí
Discounted Cumulative Gain	$DCG(b, L)$	Grande	Satisfacción y precisión	Sí	Sí
Rank-biased Precision	$RBP(p, L)$	Grande	Satisfacción y precisión	Sí	Sí
Hamming distance	$H(L)$	Grande	Inter-diversidad	No	Sí
Intra-similarity	$I(L)$	Pequeño	Intra-diversidad	No	Sí
Popularidad	$N(L)$	Pequeño	Novedad	No	Sí
Información propia	$U(L)$	Grande	Inesperado	No	Sí
Cobertura	$COV(L)$	Grande	Cobertura y diversidad	No	Sí

Tabla 1: Resumen de las métricas de recomendación. Fuente: [1].

1.5. Tipos de Sistemas de Recomendación

Existen básicamente dos tipos de sistemas de recomendaciones: basados en contenido y basados en filtrado colaborativo.

En los *sistemas de recomendación basados en contenido*, el sistema tiene en cuenta únicamente las calificaciones (ya sean implícitas o explícitas) que el propio usuario que va a recibir la recomendación ha hecho sobre algunos ítems con anterioridad. A partir de esas calificaciones, se predice la calificación que el usuario daría al resto de ítems que forman el sistema y se le ofrecen al usuario aquellos con mayor valor predicho.

En los *sistemas de recomendación de filtrado colaborativo*, se utilizan las calificaciones de todos los usuarios del sistema para ofrecer a cada uno de ellos sus propias recomendaciones. Existen diversas formas de llevar a cabo este procedimiento, cada una de ellas con sus ventajas e inconvenientes. Uno de los algoritmos más utilizados para este fin es el algoritmo de los K vecinos, que tras conocer los usuarios más similares al que se está tratando, obtiene las predicciones para éste a partir de las calificaciones de esos usuarios. En el capítulo 2 se trata en más profundidad este algoritmo.

Los sistemas de recomendación de filtrado colaborativo se dividen a su vez en dos ramas: *basados en memoria* y *basados en modelo*. En los sistemas basados en memoria se le recomiendan al usuario aquellos ítems que tienen buena calificación por parte de los usuarios más similares al que recibe la recomendación. En los basados en modelo los ítems recomendados se seleccionan a partir de modelos entrenados para identificar patrones en los datos de entrada.

Existen también aproximaciones *híbridas*, en los que los métodos colaborativos se combinan con los basados en contenido o con otras variantes de colaborativos.

1.6. Amazon: un ejemplo real de un sistema de recomendación

[3] Amazon utiliza un sistema de recomendación de filtrado colaborativo para ofrecer recomendaciones a sus usuarios. Inicialmente, cuando el usuario es nuevo, realiza una recomendación "ítem a ítem" para ofrecer recomendaciones en base al ítem que el usuario ha visto recientemente. Es decir, si un usuario ha visitado el link relacionado con un iPhone, lo siguiente que le recomendará el sistema serán otros smartphones o productos de Apple. De esta forma evita el inicio en frío que sufren los sistemas de recomendación cuando cuentan con poca información.

Más adelante, cuando el sistema consigue más información acerca del usuario (éste ha mostrado interés por más productos o ítems) y puede evitar el inicio en frío, ya utiliza un sistema de recomendación "usuario a usuario", es decir, busca para el usuario actual de entre el resto aquellos más parecidos, los que han visto o comprado los mismos ítems. A partir de los intereses de esos usuarios ofrece las recomendaciones al actual.

Utiliza también algoritmos de factorización global, es decir, que compara por cada ítem varias características y les asigna un peso para luego tener la posibilidad de calcular la similitud entre ítems teniendo en cuenta varios factores y priorizar aquellos por los que el usuario muestra más interés.

2. Algoritmos de Recomendación Colaborativos

Los sistemas de recomendación colaborativos son los más utilizados actualmente, ya que teniendo en cuenta las calificaciones de todos los usuarios de un sistema se pueden ofrecer recomendaciones más variadas pero manteniendo la apreciación del usuario. A continuación se presentan dos algoritmos de filtrado colaborativo con diferentes formas de obtener las predicciones.

2.1. El algoritmo de los K vecinos

[2] El algoritmo de los K vecinos es uno de los más referenciados en la bibliografía, dado que es un algoritmo muy simple y con resultados razonadamente precisos. El problema viene cuando se pretende utilizar en un sistema muy grande, pues la escalabilidad no es una de sus ventajas: al aumentar la cantidad de datos, el número de operaciones necesarias para ofrecer una recomendación aumenta considerablemente. Existen dos versiones de este algoritmo: algoritmo de los K vecinos "usuario a usuario" y algoritmo de los K vecinos "ítem a ítem"; en ambos se puede sustituir el número fijo de vecinos por un umbral de similitud para seleccionar sólo los vecinos que lleguen a un determinado nivel de similitud.

2.1.1. Algoritmo de los K vecinos "usuario a usuario"

Esta versión del algoritmo parte de la idea de que usuarios similares calificarán de forma similar el mismo ítem. Consta de tres pasos:

1. Utilizando la medida de similitud seleccionada, se construye el conjunto de los K vecinos del usuario objetivo a. Los K vecinos serán aquellos para los que se obtiene mayor similitud con el usuario a.
2. Una vez que ya se han obtenido los K usuarios más similares al usuario a (vecinos), para calcular la predicción de este usuario para el ítem i, se aplica una función de agregación sobre las valoraciones que los K vecinos han hecho sobre el ítem i.
3. Para obtener las n mejores recomendaciones, elegimos los n ítems que proporcionan mayor satisfacción al usuario de acuerdo a las predicciones.

Esta aproximación proporciona buenos resultados con una BBDD suficientemente poblada, pero cuando el usuario no ha emitido muchas calificaciones resulta difícil encontrar unos vecinos suficientemente similares, y se da el denominado inicio en frío.

También debe hacer frente al problema de la escalabilidad antes mencionado, pues para ofrecer las recomendaciones para un usuario se debe calcular su similitud con cada uno de los usuarios del sistema, algo con un coste computacional muy alto.

2.1.2. Algoritmo de los K vecinos "ítem a ítem"

La versión "ítem a ítem" del algoritmo reduce considerablemente el problema de la baja escalabilidad. En este caso, los vecinos se calculan para cada ítem y se almacenan para ser utilizados en el

cálculo de las predicciones. Esta información queda desfasada, pero es menos sensible almacenar información imprecisa sobre los ítems que sobre los usuarios.

Los pasos de esta aproximación son:

1. Determinar los q ítems vecinos de cada ítem del sistema.
2. Para cada ítem i no valorado por el usuario objetivo a, calcular la predicción basándose en las valoraciones que a ha hecho sobre los q vecinos de i.
3. Seleccionar las n mejores recomendaciones para el usuario a a partir de las predicciones del paso anterior.

2.1.3. Medidas de similitud

2.1.3.1. Índice del coseno

Podemos medir la similitud entre dos usuarios o dos ítems con información explícita de calificaciones a través del índice del coseno, que se define como:

$$s_{xy}^{cos} = \frac{r_x \cdot r_y}{|r_x|\,|r_y|}$$

donde los vectores r_x y r_y representan las calificaciones hechas por los usuarios x e y en el caso de la similitud entre usuarios, o las calificaciones recibidas por los ítems x e y para el caso de la similitud de ítems.

2.1.3.2. Coeficiente de Pearson

Otra medida utilizada frecuentemente en la bibliografía es el coeficiente de Pearson:

$$s_{uv}^{PC} = \frac{\sum_{\alpha \in O_{uv}} (r_{u\alpha} - \overline{r_u})(r_{v\alpha} - \overline{r_v})}{\sqrt{\sum_{\alpha \in O_{uv}} (r_{u\alpha} - \overline{r_u})^2}\ \sqrt{\sum_{\alpha \in O_{uv}} (r_{v\alpha} - \overline{r_v})^2}}$$

donde $O_{uv} = \Gamma_u \cap \Gamma_v$ indica el conjunto de objetos calificados por ambos usuarios (u y v) y $\overline{r_u}$ y $\overline{r_v}$ son, respectivamente, la media del usuario u sobre Γ_u y la media del usuario v sobre Γ_v.

Una variación de esta medida es el coeficiente de Pearson restringido, donde se sustituyen las medias de los usuarios por una calificación central (por ejemplo, en una escala de 1 a 5 se escoge el 3). La idea es tener en cuenta la diferencia entre calificaciones positivas (por encima de la central) y negativas (por debajo).

Análogamente, se puede definir la similitud de Pearson para ítems:

$$s_{\alpha\beta}^{PC} = \frac{\sum_{u \in U_{\alpha\beta}} (r_{u\alpha} - \overline{r_\alpha})(r_{u\beta} - \overline{r_\beta})}{\sqrt{\sum_{u \in U_{\alpha\beta}} (r_{u\alpha} - \overline{r_\alpha})^2}\ \sqrt{\sum_{u \in U_{\alpha\beta}} (r_{u\beta} - \overline{r_\beta})^2}}$$

donde $U_{\alpha\beta}$ es el conjunto de usuarios que han calificado ambos ítems, α y β, y $\overline{r_\alpha}$ y $\overline{r_\beta}$ son, respectivamente, las calificaciones medias sobre los ítems α y β.

2.1.4. Funciones de agregación

Las funciones de agregación toman como entrada un conjunto de datos y devuelven un único valor. Este valor es calculado a partir de los datos de entrada aplicando una simple fórmula. En este caso se utilizan para calcular las predicciones de los usuarios en función de las calificaciones de sus vecinos. Las funciones de agregación más comunes son la media aritmética, la media geométrica y la media armónica.

Media aritmética

$$M(\vec{x}) = \frac{1}{n} \sum_{1 \le i \le n} x_i$$

Media geométrica

$$G(\vec{x}) = \left(\prod_{1 \le i \le n} x_i \right)^{\frac{1}{n}}$$

Media armónica

$$H(\vec{x}) = n \left(\sum_{1 \le i \le n} \frac{1}{x_i} \right)^{-1}$$

2.1.5. Ejemplo

Sea un sistema formado por 4 usuarios y 6 ítems y sean las calificaciones recogidas en la tabla siguiente:

	Ítem 1	*Ítem 2*	*Ítem 3*	*Ítem 4*	*Ítem 5*	*Ítem 6*
Usuario 1	3	4	---	4	5	2
Usuario 2	4	3	4	4	4	1
Usuario 3	---	5	5	4	5	---
Usuario 4	2	1	3	4	---	1

Tabla 2: calificaciones dadas por los usuarios a los ítems del sistema del ejemplo

Se va a utilizar la fórmula del coseno para calcular la similitud de usuarios con K = 2. Los datos obtenidos se recogen en la siguiente tabla:

	Similitud del Coseno
Usuario 1 – Usuario 2	0.861
Usuario 1 – Usuario 3	0.764
Usuario 1 – Usuario 4	0.601
Usuario 2 – Usuario 3	0.865
Usuario 2 – Usuario 4	0.835
Usuario 3 – Usuario 4	0.678

Tabla 3: similitudes entre los usuarios del sistema del ejemplo

Teniendo en cuenta que las funciones de similitud son conmutativas, para cada usuario nos encontramos con los siguientes vecinos, a los que añadimos sus calificaciones:

		Calificaciones de los vecinos					
Usuario	Vecino	Ítem 1	Ítem 2	Ítem 3	Ítem 4	Ítem 5	Ítem 6
Usuario 1	Usuario 2	4	3	4	4	4	1
	Usuario 3		5	5	4	5	
Usuario 2	Usuario 3		5	5	4	5	
	Usuario 1	3	4		4	5	2
Usuario 3	Usuario 2	4	3	4	4	4	1
	Usuario 1	3	4		4	5	2
Usuario 4	Usuario 2	4	3	4	4	4	1
	Usuario 3		5	5	4	5	

Tabla 4: calificaciones de los vecinos de cada usuario del sistema del ejemplo

Para obtener las predicciones sólo queda agregar las calificaciones de los vecinos:

Usuario	Ítem 1	Ítem 2	Ítem 3	Ítem 4	Ítem 5	Ítem 6
Usuario 1	4	4	4.5	4	4.5	1
Usuario 2	3	4.5	5	4	5	2
Usuario 3	3.5	3.5	4	4	4.5	1.5
Usuario 4	4	4	4.5	4	4.5	1

Tabla 5: predicciones para los usuarios e ítems del sistema del ejemplo

2.2. Algoritmo de filtrado colaborativo basado en tendencias

[3] Este algoritmo nace para hacer frente a los factores externos que pueden alterar la forma en que un usuario realiza una calificación. Un usuario no valorará un ítem de la misma forma si previamente ha visto uno que le ha gustado mucho que si por el contrario no le ha gustado nada. Al segundo ítem que utilice le dará una calificación en relación al que ha valorado previamente, es decir, si al primero le ha dado una calificación buena, al segundo, si le gusta más, le dará una calificación aún mejor, porque considera que tiene que valorar este ítem mejor que el anterior, pero si el primero no le hubiese gustado, el segundo no recibiría esa recompensa.

2.2.1. Cálculo de tendencias

Para compensar estos factores externos, en este algoritmo se definen dos nuevos conceptos, la tendencia del usuario y la tendencia del ítem. El concepto de tendencias se refiere a cuándo un usuario tiende a calificar los ítems positivamente o, por el contrario, negativamente. Desde el punto de vista del ítem, la tendencia se referiría a cuándo éste tiende a ser calificado positiva o negativamente.

Se define la *tendencia de un usuario* (τ_u) como la diferencia entre sus calificaciones y la media del ítem:

$$\tau_u = \frac{\sum_{i \in I_u} (v_{ui} - \overline{v_i})}{|I_u|}$$

Se define la *tendencia de ítem* (τ_i) como la diferencia entre sus calificaciones y la media del usuario:

$$\tau_i = \frac{\sum_{u \in U_i} (v_{ui} - \overline{v_u})}{|U_i|}$$

2.2.2. Análisis de los casos

Dependiendo de los valores de estas tendencias nos podemos encontrar con diferentes casos:

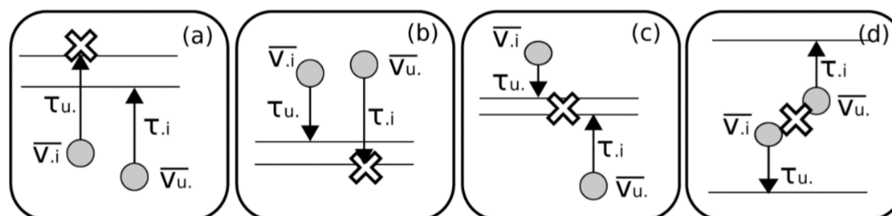

Ilustración 4: casos posibles del algoritmo basado en tendencias. Fuente: [3]

a) En el primer caso, ambos, el usuario y el ítem, tienen una tendencia positiva, lo que significa que el usuario tiende a calificar por encima de la media de calificaciones del ítem y que el ítem tiende a ser calificado por encima de la media del usuario. Para este caso utilizamos la fórmula:

$$p_{ui} = \max(\overline{v_u} + \tau_i, \overline{v_i} + \tau_u)$$

donde la razón para utilizar el máximo es dar una calificación mejor a estos ítems, cuya tendencia indica que son buenos.

b) El segundo caso es justo el opuesto. Ambos, usuario e ítem, tienen una tendencia negativa. En otras palabras, el usuario normalmente califica los ítems por debajo de su media y el ítem tiende a ser calificado por debajo de la media del usuario. En este caso, la predicción se calcula como:

$$p_{ui} = \min(\overline{v_u} + \tau_i, \overline{v_i} + \tau_u)$$

c) El tercer caso ocurre cuando la calificación viene de un usuario negativo (tiende a calificar los ítems por debajo de su media) y un ítem bueno (tiende a ser calificado por encima de la media del usuario). Si ambas medias corroboran sus tendencias (es decir, el usuario tiene media baja y el ítem alta), la predicción estará en algún punto entre ambas medias, más cercana a una u otra dependiendo del valor de las diferentes tendencias. La predicción se calcula como:

$$p_{ui} = \min\big(\max\big(\overline{v_u}, (\overline{v_i} + \tau_u)\beta + (\overline{v_u} + \tau_i)(1-\beta)\big), \overline{v_i}\big)$$

Donde β es el parámetro que controla la contribución de ambas medias.

d) El cuarto caso representa la situación donde las medias no corroboran la tendencia. Esto ocurre cuando un usuario con tendencia negativa califica un ítem con media baja (la predicción debe ser baja), pero a la vez la media del usuario es alta y la tendencia del ítem es positiva. En este caso. La predicción se calcula como:

$$p_{ui} = \overline{v_i}\beta + \overline{v_u}(1-\beta)$$

Sin embargo, no está considerado el caso (por ser inusual) en el que la tendencia de usuario sea positiva y la del ítem negativa; para este caso, llamémosle (e), se utiliza:

$$p_{ui} = \frac{(\overline{v_i} + \tau_u) + (\overline{v_u} + \tau_i)}{2}$$

2.2.3. Ejemplo

Suponiendo que tenemos un sistema de recomendaciones que consta de 4 usuarios y 6 ítems, y sea la siguiente tabla la relación de usuarios con sus respectivas calificaciones sobre los distintos ítems del sistema:

	Ítem 1	*Ítem 2*	*Ítem 3*	*Ítem 4*	*Ítem 5*	*Ítem 6*
Usuario 1	3	4	---	4	5	2
Usuario 2	4	3	4	4	4	1
Usuario 3	---	5	5	4	5	---
Usuario 4	2	1	3	4	---	1

Tabla 6: calificaciones dadas por los usuarios sobre los ítems del sistema del ejemplo.

Una vez que tenemos recopiladas estas calificaciones, calculamos las medias de los usuarios e ítems:

	Calificación media dada
Usuario 1	(3+4+4+5+2)/5 = 3.6
Usuario 2	(4+3+4+4+4+1)/6 = 3.33
Usuario 3	(5+5+4+5)/4 = 4.75
Usuario 4	(2+1+3+4+1)/5 = 2.2

Tabla 7: medias de las calificaciones dadas por los usuarios del sistema del ejemplo.

	Calificación media recibida
Ítem 1	(3+4+2)/3 = 3
Ítem 2	(4+3+5+1)/4 = 3.25
Ítem 3	(4+5+3)/3 = 4
Ítem 4	(4+4+4+4)/4 = 4
Ítem 5	(5+4+5)/3 = 4.67
Ítem 6	(2+1+1)/3 = 1.33

Tabla 8: medias de las calificaciones recibidas por los ítems del sistema del ejemplo.

Las medias son necesarias para calcular las tendencias de usuarios e ítems:

	Tendencia
Usuario 1	((3-3)+(4-3.25)+(4-4)+(5-4.67)+(2-1.33))/5 = 0.35
Usuario 2	((4-3)+(3-3.25)+(4-4)+(4-4)+(4-4.67)+(1-1.33))/6 = 0.29
Usuario 3	((5-3.25)+(5-4)+(4-4)+(5-4.67))/4 = 0.77
Usuario 4	((2-3)+(1-3.25)+(3-4)+(4-4)+(1-1.33))/5 = -0.92

Tabla 9: tendencias de los usuarios del sistema del ejemplo

	Tendencia
Ítem 1	((3-3.6)+(4-3.33)+(2-2.2))/3 = -0.04
Ítem 2	((4-3.6)+(3-3.33)+(5-4.75)+(1-2.2))/4 = -0.34
Ítem 3	((4-3.33)+(5-4.75)+(3-2.2))/3 = 0.57
Ítem 4	((4-3.6)+(4-3.33)+(4-4.75)+(4-2.2))/4 = 0.54
Ítem 5	((5-3.6)+(4-3.33)+(5-4.75))/3 = 0.77
Ítem 6	((2-3.6)+(1-3.33)+(1-2.2))/3 = -1.71

Tabla 10: tendencias de los ítems del sistema del ejemplo

Con estos datos, dependiendo del caso en el que nos encontremos, la forma de calcular cada predicción será distinta:

	Ítem 1	*Ítem 2*	*Ítem 3*	*Ítem 4*	*Ítem 5*	*Ítem 6*
Usuario 1	E	E	A	A	A	E
Usuario 2	E	E	A	A	A	E
Usuario 3	E	E	A	A	A	E
Usuario 4	B	B	C	C	C	B

Tabla 11: distribución de casos en función de las medias y tendencias de ítems y usuarios del sistema del ejemplo

Como los datos del ejemplo no han sido extraídos de un sistema real y resulta muy difícil simularlos, salen muchos casos en los que se utilizaría la fórmula E para calcular la predicción, algo que no ocurre en la realidad.

3. Sistemas de Recomendación para aplicaciones de juegos

3.1. MARS

[1] MARS es un sistema de recomendación desarrollado por investigadores de la Universidad de Zagreb en el que basan las recomendaciones que se le hacen a un usuario en la experiencia que éste ha tenido con los juegos a los que ha jugado anteriormente. Para ello utilizan indicadores como el éxito y progreso y la motivación.

3.1.1. Estimación del éxito y progreso

El tiempo que un usuario i necesita para superar satisfactoriamente el nivel j del juego viene determinado por:

$$t_{ij} = t_{ijs} + \sum_{f=1}^{p} t_{ijf}$$

donde t_{ijs} hace referencia al tiempo de juego en el nivel j cuando el usuario i lo completa, mientras que t_{ijf} es el tiempo cuando no lo supera, siendo p el número de intentos fallidos.

Si el usuario i ha jugado k veces, entonces el tiempo medio que necesita para completar el nivel j se calcula como:

$$\overline{t_{ij}} = \frac{1}{k}\sum_{l=1}^{k} t_{ij}^{l} = \frac{1}{k}\sum_{l=1}^{k}\left(t_{ijs}^{l} + \sum_{f=1}^{p} t_{ijf}^{l}\right)$$

donde t_{ij}^{l}, t_{ijs}^{l} y t_{ijf}^{l} denotan los tiempos medidos cuando el usuario juega por l-ésima vez.

El tiempo medio $\bar{t}_j$ que un total de n usuarios han necesitado para completar el nivel j se calcula:

$$\overline{t_j} = \frac{1}{n}\sum_{i=1}^{n} \overline{t_{ij}} = \frac{1}{n}\frac{1}{k}\sum_{i=1}^{n}\sum_{l=1}^{k}\left(t_{ijs}^{l} + \sum_{f=1}^{p} t_{ijf}^{l}\right)$$

3.1.2. Estimación de la motivación

El tiempo durante el cual el usuario i juega a cierto juego se define como:

$$t_i = \sum_{j=1}^{m}\left(t_{ijs} + \sum_{f=1}^{p} t_{ijf}\right)$$

donde m es el mayor nivel que el usuario ha alcanzado en el juego.

El tiempo medio $\bar{t}$ que un total de n usuarios emplean jugando a un juego se calcula como:

$$\bar{t} = \frac{1}{n}\sum_{i=1}^{n}\bar{t_i} = \frac{1}{n}\frac{1}{k_i}\sum_{i=1}^{n}\sum_{l=1}^{k_i} t_i^l = \frac{1}{n}\frac{1}{k_i}\sum_{i=1}^{n}\sum_{l=1}^{k_i}\sum_{j=1}^{m_l}\left(t_{ijs}^l + \sum_{f=1}^{p} t_{ijf}^l\right)$$

donde k_i denota cuántas veces ha jugado el usuario i y m_l es el mayor nivel que el usuario i ha alcanzado en la l-ésima vez que ha jugado.

3.1.3. Algoritmos de recomendación

3.1.3.1. *Algoritmo para determinar la categoría y dificultad de un juego nuevo*

```
categoriaJugada = categoría del juego utilizado actualmente
dificultadJugada = dificultad del juego utilizado actualmente
resultado = denota si el usuario completó la tarea con éxito
switch resultado, $\overline{t_{ij}}$, $\overline{t_j}$, $t_i$, $\bar{t}$ hacer
        caso resultado == falso && $t_i > \bar{t}$
                categoriaNueva = categoriaJugada;
                dificultadNueva = dificultadJugada - 1;
        caso resultado == falso && $t_i < \bar{t}$
                categoriaNueva = buscarOtraCategoria();
                dificultadNueva = dificultadJugada;
        caso resultado == cierto && $\overline{t_{ij}} < \overline{t_j}$ && $t_i > \bar{t}$
                categoriaNueva = categoriaJugada;
                dificultadNueva = dificultadJugada;
        caso resultado == cierto && $\overline{t_{ij}} < \overline{t_j}$ && $t_i < \bar{t}$
                categoriaNueva = categoriaJugada;
                dificultadNueva = dificultadJugada + 1;
        caso resultado == cierto && $\overline{t_{ij}} > \overline{t_j}$ && $t_i > \bar{t}$
                categoriaNueva = categoriaJugada;
                dificultadNueva = dificultadJugada - 1;
        caso resultado == cierto && $\overline{t_{ij}} > \overline{t_j}$ && $t_i < \bar{t}$
                categoriaNueva = buscarOtraCategoria();
                dificultadNueva = dificultadJugada;
fin_switch
```

3.1.3.2. *Algoritmo para seleccionar un juego para recomendar*

```
telefonoID = buscarTelefono (usuarioID);
juegos = buscarJuegos (categoriaNueva, dificultadNueva, telefonoID);
si juegos.longitud > 0 entonces
        juegos2 = buscarJuegosNoDescargados (juegos, usuarioID);
        si juegos2.longitud > 0 entonces
                devuelve juegoAleatorio (juegos2);
```

```
                si no
                        devuelve juegoAleatorio (juegos);
        si no
                juegos3 = buscarJuegos (otraCategoria, dificultadNueva, telefonoID);
                juegos4 = buscarJuegosNoDescargados (juegos3, usuarioID);
                devuelve juegoAleatorio (juegos4);
        fin_si
```

3.2. Juegos adaptativos basados en recomendaciones

[2] A la hora de diseñar un videojuego hay ciertos aspectos que hay que tener en cuenta; no todos los jugadores tienen las mismas habilidades para superar un juego ni la misma paciencia para realizar tareas repetitivas. Los juegos necesitan adaptarse a las necesidades del usuario y una forma de hacerlo es utilizar por detrás un sistema de recomendaciones. Estos sistemas se pueden aplicar en diferentes ámbitos de los juegos.

3.2.1. Ámbito externo

Sitios web de recomendación de juegos. Existen números sitios web, como Amazon.com, en los que los usuarios pueden valorar los juegos que han utilizado y recibir recomendaciones. Esto es puramente un sistema de recomendación colaborativo, ya que obtienen las recomendaciones para un usuario en función de lo que otros han calificado.

Emparejamiento. Cada día más se utilizan juegos online, en los que los usuarios se emparejan de forma que creemos generalmente aleatoria; esto no es así, pues detrás de ese emparejamiento hay un sistema que contrasta los usuarios. Para ofrecer a los jugadores una experiencia mejor, hay sistemas que utilizan el protocolo de clasificación de jugadores TrueSkill, que los empareja según su ranking de habilidades dentro de cada juego. Este protocolo utiliza algoritmos colaborativos para comparar cada usuario con el resto, obteniendo los datos necesarios de forma que el usuario no pueda modificarlos.

3.2.2. Ámbito interno

Sistemas para adaptar el nivel de dificultad. La mayoría de los juegos permiten al usuario elegir la dificultad antes de comenzar a jugar. Otros, en cambio, no dejan esta elección en manos del usuario y utilizan un sistema de ajuste del nivel de dificultad dinámico. Esto se consigue a través del concepto de rubberbanding, que consiste en dificultar el juego a los usuarios que van en cabeza y facilitárselo a aquellos que se quedan atrás.

Modelado del jugador. Cuando un jugador avanza demasiado deprisa en un juego existe el riesgo de que supere los propios límites del juego. Para solventar este problema se construyó la arquitectura IDA para modelar al jugador y así poder restringir su progreso. Con este sistema, cuando se detecta que un

jugador va a superar los límites, se toman las medidas oportunas, como distraer al usuario con alguna misión extra.

4. Tecnologías empleadas

4.1. MapReduce

[1] MapReduce es un modelo de procesamiento de datos en paralelo diseñado para ser ejecutado en un clúster de ordenadores. Su modelo de programación consiste en dos funciones definidas por el usuario (desarrollador de software): *map* y *reduce*.

Las entradas de la función map son un conjunto de pares clave / valor. Cuando se ordena al sistema ejecutar un trabajo MapReduce, las tareas de mapeo se inician en cada uno de los nodos y ejecutan la función map sobre los datos del mismo nodo. Los resultados obtenidos de esta fase (llamados resultados intermedios) se almacenan en el sistema de ficheros local, ordenados por sus claves. Cuando todas las tareas de mapeo se han completado, el motor de MapReduce notifica a las tareas de reducción que pueden comenzar. Éstas ordenan los datos intermedios y generan nuevos pares clave / valor.

4.1.1. Arquitectura de MapReduce

Los datos se dividen en particiones de igual tamaño y se dividen por todas las máquinas del clúster. Cada partición se utiliza como entrada para un mapeador. Por lo tanto, si el conjunto de datos se divide en k particiones, MapReduce creará k mapeadores para procesar los datos.

Existen dos tipos de nodos utilizados por MapReduce, el nodo master (un único nodo) y los nodos trabajadores. El nodo master controla el flujo de ejecución de las tareas de los nodos trabajadores por medio del módulo planificador. Cada uno de los nodos trabajadores es responsable de un proceso de mapeo o reducción.

La implementación básica del motor de MapReduce necesita incluir los siguientes módulos:

- *Planificador*. El planificador es el responsable de asignar las tareas de mapeo y reducción a cada nodo. Para ello tiene en cuenta dónde están los datos y la cercanía de los nodos a éstos. Cuando un proceso falla, el planificador es el encargado de reasignar el proceso a otro nodo trabajador.
- *Módulo de mapeo*. Este módulo escanea una parte de los datos e invoca la función de mapeo definida por el usuario para procesar los datos. Tras generar los resultados intermedios, los agrupa y ordena e informa al nodo master sobre las posiciones de los resultados.
- *Módulo de reducción*. El módulo de reducción coge los datos de los mapeadores tras recibir la orden del nodo master. Cuando ya tiene todos los datos, los agrupa por clave y les aplica la función de reducción definida por el usuario.

Además, para mejorar la eficiencia y usabilidad, la arquitectura básica de MapReduce normalmente se amplía con los siguientes módulos:

- *Módulos de entradas y salidas*. El módulo de entradas es el responsable de reconocer los datos de entrada con diferentes formatos y transformarlos en pares clave / valor. De esta forma se le permite al motor de procesamiento trabajar con distintos sistemas de almacenamiento. Del mismo modo, el módulo de salidas especifica el formato de las salidas de los mapeadores y reductores.
- *Módulo de combinación*. El propósito de este módulo es reducir el coste que supone mezclar los datos por medio de un proceso de reducción local de los pares clave / valor generados por el mapeador.
- *Módulo de partición*. Se utiliza para especificar cómo mezclar los pares clave / valor de los mapeadores a los reductores.
- *Módulo de agrupación*. Especifica cómo hay que juntar los datos recibidos por diferentes procesos de mapeo en uno ordenado.

4.1.2. Ejemplo de MapReduce

El ejemplo más utilizado para explicar en qué consiste MapReduce es WordCount; se trata de contar el número de ocurrencias de cada palabra en un fichero de texto. Tradicionalmente mantendríamos un array de enteros relacionados con cada palabra y al ir leyendo el fichero, por cada palabra que nos encontráramos, comprobaríamos de cuál se trata, si tiene un entero en el array relacionado con ella (si no tendríamos que añadirlo), cuál es el valor de ese entero y añadirle una unidad. Con pocos datos podría valer pero cuando el tamaño de los datos aumenta el tiempo de ejecución también lo hace considerablemente.

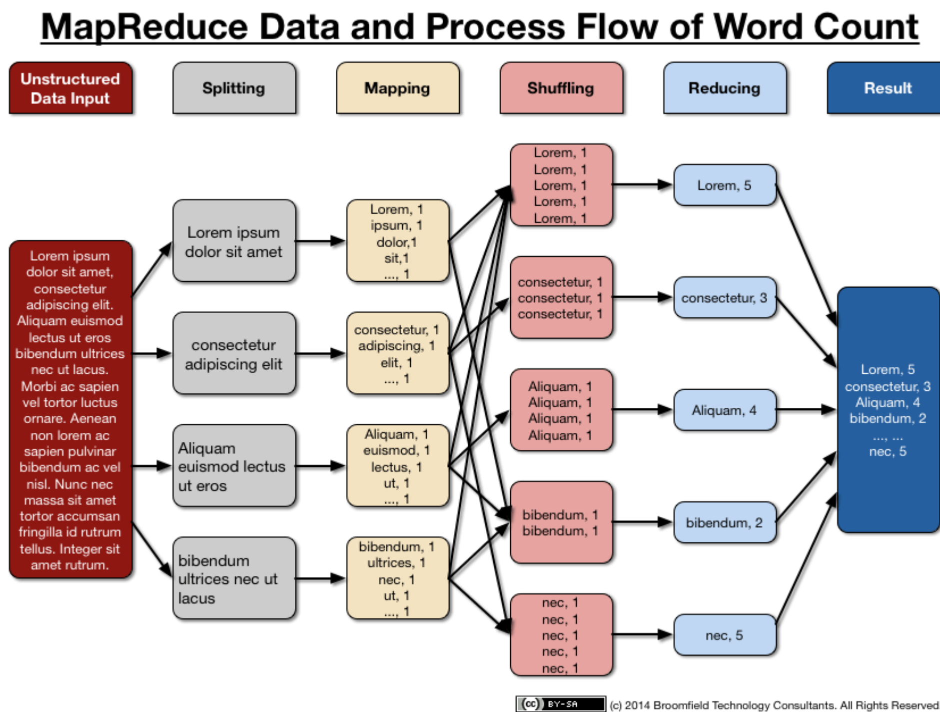

Ilustración 5: proceso de mapeo y reducción en el ejemplo de WordCount. Fuente: [7]

Con MapReduce este programa se transforma en algo muy sencillo y rápido; en primer lugar cada nodo mapeador toma una línea del fichero y separa las palabras, tras lo cual genera una pareja clave/valor con cada palabra como clave y el valor 1. Después cada nodo reductor suma los conteos de cada palabra obteniendo al final una pareja con la palabra y la suma.

4.2. Apache Hadoop

Hadoop es en la actualidad la implementación de MapReduce de código libre más popular. Está escrita en Java y testada en el clúster de Yahoo!.

[2] La librería software Apache Hadoop es un framework que permite el procesamiento distribuido de grandes conjuntos de datos sobre clústeres de computadores utilizando modelos simples de programación. Está diseñado para escalar desde un único servidor hasta miles de máquinas, cada una de las cuales ofreciendo computación y almacenamiento locales. Mejor que depender del hardware para proporcionar alta disponibilidad, la librería misma está diseñada para detectar y manejar fallos en la capa de aplicación y así proporcionar un servicio altamente disponible desde la cima de un clúster de ordenadores, cada uno de los cuales puede ser propenso a fallos.

4.2.1. HDFS: Hadoop Distributed File System

[3] El Sistema de ficheros de Hadoop es similar a otros sistemas de ficheros distribuidos pero también ofrece ventajas extra como que es altamente a prueba de fallos y está diseñado para ser utilizado con bajo coste de hardware; también proporciona un alto rendimiento de acceso a los datos de aplicaciones, especialmente útil para aplicaciones con grandes conjuntos de datos.

4.2.1.1. Arquitectura de HDFS

HDFS tiene una arquitectura maestro / esclavo. Un clúster HDFS consta de un único NameNode y varios DataNodes (normalmente uno por nodo). El NameNode es un servidor maestro que gestiona el espacio de nombres del sistema de ficheros y regula el acceso a los ficheros por parte de los clientes. Los DataNodes gestionan el almacenamiento correspondiente al nodo en el que se ejecutan.

Cuando una aplicación almacena sus datos en un fichero de HDFS, internamente éste se divide en uno o varios bloques, los cuales se almacenan en los DataNodes, que son los responsables de servir las peticiones de lectura y escritura de los clientes. Además también gestionan la creación, borrado y replicación bajo demanda del NameNode. El maestro es también el encargado de la apertura, cierre y renombrado de ficheros y directorios.

4.2.1.2. El espacio de nombres

El NameNode mantiene el espacio de nombres del sistema de ficheros. Este nodo recuerda cualquier cambio en el espacio de nombres o sus propiedades. Una aplicación puede especificar el número

de réplicas de un fichero almacenado en HDFS. El número de copias de un fichero se llama factor de replicación del mismo.

4.2.1.3. Replicación de datos

Para almacenar un fichero, éste se divide en varios bloques, cada uno de los cuales del mismo tamaño (excepto el último). Estos bloques son replicados para funcionar a prueba de fallos. El NameNode toma todas las decisiones en cuanto a la replicación de los bloques. Periódicamente recibe señales de los DataNodes que indican que siguen en funcionamiento, acompañadas de un informe de los bloques del DataNode. Cuando no recibe esta información busca otra réplica.

4.3. Apache Spark

[4] [5] Apache Spark es un framework de computación en clústeres, que proporciona APIs de alto nivel en Java, Scala y Python. Está diseñado para ser rápido y de carácter general.

En cuanto a la velocidad, Spark extiende el modelo MapReduce para soportar más tipos de cálculos sin perder eficiencia. Puede realizar estos cálculos en memoria y por lo tanto aumentar la velocidad pero también es más eficiente que MapReduce incluso con complejos algoritmos corriendo en disco.

Spark está diseñado para trabajar con grandes cantidades de datos de diversas fuentes en un mismo programa, por lo que la combinación de diferentes tipos de procesamiento no supone un problema para este framework.

4.3.1. Principales módulos de Spark

Spark SQL es un módulo de Spark para el procesamiento de datos estructurados. Proporciona una abstracción de programación llamada DataFrame y puede actuar también como un motor de consultas SQL distribuido. Un DataFrame es una colección de datos distribuida organizada en columnas. Conceptualmente, es equivalente a una tabla de una base de datos relacional, pero optimizada.

GraphX es un nuevo componente en Spark para grafos y computación paralela de grafos. En un nivel alto, GraphX extiende el RDD de Spark introduciendo una nueva abstracción Graph: un multigrafo directo con propiedades asociadas a cada vértice y enlace.

Spark Streaming es una extensión del núcleo de la API de Spark que permite el procesamiento de streaming escalable, a prueba de fallos y alto rendimiento. Proporciona una abstracción de alto nivel llamada DStream, que representa la llegada continua de datos. Internamente, una DStream se representa como una secuencia de RDDs.

MLlib es una librería de *machine learning* escalable de Spark, consistente en algoritmos de aprendizaje comunes y utilidades, incluyendo clasificación, regresión, clustering, filtrado colaborativo, reducción de dimensionalidad…

4.3.2. RDD: Resilient Distributed Dataset

[6] La abstracción de datos en Spark se realiza por medio de los RDD. Un RDD es una colección de objetos de solo lectura particionada sobre un conjunto de máquinas que puede ser reconstruida si se pierde alguna de las particiones.

Spark ofrece cuatro formas de crear un RDD:

- A partir de un fichero perteneciente a un sistema de ficheros compartidos (como HDFS).
- Paralelizando una colección Scala, es decir, dividiéndola en segmentos que serán enviados a diferentes nodos.
- Transformando otro RDD existente.
- Cambiando la persistencia de otro RDD existente.

4.4. Apache Hadoop NextGen MapReduce (YARN)

[6] La segunda versión de MapReduce de Hadoop, conocida como Yarn, introduce la idea de separar las dos grandes funcionalidades del rastreador de trabajos (JobTracker), gestión de recursos y planificación de tareas, en demonios diferentes.

El Resource Manager y su esclavo de nodo, el Node Manager, forman la infraestructura de computación de datos. El Resource Manager es la máxima autoridad en cuanto a recursos entre todas las aplicaciones del sistema.

El Application Master de cada aplicación se encarga de negociar los recursos del Resource Manager y trabajar con el gestor de nodo para ejecutar y monitorizar las tareas.

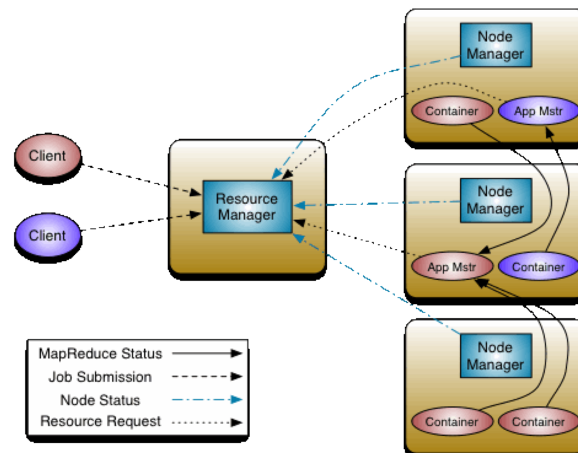

Ilustración 6: estructura de la ejecución de dos aplicaciones simultáneas sobre Yarn. Fuente: [6].

El Resource Manager se compone de dos partes: Scheduler y el Applications Manager.

El Scheduler es el responsable de distribuir los recursos a las distintas aplicaciones en ejecución, teniendo en cuenta para ello las capacidades, colas, etc.

El Applications Manager es el responsable de aceptar los trabajos, negociar el primer contenedor en el que ejecutar el maestro de cada aplicación y proporcionar el servicio de reiniciar el contenedor del maestro de aplicación en caso de fallo.

[10] Cuando se utiliza Yarn para ejecutar una aplicación de Spark, el Driver de ésta se ejecuta dentro de un proceso del Applications Master de la aplicación, el cual es manejado por Yarn en el clúster. El Applications Master obtiene la dirección del Resource Manager a partir de la configuración de Hadoop.

5. Implementación de los algoritmos de recomendación en Spark

Los algoritmos han sido implementados utilizando la API de Spark en Java y probados en un clúster proporcionado por la empresa Biko2. En el servidor utilizado se disponen de 7 máquinas, cada una de las cuales cuenta con 8 Gb de memoria RAM y CPU Intel Xeon a 2.66 GHz con cuatro núcleos. Esas máquinas tienen instalado Hadoop, Spark y Yarn.

5.1. Algoritmo basado en similitudes (vecinos)

5.1.1. Clase Neighbours

Esta clase contiene el método principal del programa, que se encarga de llamar al resto de clases para efectuar los cálculos necesarios. También contiene un método para devolver el mayor número identificador de los ítems, necesario para calcular la similitud del coseno.

Contiene también un método llamado collectData(), que a partir de la ruta de un fichero en HDFS, lee los datos contenidos en él y los devuelve convertidos en un JavaPairRDD, esto es, una extensión de tipo clave / valor de un RDD, la abstracción de objetos de Spark. El JavaPairRDD devuelto tiene como clave el par usuario y como valor contiene el ítem y la calificación (Spark permite agrupar objetos en uno único con las clases Tuple2, Tuple3, Tuple4…) que relaciona el usuario e ítem concretos, es decir, la calificación que el usuario ha dado al ítem (de haberlo hecho). Una vez obtenida esta información del fichero, en el mismo método se realiza un filtrado para limitar el número de usuarios que van a participar en el sistema de recomendación; el resto serán ignorados. Esto se hace para poder realizar pruebas variando el número de usuarios y contrastar los datos obtenidos.

Una vez obtenidas las similitudes entre usuarios, realiza una fase de agregación sobre los ratings de los usuarios “vecinos” de cada uno para obtener una predicción de la calificación que el usuario le daría a cada ítem.

En MapReduce, realizar una iteración por todos los elementos de un RDD es una operación muy costosa, pues mantiene una gran cantidad de datos. Para resolver problemas como el cálculo de medias de usuarios o la agregación de ratings se utilizan métodos reductores, como combineByKey, una implementación proporcionada por Spark. Este método necesita como parámetros tres funciones: una de creación, otra de adición y otra de fusión. Con la primera se crea un objeto auxiliar del tipo indicado, con la segunda se añaden elementos a ese objeto y con la tercera se fusionan los elementos de distintos objetos (los cálculos se realizan en paralelo y por lo tanto se crean varios objetos que luego se deben juntar).

5.1.2. Clase CosineSimilarity

La clase CosineSimilarity proporciona un constructor con el JavaPairRDD ratings como único parámetro. La clase también proporciona el método computeSimilarity, que realiza el propio cálculo. Para poder llevar a cabo la operación de un modo más eficiente, se utilizan las clases Vector, VectorWithNorm y BLAS proporcionadas por Spark. Por cada usuario, se convierten sus calificaciones en un vector donde los índices del mismo representan los ítems. Con estos vectores se crean objetos de la clase VectorWithNorm para obtener por su propio método la norma de los vectores. La clase BLAS proporciona un método para calcular el producto escalar de los dos vectores.

5.1.3. Clase PearsonSimilarity

La clase PearsonSimilarity es algo más compleja que la anterior. El constructor es igual que en CosineSimilarity. El método computeSimilarity(), como en la anterior, realiza el cálculo de la similitud entre usuarios, pero debido a las propias exigencias del método, lo hace de una forma más compleja.

Lo primero que se calcula es la media de todos los usuarios; en este caso, para calcular las medias de los usuarios, las funciones utilizadas son createAcc, addAndCount y combine que utilizan objetos de la clase Average.

Del mismo modo que para la media de los usuarios, para calcular la similitud se utiliza el método combineByKey; en este caso las funciones a utilizar son createAccumulator, addToAccumulator y combineAccumulator, que utilizan objetos de la clase PearsonAccumulator. Tanto esta clase como Average tienen métodos para una vez terminada la combinación devolver el valor del agregado.

Como PearsonSimilarity utiliza muchos mapeos, para facilitar la tarea algunas de las funciones que se utilizan en ellos se encuentran en clases diferentes: AverageMapper, JoinedMapper y AccumulatorMapper.

5.1.4. Clase Metrics

Esta clase utiliza RegressionMetrics, proporcionada por Spark, para calcular el error absoluto medio y la raíz del error cuadrático medio de las predicciones hechas por el sistema respecto de los datos con los que contábamos al principio.

5.2. Algoritmo basado en tendencias

5.2.1. Clase Trends

Ésta es la clase principal del programa, que recoge los datos del fichero de HDFS y gestiona la llamada al resto de las clases. Para obtener los datos de entrada opera de la misma forma que la clase Neighbours del algoritmo basado en similitudes.

5.2.2. Clase RatingsPredictor

Es la clase que calcula las predicciones por medio de las tendencias de ítems y usuarios. Proporciona un constructor con un único parámetro, el JavaPairRDD de ratings, ordenado en forma de clave de usuario e ítem y valor de calificación, y lo almacena tal cual en un objeto propio de la clase.

Para realizar algunos mapeos en los cálculos de tendencias y medias de usuarios e ítems se utilizan clases independientes, dejando más claro el código. Estas clases son: ReorganizerMapper y AverageMapper.

Del mismo modo que en el algoritmo de similitudes, para calcular las medias es necesario crear tres funciones con las que poder hacer la parte reductora. Estas funciones, al igual que las necesarias para el cálculo de tendencias, se encuentran en la clase Average, la cual también proporciona un constructor y métodos para realizar las tareas oportunas para la reducción.

5.2.3. Clase Metrics

La clase Metrics es la misma que en el algoritmo basado en similitudes. Calcula y devuelve los valores de error absoluto medio y raíz del error cuadrático medio de las predicciones respecto de los datos reales.

5.3. Resultados obtenidos

Para poder analizar los datos resultantes de ambos algoritmos, se han realizado varias pruebas, en cada una de las cuales se calculaban predicciones para 1000, 2000, 3000, 4000, 5000, 6000, 7000 y 8000 usuarios. Una vez obtenidos estos datos, para el tiempo de duración (que es el único resultado variable) se calcula el promedio.

El tiempo medio, el tiempo medio por usuario, el número de predicciones por segundo y los errores cometidos se representan en las siguientes gráficas.

5.3.1. Algoritmo basado en similitudes

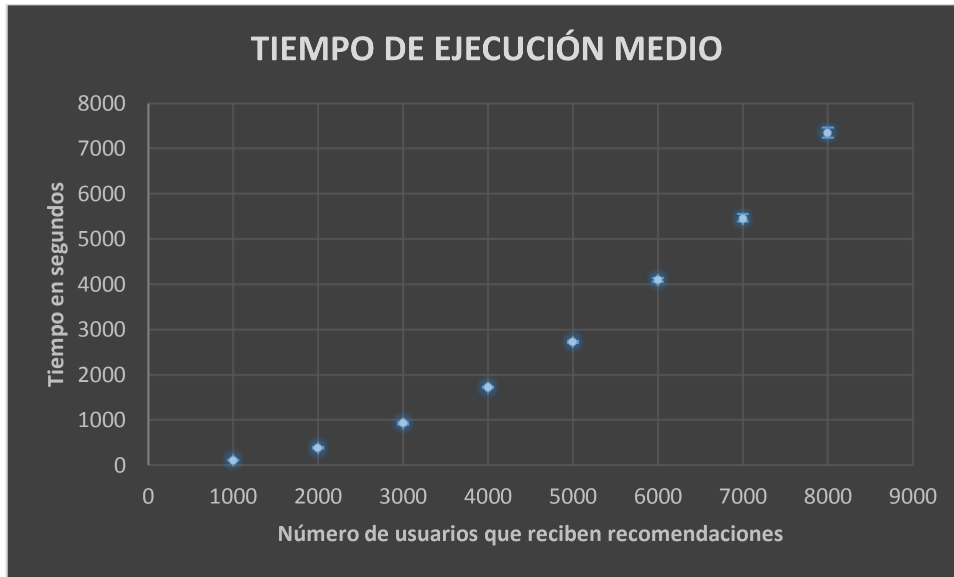

Ilustración 7: tiempo de ejecución medio en el algoritmo de los K vecinos (basado en similitudes).

Como se puede ver en la gráfica anterior, el tiempo medio de ejecución del programa aumenta a medida que lo hace el número de usuarios, llegando incluso a superar las 2 horas cuando el sistema cuenta con 8000 usuarios, algo que no augura buenos resultados.

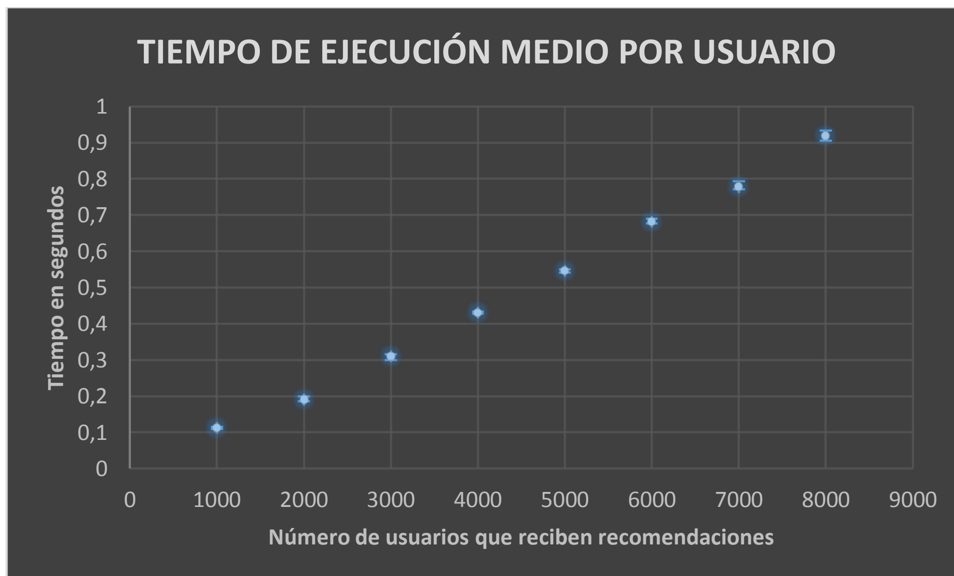

Ilustración 8: tiempo de ejecución medio por usuario en el algoritmo basado en similitudes.

El tiempo de ejecución medio por usuario, es decir, el tiempo medio de ejecución dividido por el número de usuarios a los que se ofrecen recomendaciones, crece de manera más o menos proporcional al número de usuarios. Esto es una mala señal, ya que si el algoritmo fuera escalable estos valores no deberían ser crecientes. Esta gráfica no hace sino confirmar lo que ya se había expuesto anteriormente en el apartado 2.1., que **el algoritmo de los K vecinos no es escalable** debido a su naturaleza de comparar cada par de usuarios para hallar la similitud entre los mismos.

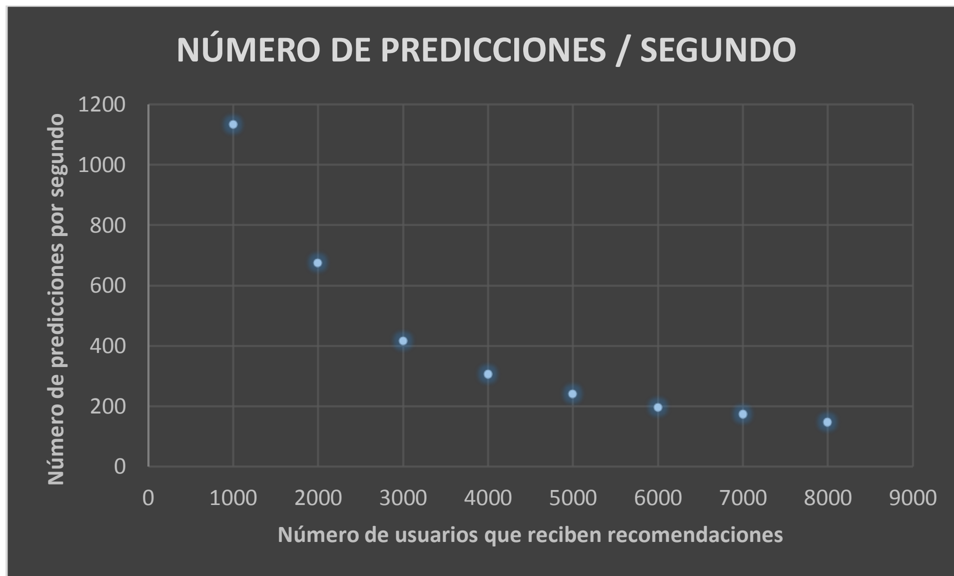

Ilustración 9: número de predicciones que realiza el algoritmo basado en similitudes por segundo.

Como consecuencia de lo visto en la gráfica anterior, el número de predicciones que el sistema realiza dividido por el tiempo de ejecución (este tiempo incluye el cálculo de similitudes) es menor a medida que incrementa el número de usuarios.

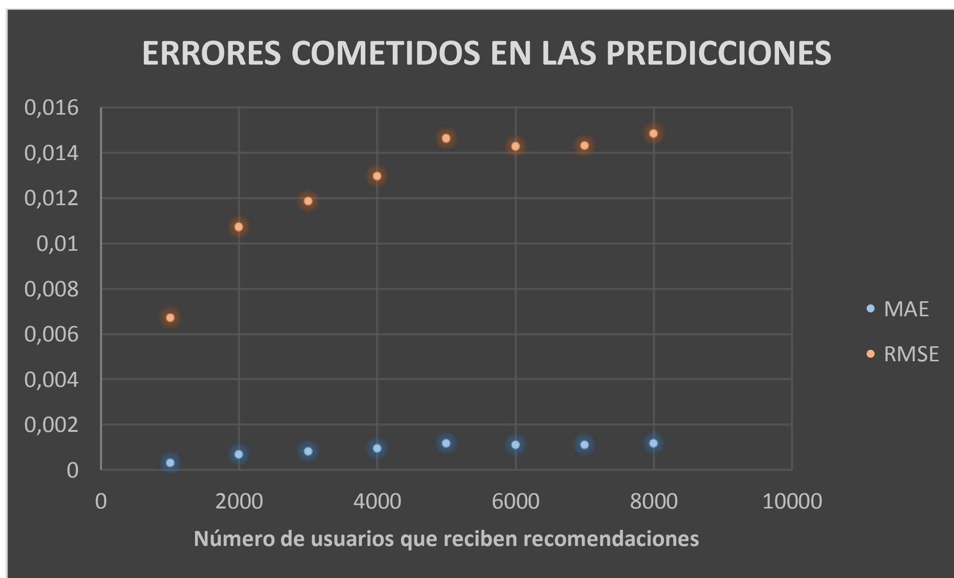

Ilustración 10: Mean Absolute Error y Root Mean Squared Error del algoritmo basado en similitudes.

Los valores de MAE y RMSE obtenidos son realmente buenos; este algoritmo, aunque le cueste, calcula las predicciones con una precisión en los datos muy alta. Los valores de RMSE con mayores que los de MAE porque ese índice crece en mayor medida por su propia definición. Aun así, los valores de RMSE también son muy buenos.

5.3.2. Algoritmo basado en tendencias

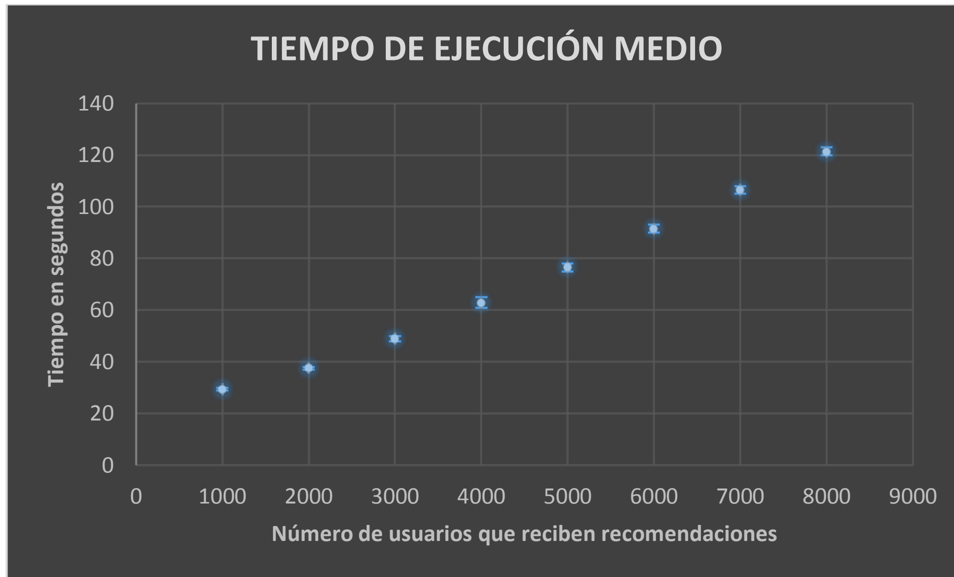

Ilustración 11: tiempo de ejecución medio del algoritmo basado en tendencias.

Este algoritmo alcanza los 2 minutos para realizar las predicciones de 8000 usuarios, un tiempo muy razonable para la cantidad de operaciones que debe hacer.

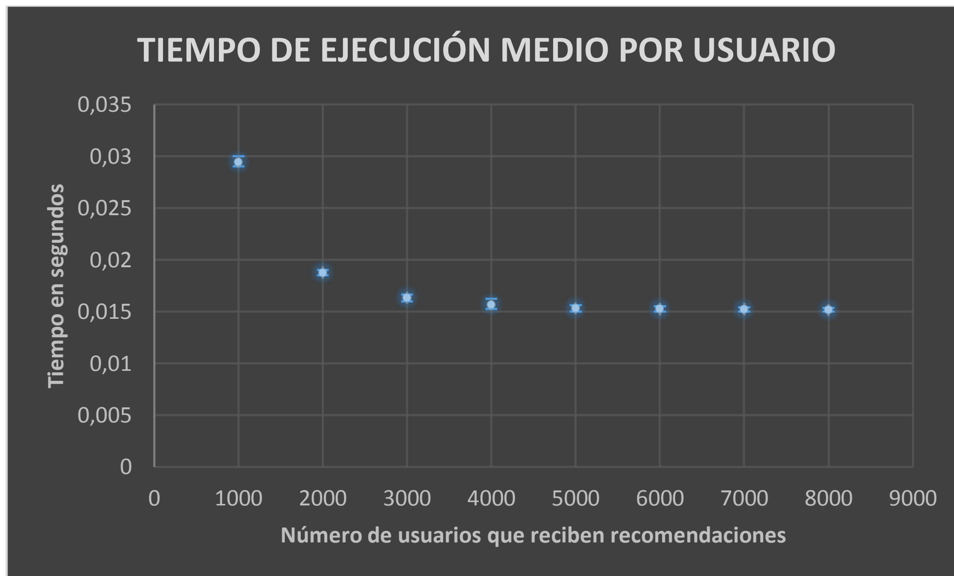

Ilustración 12: tiempo medio de ejecución por usuario en el algoritmo basado en tendencias.

Como se ve en la gráfica anterior, el tiempo medio por usuario decrece a medida que aumenta el número de usuarios con los que trabaja el sistema. Esto indica que **el algoritmo basado en tendencias es escalable**, y por lo tanto, adecuado para los intereses del proyecto.

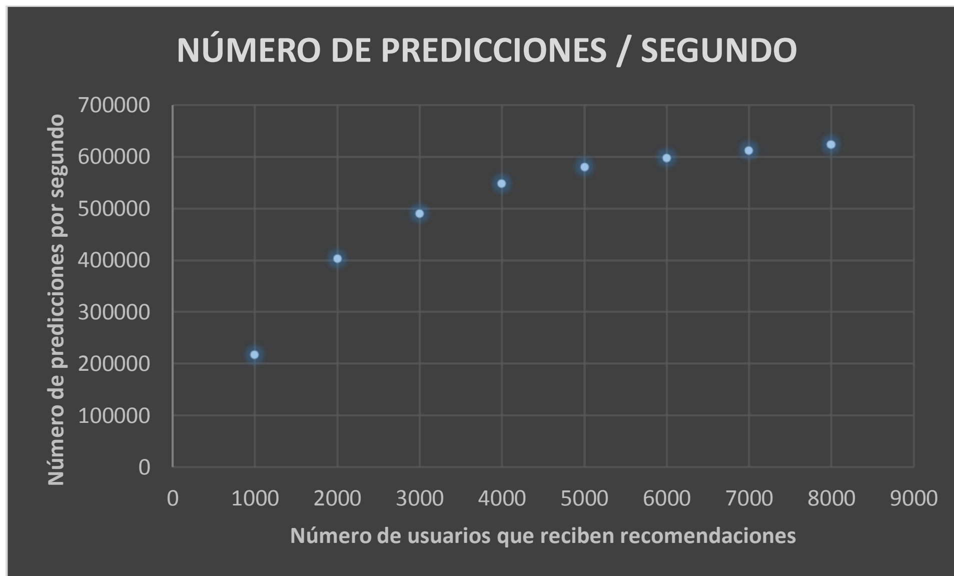

Ilustración 13: número de predicciones por segundo que realiza el algoritmo basado en tendencias.

El rendimiento incremental con el que trabaja este algoritmo influye también en el número de predicciones que realiza por unidad de tiempo, sobrepasando las 600.000 predicciones por segundo con 8000 usuarios.

Ilustración 14: errores cometidos en el algoritmo basado en tendencias.

Los errores cometidos por el algoritmo son aceptables, si bien no llegan al nivel del algoritmo basado en similitudes.

Conclusiones y trabajos futuros

Los algoritmos

Los algoritmos implementados en este proyecto permiten realizar costosas operaciones en un tiempo reducido. Más concretamente, el algoritmo basado en tendencias permite realizar millones de operaciones de media en tan solo un segundo.

Comparando ambos algoritmos, en el que uno tiene menor rendimiento pero mejor precisión de datos y el otro al contrario, habría que fijarse en la situación a la que está destinado para poder decantarse por uno de los dos.

Por un lado, el algoritmo basado en similitudes, o algoritmo de los K vecinos, tiene un rendimiento muy bajo. Para 8000 usuarios (una cifra mucho menor que la que habrá en un sistema real) el sistema tarda 2 horas en ofrecer las recomendaciones. Es impensable utilizarlo en tiempo real, por lo que la única salida viable para el algoritmo basado en similitudes sería realizar los cálculos de manera offline y posteriormente ofrecerlos a los usuarios. El inconveniente de esta forma de ofrecer los resultados es que no se tendrán en cuenta las últimas calificaciones que los usuarios han hecho, y los buenos resultados del algoritmo en cuanto a certeza de las predicciones disminuirían por este tema.

Por otro lado, el algoritmo basado en tendencias ofrece mucho mejores resultados en cuanto a tiempo. Para un sistema con 8000 usuarios calcularía las predicciones en 2 minutos, pero aun así no es aceptable hacer esperar al usuario durante 2 minutos (o más) mientras se calculan las predicciones. Habría que hacerlo también de manera offline pero, al contrario que el algoritmo anterior, podría ejecutarse más frecuentemente y ofrecer datos más actualizados y por lo tanto disminuir la desventaja que tiene en cuanto a precisión de las predicciones respecto del algoritmo basado en similitudes.

Escoger uno u otro algoritmo dependerá de la naturaleza del sistema de recomendación que se quiera implementar.

ZombieLib

Este proyecto no termina aquí, puesto que esto es una pequeña parte de un proyecto llevado a cabo conjuntamente entre la empresa Biko2 y el Grupo de Sistemas Distribuidos de la Universidad Pública de Navarra. El proyecto se llama ZombieLib y se trata de una librería insertada en el código fuente de los juegos de dispositivos móviles (previo contacto con los desarrolladores) que obtiene datos sobre la forma en la que los usuarios utilizan los juegos (tiempo jugado, dinero gastado…). Cuando el sistema detecta (por medio de ciertos indicadores) que el usuario abandonará el juego, se le recomienda otro de los que contienen la librería ZombieLib.

Se pretende, para Zombielib, realizar un sistema de recomendación que fusione las ideas del algoritmo de MARS y el basado en tendencias. Está fusión funcionará como si se tratase del algoritmo de MARS para determinar la categoría y dificultad del juego que se va a recomendar al usuario. Una vez que el sistema ya tiene una lista de los juegos que el usuario todavía no ha descargado utilizará el algoritmo de recomendación basado en tendencias para ofrecer los juegos con mejores expectativas para el usuario.

Bibliografía

[1] L. Lü, M. Medo, C. H. Yeung, Y.-C. Zhang y Z.-K. Zhang, «Recommender Systems,» *Physics Reports,* 2012.

[2] J. Bobadilla, F. Ortega, A. Hernando y A. Gutiérrez, «Recommender Systems Survey,» *Knowledge-Based Systems,* 2013.

[3] P. Skocir, L. Marusic, M. Marusic y A. Petric, «The MARS - A Multi-Agent Recommendation System for Games on Mobile Phones».

[4] B. Medler, «Using Recommendation Systems to Adapt Gameplay».

[5] F. Li, B. Chin Ooi, M. Tamer Özsu y S. Wu, «Distributed Data Management Using MapReduce,» vol. 0, nº 0, 2013.

[6] «https://hadoop.apache.org/,» 05 05 2015. [En línea]. Available: https://hadoop.apache.org/.

[7] D. Borthakur, «HDFS Architecture Guide,» The Apache Software Foundation, 2008. [En línea].

[8] «http://spark.apache.org/docs/latest/,» [En línea]. Available: http://spark.apache.org/docs/latest/.

[9] H. Karau, A. Konwinski, P. Wendell y M. Zaharia, Learning Spark, O'Reilly, 2015.

[10] M. Zaharia, M. Chowdhury y M. J. Fran, «Spark: Cluster Computing with Working Sets».

[11] «http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html,» [En línea]. Available: http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html.

[12] «https://spark.apache.org/docs/latest/running-on-yarn.html,» [En línea]. Available: https://spark.apache.org/docs/latest/running-on-yarn.html.

[13] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Google, Inc..

[14] Y. Xu, W. Qu, Z. Li, G. Min, K. Li y Z. Liu, «Efficient k-Means++ Approximation with MapReduce,» *IEEE Transactions on Parallel and Distributed Systems,* vol. 25, nº 12, 2014.

[15] S. Meng, W. Dou, X. Zhang y J. Chen, «KASR: A Keyword-Aware Service Recommendation Method on MapReduce for Big Data Applications,» *IEEE Transactions on Parallel and Distributed Systems,* 2013.

[16] Y. Koren, R. Bell y C. Volinsky, «Matrix Factorization Techniques for Recommender Systems,» IEEE Computer Society, 2009.

[17] F. Cacheda, V. Carneiro, D. Fernández y V. Formoso, «Comparison of Collaborative Filtering Algorithms: Limitations of Current Techniques and Proposals for Scalable, High-Performance Recommender Systems,» *ACM Transactions on the Web,* vol. 5, nº 1, 2011.

[18] F. Zhang y Q. Zhou, «HHT-SVM: An online method for detecting profile injection attacks in collaborative recommender systems,» *Knowledge-Based Systems,* 2014.

[19] «Microsoft Research,» [En línea]. Available: http://research.microsoft.com/en-us/projects/trueskill/.

[20] I. Pointer y L. Terlouw, «Distributed Machine Learning with Apache Mahout».

Anexo 1: código generado para el algoritmo basado en similitudes

Clase Neighbours

```
import org.apache.spark.HashPartitioner;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;
import scala.Tuple2;
import java.util.*;


public class Neighbours {

    // variables de contexto
    static SparkConf conf;
    static JavaSparkContext sc;


    static public int maxItem (final JavaPairRDD<Integer, Tuple2<Integer, Double>> ratings) {
        return ratings.map(x -> x._2()._1()).top(1).get(0);
    }


    static public JavaPairRDD<Integer, Tuple2<Integer, Double>> collectData(String path, final Integer max){
        return sc.textFile(path)
                .mapToPair(s -> {
                    List<String> separated = Arrays.asList(s.split("::"));
                    Integer u = Integer.parseInt(separated.get(0)); //usuario
                    Integer i = Integer.parseInt(separated.get(1)); // ítem
                    Double r = Double.parseDouble(separated.get(2)); // rating
                    return new Tuple2(u, new Tuple2(i, r));
                })
                .filter(new Function<Tuple2<Integer, Tuple2<Integer, Double>>, Boolean>() {
```

```
                public Boolean call(Tuple2<Integer, Tuple2<Integer, Double>> x) {
                    return (x._1().compareTo(max) <= 0); // limitar el número de usuarios
                }
            });
}

public static void main(String[] args) {

    conf= new SparkConf().setAppName("kNN");
    sc = new JavaSparkContext(conf);

    String ratingsPath = new String("hdfs:///movielens/large/ratings.dat");

    Double threshold;
    Integer typeSim, maxUser, maxItem;
    final Integer N = 20;


    if (args.length < 3) {
        typeSim = 0;
        threshold = 0.8;
        maxUser = 1000;
    } else {
        typeSim = Integer.parseInt(args[0]);
        threshold = Double.parseDouble(args[1]);
        maxUser = Integer.parseInt(args[2]);
    }

    final JavaPairRDD<Integer, Tuple2<Integer, Double>> ratings = collectData(ratingsPath, maxUser);
    maxItem = maxItem(ratings);

    long t1 = System.currentTimeMillis();

    // búsqueda de vecinos
    JavaPairRDD<Tuple2<Integer, Integer>, Double> similarities;
    if (typeSim.equals(0)){
        CosineSimilarity cs = new CosineSimilarity(ratings);
        similarities = cs.computeSimilarity(maxItem); // <vecino, similitud>
    } else {
        PearsonSimilarity ps = new PearsonSimilarity(ratings);
```

```
    similarities = ps.computeSimilarity(); // <vecino, similitud>
}

JavaPairRDD<Tuple2<Integer, Integer>, Double> predictions = similarities // <<usuario, vecino>, similitud>
        .filter(x -> x._2().compareTo(threshold) >= 0) // <<usuario, vecino>, similitudAlta>
        .mapToPair(x -> new Tuple2(x._1()._2(), x._1()._1()))
        .join(ratings) // <vecino, <ítem, rating>>
        .mapToPair(
                // entrada: <vecino, <usuario, <item, rating>>>
                // salidas: <usuario, ítem>, rating
                new PairFunction<Tuple2<Integer, Tuple2<Integer, Tuple2<Integer, Double>>>,
                        Tuple2<Integer, Integer>, Double>() {
                    public Tuple2<Tuple2<Integer, Integer>, Double>
                    call(Tuple2<Integer, Tuple2<Integer, Tuple2<Integer, Double>>> x) {
                        return new Tuple2(new Tuple2(x._2()._1(), x._2()._2()._1()), x._2()._2()._2());
                    }
                })
        .combineByKey(
                Average.createAcc,
                Average.addAndCount,
                Average.combine,
                new HashPartitioner(7)) // <<usuario, item>, Average>
        .mapToPair(new PairFunction<
                Tuple2<Tuple2<Integer, Integer>, Average>,
                Tuple2<Integer, Integer>, Double>() {
            public Tuple2<Tuple2<Integer, Integer>, Double> call(Tuple2<Tuple2<Integer, Integer>, Average> x) {
                return new Tuple2(new Tuple2(x._1()._1(), x._1()._2()), x._2().avgA());
            }
        }); // <<usuario, ítem>, media>>

long size = predictions.count();
long t2 = System.currentTimeMillis();
System.out.println("Para un total de " + maxItem + " ítems con un máximo de " + maxUser
        + " usuarios ha calculado " + size + " predicciones en " + ((t2-t1)/1000) + " segundos.");


JavaPairRDD auxRatings = ratings
        .mapToPair(x -> new Tuple2(new Tuple2(x._1(), x._2()._1()), x._2()._2()));
final Metrics metrics = new Metrics(predictions
        .join(auxRatings, new HashPartitioner(7)));
System.out.println(metrics.getMetrics());
```

```
    }
}
```

Clase CosineSimilarity

```
import org.apache.spark.HashPartitioner;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.mllib.clustering.VectorWithNorm;
import org.apache.spark.mllib.linalg.BLAS;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
import scala.Tuple2;

public class CosineSimilarity {

    final JavaPairRDD<Integer, Tuple2<Integer, Double>> ratings;


    public CosineSimilarity (JavaPairRDD<Integer, Tuple2<Integer, Double>> r) {
        ratings = r.cache();
    }


    public JavaPairRDD<Tuple2<Integer, Integer>, Double> computeSimilarity (final Integer maxItem){

        // entrada: <<usuario1, vector1>, <usuario2, vector2>>
        // salidas: <usuario1, usuario2>, similitudCoseno
        PairFunction<Tuple2<Tuple2<Integer, Vector>, Tuple2<Integer, Vector>>,
                Tuple2<Integer, Integer>, Double> compute =
                (x) -> {
                    final Double den1 = (new VectorWithNorm(x._1()._2())).norm();
                    final Double den2 = (new VectorWithNorm(x._2()._2())).norm();
                    Double num = BLAS.dot(x._2()._2(), x._1()._2());
                    return new Tuple2 (new Tuple2(x._1()._1(), x._2()._1()), (num / (den1 * den2)));
```

```
                };

        JavaRDD<Tuple2<Integer, Vector>> vectors = ratings
                .groupByKey(new HashPartitioner(7))
                .map(r -> new Tuple2(r._1(), Vectors.sparse(maxItem+1, r._2()))); // <usuario, vectorRatings>

        return vectors
                .cartesian(vectors)                       // <<usuario1, vectorRatings1>, <usuario2, vectorRatings2>>
                .filter(x -> x._1()._1() != x._2()._1()) // usuario1 != usuario2
                .mapToPair(compute);                      // <<usuario1, usuario2>, similitud>

    }

}
```

Clase PearsonSimilarity

```
import org.apache.spark.HashPartitioner;
import org.apache.spark.api.java.JavaPairRDD;
import scala.Tuple2;

public class PearsonSimilarity {

    final JavaPairRDD<Integer, Tuple2<Integer, Double>> ratings;


    public PearsonSimilarity (JavaPairRDD<Integer, Tuple2<Integer, Double>> r) {
        ratings = r.cache();
    }


    JavaPairRDD<Tuple2<Integer, Integer>, Double> computeSimilarity () {

        JavaPairRDD<Tuple2<Integer, Integer>, Tuple2<Double, Double>> userMeans = ratings
                .mapToPair(x -> new Tuple2(x._1(), x._2()._2())) // <usuario, rating>
                .combineByKey(
                        Average.createAcc,
                        Average.addAndCount,
                        Average.combine,
```

```
                new HashPartitioner(7))                        // <usuario, Average>
            .mapToPair(new AverageMapper())                    // <usuario, mediaUsuario>
            .join(ratings, new HashPartitioner(7))             // <usuario, <mediaUsuario, <ítem, rating>>>
            .mapToPair(new JoinedMapper());                    // <<usuario, ítem>, <rating, mediaUsuario>>

        return userMeans
            .cartesian(userMeans) // <<<us1, it>, <rating1, mediaUs1>>, <<us2, it2>, <rating2, mediaUs2>>>
            .filter(x ->
                ((x._2()._1()._1().compareTo(x._1()._1()._1()) != 0)) && // usuario2 != usuario1
                    (x._1()._1()._2().equals(x._2()._1()._2())))     // item1 == item2
            .mapToPair(x -> new Tuple2(
                            new Tuple2(x._1()._1()._1(), x._2()._1()._1()),     // <usuario1, usuario2>
                            new Tuple2(
                                new Tuple2(x._1()._2()._1(), x._1()._2()._2()), // <rating1, mediaUsuario1>
                                new Tuple2(x._2()._2()._1(), x._2()._2()._2())  // <rating2, mediaUsuario2>
                            )
                    )
            ) // <<usuario1, usuario2>, <<rating1, mediaUsuario1>, <rating2, mediaUsuario2>>>>
            .combineByKey(
                PearsonAccumulator.createAccumulator,
                PearsonAccumulator.addToAccumulator,
                PearsonAccumulator.combineAccumulators,
                new HashPartitioner(7)) // <usuario, PearsonAccumulator>
            .mapToPair(new AccumulatorMapper()); // <usuario, coeficientePearson>

    }

}
```

Clase PearsonAccumulator

```
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import scala.Tuple2;

import java.io.Serializable;

public class PearsonAccumulator implements Serializable {
```

```
private Double _numerator;
private Double _denominator1;
private Double _denominator2;

public Double getNumerator(){return _numerator;}
public Double getDenominator1(){return _denominator1;}
public Double getDenominator2(){return _denominator2;}

PearsonAccumulator(Double rating1, Double rating2, Double mean1, Double mean2) {
    _numerator = (rating1 - mean1) * (rating2 - mean2);
    _denominator1 = Math.pow(rating1 - mean1, 2);
    _denominator2 = Math.pow(rating2 - mean2, 2);
}

PearsonAccumulator add (Double rating1, Double rating2, Double mean1, Double mean2) {
    _numerator += (rating1 - mean1) * (rating2 - mean2);
    _denominator1 += Math.pow(rating1 - mean1, 2);
    _denominator2 += Math.pow(rating2 - mean2, 2);
    return this;
}

PearsonAccumulator combine (PearsonAccumulator other) {
    _numerator += other.getNumerator();
    _denominator1 += other.getDenominator1();
    _denominator2 += other.getDenominator2();
    return this;
}


Double getCoefficient () {
    return _numerator / ((Math.sqrt(_denominator1) * Math.sqrt(_denominator2)));
}


static Function<Tuple2<Tuple2<Double, Double>, Tuple2<Double, Double>>,
        PearsonAccumulator> createAccumulator =
        x -> new PearsonAccumulator(x._1()._1(), x._2()._1(), x._1()._2(), x._2()._2());

static Function2<PearsonAccumulator, Tuple2<Tuple2<Double, Double>, Tuple2<Double, Double>>,
        PearsonAccumulator> addToAccumulator =
        (a, x) -> a.add(x._1()._1(), x._2()._1(), x._1()._2(), x._2()._2());
```

```
    static Function2<PearsonAccumulator, PearsonAccumulator,
            PearsonAccumulator> combineAccumulators =
            (a, b) -> a.combine(b);
}
```

Clase Average

```
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;

import java.io.Serializable;

public class Average implements Serializable {
    private Double _total;
    private int _num;

    public Average (Double total, int num) {
        _total = total;
        _num = num;
    }
    public Average add (Double total, int num){
        _total += total;
        _num += num;
        return this;
    }
    public Average combine (Average other){
        _total += other.getTotal();
        _num += other.getNum();
        return this;
    }

    public Double getTotal(){return _total;}
    public int getNum(){return _num;}

    public Double avgA() {return _total / new Double(_num);}


    static Function<Double, Average> createAcc = x -> new Average(x, 1);
```

```
    static Function2<Average, Double, Average> addAndCount = (a, x) -> {a.add(x, 1); return a;};
    static Function2<Average, Average, Average> combine = (a, b) -> {a.combine(b); return a;};
}
```

Clase AverageMapper

```
import org.apache.spark.api.java.function.PairFunction;
import scala.Tuple2;

public class AverageMapper implements PairFunction<Tuple2<Integer, Average>, Integer, Double>{

    public Tuple2<Integer, Double> call(Tuple2<Integer, Average> x) {
        return new Tuple2(x._1(), x._2().avgA());
    }

}
```

Clase JoinedMapper

```
import org.apache.spark.api.java.function.PairFunction;
import scala.Tuple2;

public class JoinedMapper implements PairFunction<Tuple2<Integer, Tuple2<Double, Tuple2<Integer, Double>>>,
        Tuple2<Integer, Integer>, Tuple2<Double, Double>> {

    public Tuple2<Tuple2<Integer, Integer>, Tuple2<Double, Double>>
                call(Tuple2<Integer, Tuple2<Double, Tuple2<Integer, Double>>> x) {
        return new Tuple2(new Tuple2(x._1(), x._2()._2()._1()), new Tuple2(x._2()._2()._2(), x._2()._1()));
    }

}
```

Clase AccumulatorMapper

```
import org.apache.spark.api.java.function.PairFunction;
```

```java
import scala.Tuple2;

public class AccumulatorMapper implements PairFunction<Tuple2<Tuple2<Integer, Integer>, PearsonAccumulator>,
        Tuple2<Integer, Integer>, Double> {

        public Tuple2<Tuple2<Integer, Integer>, Double> call(Tuple2<Tuple2<Integer, Integer>, PearsonAccumulator> x) {
            return new Tuple2(new Tuple2(x._1()._1(), x._1()._2()), x._2().getCoefficient());
        }

}
```

Clase Metrics

```java
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.mllib.evaluation.RegressionMetrics;
import scala.Tuple2;

public class Metrics {

    RegressionMetrics rs;

    protected Metrics (JavaPairRDD<Tuple2<Integer, Integer>, Tuple2<Object, Object>> predictions) {
        rs = new RegressionMetrics(predictions
                .values()
                .mapToPair(x -> new Tuple2(((Double) x._1() / 5.0), ((Double) x._2() / 5.0)))
                .rdd());
    }


    private double mae () {
        return rs.meanAbsoluteError();
    }


    private double rmse () {
        return rs.rootMeanSquaredError();
    }
```

```
    protected String getMetrics() {
        return new String (mae() + "\t" + rmse());
    }

}
```

Anexo 2: código generado para el algoritmo basado en tendencias

Clase Trends

```
import org.apache.spark.HashPartitioner;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.*;
import scala.Tuple2;
import java.util.Arrays;
import java.util.List;


public class Trends {

    // variables de contexto
    static SparkConf conf;
    static public JavaSparkContext sc;


    static public JavaPairRDD<Tuple2<Integer,Integer>, Double> collectData(String path, final Integer maxUser){

        return sc.textFile(path)
            .mapToPair(s -> {
                List<String> separado = Arrays.asList(s.split("::"));
                Integer user = Integer.parseInt(separado.get(0));
                Integer item = Integer.parseInt(separado.get(1));
                Double r = Double.parseDouble(separado.get(2));
                return new Tuple2(new Tuple2(user, item), r);
            }).filter(new Function<Tuple2<Tuple2<Integer, Integer>, Double>, Boolean>() {
                    public Boolean call(Tuple2<Tuple2<Integer, Integer>, Double> x) {
                        return (x._1()._1().compareTo(maxUser) <= 0); // 1000 usuarios
                    }
                });

    }
```

```java
public static void main(String[] args){

    conf= new SparkConf().setAppName("trends");
    sc= new JavaSparkContext(conf);

    String path = new String("hdfs:///movielens/large/ratings.dat");

    Double beta;
    Integer maxUser;

    if (args.length < 2){
        beta = 0.3;
        maxUser = 1000;
    } else {
        beta = Double.parseDouble(args[0]);
        maxUser = Integer.parseDouble(args[1]);
    }

    final JavaPairRDD<Tuple2<Integer, Integer>, Double> ratings = collectData(path, maxUser);

    final long t1 = System.currentTimeMillis();

    final RatingsPredictor predictor = new RatingsPredictor (ratings);
    final JavaPairRDD<Tuple2<Integer, Integer>, Double> predictions = predictor.compute(beta);

    final long size = predictions.count();
    final long t2 = System.currentTimeMillis();
    System.out.println("Ha calculado " + size + " predicciones para " + maxUser + " usuarios con beta = "
            + beta + " en " + ((t2-t1)/1000) + " segundos.");

    final Metrics metrics = new Metrics(predictions
            .join(ratings, new HashPartitioner(7))
            .filter(x -> ((x._1()._1() != null) && (x._1()._2() != null)))
            .mapToPair(x -> new Tuple2(new Tuple2(x._1()._1(),x._1()._2()),
                    new Tuple2(x._2()._1(), x._2()._2()))));
    System.out.println(metrics.getMetrics());
```

```
    }
}
```

Clase RatingsPredictor

```
import org.apache.spark.HashPartitioner;
import org.apache.spark.api.java.JavaPairRDD;
import scala.Tuple2;

public class RatingsPredictor {

    JavaPairRDD<Tuple2<Integer,Integer>, Double> ratings;


    public RatingsPredictor (JavaPairRDD<Tuple2<Integer,Integer>, Double> r) {
        ratings = r.cache();
    }


    protected JavaPairRDD<Tuple2<Integer, Integer>, Double> compute (double beta) {

        // RDDs auxiliares
        JavaPairRDD<Integer, Double> itemMean;
        JavaPairRDD<Integer, Double> userMean;
        JavaPairRDD<Integer, Double> itemTrend;
        JavaPairRDD<Integer, Double> userTrend;

        // <item, mediaItem>
        itemMean = ratings
                .mapToPair(x -> new Tuple2(x._1()._2(), x._2()))
                .combineByKey(Average.createM, Average.addM, Average.combineM, new HashPartitioner(7))
                .mapToPair(new AverageMapper()).cache();
        // <usuario, mediaUsuario>
        userMean = ratings
                .mapToPair(x -> new Tuple2(x._1()._1(), x._2()))
                .combineByKey(Average.createM, Average.addM, Average.combineM, new HashPartitioner(7))
                .mapToPair(new AverageMapper()).cache();
        // <usuario, tendenciaUsuario>
        userTrend = ratings
```

```
            .mapToPair(x -> new Tuple2(x._1()._2(), new Tuple2(x._1()._1(), x._2())))
            .join(itemMean)
            .mapToPair(new ReorganizerMapper()) // <item, <rating, mediaItem>>
            .combineByKey(Average.createT, Average.addT, Average.combineT, new HashPartitioner(7))
            .mapToPair(new AverageMapper());
    // <item, tendenciaItem>
    itemTrend = ratings
            .mapToPair(x -> new Tuple2(x._1()._1(), new Tuple2(x._1()._2(), x._2())))
            .join(userMean, new HashPartitioner(7))
            .mapToPair(new ReorganizerMapper())
            .combineByKey(Average.createT, Average.addT, Average.combineT, new HashPartitioner(7))
            .mapToPair(new AverageMapper());

    JavaPairRDD<Tuple2<Integer, Integer>, Double> predictions = userMean
            .join(userTrend, new HashPartitioner(7))
            .cartesian(itemMean
                    .join(itemTrend, new HashPartitioner(7)))
            .mapToPair(x -> {
                Double p = 0.0;
                Double userT = x._1()._2()._2();
                Double userM = x._1()._2()._1();
                Double itemT = x._2()._2()._2();
                Double itemM = x._2()._2()._1();

                // si (tendenciaUsuario > 0)
                if (userT > 0) {
                    // si (tendenciaItem > 0) entonces p = max (mediaUsuario + tendenciaItem, mediaItem + tendenciaUsuario)
                    if (itemT > 0) {
                        p = Math.max(userM + itemT, itemM + userT);
                    }
                    // si ((tendenciaUsuario <= 0) y (tendenciaItem > 0))
                }else if (itemT > 0) {
                    // si (mediaItem - mediaUsuario > beta)
                    if (itemM - userM > beta) {
                        // p = min (max (mediaUsuario, (mediaItem + tendenciaUsuario) * beta + (mediaUsuario + tendenciaItem)
* (1 - beta)), mediaItem)
                        p = Math.min(Math.max(userM, (itemM + userT) * beta + (userM + itemT) * (1 - beta)), itemM);
                        // si no p = mediaItem * beta + mediaUsuario * (1 - beta)
                    }else {p = itemM * beta + userM * (1 - beta);}
                    // si no p = min (mediaUsuario + tendenciaItem, mediaItem + tendenciaUsuario)
                }else {p = Math.min(userM + itemT, itemM + userT);}
```

```
                    return new Tuple2(new Tuple2(x._1()._1(), x._2()._1()), p);

                });

        return predictions;

    }
}
```

Clase Average

```
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import scala.Tuple2;
import java.io.Serializable;


class Average implements Serializable{

    public Double total_;
    public int num_;

    public Average(Double total, int num) {
        total_ = total;
        num_ = num;
    }

    public Double avgA() {
        return total_ / new Double(num_);
    }

    static Function<Double, Average> createM =
            x -> new Average(x, 1);

    static Function2<Average, Double, Average> addM =
            (a, x) ->{
                a.total_ += x;
                a.num_ += 1;
```

```
                return a;
            };

    static Function2<Average, Average, Average> combineM =
            (a, b) ->{
                a.total_ += b.total_;
                a.num_ += b.num_;
                return a;
            };

    static Function<Tuple2<Double, Double>, Average> createT =
            x -> new Average(x._1() - x._2(), 1);

    static Function2<Average, Tuple2<Double, Double>, Average> addT =
            (a, x) ->{
                a.total_ += (x._1() - x._2());
                a.num_ += 1;
                return a;
            };

    static Function2<Average, Average, Average> combineT =
            (a, b) ->{
                a.total_ += b.total_;
                a.num_ += b.num_;
                return a;
            };

}
```

Clase AverageMapper

```
import org.apache.spark.api.java.function.PairFunction;
import scala.Tuple2;

public class AverageMapper implements PairFunction<Tuple2<Integer, Average>, Integer, Double>{
    public Tuple2<Integer, Double> call(Tuple2<Integer, Average> x) {
        return new Tuple2(x._1(), x._2().avgA());
    }
}
```

Clase ReorganizerMapper

```
import org.apache.spark.api.java.function.PairFunction;
import scala.Tuple2;

public class ReorganizerMapper implements PairFunction<
        Tuple2<Integer, Tuple2<Tuple2<Integer, Double>, Double>>, // <usuario, <<item, rating>, media>>
        Integer, Tuple2<Integer, Double>>{
    public Tuple2<Integer, Tuple2<Integer, Double>> call(Tuple2<Integer, Tuple2<Tuple2<Integer, Double>, Double>> x) {
        return new Tuple2(x._2()._1()._1(), new Tuple2(x._2()._1()._2(), x._2()._2())); // <item, <rating, mediaUsuario>>
    }
}
```

Clase Metrics

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.mllib.evaluation.MulticlassMetrics;
import org.apache.spark.mllib.evaluation.RegressionMetrics;
import scala.Tuple2;


public class Metrics {

    RegressionMetrics rs;

    protected Metrics (JavaPairRDD<Tuple2<Integer, Integer>, Tuple2<Object, Object>> predictions) {
        rs = new RegressionMetrics(predictions.values()
                .mapToPair(x -> new Tuple2(((Double) x._1() / 5.0), ((Double) x._2() / 5.0))).rdd());
    }

    private double mae () {
        return rs.meanAbsoluteError();
    }

    private double rmse () {
        return rs.rootMeanSquaredError();
    }
```

```
    protected String getMetrics() {
        return new String (mae() + "\t" + rmse());
    }

}
```