FACULTAD DE ESTUDIOS ESTADÍSTICOS

MÁSTER EN MINERÍA DE DATOS E INTELIGENCIA DE NEGOCIOS

Curso 2020/2021

Trabajo de Fin de Máster

TITULO: Perspectiva del DoubleDQN en los sistemas de recomendación para resolver el problema de item cold-start

***Alumno*: ZIWEI SHU**

***Tutor*: RAMÓN ALBERTO CARRASCO GONZALEZ**

Junio de 2021

UNIVERSIDAD COMPLUTENSE
MADRID

RESUMEN

Los sistemas de recomendación se utilizan ampliamente en nuestra vida. Por ejemplo, se recomiendan unos productos que le puedan interesar a partir de su historial de compra o visitas (Amazon); unas películas que le puedan gustar según su popularidad (Netflix), etc. De hecho, esos no sólo ayudan a los consumidores a encontrar productos que les puedan interesar, sino que también permiten a las empresas a atraer y retener a más clientes.

Muchas empresas aplican el filtrado colaborativo en sus sistemas de recomendación para proporcionar una lista de elementos potenciales a cada usuario. No obstante, la mayoría de los modelos del filtrado colaborativo no tienen en cuenta los cambios dinámicos de las preferencias de los usuarios y hacen recomendaciones siguiendo una estrategia estática. Otro problema grave del filtrado colaborativo es el arranque en frío (*Cold-start*, en inglés), que ocurre cuando el sistema de recomendación no ha sido capaz de detectar los productos similares debido a la falta de la información de los comportamientos de usuarios o la calificación de productos.

En consecuencia, el objetivo de este trabajo es diseñar un novedoso algoritmo de recomendación basado en el aprendizaje por refuerzo (*Reinforcement Learning*, en inglés), que favorece encontrar automáticamente la estrategia óptima de recomendación mediante las interacciones continúas con los usuarios. Al mismo tiempo, verificaría si éste tuviera capacidad de resolver el problema de *item cold-start y* el efecto temporal de las recomendaciones.

Se han usado 2 conjuntos de datos de *MovieLens*, uno de 610 usuarios que puntuaron 9724 películas en la plataforma *MovieLens* entre el 29 de marzo de 1996 y el 24 de septiembre de 2018 (MovieLensSLD), y el otro de 943 usuarios que puntuaron 1682 películas antes del año 1999 (MovieLens100K). Se han depurado los datos a través de *SAS Miner*, y posteriormente los analizaron con *Python* mediante los algoritmos clásicos de sistemas de recomendación, el algoritmo *Deep Q Network* (en adelante DQN) y *Double Deep Q-Network* (en adelante DoubleDQN) con el fin de comparar sus rendimientos en cada conjunto de dato.

Palabras clave:

Sistema de recomendación, Reinforcement Learning, Deep Q-Network, Double Deep Q-Network

Índice General

Índice de figuras

Índice de tablas

1. Introducción

Los sistemas de recomendación han tenido un fuerte impacto en diferentes áreas como son medicina, entretenimiento, comercio electrónico, educativa y entre otras (Betancur et al., 2009). Su objetivo principal es simplificar el volumen de datos y entregar a los consumidores productos o servicios cercanos a sus necesidades, cuyo origen se remonta a los años 90 (Burke, 2002).

El primer modelo exitoso de los sistemas de recomendación es el filtrado colaborativo (Goldberg et al., 1992), que se puede clasificar en los algoritmos basados en memoria y en modelos (Adomavicius & Tuzhilin, 2005). En realidad, los algoritmos basados en modelos son cada vez más importantes en los sistemas de recomendación, cuya implementación más popular es la factorización matricial (Koren et al., 2009). Otras técnicas exitosas basadas en modelos son: enfoques basados en grafos (Fouss et al., 2007), la descomposición de valor singular (Paterek, 2007), redes bayesianas (Wei & Junliang, 2013), clustering (Hu et al., 2014), SVD++ (Xian et al., 2017), etc.

Sin embargo, el arranque en frío (*Cold-start*, en inglés) todavía es un problema que influye en el rendimiento de los sistemas de recomendación. Este problema ocurre cuando faltan calificaciones de los productos sean nuevos o no; o hay usuarios (nuevos o no) que no han apuntado ningún ratings (Schein et al., 2002). Por ejemplo, la aplicación pura del filtrado colaborativo resulta insuficiente en un entorno de inicio en frío, puesto que no hay ninguna información sobre las preferencias del usuario para formar una base de recomendaciones.

En realidad, hay 3 tipos de *Cold-start* (Bobadilla Sancho et al., 2012): el *Cold-start* de los artículos nuevos, el de los usuarios nuevos y el del sistema recomendador nuevo. Existen varios estudios que han intentado resolver dicho problema utilizando distintos modelos como: Taxonomías de artículos (Weng et al., 2008), Regresión basada en características de usuarios-items (Park & Chu, 2009), Árboles de decisión (Golbandi et al., 2011), Etiquetas sociales (Zhang et al., 2012), Similitud bilineal factorizada basada en características (Sharma et al., 2015), Factorización matricial con bandidos (Mary et al., 2015), Vecinos basados en contenido (Volkovs et al., 2017), Minería de interrelaciones (Zhi-Peng Zhang et al., 2019), Redes neuronales de gráficos de atributos (Qian et al., 2020), etc.

Por otra parte, cabe destacar que la mayoría de los modelos del filtrado colaborativo no tienen en cuenta los cambios dinámicos de las preferencias de los usuarios y hacen recomendaciones siguiendo una estrategia fija, a pesar de que unos estudios (Koren, 2009; Li et al., 2011; Yu & Li, 2015; Lian et al., 2016; Guo et al., 2018) han comprobado el efecto de tiempo en los sistemas de recomendación. Es decir, las preferencias del usuario no son estáticas a lo largo de tiempo, por lo que, para optimizar la decisión de recomendación, no es suficiente recomendarles aquellos productos o servicios en función de sus calificaciones o preferencias históricas.

Para resolver el problema de las preferencias dinámicas, se puede aplicar el

aprendizaje por refuerzo (*Reinforcement Learning*, en adelante RL), dado que, por una parte, puede actualizar continuamente sus estrategias durante las interacciones hasta que genera recomendaciones adecuadas a las preferencias dinámicas de los usuarios; por otra parte, permite identificar el artículo con un *feedback* inmediato, lo cual contribuye a las recomendaciones futuras con más precisión (Zhao et al., 2019). En particular, un estudio ha comprobado que RL puede ser útil para resolver el problema de *Cold-Start* de los sistemas de recomendaciones (Dureddy & Kaden, 2018), aunque hace falta más experimentos. De todas formas, se puede considerar que el entrenamiento de RL es un proceso de agregar nuevos datos constantemente, por lo que se puede usar para resolver el problema de *Cold-Start*.

Es obvio que el RL ha logrado avances impresionantes en juegos y robóticas, pero no se ha adoptado ampliamente en los sistemas de recomendación (Association for Computing Machinery (ACM), 2019).

Por lo tanto, este trabajo va a aplicar DoubleDQN en los sistemas de recomendación tanto para diseñar un novedoso algoritmo de recomendación basado en las preferencias dinámicas de usuarios, como para resolver el problema de *Cold-Start*.

En concreto, se han utilizado 2 conjuntos de datos, uno de 610 usuarios que puntuaron 9724 películas en la plataforma *MovieLens* entre el 29 de marzo de 1996 y el 24 de septiembre de 2018. Otro conjunto utilizado en este trabajo también es de *MovieLens*, conteniendo 100.000 valoraciones de 943 usuarios sobre 1682 películas. Luego, se ha llevado a cabo la validación cruzada de K iteraciones, usando los algoritmos tradicionales de sistemas de recomendación, DQN y DoubleDQN (dos redes neuronales separadas) , con el fin de encontrar el modelo ganador.

Además de esta parte introducción, el resto de este trabajo está estructurado de las siguientes maneras. En la sección 2, se revisa brevemente la historia y los trabajos relacionados de sistemas de recomendación. En la sección 3, se presentan los objetivos principales y específicos de este trabajo. En la sección 4, se introducen los conjuntos de datos, el proceso de depuración y las técnicas aplicadas. En la sección 5, se describen los algoritmos usados para la construcción de modelos. En la sección 6, se definen distintas medidas de evaluación de los modelos. En la sección 7, se demuestran los resultados de la comparación y evaluación final de los distintos algoritmos de sistemas de recomendación. Finalmente, este trabajo termina con una conclusión y los posibles problemas pendientes de resolver como trabajo futuro. Y se incluye una sección de anexos para complementar el contenido de este trabajo (lista de acrónimos, códigos de *Python* y todos los resultados experimentales).

2. Historia y trabajos relacionados con sistema de recomendación

En esta sección, se revisa la historia del sistema de recomendación (en adelante RS). A priori, se presentan brevemente unos trabajos relevantes del problema de *cold-start* en el RS.

El origen del RS se remonta a los años 90, cuya definición ha evolucionado durante los últimos años. Antes se definía como un medio para ayudar y aumentar el proceso social de utilizar las recomendaciones de otros para tomar decisiones cuando no hay suficiente conocimiento personal o experiencia de las alternativas (Resnick & Varian, 1997). Para otros autores, era un sistema que tenía el efecto de guiar al usuario de forma personalizada a objetos interesantes o útiles en un gran espacio de posibles opciones (Burke, 2002). Hoy en día, con el crecimiento explosivo en la cantidad de información digital disponible y el número de visitantes a Internet, los RS son cada vez más parecidos al sistema de filtrado de información que trata el problema de la sobrecarga de información (Konstan & Riedl, 2012), por lo que se puede considerarlos como una parte del sistema de filtrado de información.

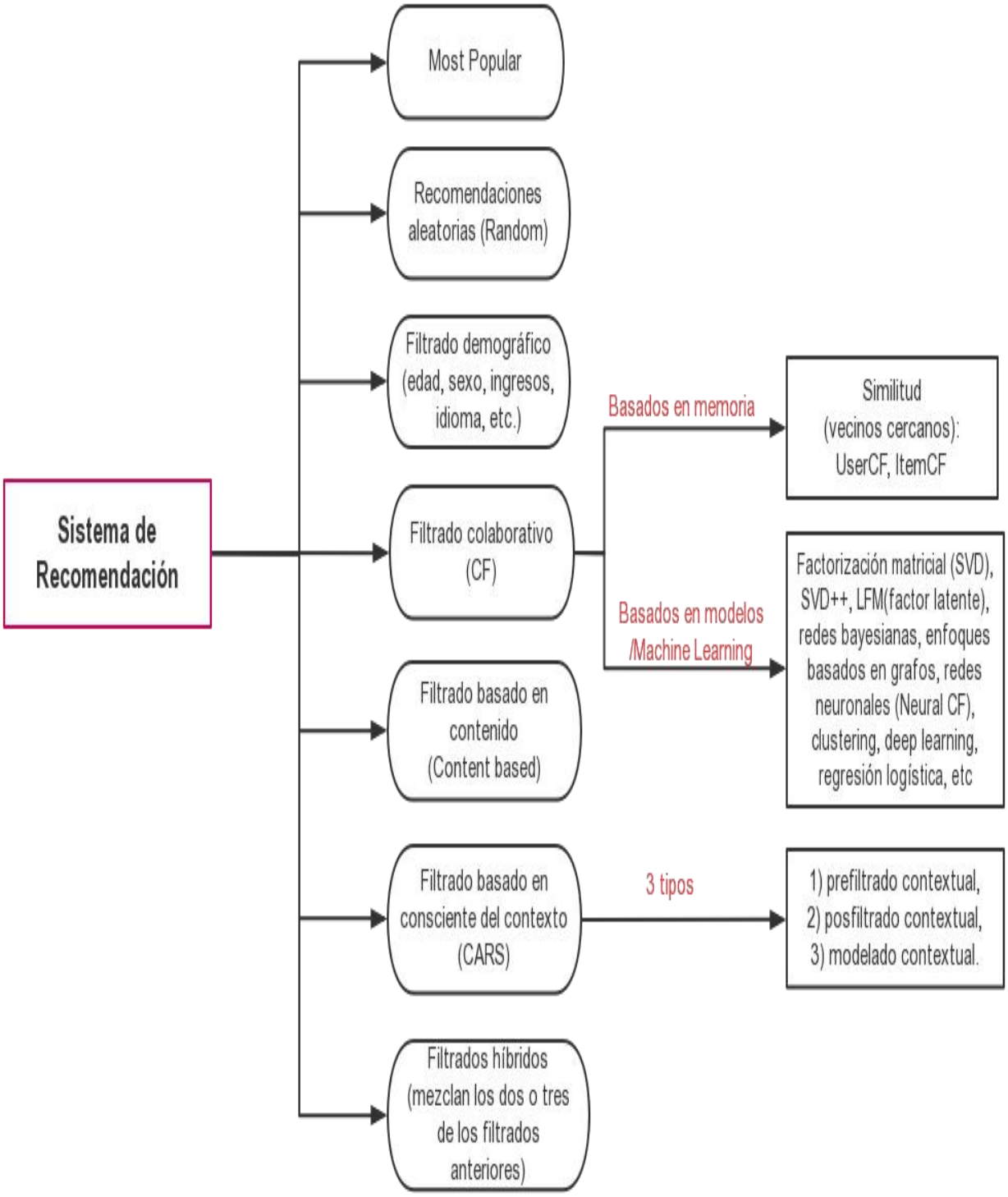
Como se puede observar en la Figura 1, desde mediados de las noventa hasta ahora, han surgido distintos tipos de los RS, de los que son: recomendar los *items* más populares a todos los usuarios por igual sin tener en cuenta sus preferencias distintas (*Most Popular*, en inglés); recomendar aleatoriamente los productos de la plataforma (*Random*, en inglés); o bien, recomendar aquellos productos o servicios de acuerdo con los datos demográficos de los usuarios (filtrado demográfico).

En concreto, uno de los más utilizados en las empresas comerciales es el filtrado colaborativo (Goldberg et al., 1992), que se puede clasificar en los algoritmos basados en memoria y en modelos (Adomavicius & Tuzhilin, 2005). Un ejemplo prominente es Amazon, que usa el filtrado colaborativo basado en la similitud de los artículos, con el propósito de asociar cada producto comprado por un usuario con una lista de productos similares. Según la investigación de *McKinsey*, el 35% de compras realizadas en Amazon provienen de sus sistemas de recomendación (Ian MacKenzie et al, 2013).

También hay otros tipos del filtrado colaborativo (en adelante CF) combinado con las técnicas de *Machine Learning*, como clustering aplicado al CF (Kohrs & Merialdo, 1999; Xue et al., 2005), el CF de las redes bayesianas (Wei & Junliang, 2013), el CF de redes neuronales (Wang et al., 2019), el CF basado en *Deep Learning* (Bobadilla et al., 2020), etc.

Por otro lado, otro modelo básico de RS es el filtrado basado en contenido, que analiza el contenido de los elementos y los recomienda en función de sus atributos de contenido asociados (Balabanovic & Shoham, 1997; Q. Li & Kim, 2003; National Institute of Technology Patna et al., 2017).

Figura 1: Modelos clásicos de sistemas de recomendación



Fuente: Elaboración propia

Sin embargo, la toma de decisiones del consumidor en marketing es variable, que depende del contexto de la toma de decisiones (Ricci et al., 2015). Por eso, durante los últimos años, aparecen más modelos de RS basados en el consciente del contexto (*Context-aware recommender systems*, en inglés), que generan consejos adaptándolos a la situación contextual específica del usuario. Este tipo de RS suele aplicar con las técnicas de *Machine Learning*, por ejemplo, red neuronal recurrente (Beutel et al.,

2018). Por supuesto, también existen muchos modelos de RS que combinan dos o varios modelos de RS para hacer frente a las limitaciones de cada enfoque, los cuales se denominan los filtrados híbridos (Li & Kim, 2003).

A pesar de que ya existen muchos modelos avanzados de RS para superar el problema de la sobrecarga de información y la elección de los *items*, otra dificultad que está enfrentando muchos años en este área es el *Cold-start*, que suele ocurrir cuando faltan calificaciones de los productos sean nuevos o no; o la información de los comportamientos de usuarios sean nuevos o no (Schein et al., 2002).

De hecho, como se puede observar en la Tabla 1, durante los últimos años, existen varios estudios que han intentado resolver dicho problema utilizando distintos modelos. La mayoría de ellos se basan principalmente en aprovechar otras fuentes de información para compensar la falta de datos de calificación, tal como la información de las redes sociales, el contenido de los *items*, información demográfica de usuarios, etc.

Tabla 1: Estudios sobre el problema cold-start en RS

Año	Autores	Técnica
2008	Li-Tung Weng; Yue Xu; Yuefeng Li; Richi Nayak	Taxonomías de artículos
2009	Seung-taek Park; Wei Chu	Regresión basada en características de usuarios-items
2011	Nadav Golbandi; Yehuda Koren; Ronny Lempel	Árboles de decisión
2012	Zi-Ke Zhang; Chuang Liu; Yi-Cheng Zhang; Tao Zhou	Etiquetas sociales
2014	Martin Saveski; Amin Mantrach	Embeddings colectivos locales basado en la factorización matricial
2015	Iman Barjasteh; Rana Forsati; Farzan Masrour; Abdol-Hossein Esfahanian; Hayder Radha	Terminación y transducción desacopladas (DCT) con información auxiliar de usuarios-items
2015	Mohit Sharma; Jiayu Zhou; Junling Hu; George Karypis	Similitud bilineal factorizada basada en características
2015	Jérémie Mary, Romaric Gaudel, Philippe Preux	Factorización matricial con bandidos
2016	Amel Hannech; Mehdi Adda; Hamid Mcheick	Grafo Social
2017	Alan V. Prando; Felipe G. Contratres; Solange N. A. Souza; Luiz S. de Souza	Filtrado basado en contenido de las redes sociales
2017	Maksims Volkovs;Guang Wei Yu;Tomi Poutanen	Vecinos basados en contenido
2018	Hima Varsha Dureddy; Zachary Kaden	*Reinforcement Learning* (DQN)
2019	Zhi-Peng Zhang; Yasuo Kudo; Tetsuya Murai; Yong-Gong Ren	Minería de interrelaciones
2020	Tieyun Qian; Yile Liang; Qing Li	Redes neuronales de gráficos de atributos (AGNN)

Fuente: Elaboración propia basada en los artículos de ACM.org, arXiv.org, Hal.archives, IEEE.org, MDPI.com, Scitepress.org

Según la Tabla 1, es obvio que la mayoría de los autores usan los métodos clásicos de RS o de *Machine Learing* para resolver el problema de *Cold-start*, apenas usan el RL. De hecho, a diferencia de los modelos clásicos de recomendaciones, el RL no se ha adoptado ampliamente en los sistemas de recomendación (Association for Computing Machinery, 2019).

Sin embargo, como el RL se trata de un proceso de aprendizaje constante y automático sobre los comportamientos de consumidores, un estudio lo aplicó para observar las respuestas de los usuarios en un recomendador interactual, así como maximizar el *feedback* acumulativo de los usuarios obteniéndolo en cada sesión de recomendación (Mahmood & Ricci, 2009). Últimamente, unos investigadores (Zhao, Xia, et al., 2018) han propuesto un marco de recomendaciones novedoso por página basado en el RL, que permite optimizar una página de elementos con una visualización adecuada en función de los comentarios de los usuarios en tiempo real.

Por consiguiente, este artículo va a usar el RL en los sistemas de recomendación, con el fin de resolver el problema de *item cold-sta*rt y aprender las preferencias dinámicas de usuarios. En concreto, se utiliza DoubleDQN para reducir el problema sobre las sobreestimaciones de la predicción Q en los algoritmos *Q-Leanring* y DQN.

3. Objetivos

En esta sección, se presentan los objetivos principales y específicos de este trabajo.

El objetivo principal es proponer un novedoso algoritmo de recomendación basado en el *DoubleDQN,* mediante lo cual intentar solucionar el problema de *item cold-start y* el efecto temporal de las recomendaciones.

Los objetivos específicos de este trabajo son los siguientes:

1) Realizar un análisis exploratorio en SAS Miner sobre los 2 conjuntos de datos, con el fin de llevar a cabo su depuración.

2) Evaluar los modelos clásicos de los sistemas de recomendación para conocer sus ventajas y desventajas.

3) Identificar las preferencias futuras de los usuarios según las puntuaciones de elementos y el efecto temporal.

4) Mejorar la novedad de las recomendaciones.

5) Ensamblar todos los algoritmos utilizados para mejorar su rendimiento.

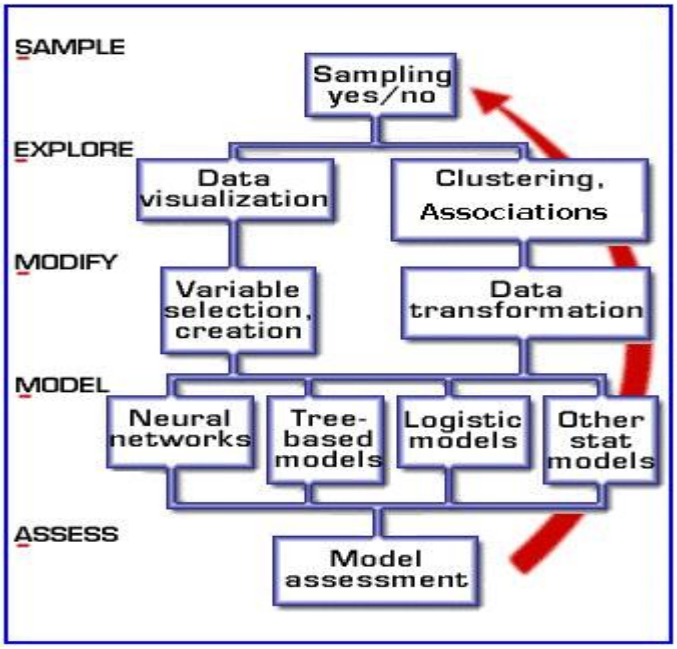
4. Metodología y datos

En esta sección, se introducen las técnicas aplicadas, los conjuntos de datos utilizados en este trabajo, así como su proceso de depuración.

4.1. Metodología de SEMMA

SEMMA es una metodología propuesta por el instituto SAS, que se define como un proceso de muestrear (Sample), explorar (Explore), modificar (Modify), Modelar (Model) y evaluar (Assess) grandes cantidades de datos para descubrir patrones desconocidos que pueden ser utilizados como una ventaja competitiva (*SAS Help Center*, 2017).

Figura 2: Proceso de la metodología SEMMA



Fuente: SAS Help Center

En concreto, como se puede ver en la Figura 2, la metodología SEMMA contiene 5 etapas:

1) ***S**ample* (muestrear): si la base de datos es demasiado grande, sería necesario tomar una muestra lo suficientemente grande para contener toda la información y lo suficientemente pequeña para poder ser procesada. Es decir, ha de seleccionar una muestra representativa del problema del estudio.

2) ***E**xplorar* (explorar): detectar relaciones anticipadas, tendencias inesperadas y anomalías para obtener comprensión e ideas.

3) ***M**odify* (modificar): realizar la depuración de datos, tal como tratamientos de los datos atípicos y datos ausentes, la transformación de las variables para facilitar la modelización.

4) ***M**odel* (modelar): modelar los datos con distintos modelos para encontrar el modelo que prediga mejor la variable objetivo.

5) ***Assess*** (evaluar): comprobar la calidad de las predicciones y comparar los modelos obtenidos a través de las medidas oportunas. Con respecto a las métricas de evaluación, se puede consultar en la sección 6.

De hecho, no hay por qué incluir todos los pasos en el análisis, sobre todo, tanto para el conjunto de datos *MovielensSLD*, como para el *Movielens100K*, no incluye la fase de muestrear al no tener una base de datos demasiado grande.

4.2. Conjuntos de datos

4.2.1. Introducción y descripción de las variables

Como se ha mencionado anteriormente, en este trabajo, se han utilizado dos conjuntos de datos:

4.2.1.1. MovieLensSLD (https://grouplens.org/datasets/movielens/latest/):

Es una muestra de 610 usuarios que puntuaron 9724 películas en la plataforma *MovieLens* entre el 29 de marzo de 1996 y el 24 de septiembre de 2018. Contiene un total de 100.836 calificaciones sobre 9724 películas, y todos los usuarios seleccionados habían calificado al menos 20 películas.

El objetivo es construir un modelo que permite resolver el problema de *item cold-sta*rt de los sistemas de recomendación. Las variables de este conjunto de datos son:

- Variables cuantitativas: Rating (0,5 estrellas - 5,0 estrellas), AñoPublicación, ComentarioTime
- Variables cualitativas: UserID, MovieID, MovieName, MovieTags

4.2.1.2. MovieLens100K (https://grouplens.org/datasets/movielens/100k/):

El conjunto de *MovieLens100K* contiene un total de 100.000 calificaciones sobre 1682 películas desde el año 1930 hasta 1998. Es una muestra de 943 usuarios y todos los usuarios seleccionados habían calificado al menos 20 películas. Este conjunto de datos consta de 100.000 observaciones, cuyas variables implicadas son las mismas que el conjunto de datos anterior explicado, salvo la variable Rating (1 - 5).

4.2.2. Modificación de la base de datos.

En el apartado 4.1, se ha comentado que se usa la metodología SEMMA en este trabajo, por lo que se ha aplicado *SAS ENTERPRISE MINER* para llevar a cabo la depuración de datos por separado para cada dataset.

4.2.2.1. MovieLensSLD

En *SAS Miner*, se ha realizado el proceso de depuración del conjunto de *MovieLensSLD* siguiendo los siguientes pasos:

1) Usar el nodo *Explorador de Estadísticos* para analizar los gráficos y estadísticos del conjunto de datos, así como detectar errores en las variables.

En este caso, hay un error en la variable de intervalo (AñoPublicación), que tiene un

mínimo igual que 2. No obstante, este conjunto contiene las películas desde 1902 hasta 2018, por lo que el límite inferior de esta variable debe ser de 1902.

2) Añadir el nodo *Reemplazo* para corregir el error detectado previamente.

En su ventana de propiedades: *"Método de límites predeterminado"* > *"Ninguno"* para asegurarnos de que sólo se apliquen las correcciones que indiquemos en el editor. Luego, pinchar *¨Variables de intervalo¨>¨Editor de reemplazo¨:*

- *¨Método de límite¨>¨Especificado por el usuario¨>¨Límite inferior del reemplazo¨>1902;*
- *¨Método de sustitución¨>¨Ausente¨.*

3) Añadir otro nodo *Explorador de Estadísticos* para verificar si algunas variables tienen datos de asimetría. En este conjunto, no hay datos atípicos.

4) Usar el nodo *Código SAS* para crear la variable numMissing**,** y añadir un nodo *DMDB* para ver el máximo de numMissing, así como verificar si existe alguna variable con demasiados datos faltantes.

En este caso, no hace falta eliminar ninguna observación, porque el máximo de numMissing es de 1, significa que una fila puede tener como máximo 1 missings, que es inferior al 50% de las variables.

Por otro lado, no hace falta recategorizar los missings como una categoría válida, porque no hay ninguna variable de clase que tiene datos ausentes. No obstante, es necesario hacer la imputación sobre la variable REP_AñoPublicación, puesto que tiene un porcentaje de missings entre el 5% y 50%.

Por lo tanto, hay que añadir un nodo *Imputar* (*¨Método de imputación predeterminado¨>¨Distribución¨*) para no quedar ningún dato ausente cuando llegue a la fase de modelización. Y en esta situación, como ninguna variable supera 5% del porcentaje de missings, no es necesario crear las variables indicadoras (*¨Variables de indicador¨>¨Tipo¨> ¨Ninguno¨*)

5) Añadir el nodo *Guardar datos* para guardar los datos en la librería para cargarlos luego en *Python*.

4.2.2.2. MovieLens100K

En este conjunto de dato se ha seguido el mismo proceso de depuración como el del conjunto de *MovieLensSLD*. No obstante, como este conjunto se ha sido limpiado por el contribuidor, no es necesario modificar nada de este conjunto, y es suficiente simplemente usar el nodo *Explorador de Estadísticos* para ver los gráficos y estadísticos del conjunto de datos, así como confirmar que no hay ningún error en este conjunto.

Después de la depuración de datos en *SAS Miner*, se han importado estos 2 datasets por separado en *Python* y hacer estadística descriptiva a través de la librería *Pandas*, cuyo resultado ha sido resumido en la Tabla 2:

Tabla 2: Tabla de datos de MovieLens

	MovieLensSLD	MovieLens100K
Número de usuarios	610	943
Número de películas	9724	1682
Número media de películas calificadas por usuario	165	106
Número de género cinematográfico	20	19
Número de rating	100.836	100.000
Rango de rating	0,5-5	1-5
Número de rating por usuario	165	106
Número de rating por película	10	59
Tasa de escasez de valoración	98,35%	93,71%

Fuente: Elaboración propia

Cabe destacar que, como se puede observar en la tabla anterior, a pesar de que los usuarios seleccionados en estos dos conjuntos habían calificado al menos 20 películas, la tasa de escasez de valoración (*Rating Sparsity,* en inglés) es muy alto. En realidad, esta tasa de los RS suele ser de hasta el 99% (Guibing Guo, 2012), por lo que es válidos usar estos dos conjuntos para experimentar con los distintos algoritmos de sistemas de recomendación.

5. Técnicas de modelado

En esta sección, lo primero, se presentan los modelos clásicos de RS. Luego, se exponen brevemente los conceptos básicos de RL y el proceso de aplicación de DoubleDQN en el RS. Por último, se construyen los modelos ensamblados mediante la combinación de distintos modelos, con el propósito de conseguir una sinergia positiva mezclando los algoritmos.

5.1. Modelos clásicos de RS

Este apartado trata de probar distintos modelos clásicos de RS, investigando sus características y conocer las diferencias entre ellos. Al mismo tiempo, se han revisado los modelos más adecuados para resolver el problema de *item cold-start*.

Con respecto a la modelización de los modelos clásicos en *Python*, se ha aplicado principalmente el paquete *Surprise,* para crear y analizar sistemas de recomendación que se ocupan de datos de clasificación explícitos. Se puede consultar más detalles sobre los códigos de algoritmos en Anexo II.

5.1.1. Random

El algoritmo Random consiste en recomendar los artículos a los usuarios aleatoriamente, basando en la distribución del conjunto de datos de entrenamiento, que se supone que es normal.

$$\hat{\mu} = \frac{1}{|R_{train}|} \sum_{r_{ui} \in R_{train}} r_{ui}$$

$$\hat{\sigma} = \sqrt{\sum_{r_{ui} \in R_{train}} r_{ui} \frac{(r_{ui} - \hat{\mu})^2}{|R_{train}|}}$$

donde r_{ui} es generado por una distribución normal $N(\hat{\mu}, \hat{\sigma}^2)$; $\hat{\mu}$ y $\hat{\sigma}$ son estimados a partir de los datos de entrenamiento utilizando la estimación de máxima verosimilitud (*Maximum Likelihood Estimation*, en inglés).

5.1.2. Popularidad

Existe varios algoritmos de RS basados en la popularidad de los artículos. En este trabajo, solo se han aplicado dos: *MostPopular* y *RecentPopular*.

El algoritmo *MostPopular* consiste en recomendar los artículos más populares de la historia a los usuarios por igual sin tener en cuenta sus preferencias distintas. En este trabajo, se recomiendan 10 películas más famosas a los usuarios.

De hecho, igual que el algoritmo *MostPopular*, el algoritmo *RecentPopular* no considera las preferencias ni la información de los usuarios, solo que éste recomienda los 10 artículos más populares recientemente.

Por lo tanto, solo hay una pequeña diferencia entre sus fórmulas para calcular la popularidad de los productos, que se puede observar de la siguiente manera:

$$n_i(T) = \sum_{(u,i,t)\in Train,\ t<T} \frac{1}{1+\alpha(-t)} \ ,cuando\ \ t > T = 0(MostPopula\ r)$$

$$n_i(T) = \sum_{(u,i,t)\in Train,\ t<T} \frac{1}{1+\alpha(T-t)} \ ,cuando\ \ T > t(\mathrm{Re}\, centPopula\ r)$$

donde *u* es el usuario, *i* es el artículo, *t* es el tiempo (*timestamp*, en inglés), α es el parámetro de pérdida de tiempo (*time loss*, en inglés), T es un momento prefijado, para *RecentPopular*, T=10.000. Y cuando T=0, este algoritmo funciona igual como el *MostPopular*.

Con los siguientes códigos de *Python*, se lleva a cabo el algoritmo de popularidad tanto para el *MostPopular* como para el *RecentPopular*:

```
alpha = 0.5
T = 0       #MostPopular
T = 10000   #RecentPopular
N = 10
def Popularity(self, records):
    result = dict()
    amount = 0
    count = 0
    for user, item, _, tm in records:
        if int(tm)/(60*24*60) <= T:
            continue
        popu = 1/(1.0+alpha*(int(tm)-T)*86400)*1000000000000000
        if user in result.keys():
            result[user][item] = popu
        else:
            result.setdefault(user, {})
            result[user][item] = popu
        amount += popu
        count += 1
    self.ave = amount/count
    return result
def predict(self, rank, user):
    rankList = rank[user]
    sortedRank = sorted(rankList,key=lambda x:(x[1]),reverse = True)
    print('Recommend {} movies for user '.format(N) + user)
    for i in range(N):
        print(sortedRank[i][0])
```

Comentario:

1) N=10 se refiere a las 10 películas a recomendar.
2) En la fórmula, α es un coeficiente negativo, pero aquí se pone positivo, porque ha sido modificado el orden (T-t) en (t-T) en la parte de código.
3) T es un parámetro importante para realizar distinta función del algoritmo popularidad, cuando T=0, es el algoritmo *MostPopular*; en cambio, si T≠0, es el *RecentPopular*.
4) Se puede consultar más códigos detalladamente en el Anexo II.

En realidad, este algoritmo aprovecha la mentalidad de manada, suponiendo que todos los usuarios siguen ciegamente lo que hacen otras personas. *Twitter Trends* e *Instagram Trends* son productos representativos del algoritmo popularidad, cuya eficacia y precisión suele ser mejor que el algoritmo *Random*.

5.1.3. Filtrado Colaborativo (CF)

Como se ha comentado en la parte introducción, el primer modelo exitoso de los sistemas de recomendación es el filtrado colaborativo (Goldberg et al., 1992), que son los recomendadores más usados. Consiste en estimar las valoraciones de productos usando la opinión o *rating* de distintas personas (Schafer et al., 2007) , emparejando aquellos productos con valoraciones similares o aquellos usuarios con gustos parecidos.

En principio, se puede clasificar en los algoritmos basados en memoria (o vecinos) y en modelos (Adomavicius & Tuzhilin, 2005)

1) **Algoritmos basados en memoria:** aplican métricas de similitud para determinar el parecido entre los usuarios o artículos.

2) **Algoritmos basados en modelos:** utilizan la matriz de *rating* para crear un modelo a través del cual establecer el conjunto de usuarios similares al usuario activo. Por ejemplo, la descomposición matricial basada en la técnica matemática del SVD, SVD++, las redes neuronales, etc.

En este apartado, se enfoca principalmente en los algoritmos basados en memoria, y se puede consultar los basados en modelos en los apartados posteriores (5.1.5, 5.1.6, 5.1.7).

De hecho, dentro de los basados en vecinos hay 2 subconjuntos importantes: el filtrado colaborativo basado en usuarios (*User Based collaborative filtering*, en inglés) y el basado en artículos (*Item Based collaborative filtering*, en inglés). Además, se investiga también el filtrado colaborativo basado en artículos con normalización (en adelante *ItemCF-Norm*) y el filtrado colaborativo basado en artículos con el efecto temporal (en adelante *TItemCF*).

5.1.3.1. Filtrado colaborativo basado en usuarios

El filtrado colaborativo basado en usuarios (en adelante *UserCF*), se refiere a predecir la valoración de un usuario sobre un producto a través de la similitud entre los usuarios, considerando las puntuaciones que aquellos usuarios han dado sobre los mismos productos.

Por ejemplo, si dos usuarios *u* y *v* tienen la misma opinión hacia ciertas películas, es muy probable que tengan una preferencia semejante, por lo que uno de los usuarios podría tener la misma valoración como el otro sobre las películas que no haya visto.

Como su objetivo es los usuarios con los gustos similares, se aplica el *KNNBasic (*K

vecinos más cercanos*)* del paquete *Surprise,* cuya fórmula para estimar la valoración es la siguiente:

$$\hat{r}_{ui} = \frac{\sum_{v \in N_i^k(u)} sim\ (u, v) \cdot r_{vi}}{\sum_{v \in N_i^k(u)} sim\ (u, v)}$$

donde r_{vi} es la puntuación dada por el usuario *v* a los *items;* $N_i^k(u)$ representa el conjunto de *k* usuarios más similares al usuario actual *u* sobre la valoración de *items; k* es el número de vecinos para la agregación, en este trabajo, k=10.

Con respecto a la similitud entre el usuario *u* y el *v*, se determina por la similitud coseno:

$$sim(u,v) = \cos(\theta) = \frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}} = \frac{\sum_{i=1}^{k} u_i v_i}{\sqrt{\left(\sum_{i=1}^{k} u_i^{\,2}\right)\left(\sum_{i=1}^{k} v_i^{\,2}\right)}}$$

donde $N(u)$ es el conjunto de *items* que el usuario *u* ha puntuado y $N(v)$ es el conjunto de *items* que el usuario *v* ha puntuado.

Por supuesto, además de la similitud de coseno, para determinar la similitud, podría usar otras medidas de similaridad como coeficiente de similitud de *Tanimoto* (*Jaccard*), similitud basada en *Log-likelihood*, coeficiente de correlación de Pearson restringido (CPCC), similitud de *Proximity-Impact-Popularity* (PIP), etc. Se compararán las diferencias de los resultados de los algoritmos sobre usar distintas medidas de similaridad en un trabajo futuro.

5.1.3.2. Filtrado colaborativo basado en artículos

El filtrado colaborativo basado en artículos (en adelante *ItemCF*), se refiere a predecir la valoración que un usuario dará sobre un producto estudiando la similitud entre los *items* (el usuario que ha visto la película Y, también vería Z)*.*

A diferencia de buscar los usuarios más vecinos, el *ItemCF* se centra en emparejar las películas vistas y valoradas por el usuario con otras películas, y posteriormente formular una lista de recomendación en la que contiene las películas con estimación de la valoración más alta, lo cual es el motivo que Amazon inventó este algoritmo en 1998.

No obstante, hay que enfatizar que el *ItemCF* no utiliza los atributos de los productos para calcular la similitud, sino las calificaciones dadas por el usuario, lo cual es la principal diferencia entre el *ItemCF* y el algoritmo *ContentItemKNN.* Sobre sus diferencias, se puede consultar más detalles en el apartado 5.1.4.

Como su objetivo es buscar *items* parecidos al *item i*, se aplica el *KNNBasic (*K

vecinos más cercanos*)* del paquete *Surprise,* cuya fórmula para estimar la valoración es la siguiente:

$$\hat{r}_{ui} = \frac{\sum_{j \in N_u^k(i)} sim\ (i, j) \cdot r_{uj}}{\sum_{j \in N_u^k(i)} sim\ (i, j)}$$

donde r_{uj} es la puntuación dada por el usuario *u* al artículo *j;* $N_u^k(i)$ representa el conjunto de *k* artículos más similares al usuario *u* sobre la valoración de *items; k* es el número de vecinos para la agregación, en este trabajo, k=10.

Con respecto a la similitud entre el artículo *i* y el *j*, se determina por la similitud coseno:

$$sim(i,j) = \cos(\theta) = \frac{|N(i) \cap N(j)|}{\sqrt{|N(i)||N(j)|}} = \frac{\sum_{u=1}^{k} i_u j_u}{\sqrt{\left(\sum_{u=1}^{k} i_u{}^2\right)\left(\sum_{u=1}^{k} j_u{}^2\right)}}$$

donde $N(i)$ es las calificaciones dadas sobre *item i*, y $N(j)$ es las calificaciones dadas sobre *item j*.

Sin embargo, tanto el *UserCF*, como el *ItemCF*, calcula la similitud entre los usuarios o los *items* a partir de las puntuaciones calificadas, por lo que puede tener el problema del *Cold-start* cuando muy pocos usuarios generan valoraciones o muy pocos productos han sido calificados. Por ejemplo, cuando faltan calificaciones de los productos nuevos, hasta que éstos no hayan sido valorados, no se podrá asociarlos a una vecindad ni es posible incluirlo dentro de una lista de recomendaciones,a no ser que combine con otros algoritmos de RS para resolver este problema.

En realidad, se define el problema del arranque en frío de los productos como *item cold-start*, y el de los usuarios como *user cold-start*. El *UserCF* es más sensible cuando hay usuarios que no comportan nada en el sistema (*user cold-start*), mientras que el *ItemCF* es más sensible ante el problema de la falta de calificaciones de productos (*item cold-start*).

Además de lo anterior, existen más diferencias entre el *UserCF* y el *ItemCF*. Según los principios previamente mencionados, la recomendación de *UserCF* se centra en conectar el interés común de un grupo de usuarios similares, por lo que su recomendación suele ser más social y tener mayor nivel de sorpresa que el *ItemCF*. Por ejemplo, se supone que Ana solamente haya visto las películas de comedia y que tenga la misma valoración como Pedro, así que se puede considerar que son usuarios similares. Entonces, se puede recomendar a Ana aquellas películas que le han gustado a Pedro.

Al cambio, el *ItemCF* recomienda al usuario aquellos productos que sean similares teniendo en cuenta sus propios gustos, por lo que teóricamente, suele ser una recomendación más personalizada con nivel de sorpresa bajo.

Se han resumido en la Tabla 3 las principales diferencias entre el *UserCF* y el *ItemCF*.

Tabla 3: Comparación entre UserCF y ItemCF

	UserCF	**ItemCF**
Característica de recomendación	Más social	Más personalizado
Situación ideal de aplicación	- Resulta más adecuado aplicar en cuanto el número de usuarios es significativamente menor que el número de productos, porque es costoso calcular la matriz de similitud de usuarios si hay demasiados usuarios. - Bajo nivel de los intereses personalizados de los usuarios.	- Resulta más adecuado aplicar en cuanto el número de productos es significativamente menor que el número de usuarios, porque es costoso calcular la matriz de similitud de productos si hay demasiados productos. - Medio-alto nivel de los intereses personalizados de los usuarios.
Tiempo real (real- time)	No actualiza los resultados de la recomendación inmediatamente, aunque el usuario tiene nuevos comportamientos.	Actualiza los resultados de la recomendación en tiempo real cuando el usuario tiene nuevos comportamientos.
Interpretabilidad sobre la recomendación	Es difícil de interpretar la causa de recomendación por la política de protección de datos.	Se interpreta el motivo de la recomendación en función del comportamiento histórico del usuario.
Nivel de sorpresa	Medio-alto	Bajo

Fuente: Elaboración propia

En definitiva, aunque tienen sus limitaciones, ambos son buenos algoritmos, que se aplica en función de las necesidades de las empresas. En los casos en que el número de usuarios es mucho mayor que el número de productos, como *Amazon*, *Aliexpress*, etc, el *ItemCF* puede producir recomendaciones más precisas (Fouss et al., 2007) porque la calificación promedio de un producto en general no cambia rápidamente, cuyo coste de calcular la matriz de similitud de *items* es menor que calcular la de usuarios.

5.1.3.3. Filtrado colaborativo basado en artículos con normalización

El filtrado colaborativo basado en artículos con normalización (en adelante *ItemCF-Norm*) consiste en normalizar la similitud entre los *items*. De hecho, *Karypis* (Karypis, 2001) descubrió en su investigación que, si la matriz de similitud de *ItemCF* se normaliza al valor máximo, puede mejorar la precisión de las recomendaciones. Se puede usar la fórmula siguiente para normalizar la similitud:

$$sim(i,j) = \cos(\theta) = \frac{|N(i) \cap N(j)|}{\sqrt{|N(i)||N(j)|}} = \frac{\sum_{i=1}^{k} i_u j_u}{\sqrt{\left(\sum_{i=1}^{k} {i_u}^2\right)\left(\sum_{i=1}^{k} {j_u}^2\right)}} \quad (ItemCF)$$

$$\longrightarrow sim(i,j)' = \frac{sim(i,j)}{\max_j sim(i,j)} \quad (ItemCF - Norm)$$

5.1.3.4. Filtrado colaborativo basado en artículos con el efecto temporal

El filtrado colaborativo basado en artículos con el efecto temporal (en adelante *TItemCF*) se refiere a agregar restricciones de tiempo para mejorar el cálculo de la similitud de los artículos:

$$sim(i,j) = \cos(\theta) = \frac{|N(i) \cap N(j)|}{\sqrt{|N(i)||N(j)|}} = \frac{\sum_{i=1}^{k} i_u j_u}{\sqrt{\left(\sum_{i=1}^{k} {i_u}^2\right)\left(\sum_{i=1}^{k} {j_u}^2\right)}} \quad (ItemCF)$$

$$\longrightarrow sim(i,j)' = \frac{\sum_{u \in N(i) \cap N(j)} f\left(\left|t_{ui} - t_{uj}\right|\right)}{\sqrt{|N(i)||N(j)|}} \quad (TItemCF)$$

donde $f\left(\left|t_{ui} - t_{uj}\right|\right)$ es una función de decaimiento temporal; t_{ui} es el tiempo en el que el usuario *u* produce un comportamiento sobre *item i* y t_{uj} es el tiempo en el que el usuario *u* produce un comportamiento sobre *item j*. Cuando mayor sea la distancia entre t_{ui} y t_{uj}, menor será $f\left(\left|t_{ui} - t_{uj}\right|\right)$

Existe muchas funciones de decaimiento, en este trabajo, se usa la siguiente:

$$f\left(\left|t_{ui} - t_{uj}\right|\right) = \frac{1}{1 + \alpha\left|t_{ui} - t_{uj}\right|}$$

donde *α* es el factor de decaimiento temporal, su valor depende del tipo de sistema de recomendación. Si el ciclo de vida de los productos es relativamente corto, por ejemplo, las noticias de los periódicos, se suele poner un α grande. Al contrario, si es relativamente largo, como en este trabajo, el de las películas es medio-largo, se puede poner un α pequeño, como 0,1.

5.1.4. Sistema de recomendación basado en contenido

El sistema de recomendación basado en contenido (en adelante *ContentItemKNN*) se orienta en torno de calcular la similitud de los atributos de los *items* (palabras claves, el género, año, autor, etc), con el propósito de recomendar al usuario un item *j* similar al *item i* que le ha gustado o comprado.

En este trabajo, se consideran principalmente dos atributos de películas: el género cinematográfico y su año de publicación. En primer lugar, se determina la similitud del género de las películas basando en la fórmula de la similitud coseno:

$$S_{género\ ij} = \frac{\sum_{x \in A_i, y \in A_j} xy}{\sqrt{\left(\sum_{x \in A_i} x^2\right)\left(\sum_{y \in A_j} y^2\right)}}$$

donde $S_{género\ ij}$ representa la similitud del género entre la película *i* y la *j;* A_i es el conjunto de género cinematográfico al que la película *i* pertenece*,* mientras que A_j es el conjunto de género cinematográfico al que la película *j* pertenece, siendo *x* y *y* géneros de cada conjunto.

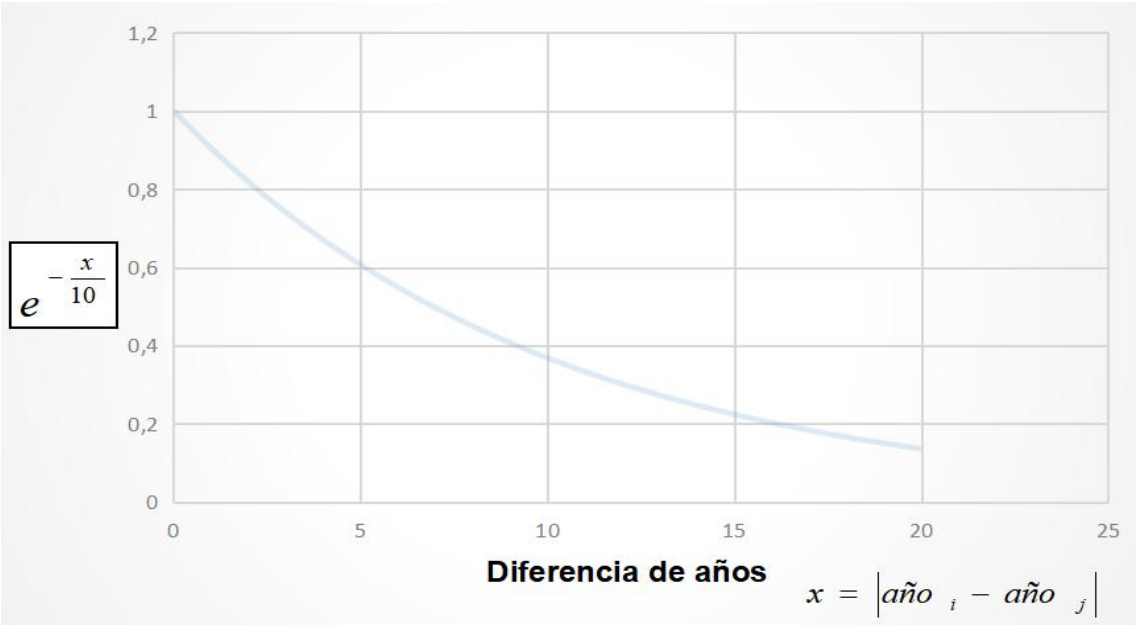
En segundo lugar, basando en la función exponencial natural, se calcula la similitud del año de publicación entre las películas:

$$S_{año\ ij} = e^{-\frac{\left|año_i - año_j\right|}{10}}$$

donde $S_{año\ ij}$ representa la similitud del año de publicación entre la película *i* y la *j* ; $año_i$ es el año de publicación de la película *i* y $año_j$ es el de la película *j* .

En realidad, dos películas pueden ser muy similares en su género, por ejemplo, ambos son de la comedia: Lluvia de dólares (*Brewster's Millions,* 1985) y *Snatch*: Cerdos y diamantes (2000). No obstante, su año de publicación es distinto, por lo que hay una posibilidad de que el proceso de creación de una película sea distinto tal como las técnicas cinematográficas, los estilos de guion, etc. Es decir, sería necesario incluir este factor en el cálculo de similitud. Como se puede ver en la Figura 3, cuando mayor sea la diferencia en el año de producción, disminuirá la similitud del año entre las películas.

Figura 3: Similitud del año de publicación entre las películas



Fuente: Elaboración propia

Por último, se multiplica la similitud del género y la del año de publicación para obtener la similitud entre las películas S_{ij}:

$$S_{ij} = S_{género\ ij}\ x\ S_{año\ ij}$$

$$= \frac{\sum_{x \in A_i, y \in A_j,} xy}{\sqrt{\left(\sum_{x \in A_i} x^2\right)\left(\sum_{y \in A_j} y^2\right)}} \cdot e^{-\frac{|año_i - año_j|}{10}}$$

Con los siguientes códigos de *Python*, se ha llevado a cabo la determinación de la similitud de los atributos:

```
def computeGenreSimilarity(self, movie1, movie2, genres):
    genres1 = genres[movie1]
    genres2 = genres[movie2]
    sumxx, sumxy, sumyy = 0, 0, 0
    for i in range(len(genres1)):
        x = genres1[i]
        y = genres2[i]
        sumxx += x * x
        sumyy += y * y
        sumxy += x * y
    return sumxy/math.sqrt(sumxx*sumyy)
```

```
def computeYearSimilarity(self, movie1, movie2, years):
    diff = abs(years[movie1] - years[movie2])
    sim = math.exp(-diff / 10.0)
    return sim
```

Comentario:

Después de calcular la similitud, se ha usado el algoritmo KNN (k=40) para recomendar aquellas películas que sean más parecidas. Se puede consultar más detalles en el Anexo II.

De hecho, el algoritmo *ContentItemKNN* supone que un usuario desee consumir *ítems* de géneros parecidos, permitiendo recomendar aquellos productos nuevos que no han sido puntuados por ningún usuario. Y su recomendación es interpretable al depender solo de los gustos del usuario. Es decir, no tiene problema de *item cold-start* como el algoritmo *ItemCF.* La principal diferencia entre el *ContentItemKNN* y el *ItemCF* es que, uno calcula la similitud de *items* a partir de sus atributos (*ContentItemKNN*), que suele recomendar cosas de la misma área; mientras que el otro la calcula a partir de sus calificaciones (*ItemCF*), que puede recomendar algo diferente desde el punto de vista de sus contenidos.

Sin embargo, hay que destacar que el análisis de contenido tiene limitación como: la polisemia y/o sinónimos de las palabras; extracción del contenido de audio o video, etc. Sobre todo, el algoritmo *ContentItemKNN* no es capaz de encontrar algo inesperado o sorprendente. En teórica, se denomina como sobreespecialización (*over-specialization*, en inglés), dado que siempre recomiendan al usuario los productos muy similares a los ya consumidos o vistos.

Por otro lado, es verdad que el *ContentItemKNN* mejora la eficacia de sistemas de recomendación cuando aparecen nuevos productos. No obstante, igual como los algoritmos del filtrado colaborativo, no es muy adecuado para productos muy personalizados o productos con los que los usuarios no suelen interactuar, por ejemplo, productos de lujo. Además, cuando los productos son complejos de describir, es más difícil de encontrar la relación entre los productos o con el perfil del usuario (Aggarwal, 2016)

Por lo tanto, vale la pena usar la técnica *Stacking* para ensamblar distintos modelos de sistemas de recomendación, combinando las ventajas de cada uno con el fin de saber si fuera más útil para resolver el problema planteado. Con respecto a este aspecto, se puede consultar más detalles en el apartado 5.3.

5.1.5. SVD

El algoritmo descomposición singular de valores (*Singular Value Descomposition*, en adelante SVD) fue popularizado por *Simon Funk* en Desafío de premios de *Netflix,* basando en la factorización matricial, que consiste en descomponer una matriz A en otras matrices U, S, V. En concreto, la descomposición se lleva a cabo como:

$$SVD(A)_{nxm} = U_{nxn} x S_{nxm} x V^{t}_{mxm}$$

donde la matriz U y la matriz V son matrices ortogonales, mientras que la matriz S es una matriz diagonal de dimensiones de *mxn* y en su diagonal hay los valores singulares.

De hecho, derivada del teorema de *Eckart-Young* (Johnson, 1963), el objetivo principal del algoritmo SVD es reducir el número de valores singulares de la matriz S a los primeros *k* valores, de tal manera que se obtendrá la mejor aproximación a la matriz original A como:

$$SVD(A)_{nxm} \cong U_{nxk} x S_{kxk} x V^{t}_{kxm}$$

En el caso del RS, consiste en descomponer la tabla de puntuaciones que los usuarios dieron a distintos *items*, por lo que se puede reescribir la fórmula de SVD como:

$$SVD(A)_{userxitems} \cong U_{userxk} x S_{kxk} x V^{t}_{kxitems}$$

$$\longrightarrow S_{kxk} = \begin{bmatrix} \lambda_1 & & \\ & \lambda_2 & \\ & & \lambda_k \end{bmatrix} = \begin{bmatrix} \sqrt{\lambda_1} & & \\ & \sqrt{\lambda_2} & \\ & & \sqrt{\lambda_k} \end{bmatrix} x \begin{bmatrix} \sqrt{\lambda_1} & & \\ & \sqrt{\lambda_2} & \\ & & \sqrt{\lambda_k} \end{bmatrix}$$

$$\longrightarrow SVD(A)_{usersxitem\ s} \cong U_{usersxk}\, x \begin{bmatrix} \sqrt{\lambda_1} & & \\ & \sqrt{\lambda_2} & \\ & & \sqrt{\lambda_k} \end{bmatrix} x \begin{bmatrix} \sqrt{\lambda_1} & & \\ & \sqrt{\lambda_2} & \\ & & \sqrt{\lambda_k} \end{bmatrix} x V^{t}_{kxitems}$$

$$\longrightarrow Matriz_{\ users} = U_{usersxk}\, x \begin{bmatrix} \sqrt{\lambda_1} & & \\ & \sqrt{\lambda_2} & \\ & & \sqrt{\lambda_k} \end{bmatrix}$$

$$\longrightarrow Matriz_{\ items} = \begin{bmatrix} \sqrt{\lambda_1} & & \\ & \sqrt{\lambda_2} & \\ & & \sqrt{\lambda_k} \end{bmatrix} x V^{t}_{kxitems}$$

$$\longrightarrow Matriz_{\ rating} = SVD(A)_{usersxitem\ s} \cong Matriz_{\ users}\, x Matriz_{\ items}$$

donde $Matriz_{users}$ y $Matriz_{\ items}$ son las matrices de factores, porque cada *rating* es el producto escalar de los factores de usuarios y los de *items*.

Así, se puede predecir el *rating* como: $\hat{r}_{ui} = q_i^T p_u$, siendo q_i el vector que represente los factores del *ítem* y p_u el vector que representa los factores del usuario.

Sin embargo, hay que destacar que la nota de los artículos no siempre depende del usuario, también hay que ver con sus atributos. Por otro lado, existe algunos usuarios tolerantes que siempre dan puntuaciones altas, con independencia de sus preferencias. Por lo tanto, es necesario añadir unos parámetros como:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$$

donde μ es la media de todas las puntuaciones registradas en el conjunto de entrenamiento; b_u es el sesgo del usuario (*user bias*, en inglés) y b_i es el sesgo del artículo (*item bias*, en inglés). Por ejemplo, si un usuario es muy quisquilloso, la nota predictiva suele ser baja. Y si es un usuario desconocido, se supone que el sesgo b_u y el vector p_u sean cero. Igualmente, si un *item* es desconocido, b_i y q_i son cero.

Por otra parte, al solo ajustarse los factores de aquellos usuarios que hayan dado una puntuación, es posible cometer un error $e_{ui} = r_{ui} - \hat{r}_{ui}$ (r_{ui} es la nota real y $\hat{r}_{ui}$ es la predictiva). Sobre todo, lo que planteó *Simon Funk* (Camilo Antonio Ramírez Morales, 2017) era minimizar el siguiente error cuadrático regularizado como:

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda (b_i^2 + b_u^2 + \|q_i\|^2 + \|p_u\|^2)$$

$$b_u \longleftarrow b_u + \gamma (e_{ui} - \lambda b_u)$$

$$b_i \longleftarrow b_i + \gamma (e_{ui} - \lambda b_i)$$

$$p_u \longleftarrow p_u + \gamma (e_{ui} \cdot q_i - \lambda p_u)$$

$$q_i \longleftarrow q_i + \gamma (e_{ui} \cdot p_u - \lambda q_i)$$

donde λ es una constante de regularización

5.1.6. SVD++

El algoritmo SVD++ es una extensión del SVD, que tiene en cuenta las calificaciones implícitas. La calificación prevista que el usuario *u* le dará al elemento *i* se calcula como:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T \left(p_u + |I_u|^{-\frac{1}{2}} \sum_{j \in I_u} y_i\right)$$

donde y_i es un nuevo conjunto de factores de elementos que capturan comentarios o calificaciones implícitas; una calificación implícita es independiente del valor de la puntuación real, porque se produce cuando un usuario *u* califica un elemento *j*. Por otro lado, como en SVD, si es un usuario desconocido, se supone que el sesgo b_u y el vector p_u son cero. Igualmente, si un *item* es desconocido, b_i , q_i y y_i son cero.

Cabe destacar que I_u representa el registro de comportamiento del usuario *u,* que incluye los artículos visitados y/o calificados por él. $|I_u|^{-\frac{1}{2}}$ puede suprimir la influencia sobre la cantidad de productos calificados por el usuario en el modelo.

5.1.7. TimeSVD++

El algoritmo TimeSVD++ es una extensión del SVD++, teniendo en cuenta el efecto temporal, de tal manera que la matriz bidimensional de *user-item* se convierte en una matriz tridimensional *user-item-time*. Por lo tanto, añadiendo el efecto temporal en el algoritmo SVD++, se puede reescribir la fórmula de predecir la puntuación como:

$$\hat{r}_{uit} = \mu + b_u(t) + b_i(t) + q_i^T \left(p_u(t) + |I_u|^{-\frac{1}{2}} \sum_{j \in I_u} y_i \right)$$

Sabiendo que los usuarios cambian sus preferencias con el tiempo. Por ejemplo, un fan de películas psicológicas puede convertirse en un fan de películas de acción un par de meses después. Igualmente, la popularidad del producto cambia con el tiempo. No obstante, los efectos temporales son los más difíciles de capturar. Según el trabajo de *Yehuda Koren* (Koren,2009), los siguientes efectos cambian con el tiempo:

1) sesgo del usuario (*user bias*, en inglés):

$$b_u(t) = b_u + \alpha_u \cdot dev_u(t)$$
$$dev_u(t) = sign(t - t_u) \cdot |t - t_u|^{\beta}$$

donde t_u es el tiempo medio de calificación de cada usuario; $|t - t_u|$ mide la distancia del tiempo entre t y t_u (por ejemplo, número de días); $dev_u(t)$ representa la desviación de tiempo asociada a la calificación dada por un usuario en el día t.

2) sesgo del artículo (*item bias*, en inglés)

$$b_i(t) = b_i + b_{i,Bin(t)}$$

donde $b_{i,Bin(t)}$ divide el sesgo del artículo en bins, que varia con el tiempo.

Dividir la línea de tiempo en bins es necesario equilibrar el objetivo de lograr una resolución más fina (es decir, bins más pequeños) con la necesidad de tener suficientes calificaciones por cada bins (es decir, bins más grandes). Para los conjuntos de datos de *MovieLens*, el tamaño de bins no afecta mucho a la precisión del algoritmo. Es decir, producen aproximadamente la misma precisión, lo cual se puede consultar más detalles en el Anexo II.

3) el vector que representa los factores del usuario:

$$p_u(t) = p_u + \alpha_u \cdot dev_u(t)$$

donde $\alpha_u \cdot dev_u(t)$ se aproxima a una posible porción que cambia linealmente con el tiempo.

5.2. DoubleDQN

Este apartado trata de explicar la idea de la aplicación del aprendizaje por refuerzo en el sistema de recomendación, con el fin de conocer si mejoraría el resultado de recomendación. En primer lugar, se explican brevemente los conceptos del aprendizaje por refuerzo. Luego, se comparan las diferencias entre *Q-Learning*, DQN y DoubleDQN . Por último, se presenta el algoritmo DoubleDQN especialmente diseñado para aplicar en el sistema de recomendación.

5.2.1 Concepto del Aprendizaje por Refuerzo

El Aprendizaje por Refuerzo (en adelante RL) es un área del Machine Learning, cuyo proceso consiste en que un “agente” interactúa con un entorno dinámico en el que experimenta distintas acciones para encontrar aquellas acciones que maximicen la recompensa (*feedback* o *reward*). Dicha técnica ha sido aplicada en muchos sistemas: *Open AI Dexterity, AlphaGo, AlphaStar, KenSci*, etc. En marketing digital, el RL promete renovar la industria, aplicándolo en la creación de recomendaciones personalizadas, optimización de presupuestos de publicidad, selección del mejor contenido de anuncios, aumento de CLV (*customer lifetime value*, en inglés), predicción sobre las respuestas de clientes sobre el cambio del precio (Alfrick Opidi, 2019).

En este trabajo, se enfoca específicamente en la aplicación del RL en el sistema de recomendación. De hecho, un sistema de recomendación ideal debería cumplir dos requisitos básicos:

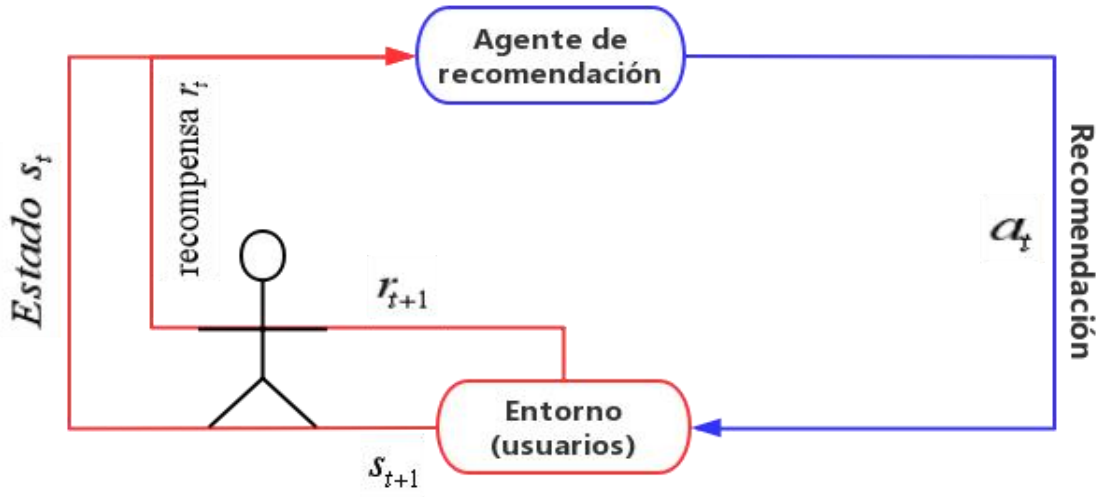
1) brindar recomendaciones basándose en los datos de retroalimentación (*rating* o *feedback*) de los usuarios;

2) optimizar los beneficios generales de todo el proceso de interacción, porque las

preferencias de usuarios son variantes.

Normalmente, un procedimiento de recomendación puede ser modelado como una interacción entre usuarios y agente de recomendación, donde el agente de recomendación aprende automáticamente las preferencias de los usuarios para generar una lista de artículos según sus gustos. En este sentido, se puede considerar los procesos de recomendación como un proceso de decisión de *Markov* (Shani et al., 2012; en adelante MDP), en el que pueda aplicar el RL como la siguiente figura:

Figura 4: MDP en el sistema de recomendación



Fuente: Elaboración propia

MDP consta de una tupla de cinco elementos (a, s, r, p, γ) de las siguientes maneras:

- **a:** acción de recomendar un *item* en el tiempo *t* (a_t), que es aleatoria con la probabilidad de decaimiento.
- **s:** estado del usuario, que es desconocido.
- **r:** la recompensa inmediata r_t en el estado s_t donde se encontraría después de la acción a_t del agente recomendador.
- **p**: la probabilidad de transición $p(s_{t+1} \mid s_t, a_t)$ cuando cambia el estado s_t hacia el s_{t+1}

γ : la tasa de descuento ($0 \leqq \gamma \leqq 1$) se aplica cuando se mide el valor presente de la recompensa futura. Si γ = 0, el agente recomendador solo considera la recompensa inmediata. Si γ = 1, los valores de acción pueden divergirse y sobre todo, sin un estado terminal, todas las historias ambientales se vuelven infinitamente largas. En caso de que γ se aproxima al 1, el agente se esfuerce por obtener una recompensa alta a largo plazo. En este trabajo, se fija una tasa de descuento del 0,9.

Como se puede ver en la Figura 4, en primer lugar, el agente recibe un estado s_t y escoge una recomendación a_t del conjunto de *items* disponibles y la envía al entorno. Luego, el entorno (en este caso, el usuario) dará una recompensa r_t sobre la película recomendada por el agente y generará un estado nuevo s_{t+1}. Así, sucesivamente, el agente interactúa con el entorno para obtener una gran cantidad de datos y encontrar cuál es la acción que genere mayor recompensa.

De hecho, en el macro del RL, el sistema de recomendación actúa como un agente, mientras que los comportamientos actuales del usuario tales como disgusta, gusta y *rating* (estado), y la película que se recomienda se considera como una acción. En cada interacción de recomendación, el agente selecciona la acción más adecuada en función del estado del usuario para maximizar el objetivo específico a largo plazo (como el número total de clics o la retención de usuarios).

Por otra parte, cabe mencionar que, basando en el MDP*,* se puede dividir el RL en dos tipos: los basados en modelo (*model-based*, en inglés) o los libres de modelo (*model-free*, en inglés). En este trabajo, solo se centra en los algoritmos libres de modelo, que son: *Q-Leanring*, DQN y DoubleDQN.

5.2.2. Comparación entre Q-Learning, DQN y DoubleDQN

Antes de realizar la comparación entre *Q-Learning*, DQN y DoubleDQN, cabe subrayar que todos ellos son métodos basados en funciones de valor, cuyo objetivo es estimar el valor de Q. No obstante, *Q-Learning* simplemente guarda información en tablas para calcular los posibles valores Q, mientras que DQN combina el algoritmo *Q-Learning* con redes neuronales profundas. Es decir, en DQN, se utiliza una red neuronal para estimar los valores.

En verdad, en el área del RS, un trabajo moderalizó el sistema de recomendación como un proceso MDP y estimó la probabilidad de transición y la tabla de valores Q (Shani et al., 2005). No obstante, resulta inflexible este método cuando crece el número de artículos para recomendar a los usuarios, así que otro estudio (Zhao, Zhang, et al., 2018) estimó la función de valor usando una red neuronal para no almacenar la tabla de valores Q y estimar la función de valor de acción por separado para cada secuencia. Se puede decir que, para gestionar gran cantidad de datos, DQN es más flexible que *Q-Learning*. Por eso, DQN es el más extendido actualmente en el aprendizaje por refuerzo (Mnih et al., 2015).

Sin embargo, como el principio de ambos métodos es maximizar el valor Q, que tiende a escoger la acción con el valor Q más alto para aproximarse a la política óptima, tienen el problema de sobreestimación del valor Q.

Tabla 4: Comparación entre Q-Learning, DQN y DoubleDQN

	Fórmulas	Características
Q-Learning	$Y_t^Q = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t)$	- Se trata de encontrar una política óptima que maximice el valor esperado de la recompensa total sobre todos los pasos sucesivos, empezando desde el estado actual. - Requiere una tabla Q grande para almacenar los datos. - Suele aplicarse cuando la cantidad de datos no es muy grande. - Tiene el problema de sobreestimación del valor Q por el operador *max*.
DQN	$Y_t^{DQN} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-)$	- Se trata de usar una red neuronal para aproximar la función Q, evitando así el uso de la tabla. - Se aplica la repetición de experiencia o Experiencia Replay para almacenar los datos y entregar la experiencia de forma aleatoria y eficiente. - Tiene el problema de sobreestimación del valor Q por el operador *max*.
DoubleDQN	$Y_t^{DoubleDQN} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1}, a; \theta_t), \theta_t^-)$	- Se trata de aplicar 2 redes neuronales, usando la segunda red para eliminar algunos efectos del error máximo. - La primera red se denomina Red Q para decidir cuál es la mejor acción entre todas las posibles. - La segunda red se denomina Red Target para evaluar acciones, conociendo el valor Q.

Fuente: Elaboración propia

Como se puede ver en la Tabla 4 donde contiene las fórmulas de estimación de Q de cada modelo, tanto en *Q-Leanring*, como en DQN, el valor Q se calcula con la recompensa agregada al valor Q máximo del siguiente estado, utilizando el operador *max* (Hado van Hasselt, 2010). En otras palabras, cada vez que el valor Q calculado para un cierto estado es muy alto, será cada vez mayor hasta que la diferencia entre cada valor de salida sea alta.

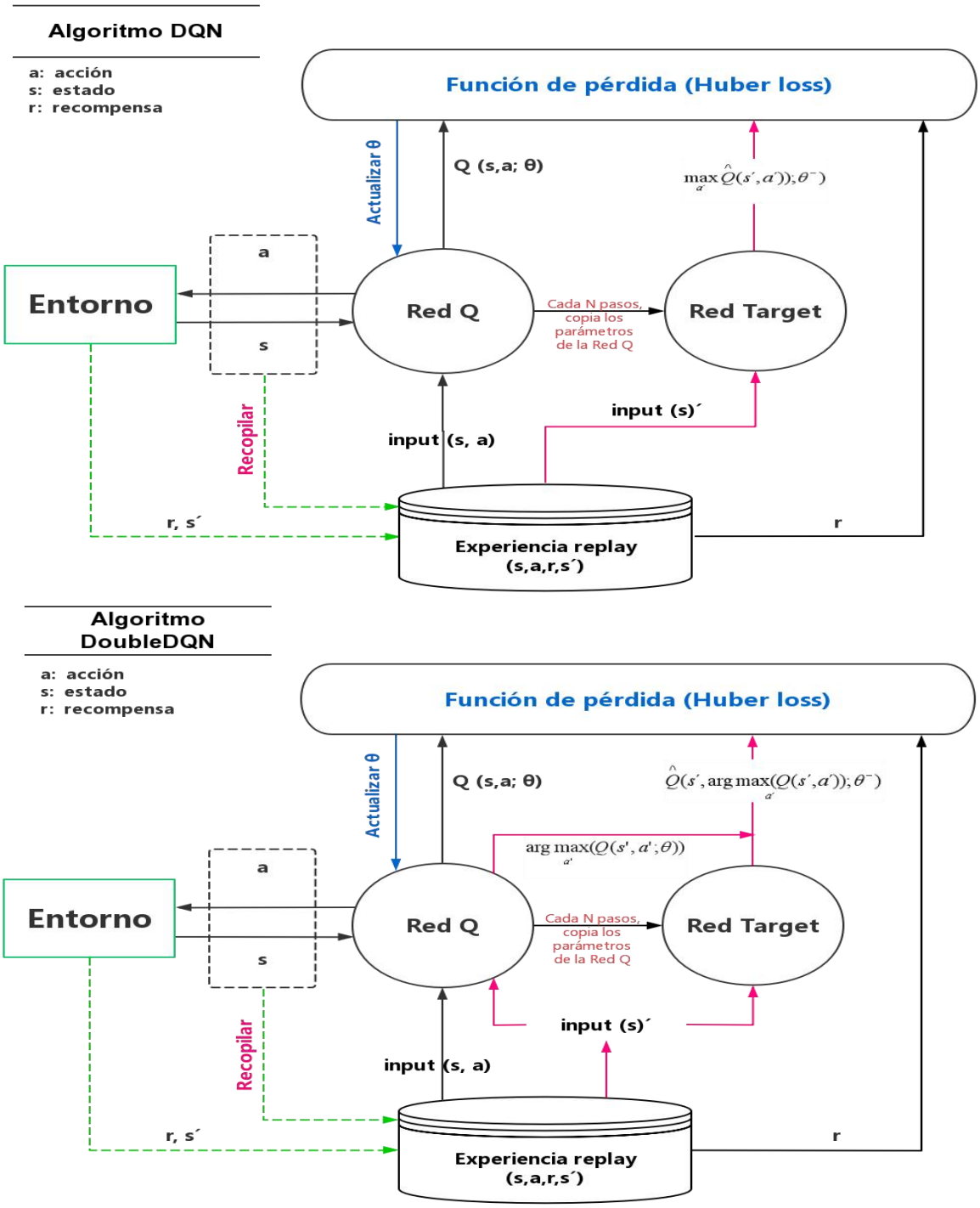
De hecho, en el trabajo de Hasselt (Hado van Hasselt et al., 2015) ha mostrado la ventaja de usar DoubleDQN para solucionar el problema de las sobreestimaciones. En concreto, se compararon DQN y DoubleDQN en seis juegos de Atari, que los valores estimados por DQN son mucho más alto que los valores reales, mientras que los

estimados por DoubleDQN son más aproximados a los reales.

Por lo tanto, en este trabajo, se aplica el algoritmo DoubleDQN para entrenar los datos y evaluar su utilidad en el sistema de recomendación. Desde luego, también se utiliza el DQN para comparar con el DoubleDQN, con el propósito de saber si hay gran diferencia sobre su aplicación en el sistema de recomendación.

A continuación, para explicar de manera clara el flujo de trabajo del DQN y DoubleDQN, se muestran dos diagramas de flujo en la Figura 5.

Figura 5: DQN y DoubleDQN



Fuente: Elaboración propia

De hecho, como se puede ver en la Figura 5, tanto el algoritmo DoubleDQN, como el DQN, utiliza dos redes neuronales para estabilizar el proceso de aprendizaje. Mejor dicho, básicamente, la estructura del DoubleDQN es la misma que la del DQN (se llama *Nature DQN* cuando se usan 2 redes). Parece contradictorio a lo mencionado en la Tabla 4, pero no hay por qué haber 2 redes en el algoritmo DQN, puede construir un DQN con una sola Red Q sin la Red Target (es DQN convencional cuando solo tiene una red).

No obstante, en el algoritmo DQN, lo que hace en primer lugar es generar el valor de Q predicho y luego el valor de Q objetivo, si utiliza solamente una red, es inestable porque una pequeña actualización de Q puede cambiar significativamente la dirección de los valores de Q objetivo predichos. Es decir, puede fluctuar mucho después de cada actualización, no ayuda a la convergencia. Si emplea una segunda red que no recibe capacitación, se puede asegurar de que los valores de Red Target permanezcan estables.

En pocas palabras, se ha descubierto que el uso de una Red Target produce un resultado más estable (Mnih et al., 2015), que la Red Q actúa como un agente que está ejecutando para producir el valor de acción de estado óptimo, y que la Red Target sirve para predecir el valor Q objetivo.

En este sentido, la principal diferencia entre el DQN y el DoubleDQN no depende del número de red usado, sino de la función para calcular el valor Q objetivo de los siguientes estados:

$$Y_t^{DQN} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-)$$

$$Y_t^{DDQN} = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t), \theta_t^-)$$

donde S_{t+1} representa el estado nuevo; ***a*** es la acción seleccionada; γ es el factor de descuento; θ es el valor de descenso de gradiente.

Como se puede ver en las fórmulas anteriores, en DQN, simplemente se toma el máximo valor Q de todas las acciones de la Red Target, que es probable seleccionar valores sobreestimados. Sobre todo, el DQN usa el parámetro θ_t^- de la Red Target que se copia de la Red Q, que es un conjunto de valores muy correlacionados.

Por consiguiente, en el DoubleDQN, desacopla la operación máxima de la Red en dos funciones: una para seleccionar acciones y la otra para evaluar(Hado van Hasselt et al., 2015). Al desacoplar la función, el DoubleDQN puede aprender intuitivamente qué estados son (o no) valiosos sin tener que aprender el efecto de cada acción en cada estado. En otras palabras, el DoubleDQN se trata de encontrar una acción correspondiente al máximo valor Q en la RED Q, determinando la acción como:

$$a^{\max}(S_{t+1},\theta_t) = \arg\max_{a'} Q(S_{t+1},a;\theta_t)$$

Y posteriormente estima la acción seleccionada en la Red Target para calcular el valor Q objetivo como:

$$\begin{aligned} Y_t^{DDQN} &= R_{t+1} + \gamma Q(S_{t+1}, a^{\max}(S_{t+1},\theta_t),\theta_t^-) \\ &= R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1},a;\theta_t),\theta_t^-) \end{aligned}$$

De esta forma, incluso si las dos aproximaciones Q tienen ruidos, pero no están muy correlacionados, por lo que el problema estaría resuelto.

5.2.3. Aplicación de DoubleDQN en el sistema de recomendación

Basando en la idea de Hasselt, se ha diseñado el DoubleDQN como el pseudocódigo siguiente:

Algoritmo DoubleDQN

Iniciar la Red Q_θ con un peso aleatorio θ;
Iniciar la Red Target $\hat{Q}_{\theta'}$ con un peso aleatorio $\theta' = \theta$;
Iniciar la Experiencia Replay D con la capacidad N;
Fijar el factor de descuento $\gamma \in [0,1]$;
Fijar el *Learning rate* $\alpha \in (0,1)$;

Para cada interacción:
  Para cada paso dentro de una interacción:
    Observar el estado s_t y seleccionar una acción a_t aleatoriamente
    con la probabilidad de decaimiento de épsilon ξ;
    en cambio, seleccionar $a_t = \arg\max_a Q(s_t,a;\theta)$;
    Ejecutar a_t, obteniendo el estado siguiente s_{t+1} y la recompensa
    $r_t = R(s_t,a_t)$;
    Recopilar (s_t,a_t,r_t,s_{t+1}) en la Experiencia Replay D;

    Muestrear un *minibatch* aleatorio de N transacciones de D;
    Para cada transacción $(s_t,a_t,r_t,s_{t+1},ter\min a_t)$ en *minibatch*, hace:
      Si $ter\min a_t$ es verdadero,
        $Q^*(s_t,a_t) = r_t$
      Si no,
        $Q^*(s_t,a_t) = r_t + \gamma Q_\theta(s_{t+1}, \arg\max_{a'} Q_{\theta'}(s_{t+1},a'))$
    Actualizar el valor de descenso de gradiente θ con la función de *Huber loss*;
    Cada T paso, actualiza los parámetros de la Red Target
    como $\theta' = \gamma * \theta + (1-\gamma) * \theta'$

Se puede consultar los códigos completos en el Anexo II, a continuación, se plantean unas respuestas sobre preguntas frecuentes sobre el diseño personalizado del DoubleDQN en este trabajo:

1) ¿Por qué necesita Experiencia Replay D?

Porque ésta interactúa con el entorno para generar datos a entrenar la Red Q, sobre todo, sirve para almacenar las experiencias de "éxito" o "fracaso" durante el proceso del entrenamiento. Es como un pendrive, que se almacenan todas las acciones y observaciones que el agente ha realizado desde el principio (limitadas por la capacidad de la memoria prefijada N, eso ya depende de la capacidad del almacenamiento del ordenador).

Por otro lado, como en las redes neuronales suelen tomar un lote de datos. Si simplemente toma una acción obteniendo una recompensa del entorno y luego lo envía a la Red Q, cada muestra y los gradientes correspondientes tendrían demasiada varianza y los pesos de la red nunca convergerían. Sobre todo, se ha comprobado que la utilización de Experiencia Replay puede mitigar los efectos del sobreajuste (Fu et al., 2019)

2) ¿Por qué selecciona una acción aleatoriamente con la probabilidad de decaimiento de épsilon *ξ*?

Porque al principio del entrenamiento, se pretende seleccionar aleatoriamente las acciones de la memoria Experiencia Replay para haber suficiente diversidad en los datos de entrenamiento, pero, dicha aleatoriedad se decrece cuando haya ejecutado muchas veces. Imagina que una persona que haya trabajado 10 años en Marketing tiene mucha experiencia y sabe muy bien tomar qué decisión puede obtener mayor beneficio, así que su probabilidad de aplicar una estrategia o una acción aleatoria es relativamente pequeña que una persona sin experiencia.

De hecho, esta técnica se llama *Epsilon-greedy*, que el término épsilon *ξ* es usado en el entrenamiento y con el que se toma una acción aleatoria.

3) ¿Por qué usa *Huber loss* en vez del *MSE loss* o *MAE loss* para actualizar el valor de descenso de gradiente θ ?

En realidad, el error cuadrático medio (en adelante, MSE) y el error absoluto medio (en adelante, MAE) son las funciones que suelen utilizarse para actualizar el valor de descenso de gradiente θ.

Sin embargo, al tener la función cuadrática, el MSE hace que la red "se preocupe" más por los errores grandes que por los pequeños. Intuitivamente, esto es algo bueno, porque importa minimizar los errores grandes en lugar de los pequeños. No obstante, en el caso de DQN y DoubleDQN, esto puede tener un impacto perverso. En concreto,

si aparece un error grande, la red cambiará radicalmente para tratar de minimizarlo, pero como la red está tratando de predecir su propia salida, un cambio radical significa que el valor Q objetivo también cambia radicalmente, habiendo gran posibilidad de que no reduzca el error tanto y haga el proceso del entrenamiento más lento. En la práctica, es posible tener el problema del gradiente explosivo cuando se acumulan grandes gradientes de error y resulta demasiado grande la actualización de los pesos de la red durante el proceso del entrenamiento (Mnih et al., 2015；zhihu_newcolour, 2020) .

Una alternativa al MSE es el MAE, que toma el valor absoluto del residual, hace que la red "se preocupe" por reducir los errores grandes tanto como los pequeños. No obstante, el MAE ignora que los errores grandes deberían de alguna manera "importar más" y no es diferenciable en 0, porque está haciendo un descenso de gradiente.

Por lo tanto, hay que encontrar un equilibrio entre el MSE y el MAE. *Huber Loss* ofrece una solución al equilibrar el MSE y el MAE juntos, que se define como:

$$L_\delta(error) = \begin{cases} \frac{1}{2} error^2, & si\ error \le \delta \\ \delta(|error| - \frac{1}{2}\delta), & si\ error > \delta \end{cases}$$

Como se puede ver en las fórmulas anteriores, *Huber Loss* combina efectivamente las ventajas tanto del MSE como del MAE. Es menos sensible a los valores atípicos en los datos que el MSE y puede ser diferenciable en 0. Cuando el error es pequeño, utiliza el MSE. Cuando el error es grande, utiliza el MAE. Para determinar el error es pequeño o grande, hay que ajustar bien el parámetro δ , por defecto, es 1.

En pocas palabras, con la aplicación de *Huber loss*, el proceso de entrenamiento sería más estable.

4) ¿Qué optimizador u optimizadores de aprendizaje profundo se usa en este trabajo? ¿Y por qué?

En realidad, existe varios optimizadores utilizados en el aprendizaje profundo: Descenso de gradiente por lotes (en adelante BGD), Descenso de gradiente estocástico (en adelante SGD), Descenso de gradiente por lotes pequeños (en adelante MBGD), *Momentum*, Gradiente acelerado de *Nesterov* (en adelante NAG), Algoritmo de gradiente adaptativo (en adelante Adagrad), *Adadelta* (una mejora de Adagrad), Propagación cuadrática media (en adelante RMSprop) y *Adaptive Moment Estimation* (en adelante Adam). Cada uno tiene distinta regla de actualización de gradiente, ventajas y desventajas en situaciones distintas (SEBASTIAN RUDER, 2016).

BGD, SGD y MBGD son las tres variantes más comunes de descenso de gradiente.

Para acelerar la tasa de convergencia de descenso de gradientes, se utiliza MBGD, un pequeño lote de *n* muestras (*batch size*, en inglés), de manera que la convergencia es más estable y puede reducir la varianza al actualizar los parámetros. Es decir, MBGD calcula el gradiente no a partir de todo el conjunto de entrenamiento (BGD) ni a partir de una única muestra (SGD). Normalmente, el tamaño del mini-lote varía de 32 hasta 512 (considerando la memoria de la computadora, ajústalo a la potencia n-ésima de 2, la velocidad de cálculo será más rápida. Por supuesto, se puede poner el valor 1, en este caso, se actúa igual como SGD).

Además del MBGD, para eliminar oscilaciones divergentes en el descenso de gradiente, se usa RMSProp, que es una variación de AdaGrad en la que, en lugar de mantener un acumulado de los gradientes, se utiliza el concepto de "ventana" para considerar solo los gradientes más recientes. En concreto, el algoritmo RMSProp utiliza un promedio ponderado exponencialmente para eliminar oscilaciones en el descenso de gradiente y dar mayor velocidad a la convergencia de descenso de gradientes.

5) ¿Por qué NO incluye *Early stopping* en la red del DoubleDQN para evitar el problema de sobreajuste?

Aunque tiene sentido usar *Early stopping* para evitar el problema de sobreajuste, se ha comprobado que aún se prefieren las arquitecturas grandes, porque el daño del sesgo de aproximación de funciones supera al daño del aumento del sobreajueste con modelos grandes. En suma, la mejor estrategia es mantener grandes arquitecturas pero seleccionar cuidadosamente el número de pasos de gradiente que se utilizan por muestra (Fu et al., 2019).

Además de lo anterior, teniendo en cuenta las características de datos de *MovieLens*, se ha decidido que para el primer conjunto *MovieLensSLD*, las variables input son: calificación media de película, la media de las calificaciones dadas por el usuario, la última película evaluada por el usuario, los 20 géneros cinematográficos de la película, el último usuario que participó en la calificación de película. En total, hay 24 neuronas de entrada. Y en el segundo conjunto *MovieLens100K*, hay 23 neuronas de entrada, porque este conjunto solo tiene 19 géneros cinematográficos de la película.

Luego, en ambos conjuntos, se utilizan dos capas ocultas de 24 neuronas con la función de activación ReLU. Aspirado por la idea de Wiering et al, se decide que la capa de salida tiene 5 neuronas, que es la probabilidad de que un usuario evalúa una película con 1, 2, 3, 4 y 5, el valor Q objetivo escoge la puntuación cuya probabilidad es la más alta. La función de activación de la capa final es lineal.

Con respecto a los valores de los hiperparámetros de la red, se puede ver en la Tabla 5:

Tabla 5: Los hiperparámetros utilizados en la red y sus valores

Hiperparámetro	Valor	Descripción
Tasa de descuento	0,9	Tasa de descuento gamma γ usada en la actualización de la red.
épsilon ξ inicial	1	Valor inicial de épsilon ξ in epsilon-greedy
épsilon ξ final	0,1	Valor final de épsilon ξ in epsilon-greedy
Paso de exploración	100000	Número de pasos sobre los que el valor inicial de épsilon ξ se suaviza linealmente a su valor final
Frecuencia de actualizar la Red Target	10	La frecuencia con la que se actualiza la Red Target
Tamaño inicial de Experiencia Replay	200	Número de pasos para completar la Experiencia Replay antes de comenzar el entrenamiento
Memoria de reproducción	400000	Número de la Experiencia Replay que el agente usa para entrenar
Train Interval	4	El número de acciones seleccionadas por el agente entre las sucesivas actualizaciones de MSGD. El agente selecciona 4 acciones entre cada par de actualizaciones sucesivas.
Save Interval	300000	La frecuencia con la que se guarda la Red Q.
Learning rate	0,00025	La tasa de aprendizaje utilizada por RMSProp.
Gradient Momentum	0,95	Momento de gradiente utilizado por RMSProp.
Min squared gradient momentum	0,01	Constante agregada al gradiente al cuadrado en el denominador de la actualización RMSProp.
Tamaño del *Minibatch*	32	Número de entrenamiento utilizado en cada iteración de MSGD.
no-op max	30	Número máximo de "no hacer nada" que el agente realiza al comienzo de un episodio.

Fuente: Elaboración propia

5.3. Ensamblado

Los modelos ensamblados son estrategias de combinar varios modelos para reducir el error y construir unos sistemas de recomendación híbridos que hacen más recomendaciones con mayor precisión.

Es obvio que cada modelo de RS funciona de manera diferente y tiene sus ventajas y desventajas a la hora de hacer recomendaciones al usuario, por lo que ensamblar distintos algoritmos de RS sirve para compensar sus errores y mejorar su generalización.

Hay cuatro tipos de ensamblados más usados, votación por mayoría, *Bagging*, *Boosting* y *Stacking* (modelos apilados). En este trabajo, se ha utilizado *Stacking* para combinar distintos modelos de RS, porque esta técnica ha sido utilizada en RS y ha conseguido buenos resultados (Bao et al., 2009) .

La arquitectura de *Stacking* puede implicar dos o más modelos de base, permitiendo usar la salida de cada estimador individual como la entrada del estimador final. En este trabajo, se ha diseñado un paquete *StackingRS* para el sistema de recomendación, dado que resulta imposible aplicar directamente el paquete *StackingClassifier* de Python para los algoritmos de RS. Con respecto a los códigos paquete *StackingRS*, se puede consultar en el Anexo II.

En concreto, solo se usa un modelo de base (nivel 1) para entrenar los datos, y un metamodelo (nivel 2) que combina las predicciones del modelo base para aprender cómo combinar mejor sus predicciones. Es decir, se considera la salida de modelo de nivel 1 como la entrada de modelo de nivel 2 para generar la predicción final del sistema híbrido de recomendación.

Por otro lado, se prepara un conjunto de datos para el metamodelo a través de la validación cruzada de k iteraciones del modelo base, que consiste en dividir los datos en 5 partes dejando uno fuera como el dato de entrenamiento para el metamodelo. De hecho, una vez que el conjunto de datos de entrenamiento esté preparado para el metamodelo, éste puede entrenarse de forma aislada en el mismo, y el modelo de base puede entrenarse en el conjunto de datos k-1.

6. Métodos de evaluación de rendimiento

Esta sección tiene como finalidad mostrar varios métodos de evaluación de rendimiento de los distintos algoritmos de RS, así como comparar sus resultados en la sección 7. Con respecto a los códigos para evaluar los algoritmos, se puede consultar en el Anexo II.

6.1. Validación cruzada repetida

La idea de validación cruzada proviene de la mejora del método de retención (se dividen los datos en 2 grupos, usando un subconjunto para entrenar y el otro para validar el análisis), porque este método no es muy preciso debido a la variación de resultados obtenidos para distintos datos de entrenamiento. Sobre todo, la evaluación puede depender en gran medida de cómo es la división entre datos de entrenamiento y de prueba, y por lo tanto puede ser significativamente diferente en función de cómo se realice esta división (Jeff Schneider, 1997).

De hecho, existe muchos tipos de validaciones cruzadas: validación cruzada de K iteraciones (*K-fold cross-validation*, en inglés), validación cruzada dejando uno fuera (*Leave-one-out cross-validation*, en inglés), validación cruzada dejando p fuera (*Leave p-out cross-validation*, en inglés) y validación cruzada de búsqueda de cuadrículas (*GridSearch cross-validation*, en inglés). En este trabajo, se utiliza validación cruzada repetida de K iteraciones, con el fin de evitar los posibles problemas relacionados con el sobreajuste y la sobregeneralización del modelo.

En concreto, se dividen los datos en 5 grupos (*5-fold cross-validation*), repitiendo 5 veces con diferente aleatorización en cada repetición. En este sentido, es obvio que realizar la validación cruzada es necesitar una gran carga computacional y sobre todo, que tardará más tiempo de ejecutar cuando el conjunto de datos sea más grande.

En este trabajo, se han probado 2 formas para llevar a cabo la validación cruzada repetida:

1) el iterador de validación cruzada *RepeatedKFold* del paquete *surprise.model_selection*

2) el método *validation.cross_validate* del paquete *surprise.model_selection*

Ambos son válidos para evaluar los algoritmos, pero la segunda forma resulta más conveniente al resumir automáticamente el tiempo de entrenamiento, el de prueba, el error cuadrado medio y la raíz del error cuadrático medio. Se puede consultar más detalles sobre los códigos de validación cruzada repetida en el Anexo II o *The model_selection package — Surprise 1 documentation.*

6.2. Error absoluto medio

El error absoluto medio (*Mean absolute error*, en adelante MAE) usa el valor absoluto para calcular el error de predicción, que se define como:

$$MAE = \frac{1}{\left|\hat{R}\right|} \sum_{\hat{r}_{ui} \in \hat{R}} \left| r_{ui} - \hat{r}_{ui} \right|$$

donde $\hat{R}$ es el conjunto de valoraciones previstas; r_{ui} es la valoración real del usuario *u* sobre el artículo *i*; $\hat{r}_{ui}$ es la predicción de la valoración del usuario *u* sobre el artículo *i.*

6.3. Raíz del error cuadrático medio

La raíz del error cuadrático medio (*Root Mean Squared Error*, en adelante RMSE) representa a la raíz cuadrada de la distancia cuadrada promedio entre el valor real y el valor pronosticado, que se define como:

$$RMSE = \sqrt{\frac{1}{\left|\hat{R}\right|} \sum_{\hat{r}_{ui} \in \hat{R}} \left(r_{ui} - \hat{r}_{ui} \right)^2}$$

En relación a las ventajas y desventajas de MAE y RMSE, se cree que RMSE ha aumentado la penalización por valoraciones inexactas de artículos de usuario y, por tanto, su evaluación de sistema de recomendación es más exigente. Un estudio sobre el sistema de recomendación (Takács et al., 2007) ha demostrado que, si el sistema de puntuación se basa en números enteros (es decir, todas las puntuaciones dadas por los usuarios son enteras), redondear las puntuaciones previstas reducirá el valor de MAE.

6.4. Precisión y Recall

Se entrenan los distintos algoritmos para brindar a los usuarios una lista personalizada de recomendaciones. La precisión y la sensibilidad (en adelante Recall) son métricas de evaluación clásicas de los algoritmos del sistema de recomendación, que se definen como:

$$\text{Pr } esición = \frac{\sum_u \left| R(u) \cap T(u) \right|}{\sum_u \left| R(u) \right|}$$

$$\text{Re } call = \frac{\sum_u \left| R(u) \cap T(u) \right|}{\sum_u \left| T(u) \right|}$$

donde $R(u)$ es el conjunto de artículos recomendados al usuario *u* ; $T(u)$ es el conjunto de artículos que le gustan.

De hecho, la precisión se preocupa por cuántas recomendaciones son relevantes entre las recomendaciones proporcionadas, mientras que la sensibilidad se preocupa por cuántas recomendaciones se proporcionan entre todas las recomendaciones relevantes.

6.5. Valor- F

El valor F (F-score, en adelante F) se considera como una media armónica que combina los valores de la precisión y de la sensibilidad, de tal forma se define como:

$$F = (1+\beta^2)\cdot\frac{\text{Pr}\,ecisión\ \cdot Sensibilid\ ad}{(\beta^2\cdot\text{Pr}\,ecisión\ )+Sensibilid\ ad}$$

Si *β* es igual a uno, significa que la precisión y la sensibilidad tienen igual importancia. Si *β* es menor que 1, significa que la precisión tiene mayor importancia que la sensibilidad. Si *β* es mayor que 1, significa que la sensibilidad tiene mayor importancia que la precisión.

En este trabajo, se supone que tengan la misma importancia, se define F1 como:

$$F1 = 2\cdot\frac{\text{Pr}\,ecisión\cdot Sensibilidad}{(\text{Pr}\,ecisión+Sensibilidad)}$$

Lógicamente, cuando F1 sea mayor, más robusto será el modelo de clasificación.

6.6. Curva ROC

La curva ROC es un gráfico que representa la proporción de verdaderos positivos (en adelante TPR) frente a la proporción de falsos positivos (en adelante FPR) según el punto de corte prefijado. En este trabajo, el punto de corte adecuado es 3.

En concreto, se crea un recomendador Top-N (una lista de recomendación con N artículos), considerando verdaderos positivos (en adelante TP) aquellos *items* de la lista Top-N que coincidan con los que el usuario prefería en el conjunto de pruebas, y siendo falsos positivos (en adelante FP) los que no coincidan.

Sucesivamente, se definen verdaderos negativos (en adelante TN) aquellos *items* que no están en la lista Top-N ni tampoco son los que el usuario prefería en el conjunto de pruebas. Y son falsos negativos (en adelante FN) aquellos *items* que no están en la

lista Top-N pero que son los que el usuario prefería en el conjunto de pruebas. Así, se definen las fórmulas para calcular la tasa de verdaderos positivos (en adelante TPR) y la de falsos positivos (en adelante FPR) como:

$$TPR = \frac{TP}{TP + FN}$$
$$FPR = \frac{FP}{FP + TN} = 1 - Especifici\ dad$$

Como todas las métricas de evaluación son numéricas, calcular el área bajo la curva ROC (en adelante AUC) para facilitar la comparación de los algoritmos. Hay distintas fórmulas para calcular el AUC, en este trabajo se define como:

$$AUC = \sum_{i \in (P+N)} \frac{\left(TPR_i + TPR_{i-1}\right) \cdot \left(FPR_i + FPR_{i-1}\right)}{2}$$

No obstante, el AUC ignora los valores de probabilidad predichos y no brinda información sobre la distribución espacial de los errores del modelo (Lobo et al., 2008; Hand, 2009). Sobre todo, tiene una gran restricción cuando hay algunos algoritmos que tengan AUC similares o iguales (Zhu & Lü, 2012). Según la literatura, el límite inferior del AUC es 0,5 para poder verificar que los modelos predictivos sean buenos. Pero, por la naturaleza del sistema de recomendación, se suele obtener un AUC cerca de 0,5. Sobre todo, para el conjunto de datos que tienen el problema de *Cold-start*.

6.7. Informedness y Markedness

Aunque la información *(Informedness*, en adelante Inform*)* y la confiabilidad (*Markedness*, en adelante Mark*)* son 2 métricas menos utilizadas para evaluar los algoritmos de RS, como la Precisión y Recall ignoran la capacidad de los modelos para manejar negativos, sería mejor incluirlas como sus métricas complementarias (Powers, 2007; Schröder et al., 2012; Ranganath, 2018).

El Mark mide la confiabilidad de las predicciones positivas y negativas del modelo, combinando la precisión y precisión inversa, que se define como:

$$\begin{aligned} Mark &= \frac{TP}{TP + FP} - \frac{FN}{TN + FN} \\ &= \frac{TP}{TP + FP} - (1 - \frac{TN}{TN + FN}) \\ &= \frac{TP}{TP + FP} - 1 + \frac{TN}{TN + FN} \\ &= \Pr ecisión + \Pr ecisión\ \ Inversa - 1 \end{aligned}$$

El Inform mide la información de las predicciones positivas y negativas del modelo, combinando el Recall *y* Recall *inverso,*que se define como:

$$\begin{aligned} Inform &= \frac{TP}{TP+FN} - \frac{FN}{TN+FP} \\ &= \frac{TP}{TP+FN} - (1 - \frac{TN}{TN+FP}) \\ &= \frac{TP}{TP+FN} - 1 + \frac{TN}{TN+FP} \\ &= \text{Re } call + \text{Re } call \;\; Inverso \;\; - 1 \end{aligned}$$

El rango del Mark y el Inform es de -1 a 1 (ambos incluidos). Un Mark de 1 implica que todas las predicciones positivas y negativas son correctas, mientras que un Mark de -1 implica que todas son incorrectas. Es decir, cuando esté más cerca del 1, será mejor el modelo al considerar correctamente tanto las predicciones positivas como las negativas. No obstante, cuando esté cerca del 0 (de -0,1 a 0,1), eso significa que solo puede confiar en las predicciones positivas del modelo y no en las negativas.

De igual manera, un Inform de 1 implica que todo positivo se identifica como positivo y todo negativo se identifica como negativo correctamente. Mientras que un Inform de -1 implica que todo positivo se identifica incorrectamente como negativo y todo negativo como positivo. Es decir, cuando esté más cerca del 1, será mejor el modelo. No obstante, cuando esté cerca del 0 (de -0,1 a 0,1), eso significa que el modelo solo está informado bien sobre los positivos y no los negativos.

6.8. Tasa de aciertos

La tasa de aciertos (*Hit Rate*, en adelante HR) es un indicador de evaluación muy útil en las recomendaciones de Top-N, por lo que también se suele llamar como *Top-N hit rate*. Consiste en generar las *n* recomendaciones principales para un usuario y comparar con las ya calificados por el usuario. Si coinciden, aumente el número de aciertos en 1.

En este trabajo, se ha usado el método dejando uno fuera (*Leave-one-out*, en adelante LOO) que consiste en emplear un conjunto de entrenamiento para generar una lista de recomendación y usar lo que deja fuera para medir el HR como:

$$HR = \frac{\left|I_{hits}\right|}{\left|I\right|}$$

donde $\left|I\right|$ es el número total de artículos del conjunto de prueba y $\left|I_{hits}\right|$ es el número de artículos calificados por el usuario que coinciden con los de la lista de recomendaciones Top-N.

6.9. Tasa de aciertos acumulativa

La tasa de aciertos acumulativa (*Cumulative Hit Rate*, en adelante cHR) es parecida al HR que se muestra antes, pero se agrega un umbral o una calificación mínima en el conjunto de prueba para verificar los artículos recomendados al usuario son los que realmente le gustan.

En concreto, en el caso del HR, el número de aciertos se aumenta en 1 cuando los artículos recomendados al usuario son los que han sido calificados en el conjunto de prueba. Mientras que en el caso del cHR, el número de aciertos se incrementa en 1 cuando los artículos recomendados al usuario son los que han sido calificados como gustados. Se define el cHR como:

$$cHR = \frac{\left|I_{hits\,\&\,like}\right|}{\left|I\right|}$$

donde $\left|I\right|$ es el número total de artículos del conjunto de prueba y $\left|I_{hits\ \&\ like}\right|$ es el número de artículos con una calificación mínima de 4 y coincidiendo con los de la lista de recomendaciones Top-N. Es decir, si predice que una película tenga una puntuación menor que 4, no se acumula el número de aciertos.

6.10. Rango medio de aciertos recíprocos

El rango medio de aciertos recíprocos (*Average Reciprocal Hit Rank*, en adelante ARHR) también es una métrica similar al HR. De hecho, es una métrica centrada en usuarios, puesto que las personas tienden a concentrarse más en lo que ven al comienzo de las listas de Top-N. Así que se le da más peso a los atributos que aparecen en la parte superior de la lista, y penaliza a aquellos que tengan posiciones bajas.

En concreto, para cada usuario que tenga un acierto, suma el recíproco del rango del *itemid* (por ejemplo, *movieId*) en la lista de Top-N y luego se divide por el número total de usuarios como:

$$ARHR = \frac{\sum_{i=1}^{N} \frac{1}{rank_i}}{\left|U\right|}$$

donde $rank_i$ representa el ranking de los artículos; $\left|U\right|$ es el número total de usuarios.

6.11. Diversidad

En la búsqueda de información, la diversidad se considera una medida para evitar la redundancia y encontrar resultados que cubran diferentes aspectos de una necesidad de información (Radlinski et al., 2009). En general, se puede considerar la diversidad como lo opuesto a la similitud, por lo que se suele calcular primero la similitud entre los artículos recomendados al usuario y luego restándola 1 para encontrar la diversidad.

En este trabajo, se define como:

$$Diversidad = 1 - \frac{\sum_u (\sum_{i,j \in T_u} Sim_{ij})}{\sum_u N_u}$$

donde T_u es el conjunto de los Top-N recomendados para el usuario *u*; *i* y *j* son los elementos dentro de T_u; Sim_{ij} es la similitud entre los elementos *i* y *j*; N_u representa todas las combinaciones de los elementos *i* y *j* para el usuario *u*.

Esta métrica es importante para medir la capacidad del RS para descubrir los diferentes intereses de usuarios. Si la lista de recomendaciones es más diversa y cubre la mayoría de los puntos de interés del usuario, aumentará la probabilidad de que el usuario encuentre el artículo que le guste.

Sin embargo, hay que mencionar que, es posible lograr una alta diversidad recomendando cosas completamente aleatorias. Es decir, una diversidad inusualmente alta puede conducir a malas recomendaciones, que tampoco sea lo ideal.

6.12. Novedad

La novedad de las recomendaciones se refiere a recomendar los artículos desconocidos a los usuarios. La novedad se ha evaluado de muchas formas en la literatura, utilizando diferentes métricas y definiciones.

En este trabajo, se aplica la novedad definida por *Oscar Celma* (Celma, 2010), calculando el promedio de la popularidad sobre los Top-N recomendados, porque hay gran posibilidad de que los artículos menos populares hagan que los usuarios se sientan novedosos. Es decir, si el valor de novedad del modelo es bajo, se recomendarán aquellos productos más populares a usuarios, y viceversa.

Por lo tanto, en primer lugar, es necesario determinar el ranking de popularidad sobre los Top-N recomendados. Luego, se revisa la lista de Top-N de cada usuario, para calcular el promedio de su popularidad. Así, se define la novedad como:

$$Novedad = \frac{\sum_u (\sum_{i \in T_u} rank_i)}{\sum_u N_u}$$

donde $rank_i$ representa el ranking de los artículos; T_u es el conjunto de los Top-N recomendados para el usuario *u*; *i* representa los elementos dentro de T_u; N_u es el número total de los artículos recomendados al usuario *u*.

6.13. Tiempo de ejecución

Con respecto al tiempo de ejecución, en este trabajo, a través del paquete *surprise.model_selection*, se calcula automáticamente el tiempo de entrenamiento (en adelante, Fit Time) para saber cuánto tiempo se ha tardado en entrenar un algoritmo, y el tiempo de prueba (en adelante, Test Time) en la validación cruzada. Cuando se realiza una prueba con los datos de *train-test* una sola vez, se usará el método *time()* del paquete *time*, lo cual se puede consultar más detalles en el Anexo II.

Para resumir, en la fase de evaluación, podrá haber una dificultad, porque conseguir el mínimo error cuadrático o la alta tasa de precisión no sería suficiente para concluir que los usuarios tengan experiencias satisfactorias. En concreto, unos estudios han indicado que a veces las recomendaciones más precisas según las métricas matemáticas no son las más útiles para los usuarios (McNee et al., 2006). Por otro lado, este trabajo solo ha usado dos conjuntos de datos para entrenar los distintos algoritmos, es posible que no sean suficientes para llegar una conclusión muy precisa sobre sus utilidades.

En concreto, el procedimiento de evaluación del comportamiento de un sistema de recomendación no debería limitarse a datos estadísticos. Podría usar otras formas como la encuesta de satisfacción o la prueba A/B para evaluar los algoritmos. Dicho problema se espera resolver en un trabajo futuro.

7. Evaluación experimental

En la sección 7, en primer lugar, se introduce el entorno experimental de este trabajo. Luego, se demuestran los resultados de los algoritmos planteados de este trabajo, de tal manera que se compara sus eficiencias.

7.1. Entorno experimental

Se ha creado un entorno artificial con Anaconda Python 3.6 para simular los modelos, porque existen unas librerías (por ejemplo,tensorflow, keras,etc) que todavía no pueden soportar la última versión Python 3.9.

Por otro lado, el tiempo de ejecución se ve afectado por el sistema y la capacidad de procesador de cada equipamiento, como por ejemplo, los procesadores i7 son más rápidos que los i5.

A continuación, se presenta una tabla en la que se puede conocer el entorno experimental de este trabajo.

Tabla 6: Entorno de experimento

Sistema	Windows 10 Home	
Procesador	i7-10510U CPU @ 1.80GHz 2.30 GHz	
RAM	16G	
Versión de Python y las librerías usadas	Python 3.6.	pandas
	numpy==1.16.0	keras==2.2.5
	scikit-surprise	gym
	sklearn	requests
	tensorflow==1.14.0	surprise
	stable_baselines	matplotlib

Fuente: Elaboración propia

7.2. Resultados de DoubleDQN y DQN

En este apartado, se comparan los resultados de DoubleDQN y DQN de acuerdo con las medidas de evaluación presentadas en la sección 6.

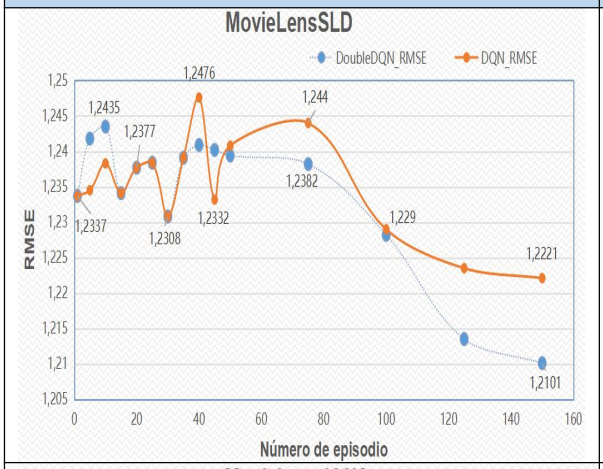
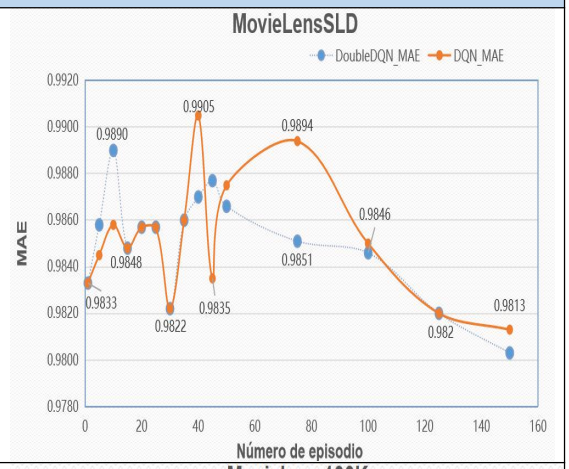
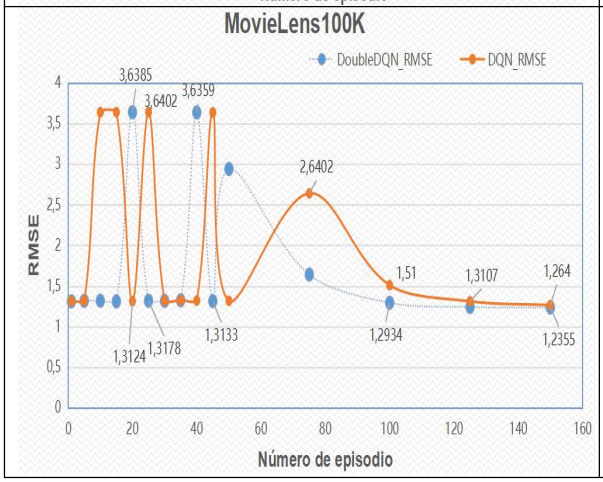
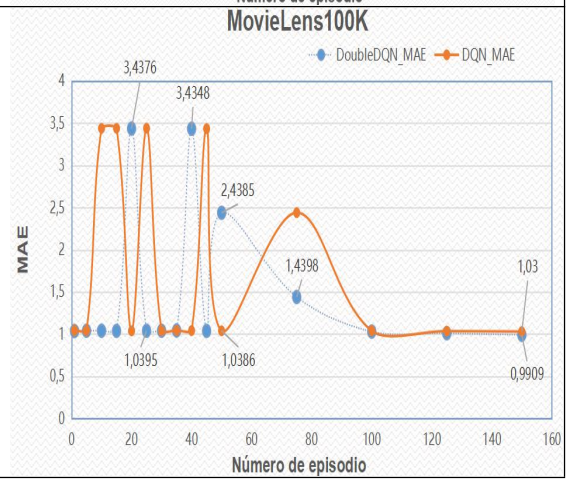
Antes de nada, cabe mencionar que, ha tardado mucho tiempo en ejecutar DoubleDQN y DQN, cuyo tiempo promedio de ejecución puede llegar hasta 40 veces más que el de los algoritmos clásicos de sistema de recomendación. De hecho, pese a que generalmente el DQN tarda menos tiempo de ejecución que el DoubleDQN, ambos algoritmos tardarán más tiempo de entrenar si aumenta el número de episodio. Por ejemplo, si prueba un episodio grande (10000), a lo mejor tarda más de un día para entrenar un modelo.

Por otro lado, debido al límite de la capacidad computacional, no podía fijar un

número demasiado grande de episodio. Por lo tanto, en este trabajo, lo que ha experimentado era probar distintos números de episodio (desde 1 hasta 150) para encontrar el número "óptimo" de episodio con el que el algoritmo tenga una eficiencia más alta.

A continuación, se presenta una tabla (Tabla 7) en la que se compara la evolución de RMSE y MAE de DoubleDQN y DQN.

Tabla 7: RMSE y MAE (DoubleDQN y DQN)

RMSE	MAE
	
	

Fuente: Elaboración propia

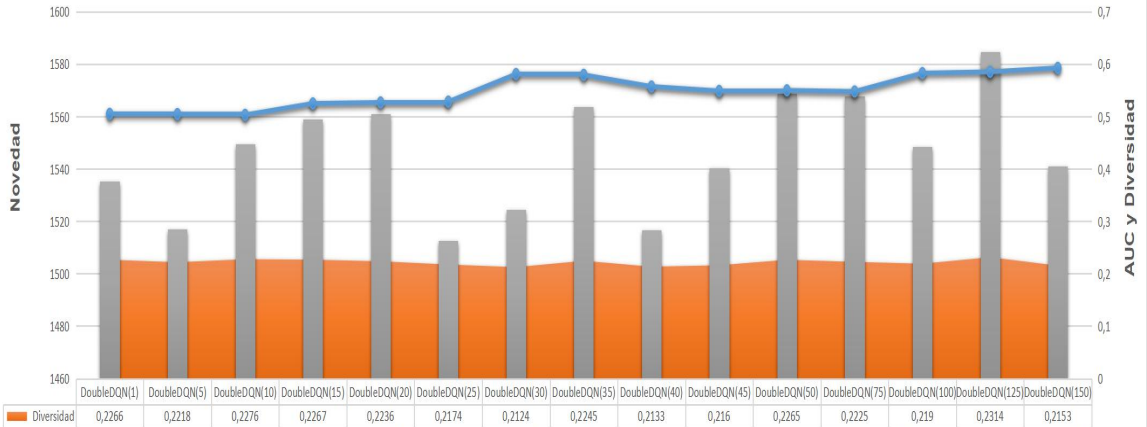
Como se puede observar en la Tabla 7, para el conjunto de datos *MovieLensSLD*, el DoubleDQN comete más error que el DQN, pero desde 15 episodios hasta 35, tienen el mismo nivel de error. Después de 35 episodios, comparando con el DQN, el RMSE y MAE de DoubleDQN no fluctúan mucho, y a partir de 75 episodios tiene una tendencia decreciente. Para el conjunto de datos *MovieLens100K*, desde un principio, el RMSE y MAE de DoubleDQN casi no varían. A partir de 20 episodios, empieza a fluctuar como el DQN, teniendo un nivel de error similar y grande. No obstante, el nivel e error del DoubleDQN empieza a tener tendencia decreciente desde 50 episodios, mientras que el otro desde 75 episodios.

De lo anterior se desprende que, cuando el número de episodio es pequeño, el

DoubleDQN produce un resultado más estable que el DQN, aunque ambos tienen un nivel de error relativamente grande. Mientras que en función del aumento del número de episodio, el RMSE y MAE de ambos algoritmos presentan una tendencia decreciente.

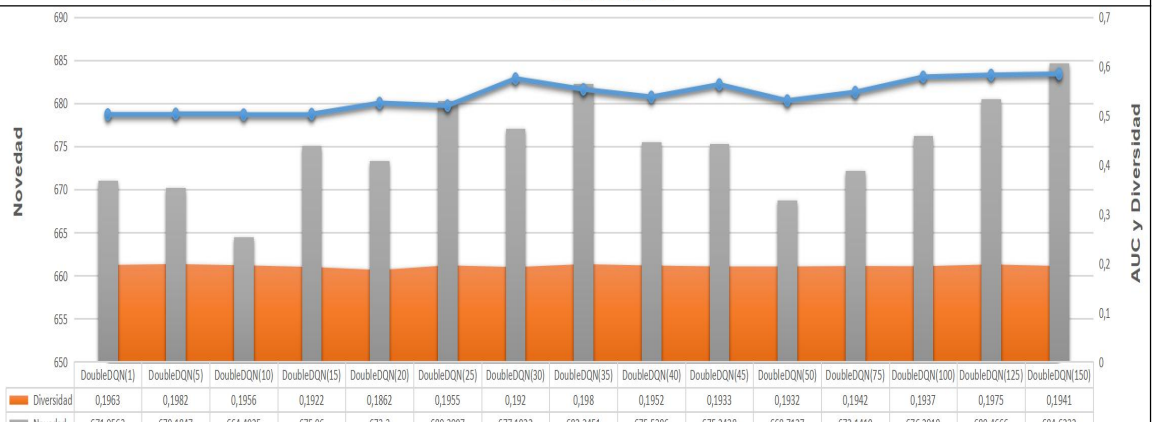
Tabla 8: AUC, Novedad y Diversidad (DoubleDQN y DQN)

DoubleDQN



	DoubleDQN(1)	DoubleDQN(5)	DoubleDQN(10)	DoubleDQN(15)	DoubleDQN(20)	DoubleDQN(25)	DoubleDQN(30)	DoubleDQN(35)	DoubleDQN(40)	DoubleDQN(45)	DoubleDQN(50)	DoubleDQN(75)	DoubleDQN(100)	DoubleDQN(125)	DoubleDQN(150)
Diversidad	0,2266	0,2218	0,2276	0,2267	0,2236	0,2174	0,2124	0,2245	0,2133	0,216	0,2265	0,2225	0,219	0,2314	0,2153
Novedad	1535,3508	1517,008	1549,6995	1559,019	1561,1644	1512,6098	1524,6525	1563,8785	1516,7256	1540,2775	1568,9231	1567,832	1548,4616	1584,6244	1541,1133
AUC	0,5050	0,5046	0,5031	0,5249	0,5268	0,5283	0,5810	0,5795	0,5572	0,5488	0,5499	0,5477	0,5828	0,5853	0,5928

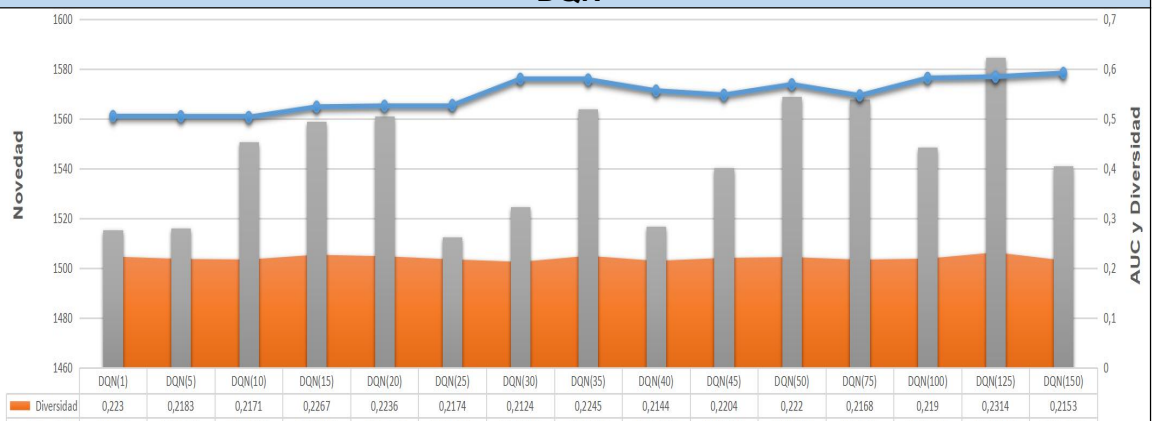
MovieLensSLD



	DoubleDQN(1)	DoubleDQN(5)	DoubleDQN(10)	DoubleDQN(15)	DoubleDQN(20)	DoubleDQN(25)	DoubleDQN(30)	DoubleDQN(35)	DoubleDQN(40)	DoubleDQN(45)	DoubleDQN(50)	DoubleDQN(75)	DoubleDQN(100)	DoubleDQN(125)	DoubleDQN(150)
Diversidad	0,1963	0,1982	0,1956	0,1922	0,1862	0,1955	0,192	0,198	0,1952	0,1933	0,1932	0,1942	0,1937	0,1975	0,1941
Novedad	671,0563	670,1847	664,4925	675,06	673,3	680,3097	677,1022	682,2451	675,5386	675,2438	668,7127	672,1419	676,2818	680,4666	684,6232
AUC	0,5018	0,5030	0,5014	0,5026	0,5250	0,5197	0,5746	0,5528	0,5374	0,5626	0,5300	0,5472	0,5781	0,5819	0,5842

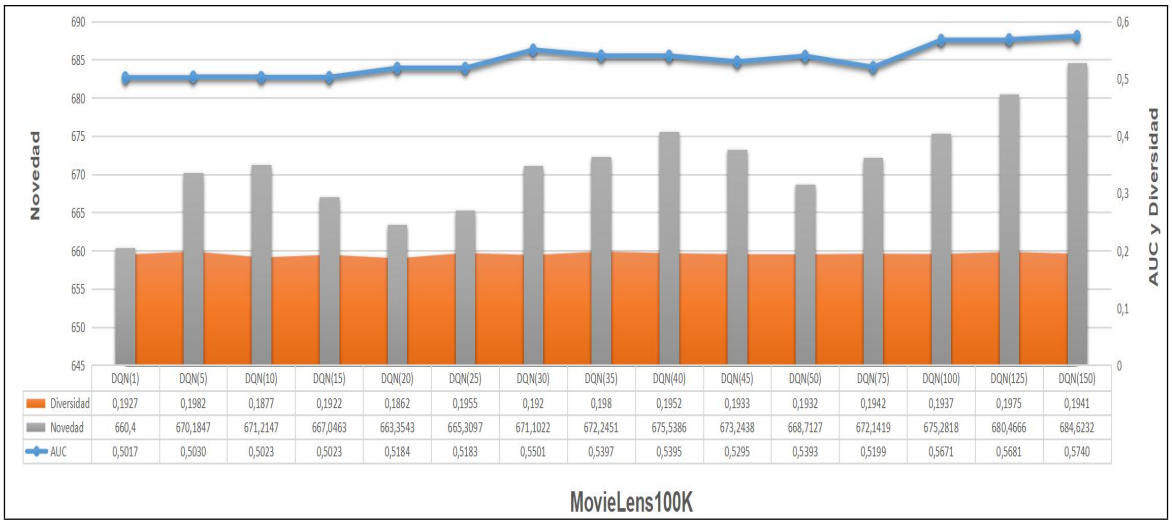
MovieLens100K

DQN



	DQN(1)	DQN(5)	DQN(10)	DQN(15)	DQN(20)	DQN(25)	DQN(30)	DQN(35)	DQN(40)	DQN(45)	DQN(50)	DQN(75)	DQN(100)	DQN(125)	DQN(150)
Diversidad	0,223	0,2183	0,2171	0,2267	0,2236	0,2174	0,2124	0,2245	0,2144	0,2204	0,222	0,2168	0,219	0,2314	0,2153
Novedad	1515,35	1516,008	1550,6995	1559,019	1561,1644	1512,6098	1524,6525	1563,8785	1516,7256	1540,2775	1568,9231	1567,832	1548,4616	1584,6244	1541,1133
AUC	0,5050	0,5047	0,5032	0,5239	0,5258	0,5273	0,5800	0,5785	0,5562	0,5478	0,5689	0,5467	0,5818	0,5843	0,5917

MovieLensSLD



	DQN(1)	DQN(5)	DQN(10)	DQN(15)	DQN(20)	DQN(25)	DQN(30)	DQN(35)	DQN(40)	DQN(45)	DQN(50)	DQN(75)	DQN(100)	DQN(125)	DQN(150)
Diversidad	0,1927	0,1982	0,1877	0,1922	0,1862	0,1955	0,192	0,198	0,1952	0,1933	0,1932	0,1942	0,1937	0,1975	0,1941
Novedad	660,4	670,1847	671,2147	667,0463	663,3543	665,3097	671,1022	672,2451	675,5386	673,2438	668,7127	672,1419	675,2818	680,4666	684,6232
AUC	0,5017	0,5030	0,5023	0,5023	0,5184	0,5183	0,5501	0,5397	0,5395	0,5295	0,5393	0,5199	0,5671	0,5681	0,5740

Fuente: Elaboración propia

Luego, como se puede ver en la Tabla 8, para ambos conjuntos de datos, el DoubleDQN y DQN obtienen un valor de AUC prácticamente igual.

En concreto, el AUC de ambos algoritmos está cerca de 0,6, y tiene tendencia creciente con el incremento del número de episodio. Bajo esta situación, el valor de precisión también presenta una subida constante aunque el resultado de Recall no siempre es así.

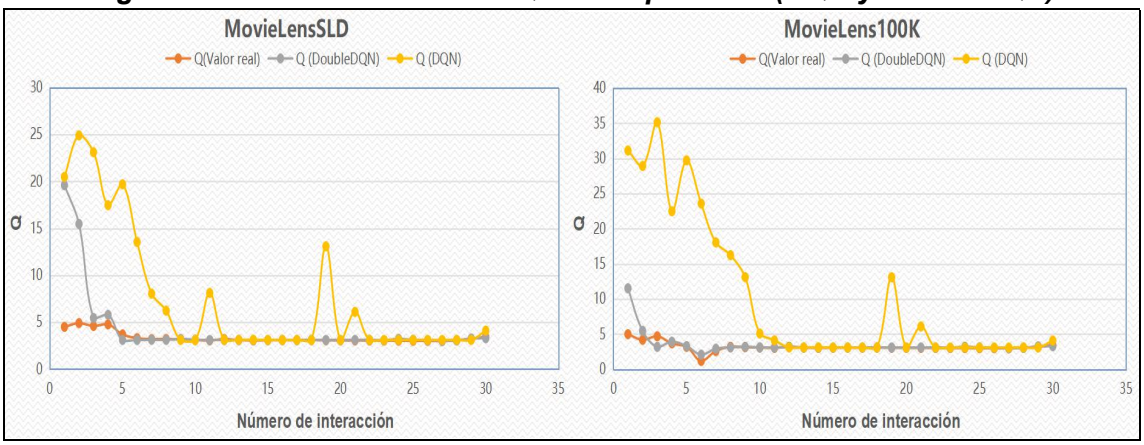
Sobre la novedad, se puede concluir que el DoubleDQN y el DQN tienen la misma capacidad de recomendar las películas desconocidas y novedosas a los usuarios. No obstante, su valor puede variar levemente con distintos números de episodio, por lo que en práctica, es necesario fijar un número adecuado para obtener mayor novedad.

En cuanto a la diversidad, se puede decir que para el conjunto de datos *MovieLensSLD*, su valor oscila entre 0,21 y 0,24. Y para el conjunto de datos *MovieLens100K*, dicho valor baja un poco, fluctuando entre 0,18 y 0,2. Habitualmente, el valor de diversidad no varía mucho con el cambio del número de episodio. Por eso, para una empresa que quiera tardar menor tiempo de ejecución para obtener una mayor diversidad de recomendación, podría fijar un número de episodio pequeño conforme a la potencia computacional de su sistema.

Con respecto al resto de resultados sobre las métricas de evaluación, se puede consultar más detalle en el Anexo III, porque no hay gran diferencia entre los resultados del DoubleDQN y DQN.

En resumen, la principal diferencia entre el DoubleDQN y el DQN es generar distintos niveles de error en función del número de episodio. De hecho, aunque a veces tienen el mismo nivel del error, el DoubleDQN es más estable que el DQN, porque su estimación del valor Q es más precisa y cerca al valor real de rating (Figura 6).

Figura 6: Estimación del valor Q en 30 episodios (DQN y DoubleDQN)



Fuente: Elaboración propia

7.3. Comparación de DoubleDQN con los modelos tradicionales de RS

Se han usado dos conjuntos de datos para comparar el algoritmo DoubleDQN con los modelos clásicos de RS. Aunque ambos son de *MovieLens*, contienen información sobre distintos usuarios, por lo que puede producir algunas divergencias en los resultados experimentales. De hecho, es posible sacar una conclusión distinta sobre la eficiencia algorítmica cuando se aplica otro conjunto de datos.

Por lo tanto, basando en distintos conjuntos, se presentan los principales resultados de todos los algoritmos por separado en la Tabla 9. Se puede consultar los resultados experimentales sobre todas las métricas de evaluación en el Anexo III.

Tabla 9: Resultados de todos los algoritmos

Algoritmo	Precisión	Recall	AUC	Diversidad	Novedad
MovieLensSLD					
MostPopular	66,43%	8,24%	0,5267	0,0489	502,46
RecentPopular	67,89%	8,46%	0,5284	0,0489	502,46
Random	54,51%	14,69%	0,5021	0,0482	814,80
UserCF	83,18%	22,94%	0,5947	0,8882	6126,98
ItemCF	85,07%	14,60%	0,5633	0,7204	6933,13
ItemCFNorm	90,07%	13,86%	0,5634	0,7184	7142,89
TItemCF	88,31%	20,04%	0,5891	0,7514	7041,83
ContentKNN	88,34%	17,93%	0,5800	0,6239	4892,58
SVD	88,14%	20,50%	0,5910	0,0305	519,80
SVD++	88,27%	21,22%	0,5941	0,0740	896,99
TimeSVD++	88,21%	20,86%	0,5943	0,0523	708,39
DoubleDQN*	64,99%	17,8%	0,5810	0,2124	1524,65
DQN*	64,71%	15,9%	0,5689	0,2220	1568,92
MovieLens100K					
MostPopular	58,59%	9,83%	0,5191	0,0631	320,04
RecentPopular	60,25%	10,21%	0,5219	0,0631	320,04
Random	57,43%	9,72%	0,5175	0,0732	355,34
UserCF	78,46%	26,89%	0,5911	0,9934	1541,87
ItemCF	82,75%	16,33%	0,5641	0,5472	1282,77
ItemCFNorm	85,62%	12,27%	0,5512	0,6970	1573,37
TItemCF	84,74%	20,03%	0,5803	0,7700	1509,55
ContentKNN	80,75%	19,30%	0,5721	0,2558	808,41
SVD	84,05%	24,51%	0,5952	0,0282	213,12
SVD++	85,96%	25,42%	0,6021	0,0302	230,49
TimeSVD++	85,01%	25,07%	0,5950	0,0292	221,98
DoubleDQN*	75,41%	15,51%	0,5746	0,1920	677,10
DQN*	73,89%	10,79%	0,5501	0,1920	671,10
*Se ha sacado el mejor resultado según el número de episodio, considerando todas las medidas de evaluación.					

Fuente: Elaboración propia

- **MovielensSLD**

Según la Tabla 9, es obvio que la precisión del DoubleDQN es cerca del 65%, y que es mayor que el algoritmo Random. No obstante, cabe destacar que, el valor de precisión tanto del DoubleDQN, como de DQN, aumenta en función del número de episodio, por lo que tendría la posibilidad de llegar a una precisión más alta que el resto de los algoritmos. Esto ocurre porque la idea del DoubleDQN y DQN es interactuar continuamente con el entorno recomendando distintos productos aleatoriamente con probabilidad decreciente, para conocer las preferencias del usuario. De este modo, el agente recomendador conocería mejor los comportamientos del usuario en caso del aumento de interacción.

Sobre el recall del DoubleDQN, comparando con otros algoritmos, cabe apreciar que es mayor que MostPopular, RecentPopular, Random, ItemCF y ItemCFNorm. En concreto, dicho modelo tiene un recall similar al algoritmo ContentKNN, y es capaz de identificar un 17,8% de las películas que los usuarios interesarían ver. Esto significa que el DoubleDQN es capaz de identificar 1 de cada 5 de las películas que sí gustarían a los usuarios.

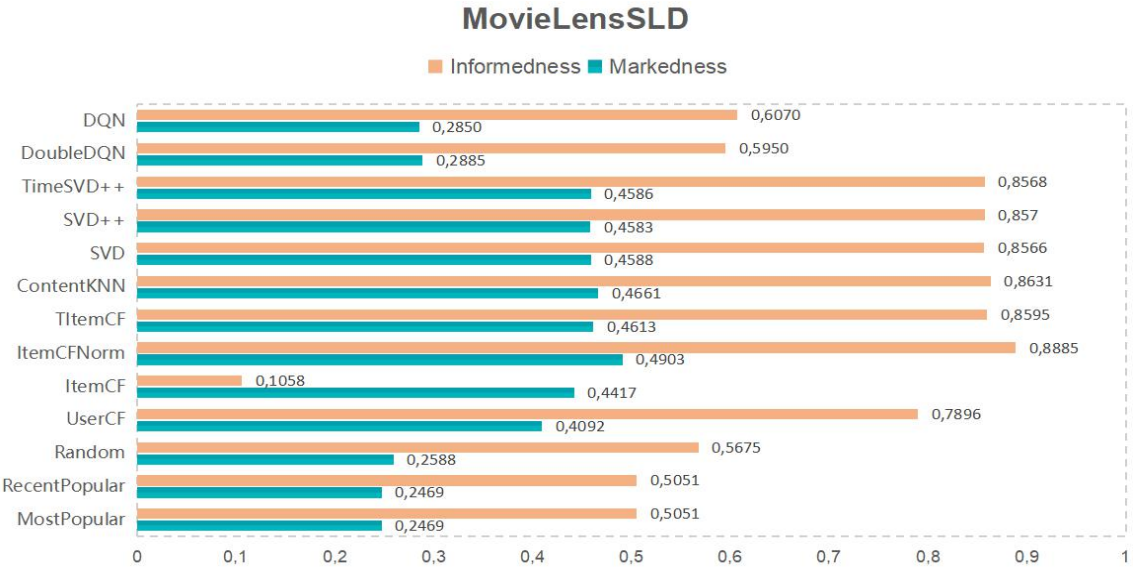
Con respecto al AUC de cada algoritmo, se puede resumir que no existe una gran diferencia entre todos los algoritmos, aunque el UserCF tiene el mayor AUC. En realidad, el AUC del DoubleDQN sólo es mayor que Random, ItemCF, ItemCFNorm, ContentKNN y DQN, pero igual como lo comentado en el apartado 7.2, dicho valor podría aumentar con un número grande de episodio. Por supuesto, si prueba más episodios, tardará más tiempo de entrenar el modelo, lo cual es una gran desventaja tanto del DoubleDQN como del DQN.

Por otra parte, los productos recomendados por el DoubleDQN y DQN son más novedosos que MostPopular, RecentPopular, Random y los algoritmos basados en modelos (SVD, SVD++, TimeSVD++), a pesar de que no tan alto como el nivel de novedad del ItemCFNorm.

Así mismo, se puede sacar la misma conclusión sobre la diversidad del DoubleDQN y DQN como lo anterior. Sin embargo, los otros algoritmos, tales como UserCF y TItemCF, tienen una diversidad extremadamente alta, no es lo ideal porque puede conducir a malas recomendaciones. De esta forma, se puede considerar que el DoubleDQN y DQN han conseguido un nivel de diversidad adecuado para no afectar las experiencias de los consumidores.

Sin perjuicio de lo anterior, si observa la Figura 7, se puede decir que el ItemCF solo está informado bien sobre las predicciones positivas (los artículos que sean preferibles por usuarios) y no las negativas. Es decir, es posible recomendar al mismo tiempo los productos que le puedan interesar y los que no. De hecho, el *Informedness* del DoubleDQN es mayor que ItemCF, Random, RecentPopular y MostPopular, cuyo valor de *Markedness* también es relativamente alto (0,595).

Figura 7: Informedness y Markedness (MovieLensSLD)



Fuente: Elaboración propia

No obstante, el *Informedness* del ItemCFNorm es 0,4903, mayor que el resto de los modelos, y sobre todo, su valor de *Markedness* es el más grande. En realidad, si se comparan todas las métricas de evaluación a la vez, se puede considerar que el ItemCFNorm es el mejor algoritmo de RS para el conjunto de datos *MovielesSLD*. También merece la pena usarlo para ensamblar con el DoubleDQN, a ver si podría mejorar los resultados de recomendación.

- **Movielens100K**

Según la Tabla 9, en cuanto a la comparación de la precisión del DoubleDQN con otros algoritmos, se puede sacar una conclusión un poco distinta del primer conjunto de datos, que la precisión del DoubleDQN solamente es mayor que MostPopular, RecentPopular y Random. Pero igual como lo mencionado anteriormente, que tendría la posibilidad de llegar a una precisión más alta que el resto de los algoritmos si aumenta el número de episodio.

En cuanto a la comparación sobre el recall del DoubleDQN con el resto de los algoritmos, se puede concluir lo mismo como el primer conjunto de datos, aunque en este caso, su recall no es muy similar al algoritmo ContentKNN. En concreto, el DoubleDQN es capaz de identificar un 15,51% de las películas que los usuarios interesarían ver.

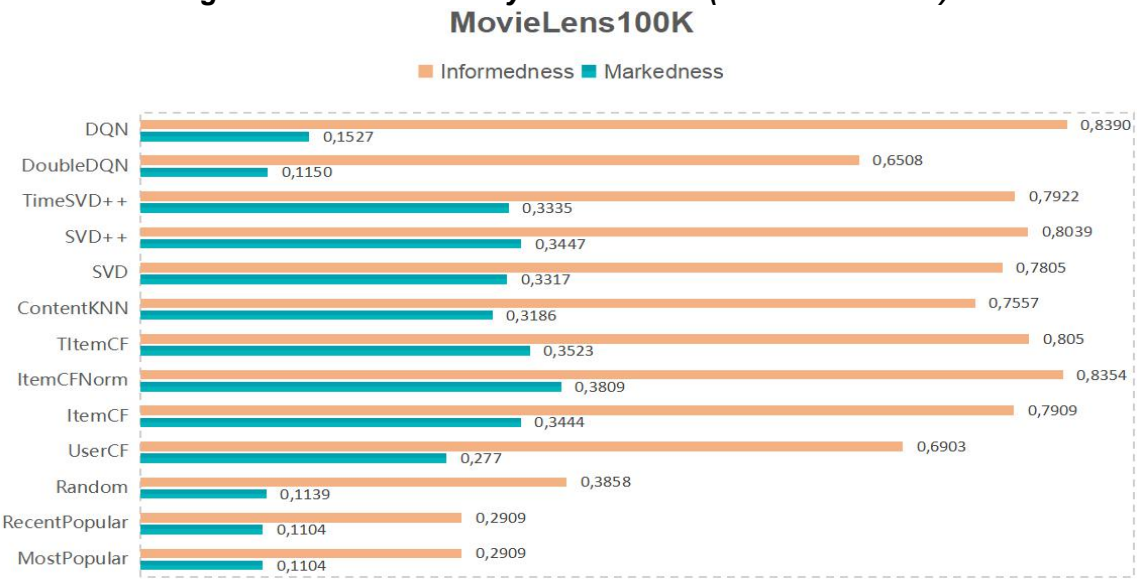
Con relación al AUC de cada algoritmo, se puede concluir que no existe una gran diferencia entre todos los algoritmos, aunque el SVD++ tiene el mayor AUC. En realidad, el AUC del DoubleDQN sólo es mayor que Random, MostPopular, RecentPopular, ItemCF, ItemCFNorm, ContentKNN y DQN. No obstante, de acuerdo con lo comentado anteriormente, dicho valor podría aumentar con un número grande de episodio. Por ejemplo, con 150 episodios, el DoubleDQN puede conseguir una

buena precisión (78,59%), pero tarda mucho más tiempo de ejecución que el SVD++.

Por otra parte, los productos recomendados por el DoubleDQN y DQN son más novedosos que MostPopular, RecentPopular, Random y los algoritmos basados en modelos (SVD, SVD++, TimeSVD++), a pesar de que no tan alto como el nivel de novedad del ItemCFNorm. Igualmente, sobre la diversidad del DoubleDQN, se puede sacar la misma conclusión como lo anterior. De hecho, aunque algunos algoritmos, tales como ItemCFNorm y TItemCF, tienen una diversidad extremadamente alta, no es lo ideal puesto que puede conducir a malas recomendaciones. De este modo, se puede considerar que el DoubleDQN y DQN han conseguido un nivel de diversidad adecuado para no afectar las experiencias de los consumidores.

Además de lo anterior,a partir de la Figura 8, se puede resumir que en este conjunto de datos, el DQN tiene la capacidad de informar bien tanto sobre las predicciones positivas como sobre las negativas, porque su valor de *Informedness* es el más grande (0,839). Mientras que dicho indicador de DoubleDQN no puede llegar ni al 0,7. No obstante, tanto el DQN como el DoubleDQN, tiene un valor de *Markedness* (confiabilidad) cerca del 0,1, eso significa que solo puede confiar en sus predicciones positivas del modelo y no en las negativas. Es decir, el DQN es capaz de informar todas las predicciones, pero solamente son confiables las predicciones positivas.

Figura 8: Informedness y Markedness (MovieLens100K)



Fuente: Elaboración propia

En realidad, se puede considerar que el ItemCFNorm es el mejor algoritmo de RS para el conjunto de datos *Movieles100K*, puesto que su precisión solamente es un poco menor que SVD++ pero su capacidad de recomendar películas novedosas es mucho más alta que la de SVD++. Sobre todo, el *Informedness* del ItemCFNorm es 0,3809, mayor que el resto de los modelos, además, su valor de *Markedness* también es el más grande. Por lo tanto, teniendo en cuenta todas las medidas de evaluación, se puede considerar que el ItemCFNorm es el mejor algoritmo de RS para este conjunto de datos.

7.4. Resultados de los modelos ensamblados

En este trabajo, se han examinado 13 algoritmos de RS, por lo que cada uno de ellos puede apilar con otro algoritmo distinto para crear modelos ensamblados, a ver si es posible aumentar el AUC y reducir el nivel del sesgo de los modelos obtenidos anteriormente.

Igual que el apartado anterior, se han usado dos conjuntos de datos para evaluar los resultados de los modelos ensamblados. Por lo tanto, para cada dataset, se han generado 156 modelos ensamblados, cuyos resultados experimentales sobre todas las métricas de evaluación se puede consultar en el Anexo III. Cabe mencionar que ensamblar ItemCFNorm con ContentKNN no es lo mismo que ensamblar ContentKNN con ItemCFNorm, porque en la primera situación se usa la salida del ItemCFNorm como la entrada del ContentKNN, mientras que en la segunda es lo contrario.

A continuación, sólo se presentan los resultados sobre los modelos ensamblados cuyo AUC es mayor que 0,75, porque en teoría aquellos modelos que cumple este criterio son buenos. Por supuesto, también podría considerarse aquellos modelos que cometen menor error como los mejores. No obstante, el objetivo de este trabajo es resolver el problema de *item cold-start* y sobre todo, las salidas del algoritmo DoubleDQN y DQN son distintas categorías (la probabilidad de que un usuario evalúa una película con 1, 2, 3, 4 y 5) de las películas dadas por el usuario (aprendizaje reforzado combinado con la clasificación), sería mejor evaluar el rendimiento de los modelos a partir del AUC.

Tabla 10: Resultados de los modelos ensamblados

Algoritmo	Precisión	Recall	AUC	Diversidad	Novedad
MovieLensSLD					
ItemCFNorm & ContentKNN	88,92%	23,18%	0,8041	0,6447	5084,8910
SVD* & ItemCF	90,85%	22,96%	0,7918	0,7571	7412,6379
TimeSVD++ &TItemCF	80,06%	15,52%	0,7786	0,2737	1450,6526
DoubleDQN* & ItemCF	68,24%	10,81%	0,7870	0,4765	7382,7030
DoubleDQN* & TItemCF	71,60%	28,32%	0,7710	0,4893	7454,6032
DQN* & TimeSVD++	69,15%	10,82%	0,7675	0,2260	2736,5357
DQN* & ItemCF	65,65%	23,78%	0,7571	0,4765	7382,7032
DoubleDQN* & ItemCFNorm	70,20%	25,32%	0,7559	0,4893	7454,6032
MovieLens100K					
ContentKNN & TItemCF	90,08%	39,80%	0,8564	0,7406	1461,6355
ContentKNN & ItemCFNorm	81,76%	29,19%	0,8310	0,7324	1558,9438
DoubleDQN* & ContentKNN	83,73%	22,72%	0,8230	0,1353	701,2426
DoubleDQN* & ItemCFNorm	82,20%	21,69%	0,8079	0,6448	1573,9375
DoubleDQN* & SVD	81,45%	21,67%	0,8005	0,2653	105,0000
DoubleDQN* & TItemCF	81,05%	20,67%	0,7966	0,6448	1573,9375
DoubleDQN* & TimeSVD++	80,52%	19,66%	0,7914	0,1652	118,2468
DQN* & ItemCFNorm	81,28%	18,86%	0,7664	0,1700	1573,9375

*Se ha sacado el mejor resultado según el número de episodio, considerando todas las medidas de evaluación.

Fuente: Elaboración propia

- **MovielensSLD**

Según la Tabla 10, es obvio que el *Stacking* de los algoritmos de RS ha mejorado su AUC hasta 0,8041, porque el AUC máximo del algoritmo simple no llega al 0,6. Al mismo tiempo, los ensamblados proporcionan mayor diversidad y novedad sobre las películas recomendadas.

Se puede observar que el modelo ensamblado cuyo AUC es el mayor es ItemCFNorm&ContentKNN, lo cual es una forma tradicional para solucionar el problema de *Cold Start*. Aunque su nivel de diversidad no es tan alto como el ensamblado SVD&ItemCF, no afecta mucho a su bondad, porque tampoco es lo ideal obtener una diversidad demasiada alta al poder conducir a malas recomendaciones.

Parece que ensamblar con los algoritmos tradicionales obtengan mejor rendimiento que los ensamblados con DoubleDQN y DQN, porque no sólo aumentan el AUC, sino que también reducen el nivel del sesgo. No obstante, si se fija en la métrica Recall de los ensamblados, se puede decir que, el modelo DoubleDQN&TItemCF es capaz de identificar 28,32% de las películas que los usuarios interesarían ver, casi un 5% más que el ItemCFNorm&ContentKNN. En otras palabras, a pesar de que el DoubleDQN&TItemCF tiene un AUC menor que el ItemCFNorm&ContentKNN, también se puede considerar como un modelo ensamblado bueno.

En suma, para el conjunto de datos *MovielesSLD,* la técnica de *Stacking* ha mejorado el rendimiento de los algoritmos de AUC, demostrando más formas para resolver el problema de *item cold-start*. En concreto, además de las maneras tradicionales (ItemCFNorm&ContentKNN, por ejemplo), se puede incluir el aprendizaje reforzado (DoubleDQN& TItemCF, DoubleDQN &ItemCFNorm, etc.)

- **Movielens100K**

Con respecto al AUC de los modelos ensamblados, se puede concluir lo mismo como el primer conjunto de datos. Sin embargo, los ensamblados ofrecen aproximadamente el mismo grado de novedad que los algoritmos simples, aunque algunos de ellos han conseguido mayor diversidad sobre las películas recomendadas.

Para el conjunto de datos *Movieles100K,* el mejor modelo ensamblado es el ContentKNN&TItemCF, lo cual es una forma tradicional añadiendo el efecto temporal para solucionar el problema de *Cold Start*. En concreto, dicho modelo tiene capacidad de reconocer casi 4 de cada 10 de las películas que sí gustarían a los usuarios, lo cual es una gran mejora sobre el rendimiento de los algoritmos de RS.

Por otro lado, se puede observar que, en este conjunto de datos, aunque el *Stacking* de DoubleDQN con otros algoritmos tradicionales de RS puede aumentar su AUC y precisión, no resulta muy evidente para mejorar su recall (un aumento alrededor del 3%). Por lo tanto, para este dataset, aunque en general la técnica de *Stacking* ha mejorado el rendimiento de los algoritmos de AUC, sería recomendable ensamblar el ContentKNN con TItemCF para resolver el problema de *item cold-start*, en lugar de aplicar el DoubleDQN o DQN.

8. Conclusiones y trabajo futuro

Hoy en día, con la imparable expansión del Internet, la cantidad de información digital disponible está creciendo a una velocidad vertiginosa, causando el problema de la sobrecarga de información. Es decir, frente a miles productos o servicios, los consumidores tienen dificultad para elegir y tomar decisiones. Por eso, surgen los sistemas de recomendación, cuyo objetivo es simplificar el volumen de datos y proponer a los usuarios productos o servicios en base a sus preferencias.

Sin embargo, aunque los sistemas de recomendación han sido aplicados en muchas plataformas famosas (*Amazon, Netflix, Spotify, Aliexpress*, etc), el *Cold-start* todavía es un problema que influye en su rendimiento cuando faltan calificaciones de los productos o hay usuarios que no han apuntado ningún rating. Otro problema es los cambios dinámicos de las preferencias de los usuarios, porque la mayoría de los modelos del filtrado colaborativo no tienen en cuenta el efecto temporal, aunque se han sido comprobado varios beneficios de incluirlo.

En consecuencia, en este trabajo se ha diseñado un novedoso algoritmo de recomendación basado en el aprendizaje por refuerzo (DoubleDQN), para encontrar automáticamente la estrategia óptima de recomendación mediante las interacciones continúas con los usuarios. Al mismo tiempo, se ha comparado su eficiencia con los modelos clásicos de sistemas de recomendación, con el propósito de verificar si éste tuviera capacidad de resolver el problema de *item cold-start y* el efecto temporal de las recomendaciones.

A partir de analizar los resultados experimentales de 2 conjuntos de datos de *MovieLens*, se puede decir que el algoritmo propuesto (DoubleDQN) supera a algunos algoritmos de sistemas de recomendación (MostPopular, RecentPopular, Random, ItemCF y ItemCFNorm) en cuanto a la capacidad de identificar las películas que los usuarios interesarían ver. Y dicha capacidad tiene posibilidad de crecer con el aumento de episodio, a costa de tardar más tiempo computacional. Es una gran desventaja tanto del DoubleDQN como del DQN, mientras que los algoritmos clásicos de sistemas de recomendación son más rápidos.

Por otro lado, el DoubleDQN ha conseguido un aumento de novedad sobre las películas recomendadas y un nivel de diversidad adecuado para no afectar las experiencias de los consumidores. No obstante, con respecto al AUC, se puede resumir que no existe una gran diferencia entre todos los algoritmos, porque casi todos tienen un AUC muy cerca del 0,6. Esto ocurre porque los dos conjuntos de datos utilizados tienen el problema de *Cold-start*, con un solo algoritmo no es suficiente para resolverlo, aunque ha mejorado unos resultados de evaluación (recall, novedad, diversidad, etc).

Por lo tanto, se han ensamblado los algoritmos con la técnica *Stacking*, para aumentar el AUC y reducir el nivel del sesgo de los modelos obtenidos hasta ahora. Con esta técnica, se han conseguido unos modelos con AUC relativamente alto:

- ContentKNN&TItemCF (AUC=0,8564)

- ContentKNN&ItemCFNorm (AUC=0,831)
- DoubleDQN&ContentKNN (AUC=0,8230)
- ItemCFNorm&ContentKNN (AUC=0,8041)
- DoubleDQN&TItemCF (AUC=0,771)
- DoubleDQN&ItemCFNorm (AUC=0,7559)

En definitiva, los ensamblados con los algoritmos clásicos tienen mayor AUC, comprobando que las formas tradicionales resultan útiles para resolver el problema de *item cold-start*. Además, también podría incluirse el aprendizaje reforzado (DoubleDQN) para resolver dicho problema, aunque su AUC es un poco pequeño y tarda más tiempo de ejecución.

Con todo lo anterior, se puede concluir que este trabajo ha cumplido todos los objetivos planteados, y también ha demostrado la factibilidad sobre la aplicación del aprendizaje reforzado en los sistemas de recomendación para aliviar el problema de *Cold-Start.*

Desde luego, la conclusión de este trabajo se ha obtenido por sólo 2 conjuntos de datos offline, por lo que no es suficiente para determinar el mejor o mejores algoritmos. Sobre todo, en la vida real, la selección de algoritmos adecuados no sólo depende de los indicadores estadísticos de evaluación, sino también de las características de cada empresa.

A continuación, se proponen unas formas para mejorar y continuar este estudio en el futuro:

- Llevar a cabo más experimentos sobre DoubleDQN y DQN con distintos números de episodios, porque en este trabajo no podía fijar un número demasiado grande de episodio por la limitación computacional y temporal.
- Extender el método propuesto mediante la integración de la información contextual.
- Comparar el rendimiento de DoubleDQN con los modelos supervisados.
- Probar otro conjunto de datos con el problema de *Cold-Start*, cuya calificación es implícita.
- Aplicar otras medidas de similaridad para determinar la similitud entre los usuarios y *items*, así como comparar las diferencias de los resultados de los algoritmos.
- Evaluar el rendimiento de los algoritmos de sistemas de recomendación con métricas no matemáticas, porque conseguir el mínimo error cuadrático o la alta AUC no sería suficiente para concluir que los usuarios tengan experiencias satisfactorias.

9. Bibliografía

(1) Adomavicius, G., & Tuzhilin, A. (2005). Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, *17*(6), 734-749. https://doi.org/10.1109/TKDE.2005.99

(2) Aggarwal, C. C. (2016). *Recommender Systems: The Textbook*. Springer International Publishing. https://doi.org/10.1007/978-3-319-29659-3

(3) Alfrick Opidi. (2019, junio 26). *5 Ways Tech Companies Apply Reinforcement Learning To Marketing*. TOPBOTS. http://www.topbots.com/reinforcement-learning-in-marketing/

(4) Association for Computing Machinery (ACM). (2019, marzo 28). *«Reinforcement Learning for Recommender Systems: A Case Study on Youtube» by Minmin Chen*. https://www.youtube.com/watch?v=HEqQ2_1XRTs

(5) Balabanovic, M., & Shoham, Y. (1997). *Fab: Content-based, collaborative recommendation.(Special Section: Recommender Systems)*. Undefined. /paper/Fab%3A-content-based%2C-collaborative-Section%3A-Systems)-Balabanovic-Shoham/8dd8dd27a73d63135e9f2cfaf39e2b4339b394c8

(6) Bao, X., Bergman, L., & Thompson, R. (2009). Stacking recommendation engines with additional meta-features. *Proceedings of the third ACM conference on Recommender systems*, 109-116. https://doi.org/10.1145/1639714.1639734

(7) Barjasteh, I., Forsati, R., Masrour, F., Esfahanian, A.-H., & Radha, H. (2015). Cold-Start Item and User Recommendation with Decoupled Completion and Transduction. *Proceedings of the 9th ACM Conference on Recommender Systems*, 91-98. https://doi.org/10.1145/2792838.2800196

(8) Betancur, D. B., Julián Moreno, & Demetrio A. Ovalle. (2009). Modelo para la recomendación y recuperación de objetos de aprendizaje en entornos virtuales de enseñanza/aprendizaje. *Revista Avances en Sistemas e Informática*, *6*(1), 45-56.

(9) Beutel, A., Covington, P., Jain, S., Xu, C., Li, J., Gatto, V., & Chi, E. H. (2018). Latent Cross: Making Use of Context in Recurrent Recommender Systems. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 46-54. https://doi.org/10.1145/3159652.3159727

(10) Bobadilla, J., Alonso, S., & Hernando, A. (2020). Deep Learning Architecture for Collaborative Filtering Recommender Systems. *Applied Sciences*, *10*(7), 2441. https://doi.org/10.3390/app10072441

(11) Bobadilla Sancho, J., Ortega Requena, F., Hernando Esteban, A., & Bernal Bermúdez, J. (2012). A collaborative filtering approach to mitigate the new user cold start problem. *Knowledge-Based Systems*, *26*, 225-238.

(12) Burke, R. (2002). Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction*, *12*. https://doi.org/10.1023/A:1021240730564

(13) Camilo Antonio Ramírez Morales. (2018). Algoritmo SVD aplicado a los sistemas de recomendación en el comercio | Tecnología Investigación y Academia. *2017*. Recuperado 1 de abril de 2021, de https://revistas.udistrital.edu.co/index.php/tia/article/view/11827

(14) Celma, Ò. (2010). *Music Recommendation and Discovery: The Long Tail, Long Fail, and Long Play in the Digital Music Space*. Springer-Verlag. https://doi.org/10.1007/978-3-642-13287-2

(15) Dureddy, H. V., & Kaden, Z. (2018). Handling Cold-Start Collaborative Filtering with Reinforcement Learning. *arXiv:1806.06192 [cs]*. http://arxiv.org/abs/1806.06192

(16) Fouss, F., Pirotte, A., Renders, J., & Saerens, M. (2007). Random-Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation. *IEEE Transactions on Knowledge and Data Engineering*, *19*(3), 355-369. https://doi.org/10.1109/TKDE.2007.46

(17) Fu, J., Kumar, A., Soh, M., & Levine, S. (2019). Diagnosing Bottlenecks in Deep Q-learning Algorithms. *arXiv:1902.10250 [cs, stat]*. http://arxiv.org/abs/1902.10250

(18) Golbandi, N., Koren, Y., & Lempel, R. (2011). Adaptive bootstrapping of recommender systems using decision trees. *Proceedings of the fourth ACM international conference on Web search and data mining*, 595-604. https://doi.org/10.1145/1935826.1935910

(19) Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, *35*(12), 61-70. https://doi.org/10.1145/138859.138867

(20) Guibing Guo. (2012). *[PDF] Improving the Performance of Recommender Systems by—Free Download PDF*. https://nanopdf.com/download/improving-the-performance-of-recommender-systems-by_pdf

(21) Guo, G., Zhu, F., Qu, S., & Wang, X. (2018). PCCF: Periodic and continual temporal co-factorization for recommender systems. *Information Sciences*, *436-437*, 56-73. https://doi.org/10.1016/j.ins.2018.01.019

(22) Hado van Hasselt. (2010). *DoubleQ-learning Hado van Hasselt Multi-agent and Adaptive Computation Group*.

(23) Hado van Hasselt, H., Guez, A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. *arXiv:1509.06461 [cs]*. http://arxiv.org/abs/1509.06461

(24) Hand, D. J. (2009). Measuring classifier performance: A coherent alternative to the area under the ROC curve. *Machine Learning*, *77*(1), 103-123. https://doi.org/10.1007/s10994-009-5119-5

(25) Hannech, A., Adda, M., & Mcheick, H. (2016). Cold-start recommendation strategy based on social graphs. *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 1-7. https://doi.org/10.1109/IEMCON.2016.7746324

(26) Hu, R., Dou, W., & Liu, J. (2014). ClubCF: A Clustering-Based Collaborative

Filtering Approach for Big Data Application. *IEEE Transactions on Emerging Topics in Computing*, *2*(3), 302-313. https://doi.org/10.1109/TETC.2014.2310485

(27) Ian MacKenzie et al. (2013). *How retailers can keep up with consumers | McKinsey*. https://www.mckinsey.com/industries/retail/our-insights/how-retailers-can-keep-up-with-consumers#

(28) Jeff Schneider. (1997). *Cross Validation*. http://www.cs.cmu.edu/~schneide/tut5/node42.html

(29) Johnson, R. M. (1963). On a theorem stated by eckart and young. *Psychometrika*, *28*(3), 259-263. https://doi.org/10.1007/BF02289573

(30) Karypis, G. (2001). Evaluation of Item-Based Top-*N* Recommendation Algorithms. *Proceedings of the tenth international conference on Information and knowledge management*, 247-254. https://doi.org/10.1145/502585.502627

(31) Kohrs, A., & Merialdo, B. (1999). Clustering for Collaborative Filtering Applications. *In Computational Intelligence for Modelling, Control & Automation. Ios*.

(32) Konstan, J. A., & Riedl, J. (2012). Recommender systems: From algorithms to user experience. *User Modeling and User-Adapted Interaction*, *22*(1), 101-123. https://doi.org/10.1007/s11257-011-9112-x

(33) Koren, Y. (2009). *Collaborative Filtering with Temporal Dynamics*. 9.

(34) Koren, Y., Bell, R., & Volinsky, C. (2009). *Matrix Factorization Techniques for Recommender Systems*.

(35) Li, B., Zhu, X., Li, R., Zhang, C., Xue, X., & Wu, X. (2011). Cross-domain collaborative filtering over time. *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Three*, 2293-2298.

(36) Li, Q., & Kim, B. M. (2003). An Approach for Combining Content-based and Collaborative Filters. *Proceedings of the Sixth International Workshop on Information Retrieval with Asian Languages*, 17-24. https://doi.org/10.3115/1118935.1118938

(37) Lian, D., Zhang, Z., Ge, Y., Zhang, F., Yuan, N. J., & Xie, X. (2016). Regularized Content-Aware Tensor Factorization Meets Temporal-Aware Location Recommendation. *2016 IEEE 16th International Conference on Data Mining (ICDM)*, 1029-1034. https://doi.org/10.1109/ICDM.2016.0131

(38) Lobo, J. M., Jiménez‐Valverde, A., & Real, R. (2008). AUC: A misleading measure of the performance of predictive distribution models. *Global Ecology and Biogeography*, *17*(2), 145-151. https://doi.org/10.1111/j.1466-8238.2007.00358.x

(39) Mahmood, T., & Ricci, F. (2009). Improving recommender systems with adaptive conversational strategies. *Proceedings of the 20th ACM conference on Hypertext and hypermedia*, 73-82. https://doi.org/10.1145/1557914.1557930

(40) Mary, J., Gaudel, R., & Preux, P. (2015). Bandits and Recommender Systems. En P. Pardalos, M. Pavone, G. M. Farinella, & V. Cutello (Eds.), *Machine Learning, Optimization, and Big Data* (Vol. 9432, pp. 325-336). Springer International Publishing. https://doi.org/10.1007/978-3-319-27926-8_29

(41) McNee, S. M., Riedl, J., & Konstan, J. A. (2006). Being accurate is not enough: How accuracy metrics have hurt recommender systems. *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, 1097-1101. https://doi.org/10.1145/1125451.1125659

(42) Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529-533. https://doi.org/10.1038/nature14236

(43) National Institute of Technology Patna, Tewari, A., Barman, A., & National Institute of Technology Patna. (2017). Collaborative Recommendation System Using Dynamic Content based Filtering, Association Rule Mining and Opinion Mining. *International Journal of Intelligent Engineering and Systems*, *10*(5), 57-66. https://doi.org/10.22266/ijies2017.1031.07

(44) Park, S.-T., & Chu, W. (2009). Pairwise preference regression for cold-start recommendation. *Proceedings of the third ACM conference on Recommender systems*, 21-28. https://doi.org/10.1145/1639714.1639720

(45) Paterek, A. (2007). *Improving regularized singular value decomposition for collaborative filtering*. 4.

(46) Prando, A. V., Contratres, F. G., Souza, S. N. A., & de Souza, L. S. (2017). Content-based Recommender System using Social Networks for Cold-start Users: *Proceedings of the 9th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, 181-189. https://doi.org/10.5220/0006496301810189

(47) Qian, T., Liang, Y., & Li, Q. (2020). Solving Cold Start Problem in Recommendation with Attribute Graph Neural Networks. *arXiv:1912.12398 [cs]*. http://arxiv.org/abs/1912.12398

(48) Radlinski, F., Bennett, P. N., Carterette, B., & Joachims, T. (2009). Redundancy, diversity and interdependent document relevance. *ACM SIGIR Forum*, *43*(2), 46-52. https://doi.org/10.1145/1670564.1670572

(49) Resnick, P., & Varian, H. R. (1997). Recommender systems. *Communications of the ACM*, *40*(3), 56-58. https://doi.org/10.1145/245108.245121

(50) Ricci, F., Rokach, L., & Shapira, B. (Eds.). (2015). *Recommender Systems Handbook*. Springer US. https://doi.org/10.1007/978-1-4899-7637-6

(51) *SAS Help Center*. (2017). https://documentation.sas.com/?docsetId=emref&docsetTarget=n061bzurmej4j3n1jnj8bbjjm1a2.htm&docsetVersion=14.3&locale=en

(52) Saveski, M., & Mantrach, A. (2014). Item cold-start recommendations: Learning local collective embeddings. *Proceedings of the 8th ACM Conference on Recommender systems*, 89-96. https://doi.org/10.1145/2645710.2645751

(53) Schafer, J. B., Frankowski, D., Herlocker, J., & Sen, S. (2007). Collaborative

Filtering Recommender Systems. En P. Brusilovsky, A. Kobsa, & W. Nejdl (Eds.), *The Adaptive Web: Methods and Strategies of Web Personalization* (pp. 291-324). Springer. https://doi.org/10.1007/978-3-540-72079-9_9

(54) Schein, A. I., Popescul, A., Ungar, L. H., & Pennock, D. M. (2002). Methods and metrics for cold-start recommendations. *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR '02*, 253. https://doi.org/10.1145/564376.564421

(55) SEBASTIAN RUDER. (2016, enero 19). *An overview of gradient descent optimization algorithms*. Sebastian Ruder. https://ruder.io/optimizing-gradient-descent/

(56) Shani, G., Brafman, R., & Heckerman, D. (2012). An MDP-based recommender system. *Journal of Machine Learning Research*, *6*.

(57) Shani, G., Heckerman, D., & Brafman, R. I. (2005). *An MDP-Based Recommender System*. 31.

(58) Sharma, M., Zhou, J., Hu, J., & Karypis, G. (2015). Feature-based factorized Bilinear Similarity Model for Cold-Start Top-n Item Recommendation. *Proceedings of the 2015 SIAM International Conference on Data Mining*, 190-198. https://doi.org/10.1137/1.9781611974010.22

(59) *The model_selection package—Surprise 1 documentation*. https://surprise.readthedocs.io/en/stable/model_selection.html#cross-validation

(60) Volkovs, M., Yu, G. W., & Poutanen, T. (2017). Content-based Neighbor Models for Cold Start in Recommender Systems. *Proceedings of the Recommender Systems Challenge 2017 on ZZZ - RecSys Challenge '17*, 1-6. https://doi.org/10.1145/3124791.3124792

(61) Wang, X., He, X., Wang, M., Feng, F., & Chua, T.-S. (2019). Neural Graph Collaborative Filtering. *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 165-174. https://doi.org/10.1145/3331184.3331267

(62) Wei, D., & Junliang, C. (2013). The Bayesian Network and Trust Model Based Movie Recommendation System. En Z. Du (Ed.), *Intelligence Computation and Evolutionary Computation* (pp. 797-803). Springer. https://doi.org/10.1007/978-3-642-31656-2_107

(63) Weng, L., Xu, Y., Li, Y., & Nayak, R. (2008). Exploiting Item Taxonomy for Solving Cold-Start Problem in Recommendation Making. *2008 20th IEEE International Conference on Tools with Artificial Intelligence*, *2*, 113-120. https://doi.org/10.1109/ICTAI.2008.97

(64) Wiering, M., Van Hasselt, H., Pietersma, A.-D., & Schomaker, L. (2011). *Reinforcement learning algorithms for solving classification problems*. 91-96. https://doi.org/10.1109/ADPRL.2011.5967372

(65) Xian, Z., Li, Q., Li, G., & Li, L. (2017, marzo 19). *New Collaborative Filtering Algorithms Based on SVD++ and Differential Privacy* [Research Article].

Mathematical Problems in Engineering; Hindawi. https://doi.org/10.1155/2017/1975719

(66) Xue, G.-R., Lin, C., Yang, Q., Xi, W., Zeng, H.-J., Yu, Y., & Chen, Z. (2005). Scalable collaborative filtering using cluster-based smoothing. *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, 114-121. https://doi.org/10.1145/1076034.1076056

(67) Yu, H., & Li, J.-H. (2015). Algorithm to solve the cold-start problem in new item recommendations. *Ruan Jian Xue Bao/Journal of Software*, *26*, 1395-1408. https://doi.org/10.13328/j.cnki.jos.004587

(68) Zhang, Z.-K., Liu, C., Zhang, Y.-C., & Zhou, T. (2012). Solving the Cold-Start Problem in Recommender Systems with Social Tags. *Expert Systems with Applications*, *39*(12), 10990-11000. https://doi.org/10.1016/j.eswa.2012.03.025

(69) Zhao, X., Xia, L., Zhang, L., Ding, Z., Yin, D., & Tang, J. (2018). Deep reinforcement learning for page-wise recommendations. *Proceedings of the 12th ACM Conference on Recommender Systems*, 95-103. https://doi.org/10.1145/3240323.3240374

(70) Zhao, X., Zhang, L., Ding, Z., Xia, L., Tang, J., & Yin, D. (2018). Recommendations with Negative Feedback via Pairwise Deep Reinforcement Learning. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 1040-1048. https://doi.org/10.1145/3219819.3219886

(71) Zhao, X., Zhang, L., Xia, L., Ding, Z., Yin, D., & Tang, J. (2019). Deep Reinforcement Learning for List-wise Recommendations. *arXiv:1801.00209 [cs, stat]*. http://arxiv.org/abs/1801.00209

(72) Zhi-Peng Zhang & Yasuo Kudo, Tetsuya Murai, Yong-Gong Ren. (2019). *Addressing Complete New Item Cold-Start Recommendation: A Niche Item-Based Collaborative Filtering via Interrelationship Mining*.

(73) Zhu, Y.-X., & Lü, L.-Y. (2012). Evaluation metrics for recommender systems. *Dianzi Keji Daxue Xuebao/Journal of the University of Electronic Science and Technology of China*, *41*, 163-175. https://doi.org/10.3969/j.issn.1001-0548.2012.02.001

10. Anexos

Anexo I Lista de acrónimos

Adagrad: Adaptive gradient algorithm

Adadelta: Adaptive Learning Rate Method

Adam: ADAptative Moment estimation

BGD: Batch Gradient Descent

DQN: Deep Q Network

DoubleDQN: Double Deep Q-Netwoks

MAE: Mean Absolute Error

MDP: Markov Decission Process

MBGD: Mini-Batch Gradient Descent

MSE: Mean Squared Error

NAG: Nesterov Accelerated Gradient

ReLU: Rectified Linear Unit

RL: Reinforcement Learning

RMSProp: Root Mean Square Propagation.

RS: Recommender System

SGD: Stochastic Gradient Descent

SVD: Singular Value Decomposition

Anexo II Códigos de Python

1. Main.py (incluye validación cruzada repetida)

```
#Librerías
import random
import numpy as np
from surprise import accuracy
from surprise.model_selection import RepeatedKFold
from surprise.model_selection import train_test_split
from surprise.model_selection import cross_validate
from surprise.prediction_algorithms.predictions import Prediction
from AllAlgorithms.ItemCF import ItemCF
from AllAlgorithms.UserCF import UserCF
from AllAlgorithms.ContentKNN import ContentKNNAL
from AllAlgorithms.SVD import SVDAL
from AllAlgorithms.SVDpp import SVDppAL
from AllAlgorithms.TimeSVDpp import TimeSVDpp
from AllAlgorithms.Random import RandomAL
from AllAlgorithms.TItemCF import TItemCF
from AllAlgorithms.ItemCFNorm import ItemCFNorm
from AllAlgorithms.RecentPopular import RecentPopular
from AllAlgorithms.MostPopular import MostPopular
from DDQNModule.ddqn import DDQNAL
from DDQNModule.dqn import DQNAL
from BaseModules.DataCheck import checkData
from BaseModules.DataCheck import viewData
from BaseModules.MovieLens import MovieLens
from BaseModules.Evaluator import Evaluator
from StackingRS import Stacking

def chooseModel(modelName):
    if modelName == 'MostPopular':
        model = MostPopular()
    elif modelName == 'RecentPopular':
        model = RecentPopular()
    elif modelName == 'Random':
        model = RandomAL()
    elif modelName == 'ItemCF':
        model = ItemCF()
    elif modelName == 'TItemCF':
        model = TItemCF()
    elif modelName == 'ItemCFNorm':
        model = ItemCFNorm()
    elif modelName == 'UserCF':
```

```
            model = UserCF()
        elif modelName == 'ContentKNN':
            model = ContentKNNAL()
        elif modelName == 'SVD':
            model = SVDAL()
        elif modelName == 'SVDpp':
            model = SVDppAL()
        elif modelName == 'TimeSVDpp':
            model = TimeSVDpp()
        elif modelName == 'DoubleDQN':
            model = DDQNAL()
        elif modelName == 'DQN':
            model = DQNAL()
        else:
            print('Wrong model!')
            return
        return model

###### MODELO SIMPLE SIN Stacking ######
#1. Validación Cruzada (sólo una vez)
def testAllSingleModel():
        def LoadMovieLensData():
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall()     #MovieLensSLD
            #data = ml.loadMovieLens100k() #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        evaluator = Evaluator(evaluationData, rankings)
        # lista de los modelos
        MODELS = ["MostPopular", "RecentPopular", "Random", "ItemCF", "TItemCF",
"ItemCFNorm", "UserCF", "ContentKNN","SVD", "SVDpp", "TimeSVDpp", "DoubleDQN",
"DQN"]
        for modelName in MODELS:
            model = chooseModel(modelName)
            evaluator.AddAlgorithm(model.algo, model.name)
        evaluator.Evaluate(True)

#2. Validación Cruzada Repetida (k=5, rept=5)
#Forma 1: cross_validate
def testAllCrossValidate():
```

```
    def LoadMovieLensData():
        ml = MovieLens()
        data = ml.loadMovieLensLatestSmall()   #MovieLensSLD
        #data = ml.loadMovieLens100k() #MovieLens100K
        rankings = ml.getPopularityRanks()
        return (ml, data, rankings)

    np.random.seed(0)
    random.seed(0)
    (ml, evaluationData, rankings) = LoadMovieLensData()
    evaluator = Evaluator(evaluationData, rankings)
    #lista de los modelos
    MODELS =["MostPopular", "RecentPopular", "Random", "ItemCF", "TItemCF",
"ItemCFNorm", "UserCF", "ContentKNN","SVD", "SVDpp", "TimeSVDpp", "DoubleDQN",
"DQN"]
    for modelName in MODELS:
        model = chooseModel(modelName)
        rept=5   #repite 5 veces
        while rept>0:
            cross_validate(model.algo, evaluationData, measures=['RMSE', 'MAE'],
cv=5, verbose=True)
            rept -=1
        evaluator.AddAlgorithm(model.algo, model.name)
        evaluator.Evaluate(True)

#Forma 2: RepeatedKFold
def CrossValidateRepetida():
    def LoadMovieLensData():
        ml = MovieLens()
        data = ml.loadMovieLensLatestSmall()    #MovieLensSLD
        #data = ml.loadMovieLens100k() #MovieLens100K
        rankings = ml.getPopularityRanks()
        return (ml, data, rankings)

    np.random.seed(0)
    random.seed(0)
    (ml, evaluationData, rankings) = LoadMovieLensData()
    evaluator = Evaluator(evaluationData, rankings)
    # lista de los modelos
    MODELS = ["MostPopular", "RecentPopular", "Random", "ItemCF", "TItemCF",
"ItemCFNorm", "UserCF", "ContentKNN","SVD", "SVDpp", "TimeSVDpp", "DoubleDQN",
"DQN"]
    for modelName in MODELS:
        model = chooseModel(modelName)
```

```
        kf=RepeatedKFold(n_splits=5,n_repeats=5)
        for trainset, testset in kf.split(evaluationData):
            model.fit(trainset)
            predictions = model.test(testset)
            mae=accuracy.mae(predictions, verbose=False)
            rmse=accuracy.rmse(predictions, verbose=False)
            print("MAE del algoritmo {}: {}".format(modelName, mae))
            print("RMSE del algoritmo {}: {}". format(modelName,rmse))
        evaluator.AddAlgorithm(model.algo, model.name)
        evaluator.Evaluate(True)


###### MODELO ENSAMBLADO ######
# para ensamblar los modelos automáticamente, en total hay 156 modelos
def testAllStackingModel():
    def LoadMovieLensData():
        ml = MovieLens()
        data = ml.loadMovieLensLatestSmall()   #MovieLensSLD
        #data = ml.loadMovieLens100k() #MovieLens100K
        rankings = ml.getPopularityRanks()
        return (ml, data, rankings)

    np.random.seed(0)
    random.seed(0)
    (ml, evaluationData, rankings) = LoadMovieLensData()
    evaluator = Evaluator(evaluationData, rankings)
    # lista de los modelos
    MODELS = ["MostPopular", "RecentPopular", "Random", "ItemCF", "TItemCF",
"ItemCFNorm", "UserCF", "ContentKNN","SVD", "SVDpp", "TimeSVDpp", "DoubleDQN",
"DQN"]

    for i in range(13):
        modelName_a=MODELS[i]
        model_a = chooseModel(modelName_a)   # primer modelo
        for modelName_b in MODELS:
            if modelName_a == modelName_b:
                pass
            else:
                model_b = chooseModel(modelName_b)   # segundo modelo
                Stacking(model_a, model_b).run()

    for modelName in MODELS:
        model = chooseModel(modelName)
        evaluator.AddAlgorithm(model.algo, model.name)
```

```
        evaluator.Evaluate(True)

#ejecutar el programa principal
if __name__ == '__main__':
        ######## 1. Estudiar los datos ####################
        checkData()
        viewData()

        ######## 2. Modelo Simple SIN Stacking ###################
        # Validación cruzada
        # Manualmente
        model = chooseModel("DoubleDQN")
        model.run()
        #Automáticamente
        testAllSingleModel()

        #2. Validación Cruzada Repetida (k=5, rept=5)
        #Forma 1: cross_validate
        testAllCrossValidate()

        #Forma 2: RepeatedKFold
        CrossValidateRepetida()

        #### 3. Modelos Emsablados ###################
        # Manualmente
        modelName_a = 'ItemCF'
        modelName_b = 'DoubleDQN'
        model_a = chooseModel(modelName_a)
        model_b = chooseModel(modelName_b)
        Stacking(model_a,model_b).run()

        # Automáticamente
        testAllStackingModel()
```

2. StackingRS.py (para ensamblar modelos)

```
#Librerías
import math
import random
import pandas as pd
import numpy as np
from surprise import Dataset
from surprise.model_selection import train_test_split
from surprise.model_selection import LeaveOneOut
```

```
from surprise.model_selection import KFold
from surprise import Reader
from surprise.prediction_algorithms.predictions import Prediction
from surprise.model_selection import cross_validate
from operator import itemgetter
from BaseModules.MovieLens import MovieLens
from BaseModules.RecommenderMetrics import RecommenderMetrics
from BaseModules.Evaluator import Evaluator
from BaseModules.Evaluator import EvaluatorForStacking

#MovieLensSLD
PATH = 'data/ml-latest-small/ratings.csv'
TRAIN_PATH = 'data/ml-latest-small/ratings_train.csv'
TEST_PATH = 'data/ml-latest-small/ratings_test.csv'

"""
#MovieLens100K
PATH = 'data/ml-100k/u.data'
TRAIN_PATH = 'data/ml-100k/ratings_train.data'
TEST_PATH = 'data/ml-100k/ratings_test.data'
"""

class Stacking():
    def __init__(self, model_a, model_b):
        self.algo = self
        self.model1 = model_a    # nombre del modelo 1
        self.model2 = model_b    # nombre del modelo 2
        self.algo1 = model_a.algo
        self.algo2 = model_b.algo
        self.cutFile()

    def fit(self):
        ml = MovieLens()
        data = ml.loadMovieLensLatestSmall()     #MovieLensSLD
        #data = ml.loadMovieLens100k() #MovieLens100K
        kf = KFold(n_splits=5, random_state=1)
        dfNewTrainSet = pd.DataFrame(columns=['uid', 'iid', 'rate'])
        allTestPredictions = list()
        allNewTestSet = list()

        for trainSet, testSet in kf.split(data):
            self.algo1.fit(trainSet)
            leftOutPredictions = self.algo1.test(testSet)
            allTestPredictions.extend(leftOutPredictions)
```

```
                for prediction in leftOutPredictions:
                    df = pd.DataFrame({'uid':[prediction.uid], 'iid':[prediction.iid],
'rate':[prediction.est]})
                    dfNewTrainSet = dfNewTrainSet.append({'uid':prediction.uid,
'iid':prediction.iid, 'rate':prediction.est}, ignore_index=True)
            NewTrainSet = Dataset.load_from_df(dfNewTrainSet[['uid', 'iid', 'rate']],
Reader(rating_scale=(1, 5))).build_full_trainset()
            self.algo2.fit(NewTrainSet)

        def test(self):
            def loadTestSet():
                testSet = list()
                with open(TEST_PATH, 'r') as f:
                    for i, line in enumerate(f):
                        line_list = line.strip('\r\n').split(',')   #MovieLensSLD
                        #line_list=line.replace('\t', ',').strip('\n').split(',')   #MovieLens100K
                        user = line_list[0]
                        item = line_list[1]
                        rating = line_list[2]
                        testSet.append((user, item, float(rating)))
                return testSet
            testSet = loadTestSet()
            return self.algo2.test(testSet)

        def testWithSet(self, testSet):
            return self.algo2.test(testSet)
        def cutFile(self):
            train_f = open(TRAIN_PATH, 'w+')
            test_f = open(TEST_PATH, 'w+')
            pivot = 0.8
            with open(PATH, 'r') as f:
                for i, line in enumerate(f):
                    if i == 0:
                        train_f.write(line)
                        continue
                    if (random.random() < pivot):
                        train_f.write(line)
                    else:
                        test_f.write(line)
            train_f.close()
            test_f.close()

        def run(self):
            def LoadMovieLensData():
```

```
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall()     #MovieLensSLD
            #data = ml.loadMovieLens100k() #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        # validación cruzada
        cross_validate(self.algo, evaluationData, measures=['RMSE', 'MAE'], cv=5,
verbose=True)
        #evaluación
        evaluator = EvaluatorForStacking(evaluationData, rankings)
        evaluator.AddAlgorithm(self, self.model1.name+"&"+self.model2.name)
        evaluator.Evaluate(True)
```

3. Distintos Algoritmos

Los Algoritmos (1) -(11) se guardan en la carpeta nombrada *AllAlgorithms*, y los (12)-(13) en la carpeta nombrada *DDQNModule* junto con el entorno específico para realizar el aprendizaje por refuerzo.

(1) MostPopular

```
#Librerías
import numpy as np
import random
from BaseModules.MovieLens import MovieLens
from BaseModules.Evaluator import Evaluator
from surprise.prediction_algorithms.predictions import Prediction
from surprise.model_selection import cross_validate

PATH = 'data/ml-latest-small/ratings.csv' #MovieLensSLD
# PATH = 'data/ml-100k/u.data' #MovieLens100K
alpha = 0.5
T = 0
N = 10   #Top 10 películas

# Algortimo MostPopular
class MostPopular:
    def __init__(self):
        self.algo = self
        self.ifNeedTime = False
        self.train_set = None
```

```
        self.name = 'MostPopular'    #nombre del algoritmo
        self.rank = {}
        self.ave = 0


def load_records(self):
        records = list()
        with open(PATH, 'r') as f:
                for i, line in enumerate(f):
                        if i == 0:
                                continue
                        records.append(line.split('\n')[0].split(',')) #MovieLensSLD
                        #records.append(line.split('\n')[0].split('\t')) #MovieLens100K
        return records


def MostPopularity(self, records):
        result = dict()
        amount = 0
        count = 0
        for user, item, _, tm in records:
                if int(tm)/(60*24*60) <= T:    #1 day=24h=24*60*60=86400s
                        continue
                popu = 1/(1.0+alpha*(int(tm)-T)*86400)*1000000000000000
                if user in result.keys():
                        result[user][item] = popu
                else:
                        result.setdefault(user, {})
                        result[user][item] = popu
                amount += popu
                count += 1
        self.ave = amount/count
        return result

def predict(self, rank, user):
        rankList = rank[user]
        sortedRank = sorted(rankList,key=lambda x:(x[1]),reverse = True)
        print('Recommend {} movies for user '.format(N) + user)
        for i in range(N):
                print(sortedRank[i][0])

def makeDataSetsForElse(self):
        pivot = 2
        def load_file():
```

```
                rating_path = PATH
                with open(rating_path, 'r') as f:
                    for i, line in enumerate(f):
                        if i == 0:
                            continue
                        yield line.strip('\r\n')

        train_set = {}
        test_set = {}
        trainSet_len = 0
        testSet_len = 0
        for line in load_file():
            user, movie, rating, timestamp = line.split(',') #MovieLensSLD
            #user, movie, rating, timestamp = line.split('\t') #MovieLens100K
            if (random.random() <= pivot):
                train_set.setdefault(user, {})
                train_set[user][movie] = [rating,timestamp]
                trainSet_len += 1
            else:
                test_set.setdefault(user, {})
                test_set[user][movie] = [rating, timestamp]
                testSet_len += 1
        return [train_set, test_set]


    def fit(self, trainSet):
        [train_set, _] = self.makeDataSetsForElse()
        def trainSetToAvaliable(trainSet):
            train_list = list()

            for inner_user in trainSet.ur.keys():
                outer_user = trainSet.to_raw_uid(inner_user)
                for (inner_item, rating) in trainSet.ur[inner_user]:
                    outer_item = trainSet.to_raw_iid(inner_item)
                    oneRecord = [str(outer_user), str(outer_item), str(rating),
train_set[str(outer_user)][str(outer_item)][1] ]
                    train_list.append(oneRecord)
            return train_list

        self.train_set = trainSetToAvaliable(trainSet)
        self.rank = self.MostPopularity(self.train_set)

    def pred(self, user, item):
        if user in self.rank.keys() and item in self.rank[user].keys():
```

```
                return self.rank[user][item]
            else:
                return self.ave

    def test(self, testSet):
        predictions = list()
        for (user, item, rating) in testSet:
            est = self.pred(user, item)
            predictions.append(Prediction(uid=user, iid=item, r_ui=rating, est=est,
details={}))
        return predictions

    def run(self):
        def LoadMovieLensData():
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall() #MovieLensSLD
            #data = ml.loadMovieLens100k() #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        # validación cruzada
        cross_validate(self.algo, evaluationData, measures=['RMSE', 'MAE'], cv=5,
verbose=True)

        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
        evaluator.Evaluate(True)
```

(2) RecentPopular

```
#Librerías
import numpy as np
import random
from BaseModules.MovieLens import MovieLens
from BaseModules.Evaluator import Evaluator
from surprise.prediction_algorithms.predictions import Prediction
from surprise.model_selection import cross_validate

PATH = 'data/ml-latest-small/ratings.csv' #MovieLensSLD
# PATH = 'data/ml-100k/u.data' #MovieLens100K
```

```
alpha = 0.5
T = 10000
# T días después de 1970.1.1 (aquí se supone que es 8000 días)
# 1970 es el año de publicación que empezó a contar el timestamp(tm)
N = 10   #Top 10 películas

# RecentPopular
class RecentPopular:
    def __init__(self):
        self.algo = self
        self.ifNeedTime = False
        self.train_set = None
        self.name = 'RecentPopular'   #nombre del algoritmo
        self.rank = {}
        self.ave = 0

    def load_records(self):
        records = list()
        with open(PATH, 'r') as f:
            for i, line in enumerate(f):
                if i == 0:
                    continue
                records.append(line.split('\n')[0].split(',')) #MovieLensSLD
                #records.append(line.split('\n')[0].split('\t')) #MovieLens100K
        return records

    def RecentPopularity(self, records):
        result = dict()
        amount = 0
        count = 0
        for user, item, _, tm in records:
            if int(tm)/(60*24*60) <= T: #1 day=24h=24*60*60=86400s
                continue
            popu = 1/(1.0+alpha*(int(tm)-T)*86400)*1000000000000000
            if user in result.keys():
                result[user][item] = popu
            else:
                result.setdefault(user, {})
                result[user][item] = popu
            amount += popu
            count += 1
        self.ave = amount/count
        return result
```

```
def predict(self, rank, user):
    rankList = rank[user]
    sortedRank = sorted(rankList,key=lambda x:(x[1]),reverse = True)
    print('Recommend {} movies for user '.format(N) + user)
    for i in range(N):
        print(sortedRank[i][0])

def makeDataSetsForElse(self):
    pivot = 2
    def load_file():
        rating_path = PATH
        with open(rating_path, 'r') as f:
            for i, line in enumerate(f):
                if i == 0:    # except title
                    continue
                yield line.strip('\r\n')

    train_set = {}
    test_set = {}
    trainSet_len = 0
    testSet_len = 0
    for line in load_file():    # user, movie, rating, timestamp
        user, movie, rating, timestamp = line.split(',') #MovieLensSLD
        #user, movie, rating, timestamp = line.split('\t') #MovieLens100K
        if (random.random() <= pivot):
            train_set.setdefault(user, {})
            train_set[user][movie] = [rating,timestamp]
            trainSet_len += 1
        else:
            test_set.setdefault(user, {})
            test_set[user][movie] = [rating, timestamp]
            testSet_len += 1
    return [train_set, test_set]

def fit(self, trainSet):
    [train_set, _] = self.makeDataSetsForElse()
    def trainSetToAvaliable(trainSet):
        train_list = list()

        for inner_user in trainSet.ur.keys():
            outer_user = trainSet.to_raw_uid(inner_user)
            for (inner_item, rating) in trainSet.ur[inner_user]:
                outer_item = trainSet.to_raw_iid(inner_item)
                oneRecord    =    [str(outer_user),    str(outer_item),    str(rating),
```

```
train_set[str(outer_user)][str(outer_item)][1] ]
                        train_list.append(oneRecord)
            return train_list

        self.train_set = trainSetToAvaliable(trainSet)
        self.rank = self.RecentPopularity(self.train_set)

    def pred(self, user, item):
        if user in self.rank.keys() and item in self.rank[user].keys():
            return self.rank[user][item]
        else:
            return self.ave

    def test(self, testSet):
        predictions = list()
        for (user, item, rating) in testSet:
            est = self.pred(user, item)
            predictions.append(Prediction(uid=user, iid=item, r_ui=rating, est=est,
details={}))
        return predictions

    def run(self):
        def LoadMovieLensData():
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall()  #MovieLensSLD
            #data = ml.loadMovieLens100k()  #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        # validación cruzada
        cross_validate(self.algo, evaluationData, measures=['RMSE', 'MAE'], cv=5,
verbose=True)

        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
        evaluator.Evaluate(True)
```

(3) Random

```
#Librerías
```

```
from BaseModules.MovieLens import MovieLens
from surprise import NormalPredictor
from BaseModules.Evaluator import Evaluator
import random
import numpy as np
from surprise.model_selection import cross_validate

# Random
class RandomAL:
    def __init__(self):
        self.algo = NormalPredictor()
        self.ifNeedTime = False
        self.name = 'Random'     # nombre del algoritmo

    def fit(self, trainSet):
        self.algo.fit(trainSet)

    def test(self, testSet):
        return self.algo.test(testSet)

    def run(self):
        def LoadMovieLensData():
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall() #MovieLensSLD
            #data = ml.loadMovieLens100k()   #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        # validación cruzada
        cross_validate(self.algo,  evaluationData,  measures=['RMSE',  'MAE'],  cv=5,
verbose=True)

        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
        evaluator.Evaluate(True)
```

(4) UserCF

```
#Librerías
from BaseModules.MovieLens import MovieLens
```

```
from surprise import KNNBasic
import heapq
from collections import defaultdict
from operator import itemgetter
import random
import numpy as np
from BaseModules.Evaluator import Evaluator
from surprise.model_selection import cross_validate

# UserCF
class UserCF:
    def __init__(self):
        sim_options = {'name': 'cosine','user_based': True}
        self.algo = KNNBasic(sim_options=sim_options)   # k vecinos
        self.ifNeedTime = False
        self.name = 'UserCF'     #nombre del algoritmo

    def fit(self, trainSet):
        self.algo.fit(trainSet)

    def test(self, testSet):
        return self.algo.test(testSet)

    def predict(self, testSubject = '85'):
        testSubject = testSubject
        k = 10   #Top 10 películas
        # Cargar dataset para computar user similarity matrix
        ml = MovieLens()
        data = ml.loadMovieLensLatestSmall()   #MovieLensSLD
        #data = ml.loadMovieLens100k()   #MovieLens100K
        trainSet = data.build_full_trainset()
        sim_options = {'name': 'cosine','user_based': True}

        model = KNNBasic(sim_options=sim_options)
        model.fit(trainSet)
        simsMatrix = model.compute_similarities()

        # Top N similar users (test)
        testUserInnerID = trainSet.to_inner_uid(testSubject)
        similarityRow = simsMatrix[testUserInnerID]

        similarUsers = []
        for innerID, score in enumerate(similarityRow):
            if (innerID != testUserInnerID):
```

```
                similarUsers.append( (innerID, score) )

        kNeighbors = heapq.nlargest(k, similarUsers, key=lambda t: t[1])
        candidates = defaultdict(float)
        for similarUser in kNeighbors:
            innerID = similarUser[0]
            userSimilarityScore = similarUser[1]
            theirRatings = trainSet.ur[innerID]
            for rating in theirRatings:
                candidates[rating[0]] += (rating[1] / 5.0) * userSimilarityScore

        # Construir un diccionario de las películas que el usuario ya ha visto
        watched = {}
        for itemID, rating in trainSet.ur[testUserInnerID]:
            watched[itemID] = 1
        #Obtener Top-rated items desde similar users:
        pos = 0
        for itemID, ratingSum in sorted(candidates.items(), key=itemgetter(1),
reverse=True):
            if not itemID in watched:
                movieID = trainSet.to_raw_iid(itemID)
                print(ml.getMovieName(int(movieID)), ratingSum)
                pos += 1
                if (pos > 10):
                    break

    def run(self):
        def LoadMovieLensData():
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall() #MovieLensSLD
            #data = ml.loadMovieLens100k() #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        # validación cruzada
        cross_validate(self.algo, evaluationData, measures=['RMSE', 'MAE'], cv=5,
verbose=True)

        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
```

```
        evaluator.Evaluate(True)
```

(5) ItemCF

```
#Librerías
from BaseModules.MovieLens import MovieLens
from surprise import KNNBasic
import heapq
from collections import defaultdict
from operator import itemgetter
import random
import numpy as np
from BaseModules.Evaluator import Evaluator
from surprise.model_selection import cross_validate

# ItemCF
class ItemCF:
    def __init__(self):
        sim_options = {'name': 'cosine','user_based': False}
        self.algo = KNNBasic(sim_options=sim_options)
        self.ifNeedTime = False
        self.name = 'ItemCF'    #nombre del algoritmo

    def fit(self, trainSet):
        self.algo.fit(trainSet)

    def test(self, testSet):
        return self.algo.test(testSet)

    def predict(self,testSubject='85'):
        testSubject = testSubject
        k = 10    #Top 10 películas
        # Cargar dataset
        ml = MovieLens()
        data = ml.loadMovieLensLatestSmall() #MovieLensSLD
        #data = ml.loadMovieLens100k() #MovieLens100K
        trainSet = data.build_full_trainset()


        sim_options = {'name': 'cosine', 'user_based': False}
        model = KNNBasic(sim_options=sim_options)
        model.fit(trainSet)
        simsMatrix = model.compute_similarities()
        testUserInnerID = trainSet.to_inner_uid(testSubject)
```

```
            testUserRatings = trainSet.ur[testUserInnerID]
            kNeighbors = heapq.nlargest(k, testUserRatings, key=lambda t: t[1])
            # Obtener productos similares a los que al usuario le gusta
            candidates = defaultdict(float)
            for itemID, rating in kNeighbors:
                similarityRow = simsMatrix[itemID]
                for innerID, score in enumerate(similarityRow):
                    candidates[innerID] += score * (rating / 5.0)

            watched = {}
            for itemID, rating in trainSet.ur[testUserInnerID]:
                watched[itemID] = 1

            pos = 0
            for itemID, ratingSum in sorted(candidates.items(), key=itemgetter(1),
reverse=True):
                if not itemID in watched:
                    movieID = trainSet.to_raw_iid(itemID)
                    print(ml.getMovieName(int(movieID)), ratingSum)
                    pos += 1
                    if (pos > 10):
                        break

        def run(self):
            def LoadMovieLensData():
                ml = MovieLens()
                data = ml.loadMovieLensLatestSmall()   #MovieLensSLD
                #data = ml.loadMovieLens100k()   #MovieLens100K
                rankings = ml.getPopularityRanks()
                return (ml, data, rankings)

            np.random.seed(0)
            random.seed(0)
            (ml, evaluationData, rankings) = LoadMovieLensData()
            # validación cruzada
            cross_validate(self.algo, evaluationData, measures=['RMSE', 'MAE'], cv=5,
verbose=True)

            # evaluación
            evaluator = Evaluator(evaluationData, rankings)
            evaluator.AddAlgorithm(self.algo, self.name)
            evaluator.Evaluate(True)
```

(6) ItemCFNorm

```
#Librerías
import math
import random
import numpy as np
from BaseModules.MovieLens import MovieLens
from BaseModules.Evaluator import Evaluator
from operator import itemgetter
from surprise.prediction_algorithms.predictions import Prediction
from surprise.model_selection import cross_validate

PATH = 'data/ml-latest-small/ratings.csv' #MovieLensSLD
# PATH = 'data/ml-100k/u.data'    #MovieLens100K

# ItemCF-Norm
class ItemCFNorm:
    def __init__(self, n_sim_movie=20, n_rec_movie=10):
        self.algo =self
        self.ifNeedTime = False
        self.name = 'ItemCFNorm'     #nombre del algoritmo
        pivot = 0.75
        def makeDataSetsForElse():
            def load_file():
                rating_path = PATH
                with open(rating_path, 'r') as f:
                    for i, line in enumerate(f):
                        if i == 0:
                            continue
                        yield line.strip('\r\n')

            train_set = {}
            test_set = {}
            trainSet_len = 0
            testSet_len = 0
            head=0
            for line in load_file():
                user, movie, rating, timestamp = line.split(',') #MovieLensSLD
                #user, movie, rating, timestamp = line.replace('\t', ',').strip('\n').split(',')
#MovieLens100K
                if (random.random() < pivot):
                    train_set.setdefault(user, {})
                    train_set[user][movie] = [rating,timestamp]
                    trainSet_len += 1
                else:
```

```
                    test_set.setdefault(user, {})
                    test_set[user][movie] = [rating, timestamp]
                    testSet_len += 1
            return [train_set, test_set]

        [train_set, test_set] = makeDataSetsForElse()
        # train set & test set
        self.train_set = train_set
        self.test_set = test_set
        self.n_sim_movie = n_sim_movie   #películas similares
        self.n_rec_movie = n_rec_movie   #películas recomendadas
        self.movie_sim_matrix = {}
        self.norm_movie_sim_matrix = {}
        self.movie_popular = {}
        self.movie_count = 0
        self.users_and_rec_movies = {}
        self.ave = 0

    # calcular la similitud entre las películas
    def calc_movie_sim(self):
        for user, movies in self.train_set.items():
            for movie in movies:
                if movie not in self.movie_popular:
                    self.movie_popular[movie] = 0
                self.movie_popular[movie] += 1

        self.movie_count = len(self.movie_popular)

        for user, movies in self.train_set.items():
            for m1 in movies:
                for m2 in movies:
                    if m1 == m2:
                        continue
                    self.movie_sim_matrix.setdefault(m1, {})
                    self.movie_sim_matrix[m1].setdefault(m2, 0)
                    self.movie_sim_matrix[m1][m2] += 1

        for m1, related_movies in self.movie_sim_matrix.items():
            for m2, count in related_movies.items():
                if self.movie_popular[m1] == 0 or self.movie_popular[m2] == 0:
                    self.movie_sim_matrix[m1][m2] = 0
                else:
                    self.movie_sim_matrix[m1][m2] = count /
math.sqrt(self.movie_popular[m1] * self.movie_popular[m2])
```

```
    # K similar movies and N recommend movies
    def recommend(self, user):
        K = self.n_sim_movie
        N = self.n_rec_movie
        rank = {}
        watched_movies = self.train_set[user]
        for movie, [rating] in watched_movies.items():
            for related_movie, w in sorted(self.norm_movie_sim_matrix[movie].items(),
key=itemgetter(1), reverse=True)[:K]:
                # K películas que son similares a las películas vistas
                if related_movie in watched_movies:
                    continue
                rank.setdefault(related_movie, 0)
                rank[related_movie] += w * float(rating)    # rank = {movie, rank}
        return sorted(rank.items(), key=itemgetter(1), reverse=True)[:N]

    #normalización
    def normalization(self):
        for i in self.movie_sim_matrix:
            self.norm_movie_sim_matrix.setdefault(i, {})
            maxValue = max(self.movie_sim_matrix[i].values())
            for j in self.movie_sim_matrix[i]:
                self.norm_movie_sim_matrix[i][j]    =    self.movie_sim_matrix[i][j]    /
maxValue

    def fit(self, trainSet):
        def trainSetToAvaliable(trainSet):
            train_dict = dict()
            ratings = 0
            ratings_amount = 0
            for user in trainSet.ur.keys():
                for (item, rating) in trainSet.ur[user]:
                    ratings_amount += 1
                    ratings += rating
                    outer_user = trainSet.to_raw_uid(user)
                    outer_item = trainSet.to_raw_iid(item)
                    train_dict.setdefault(str(outer_user), {})
                    train_dict[outer_user][outer_item] = [str(rating)]

            return train_dict, ratings/ratings_amount

        self.train_set, self.ave = trainSetToAvaliable(trainSet)
        self.calc_movie_sim()
```

```
            self.normalization()

        def test(self, testSet):
            predictions = list()
            for (user, item, rating) in testSet:
                est = self.pred(user, item)
                predictions.append(Prediction(uid=user, iid=item, r_ui=rating, est=est,
details={}))
            return predictions

        def pred(self, uid, mid):
            if uid not in self.train_set.keys():
                return self.ave
            sim_accumulate = 0.0
            rat_acc = 0.0
            for item in self.train_set[uid].keys():
                if item in self.norm_movie_sim_matrix.keys() and mid in
self.norm_movie_sim_matrix[item].keys():
                    sim = self.norm_movie_sim_matrix[item][mid] # sji, i=mid
                else:
                    continue
                if sim < 0:
                    continue
                rat_acc += sim * float(self.train_set[uid][item][0])
                sim_accumulate += sim

            if sim_accumulate == 0:
                return self.ave
            return rat_acc / sim_accumulate


        def run(self):
            def LoadMovieLensData():
                ml = MovieLens()
                data = ml.loadMovieLensLatestSmall() #MovieLensSLD
                #data = ml.loadMovieLens100k() #MovieLens100K
                rankings = ml.getPopularityRanks()
                return (ml, data, rankings)

            np.random.seed(0)
            random.seed(0)
            (ml, evaluationData, rankings) = LoadMovieLensData()
            # validación cruzada
            cross_validate(self, evaluationData, measures=['RMSE', 'MAE'], cv=5,
```

```
verbose=True)

            # evaluación
              evaluator = Evaluator(evaluationData, rankings)
              evaluator.AddAlgorithm(self.algo, self.name)
              evaluator.Evaluate(True)
```

(7) TItemCF

```
#Librerías
from operator import itemgetter
import numpy as np
import math
import random
from BaseModules.MovieLens import MovieLens
from BaseModules.Evaluator import Evaluator
from surprise.prediction_algorithms.predictions import Prediction
from surprise.model_selection import cross_validate

PATH = 'data/ml-latest-small/ratings.csv'    #MovieLensSLD
# PATH = 'data/ml-100k/u.data'    #MovieLens100K
# TItemCF
class TItemCF:
      def __init__(self, n_sim_movie=20, n_rec_movie=10):
            self.algo = self
            self.ifNeedTime = False
            self.name = 'TItemCF'    #nombre del algoritmo
            [train_set, test_set] = self.makeDataSetsForElse()
            # train set & test set
            self.train_set = train_set
            self.test_set = test_set
            self.n_sim_movie = n_sim_movie    #películas similares
            self.n_rec_movie = n_rec_movie    #películas recomendadas
            self.movie_sim_matrix = {}
            self.movie_popular = {}
            self.movie_count = 0
            self.users_and_rec_movies = {}

      def makeDataSetsForElse(self):
            pivot = 2
            def load_file():
                  rating_path = PATH
                  with open(rating_path, 'r') as f:
                        for i, line in enumerate(f):
                              if i == 0:
```

```
                    continue
                yield line.strip('\r\n')
        train_set = {}
        test_set = {}
        trainSet_len = 0
        testSet_len = 0
        for line in load_file():
            user, movie, rating, timestamp = line.split(',') #MovieLensSLD
            # user, movie, rating, timestamp = line.replace('\t', ',').strip('\n').split(',')
#MovieLens100K
            if (random.random() <= pivot):
                train_set.setdefault(user, {})
                train_set[user][movie] = [rating, timestamp]
                trainSet_len += 1
            else:
                test_set.setdefault(user, {})
                test_set[user][movie] = [rating, timestamp]
                testSet_len += 1
        return [train_set, test_set]

    # calcular la similitud entre las películas
    def calc_movie_sim(self, alpha):        #alpha es un parámetro de decay
        for user in self.train_set.keys():
            movies = self.train_set[user]
            for movie in movies:
                if movie not in self.movie_popular:
                    self.movie_popular[movie] = 0
                self.movie_popular[movie] += 1
        self.movie_count = len(self.movie_popular)
        for user in self.train_set.keys():
            movies = self.train_set[user]
            for m1 in movies.keys():
                for m2 in movies.keys():
                    if m1 == m2:
                        continue
                    self.movie_sim_matrix.setdefault(m1, {})
                    self.movie_sim_matrix[m1].setdefault(m2, 0)
                    try:
                        self.movie_sim_matrix[m1][m2]                                    +=
1/(1+alpha*abs(float(movies[m1][1])-float(movies[m2][1])))
                        #movies[m1][1] es t_ui, movies[m2][1] es t_uj
                        except IndexError:
                            pass
        for m1, related_movies in self.movie_sim_matrix.items():
```

```
                for m2, count in related_movies.items():
                    if self.movie_popular[m1] == 0 or self.movie_popular[m2] == 0:
                        self.movie_sim_matrix[m1][m2] = 0
                    else:
                        self.movie_sim_matrix[m1][m2] = count / math.sqrt(self.movie_popular[m1] * self.movie_popular[m2])
    # K similar movies and N recommend movies
    def recommend(self, user):
        K = self.n_sim_movie
        N = self.n_rec_movie
        rank = {}
        watched_movies = self.train_set[user]
        for movie, [rating, _] in watched_movies.items():
            for related_movie, w in sorted(self.movie_sim_matrix[movie].items(), key=itemgetter(1), reverse=True)[:K]:
                if related_movie in watched_movies:
                    continue
                rank.setdefault(related_movie, 0)
                rank[related_movie] += w * float(rating)    # rank = {movie, rank}
        return sorted(rank.items(), key=itemgetter(1), reverse=True)[:N]

    def fit(self, trainSet):
        [train_set, _] = self.makeDataSetsForElse()
        def trainSetToAvaliable(trainSet):
            train_dict = dict()
            ratings = 0
            ratings_amount = 0
            for inner_user in trainSet.ur.keys():
                outer_user = trainSet.to_raw_uid(inner_user)
                for (inner_item, rating) in trainSet.ur[inner_user]:
                    outer_item = trainSet.to_raw_iid(inner_item)
                    ratings_amount += 1
                    ratings += rating
                    train_dict.setdefault(str(outer_user), {})
                    train_dict[str(outer_user)].setdefault(str(outer_item), [])
                    try:
                        train_dict[str(outer_user)][str(outer_item)] = [str(rating),train_set[str(outer_user)][str(outer_item)][1]]
                    except KeyError:
                        train_dict[str(outer_user)].get(str(outer_item),[str(rating),str(881250949)])

            return train_dict, ratings/ratings_amount
```

```
            self.train_set, self.ave = trainSetToAvaliable(trainSet)
            self.calc_movie_sim(0.1)   #alpha=0,1
        def test(self, testSet):
            predictions = list()
            for (user, item, rating) in testSet:
                est = self.pred(user, item)
                predictions.append(Prediction(uid=user,   iid=item,   r_ui=rating,   est=est,
details={}))
            return predictions

        def pred(self, uid, mid):
            if uid not in self.train_set.keys():
                return self.ave
            sim_accumulate = 0.0
            rat_acc = 0.0
            for item in self.train_set[uid].keys():
                if      item      in      self.movie_sim_matrix.keys()      and      mid      in
self.movie_sim_matrix[item].keys():
                    sim = self.movie_sim_matrix[item][mid] # sji, i=mid
                else:
                    continue
                if sim < 0:
                    continue
                try:
                    rat_acc += sim * float(self.train_set[uid][item][0])
                    sim_accumulate += sim    # sum(sij)
                except IndexError:
                    pass
            if sim_accumulate == 0:
                return self.ave
            return rat_acc / sim_accumulate


        def run(self):
            def LoadMovieLensData():
                ml = MovieLens()
                data = ml.loadMovieLensLatestSmall() #MovieLensSLD
                #data = ml.loadMovieLens100k() #MovieLens100K
                rankings = ml.getPopularityRanks()
                return (ml, data, rankings)
            np.random.seed(0)
            random.seed(0)
            (ml, evaluationData, rankings) = LoadMovieLensData()
            # validación cruzada
```

```
        cross_validate(self.algo, evaluationData, measures=['RMSE', 'MAE'], cv=5,
verbose=True)
        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
        evaluator.Evaluate(True)
```

(8) ContentKNN

```
#Librerías
from surprise import AlgoBase
from surprise import PredictionImpossible
import math
import heapq
from BaseModules.MovieLens import MovieLens
from BaseModules.Evaluator import Evaluator
from surprise.model_selection import cross_validate
import random
import numpy as np

# ContentKNN
class ContentKNNAlgorithm(AlgoBase):
    def __init__(self, k=40, sim_options={}):
        AlgoBase.__init__(self)
        self.k = k

    def fit(self, trainset):
        AlgoBase.fit(self, trainset)
        # Calcular la matriz de la similitud de items basando en sus atributos
        ml = MovieLens()
        genres = ml.getGenres()
        years = ml.getYears()
        mes = ml.getMiseEnScene()
        # 2x2 matrix
        self.similarities = np.zeros((self.trainset.n_items, self.trainset.n_items))
        for thisRating in range(self.trainset.n_items):
            if (thisRating % 100 == 0):
                pass
                # print(thisRating, " of ", self.trainset.n_items)
            for otherRating in range(thisRating+1, self.trainset.n_items):
                thisMovieID = int(self.trainset.to_raw_iid(thisRating))
                otherMovieID = int(self.trainset.to_raw_iid(otherRating))
                genreSimilarity = self.computeGenreSimilarity(thisMovieID,
otherMovieID, genres)
```

```
                    yearSimilarity = self.computeYearSimilarity(thisMovieID,
otherMovieID, years)
                    self.similarities[thisRating, otherRating] = genreSimilarity *
yearSimilarity
                    self.similarities[otherRating, thisRating] = self.similarities[thisRating,
otherRating]
        return self
    # Similitud del género cinematográfico
    def computeGenreSimilarity(self, movie1, movie2, genres):
        genres1 = genres[movie1]
        genres2 = genres[movie2]
        sumxx, sumxy, sumyy = 0, 0, 0
        for i in range(len(genres1)):
            x = genres1[i]
            y = genres2[i]
            sumxx += x * x
            sumyy += y * y
            sumxy += x * y
        return sumxy/math.sqrt(sumxx*sumyy)

    # Similitud del año de publicación (función exponencial natural)
    def computeYearSimilarity(self, movie1, movie2, years):
        diff = abs(years[movie1] - years[movie2])
        sim = math.exp(-diff / 10.0)
        return sim

    def estimate(self, u, i):
        if not (self.trainset.knows_user(u) and self.trainset.knows_item(i)):
            raise PredictionImpossible('User and/or item is unkown.')
        neighbors = []
        for rating in self.trainset.ur[u]:
            genreSimilarity = self.similarities[i,rating[0]]
            neighbors.append( (genreSimilarity, rating[1]) )

        k_neighbors = heapq.nlargest(self.k, neighbors, key=lambda t: t[0])
        simTotal = weightedSum = 0
        for (simScore, rating) in k_neighbors:
            if (simScore > 0):
                simTotal += simScore
                weightedSum += simScore * rating
        if (simTotal == 0):
            raise PredictionImpossible('No neighbors')
        predictedRating = weightedSum / simTotal
```

```
            return predictedRating

class ContentKNNAL():
    def __init__(self):
        self.algo = ContentKNNAlgorithm()
        self.ifNeedTime = False
        self.name = 'ContentKNN'    # nombre del algoritmo

    def fit(self, trainset):
        self.algo.fit(trainset)

    def test(self, testSet):
        return self.algo.test(testSet)

    def run(self):
        def LoadMovieLensData():
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall() #MovieLensSLD
            #data = ml.loadMovieLens100k() #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        # validación cruzada
        cross_validate(self.algo,  evaluationData,  measures=['RMSE',  'MAE'],  cv=5,
verbose=True)
        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
        evaluator.Evaluate(True)
```

(9) SVD

```
#Librerías
import random
import numpy as np
from surprise import SVD
from surprise.model_selection import cross_validate
from BaseModules.Evaluator import Evaluator
from BaseModules.MovieLens import MovieLens

#SVD
```

```
class SVDAL:
    def __init__(self):
        self.algo = SVD()
        self.ifNeedTime = False
        self.name = 'SVD'    #nombre del algoritmo
    def fit(self, trainSet):
        self.algo.fit(trainSet)

    def test(self, testSet):
        return self.algo.test(testSet)

    def run(self):
        def LoadMovieLensData():
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall() #MovieLensSLD
            #data = ml.loadMovieLens100k() #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        # validación cruzada
        cross_validate(self.algo,  evaluationData,  measures=['RMSE',  'MAE'],  cv=5,
verbose=True)
        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
        evaluator.Evaluate(True)
```

(10) SVD++

```
#Librerías
import random
import numpy as np
from surprise import SVDpp
from BaseModules.Evaluator import Evaluator
from BaseModules.MovieLens import MovieLens
from surprise.model_selection import cross_validate

#SVD++
class SVDppAL:
    def __init__(self):
        self.algo = SVDpp()
```

```
            self.ifNeedTime = False
            self.name = 'SVDpp'    #nombre del algoritmo

        def fit(self, trainSet):
            self.algo.fit(trainSet)
        def test(self, testSet):
            return self.algo.test(testSet)

        def run(self):
            def LoadMovieLensData():
                ml = MovieLens()
                data = ml.loadMovieLensLatestSmall() #MovieLensSLD
                #data = ml.loadMovieLens100k() #MovieLens100K
                rankings = ml.getPopularityRanks()
                return (ml, data, rankings)

            np.random.seed(0)
            random.seed(0)
            (ml, evaluationData, rankings) = LoadMovieLensData()
            # validación cruzada
            cross_validate(self.algo,  evaluationData,  measures=['RMSE',  'MAE'],  cv=5,
verbose=True)
            # evaluación
            evaluator = Evaluator(evaluationData, rankings)
            evaluator.AddAlgorithm(self.algo, self.name)
            evaluator.Evaluate(True)
```

(11) TimeSVD++

```
#Librerías
import csv
import time
import math
import random
import numpy as np
from BaseModules.MovieLens import MovieLens
from BaseModules.Evaluator import Evaluator
from surprise.prediction_algorithms.predictions import Prediction
from surprise.model_selection import cross_validate

PATH = 'data/ml-latest-small/ratings.csv'    #MovieLensSLD
# PATH = 'data/ml-100k/u.data'    #MovieLens100K

class loadMovieData:
```

```
def read_training_data(self, path):
    with open(path, 'r') as f:
        matrix = []
        userItems = {}
        itemUsers = {}
        max_item = []
        timestamps = []
        b = 0

        for line in f:
            row = []
            a = line.split('\t')
            user = int(a[0])
            item = int(a[1])
            rating = float(a[2])
            time_ = int(a[3])

            row.append(user)
            row.append(item)
            row.append(rating)
            row.append(time_)
            matrix.append(row)
            max_item.append(item)
            timestamps.append(time_)

            #usuarios
            if user not in userItems:
                userItems[user] = [(item, rating, time_)]
            else:
                if item not in userItems[user]:
                    userItems[user].append((item, rating, time_))

            #items
            if item not in itemUsers:
                itemUsers[item] = [(user, rating, time_)]
            else:
                if user not in itemUsers[item]:
                    itemUsers[item].append((user, rating, time_))
            b += 1
        min_timestamp = min(timestamps)
        max_timestamp = max(timestamps)

        return matrix, userItems, itemUsers, min_timestamp, max_timestamp
```

```
    def num_of_days(self, min_timestamp, max_timestamp):
        nDays = int((max_timestamp - min_timestamp) / 86400)
        return nDays

    def cal_day(self, timestamp, num_of_days, min_timestamp):
        day_ind = np.minimum(num_of_days - 1, int((timestamp - min_timestamp) /
86400))
        return day_ind

    def timestamp_to_day(self, matrix, n_days, min_timestamp):
        userItems = {}
        itemUsers = {}
        for i in range(len(matrix)):
            user = matrix[i][0]
            item = matrix[i][1]
            rating = matrix[i][2]
            timestamp = matrix[i][3]
            day_ind = self.cal_day(timestamp, n_days, min_timestamp)

            #usuarios
            if user not in userItems:
                userItems[user] = [(item, rating, day_ind)]
            else:
                if item not in userItems[user]:
                    userItems[user].append((item, rating, day_ind))

            # items
            if item not in itemUsers:
                itemUsers[item] = [(user, rating, day_ind)]
            else:
                if user not in itemUsers[item]:
                    itemUsers[item].append((user, rating, day_ind))
        return userItems, itemUsers

    def main(self, matrix, userItems, itemUsers, min_timestamp, max_timestamp):
        num_days = self.num_of_days(min_timestamp, max_timestamp)
        new_userItems, new_itemUsers = self.timestamp_to_day(matrix, num_days,
min_timestamp)
        nUsers = len(new_userItems)
        nItems = max(new_itemUsers.keys())
        return new_userItems, nUsers, nItems, num_days, min_timestamp

#TimeSVD++
class TimeSVDpp:
```

```
    def __init__(self):
        self.algo = self
        self.ifNeedTime = True
        self.ave = 0
        self.name = 'TimeSVDpp'    #nombre del algoritmo

    def runner(self, iter, nFactors, nUsers, nItems, userItems, nBins, nDays,
min_time_in_seconds):
        self.gamma_1 = 0.005
        self.gamma_2 = 0.007
        self.gamma_3 = 0.001
        self.g_alpha = 0.00001
        self.tau_6 = 0.005
        self.tau_7 = 0.015
        self.tau_8 = 0.015
        self.l_alpha = 0.0004
        self.max_time = nDays
        self.min_time = 0
        self.min_time_in_seconds = min_time_in_seconds
        self.iterations = iter
        self.userItems = userItems
        self.factors = nFactors + 1
        self.nUsers = nUsers
        self.nItems = nItems
        self.nBins = nBins
        self.nDays = nDays


        b_u, b_i, u_f, i_f, y_j, sumMW, bi_bin, alpha_u, bu_t, alpha_u_k, userFactors_t
= self.init(self.nUsers,self.nItems,self.factors,self.nBins)
        self.bu = b_u
        self.bi = b_i
        self.bi_bin = bi_bin
        self.alpha_u = alpha_u
        self.bu_t = bu_t

        self.userFactors = u_f
        self.itemFactors = i_f
        self.y_j = y_j
        self.sumMW = sumMW

        self.alpha_u_k = alpha_u_k
        self.userFactors_t = userFactors_t
```

```
        self.average = self.avg()
        self.train(self.iterations)


    def init(self, nUsers, nItems, nFactors, nBins):
        bu = np.zeros(nUsers + 1)
        bi = np.zeros(nItems + 1, dtype='float64')
        bi_bin = np.zeros((nItems + 1, nBins))
        alpha_u = np.zeros(nUsers + 1, dtype='float64')
        bu_t = np.zeros((nUsers + 1, self.nDays))

        # los factors
        userFactors = np.random.random((nUsers + 1, nFactors))
        itemFactors = np.random.random((nItems + 1, nFactors))
        y_j = np.zeros((nItems + 1, nFactors), np.float64)
        sumMW = np.random.random((nUsers + 1, nFactors))

        alpha_u_k = np.zeros((nUsers + 1, nFactors))
        userFactors_t = []
        for j in range(nUsers + 1):
            f_d = np.random.random((nFactors, self.nDays))
            for i in range(len(f_d)):
                for k in range(len(f_d[i])):
                    f_d[i][k] = f_d[i][k] / math.sqrt(20)
            userFactors_t.append(f_d)

        for i in range(len(userFactors)):
            for j in range(len(userFactors[i])):
                userFactors[i][j] = userFactors[i][j] / math.sqrt(20)

        for i in range(len(itemFactors)):
            for j in range(len(itemFactors[i])):
                itemFactors[i][j] = itemFactors[i][j] / math.sqrt(20)

        for i in range(len(sumMW)):
            for j in range(len(sumMW[i])):
                sumMW[i][j] = sumMW[i][j] / math.sqrt(20)
        return bu, bi, userFactors, itemFactors, y_j, sumMW, bi_bin, alpha_u, bu_t,
alpha_u_k, userFactors_t

    def train(self, iter):
        start = time.time()
        for i in range(iter):
            self.oneIteration()
```

```
    def oneIteration(self):
        for userId in range(1, len(self.userItems) + 1):
            tmpSum = np.zeros(self.factors, dtype='float')
            sz = len(self.userItems[userId])
            if sz > 0:
                sqrtNum = 1 / (math.sqrt(sz))
                for f in range(self.factors):
                    sum_y = 0
                    for j in range(sz):
                        pos_item = self.userItems[userId][j][0]
                        sum_y += self.y_j[pos_item][f]
                    self.sumMW[userId][f] = sum_y

                for it in range(sz):
                    itemid = self.userItems[userId][it][0]
                    rating = self.userItems[userId][it][1]
                    timestamp_ = self.userItems[userId][it][2]
                    prediction = self.prediction(userId, itemid, timestamp_)
                    error = rating - prediction
                    self.bu[userId] += 0.01 * error - 0.01 * self.bu[userId]
                    self.bi[itemid] += 0.01 * error - 0.01 * self.bi[itemid]
                    self.bu_t[userId][timestamp_] += 0.01 * (error - 0.01 *
self.bu_t[userId][timestamp_])
                    self.bi_bin[itemid][self.calBin(timestamp_)] += 0.01 * (error - 0.01
* self.bi_bin[itemid][self.calBin(timestamp_)])
                    self.alpha_u[userId] += 0.01 * (error * self.dev(userId,
timestamp_) - 0.01 * self.alpha_u[userId])
                    # Actualizar los factores
                    for k in range(self.factors):
                        u_f = self.userFactors[userId][k]
                        i_f = self.itemFactors[itemid][k]
                        u_f_t = self.userFactors_t[userId][k][timestamp_]
                        self.userFactors[userId][k] += 0.01 * (error * i_f - 0.01 * u_f)
                        self.itemFactors[itemid][k] += 0.01 * (error * (u_f + sqrtNum *
self.sumMW[userId][k]) - 0.01 * i_f)
                        self.alpha_u_k[userId][k] += 0.01 * (error * self.dev(userId,
timestamp_) - 0.01 * self.alpha_u_k[userId][k])
                        self.userFactors_t[userId][k][timestamp_] += 0.01 * (error *
i_f - 0.01 * u_f_t)
                        tmpSum[k] += error * sqrtNum * i_f

                for j in range(sz):
                    itID = self.userItems[userId][j][0]
```

```
                        for f in range(self.factors):
                            tmpMW = self.y_j[itID][f]
                            self.y_j[itID][f] += 0.01 * (tmpSum[f] - 0.01 * tmpMW)
                            self.y_j[itID][f] = round(self.y_j[itID][f], 4)
                            self.sumMW[userId][f] += self.y_j[itID][f] - tmpMW

            for userId in range(1, len(self.userItems) + 1):
                sz = len(self.userItems[userId])
                if sz > 0:
                    sqrtNum = 1 / (math.sqrt(sz))
                    for k in range(self.factors):
                        sumy = 0
                        for i in range(sz):
                            itID = self.userItems[userId][i][0]
                            sumy += self.y_j[itID][k]
                        self.sumMW[userId][k] = sumy

            self.gamma_1 *= 0.9
            self.g_alpha *= 0.9

        # la media de las puntuaciones
        def avg(self):
            s = 0
            count = 0
            l = len(self.userItems)
            for i in range(1, l + 1):
                sz = len(self.userItems[i])
                for j in range(sz):
                    s += self.userItems[i][j][1]
                    count += 1
            avg = s / count
            return avg

        def calBin(self, day_of_rating):
            interval = (self.max_time - self.min_time) / self.nBins
            bin_ind = np.minimum(self.nBins - 1, int((day_of_rating - self.min_time) /
interval))
            return bin_ind

        # la desviación de tiempo asociada de la calificación dada por un usuario en el día t
        def dev(self, userID, t):
            deviation = np.sign(t - self.meanTime(userID)) * pow(abs(t -
self.meanTime(userID)), 0.015)
            return deviation
```

```
    # tu: tiempo medio de calificación de cada usuario
    def meanTime(self, userID):
        s = 0
        count = 0
        sz = len(self.userItems[userID])
        if sz > 0:
            for i in range(sz):
                s += self.userItems[userID][i][2]
                count += 1
            return s / count
        else:
            return 0

    def prediction(self, u, i, day_ind):
        if u not in self.userItems.keys():
            return self.ave
        sz = len(self.userItems[u])
        if sz > 0:
            sqrtNum = 1 / math.sqrt(sz)
        else:
            sqrtNum = 0
        tmp = 0
        for k in range(self.factors):
            tmp += ((self.userFactors[u][k] + self.alpha_u_k[u][k] * self.dev(u, day_ind)
+ self.userFactors_t[u][k][day_ind]) + (sqrtNum * self.sumMW[u][k])) *
self.itemFactors[i][k]
        prediction = self.average + self.bu[u] + self.bi[i] +
self.bi_bin[i][self.calBin(day_ind)] + self.alpha_u[u] * self.dev(u, day_ind) +
self.bu_t[u][day_ind] + tmp

        return prediction

    def RMSE(self):
        with open(PATH, 'r') as f:
            data = csv.reader(f, delimiter=',')
            mean_squared_error = 0
            c = 0
            for row in data:
                userid = int(row[0])
                itemid = int(row[1])
                rating = float(row[2])
                tmp = int(row[3])
                day = min(self.nDays - 1, int((tmp - self.min_time_in_seconds) /
```

```
86400))
                    predict = self.prediction(userid, itemid, day)
                    mean_squared_error += math.pow((rating - predict), 2)
                    c += 1
            meanSuaredError = mean_squared_error / c
            meanSuaredError = math.sqrt(meanSuaredError)
            return meanSuaredError

    def makeDataSetsForElse(self):
        pivot = 2
        def load_file():
            rating_path = PATH
            with open(rating_path, 'r') as f:
                for i, line in enumerate(f):
                    if i == 0:
                        continue
                    yield line.strip('\r\n')

        train_set = {}
        test_set = {}
        trainSet_len = 0
        testSet_len = 0
        for line in load_file():
            user, movie, rating, timestamp = line.split(',')   #MovieLensSLD
            # user, movie, rating, timestamp = line.split('\t')   #MovieLens100K
            if (random.random() <= pivot):
                train_set.setdefault(user, {})
                train_set[user][movie] = [rating,timestamp]
                trainSet_len += 1
            else:
                test_set.setdefault(user, {})
                test_set[user][movie] = [rating, timestamp]
                testSet_len += 1
        return [train_set, test_set]

    def fit(self, trainSet):
        [train_set, _] = self.makeDataSetsForElse()
        def trainSetToAvaliable(trainSet):
            matrix = []
            userItems = {}
            itemUsers = {}
            max_item = []
            timestamps = []
            b = 0
```

```
amount = 0
count = 0
for inner_user in trainSet.ur.keys():
    outer_user = trainSet.to_raw_uid(inner_user)
    for (inner_item, rating) in trainSet.ur[inner_user]:
        outer_item = trainSet.to_raw_iid(inner_item)
        row = []
        user = int(outer_user)
        item = int(outer_item)
        rating = float(rating)
        try:
            time_ = int(train_set[str(outer_user)][str(outer_item)][1])
        except KeyError:
            time_ = int(881250949)
        amount += rating
        count += 1

        row.append(user)
        row.append(item)
        row.append(rating)
        row.append(time_)
        matrix.append(row)
        max_item.append(item)
        timestamps.append(time_)

        # usuarios
        if user not in userItems:
            userItems[user] = [(item, rating, time_)]
        else:
            if item not in userItems[user]:
                userItems[user].append((item, rating, time_))

        # items
        if item not in itemUsers:
            itemUsers[item] = [(user, rating, time_)]
        else:
            if user not in itemUsers[item]:
                itemUsers[item].append((user, rating, time_))

        b += 1

min_timestamp = min(timestamps)
max_timestamp = max(timestamps)
return matrix, userItems, itemUsers, min_timestamp, max_timestamp,
```

```
amount / count

        lm = loadMovieData()
        matrix, userItems, itemUsers, min_timestamp, max_timestamp, self.ave = trainSetToAvaliable(trainSet)
        userItems, nUsers, nItems, nDays, minTimestamp = lm.main(matrix, userItems, itemUsers, min_timestamp, max_timestamp)
        nFactors = 10
        nBins = 6
        iter = 1
        timesvd_pp = self.runner(iter, nFactors, nUsers, nItems, userItems, nBins, nDays, minTimestamp)

    def test(self, testSet):
        [train_set, _] = self.makeDataSetsForElse()
        predictions = list()
        for (user, item, rating) in testSet:
            if user not in train_set.keys() or item not in train_set[user].keys():
                est = self.ave
            else:
                est = self.prediction(user, item, train_set[user][item][1])
            predictions.append(Prediction(uid=user, iid=item, r_ui=rating, est=est, details={}))
        return predictions

    def pred(self, uid, mid):
        if uid not in self.train_set.keys():
            return self.ave
        sim_accumulate = 0.0
        rat_acc = 0.0
        for item in self.train_set[uid].keys():
            if item in self.movie_sim_matrix.keys() and mid in self.movie_sim_matrix[item].keys():
                sim = self.movie_sim_matrix[item][mid]
            else:
                continue
            if sim < 0:
                continue
            rat_acc += sim * float(self.train_set[uid][item][0])
            sim_accumulate += sim    # sum(sij)

        if sim_accumulate == 0:
            return self.ave
        return rat_acc / sim_accumulate
```

```
    def run(self):
        def LoadMovieLensData():
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall()     #MovieLensSLD
            #data = ml.loadMovieLens100k() #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        # validación cruzada
        cross_validate(self.algo,  evaluationData,  measures=['RMSE',  'MAE'],  cv=5,
verbose=True)
        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
        evaluator.Evaluate(True)
```

(12) DQN

```
#Librerías
import os
import math
import gym
import random
import numpy as np
import pandas as pd
import tensorflow as tf
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras import backend as K
from skimage.color import rgb2gray
from skimage.transform import resize
from DDQNModule.gym_recommendation import RecoEnvLatest
from BaseModules.MovieLens import MovieLens
from BaseModules.Evaluator import Evaluator
from surprise.prediction_algorithms.predictions import Prediction
from surprise.model_selection import cross_validate


ENV_NAME = 'rec_dqn'   # Environment name
```

```
NUM_EPISODES = 1000 # Número de episodes
FRAME_WIDTH = 84
FRAME_HEIGHT = 84
GAMMA = 0.9    # Discount factor
EXPLORATION_STEPS = 100000
INITIAL_EPSILON = 1.0
FINAL_EPSILON = 0.1
INITIAL_REPLAY_SIZE = 200
NUM_REPLAY_MEMORY = 400000
BATCH_SIZE = 32    # Mini batch size
TARGET_UPDATE_INTERVAL = 10
TRAIN_INTERVAL = 4
LEARNING_RATE = 0.00025    # Learning rate
MOMENTUM = 0.95
MIN_GRAD = 0.01
SAVE_INTERVAL = 300000
NO_OP_STEPS = 30
LOAD_NETWORK = False
TRAIN = False
SAVE_NETWORK_PATH = 'saved_networks/' + ENV_NAME
SAVE_SUMMARY_PATH = 'summary/' + ENV_NAME

"""
#MovieLens100K
ITEM_HEADER = "movie id | movie title | release date | video release date | IMDb URL |
" \
                "unknown | Action | Adventure | Animation | Children's | Comedy | Crime
| " \
                "Documentary | Drama | Fantasy | Film-Noir | Horror | Musical | Mystery
| " \
                "Romance | Sci-Fi | Thriller | War | Western"
"""

class Agent():
    def __init__(self, num_states, num_actions):
        self.num_states = num_states
        self.num_actions = num_actions
        self.epsilon = INITIAL_EPSILON
        self.epsilon_step = (INITIAL_EPSILON - FINAL_EPSILON) /
EXPLORATION_STEPS
        self.t = 0
        self.total_reward = 0
        self.total_q_max = 0
        self.total_loss = 0
```

```
            self.duration = 0
            self.episode = 0

            # Experiencia Replay
            self.replay_memory = deque()

            # RED Q
            self.s, self.q_values, q_network = self.build_network()
            q_network_weights = q_network.trainable_weights

            # RED Target
            self.st, self.target_q_values, target_network = self.build_network()
            target_network_weights = target_network.trainable_weights

            self.update_target_network = [target_network_weights[i].assign(q_network_weights[i]) for i in range(len(target_network_weights))]
            self.a, self.y, self.loss, self.grads_update = self.build_training_op(q_network_weights)
            self.sess = tf.InteractiveSession()
            self.saver = tf.train.Saver(q_network_weights)
            self.summary_placeholders, self.update_ops, self.summary_op = self.setup_summary()
            self.summary_writer = tf.summary.FileWriter(SAVE_SUMMARY_PATH, self.sess.graph)

            if not os.path.exists(SAVE_NETWORK_PATH):
                os.makedirs(SAVE_NETWORK_PATH)
            self.sess.run(tf.initialize_all_variables())

            # Cargar RED Q
            if LOAD_NETWORK:
                self.load_network()
            # Inicializar RED Target
            self.sess.run(self.update_target_network)

        #Huber Loss
        def _huber_loss(self, y_true, y_pred, clip_delta=1.0):
            error = y_true - y_pred
            cond  = K.abs(error) <= clip_delta
            squared_loss = 0.5 * K.square(error)    #mse
            quadratic_loss = clip_delta * (K.abs(error) - 0.5 * clip_delta)    #mae
            return K.mean(tf.where(cond, squared_loss, quadratic_loss))
```

```
    # RED Q
    def build_network(self):
        model = Sequential()
        model.add(Dense(24, input_dim=self.num_states, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.num_actions, activation='linear'))
        s = tf.placeholder(tf.float32, [None, self.num_states])
        q_values = model(s)
        return s, q_values, model

    def build_training_op(self, q_network_weights):
        a = tf.placeholder(tf.int64, [None])
        y = tf.placeholder(tf.float32, [None])

        a_one_hot = tf.one_hot(a, self.num_actions, 1.0, 0.0)
        q_value = tf.reduce_sum(tf.multiply(self.q_values, a_one_hot),
reduction_indices=1)
        # Huber_loss para actualizar los parámetros
        loss = self._huber_loss(y,q_value)
        optimizer = tf.train.RMSPropOptimizer(LEARNING_RATE,
momentum=MOMENTUM, epsilon=MIN_GRAD)
        grads_update = optimizer.minimize(loss, var_list=q_network_weights)
        return a, y, loss, grads_update

    # Acción aleatoria con la probabilidad de decaimiento de épsilon
    def get_action(self, state):
        if self.epsilon >= random.random() or self.t < INITIAL_REPLAY_SIZE:
            action = random.randrange(self.num_actions)
        else:
            action = np.argmax(self.q_values.eval(feed_dict={self.s:
[np.float32(state)]}))

        if self.epsilon > FINAL_EPSILON and self.t >= INITIAL_REPLAY_SIZE:
            self.epsilon -= self.epsilon_step
        return action

    def run(self, state, action, reward, terminal, next_state):
        reward = np.clip(reward, -1, 1)
        # Guardar en Experiencia Replay
        self.replay_memory.append((state, action, reward, next_state, terminal))
        if len(self.replay_memory) > NUM_REPLAY_MEMORY:
            self.replay_memory.popleft()

        if self.t >= INITIAL_REPLAY_SIZE:
```

```
                # RED Q
                if self.t % TRAIN_INTERVAL == 0:
                    self.train_network()

                # Actualizar RED Target
                if self.t % TARGET_UPDATE_INTERVAL == 0:
                    self.sess.run(self.update_target_network)

                # Guardar RED Q
                if self.t % SAVE_INTERVAL == 0:
                    save_path = self.saver.save(self.sess, SAVE_NETWORK_PATH + '/'
+ ENV_NAME, global_step=self.t)
                    print('Successfully saved: ' + save_path)

            self.total_reward += reward
            self.total_q_max          +=          np.max(self.q_values.eval(feed_dict={self.s:
[np.float32(state )]}))
            self.duration += 1

            if terminal:
                if self.t >= INITIAL_REPLAY_SIZE:
                    stats = [self.total_reward, self.total_q_max / float(self.duration),
                             self.duration,    self.total_loss    /    (float(self.duration)    /
float(TRAIN_INTERVAL))]
                    for i in range(len(stats)):
                        self.sess.run(self.update_ops[i], feed_dict={
                            self.summary_placeholders[i]: float(stats[i])
                        })
                    summary_str = self.sess.run(self.summary_op)
                    self.summary_writer.add_summary(summary_str, self.episode + 1)

                # Debug
                if self.t < INITIAL_REPLAY_SIZE:
                    mode = 'random'
                elif  INITIAL_REPLAY_SIZE  <=  self.t  <  INITIAL_REPLAY_SIZE  +
EXPLORATION_STEPS:
                    mode = 'explore'
                else:
                    mode = 'exploit'
                print('EPISODE:  {0:6d}  /  TIMESTEP:  {1:8d}  /  DURATION:  {2:5d}  /
EPSILON: {3:.5f} / TOTAL_REWARD: {4:3.0f} / AVG_MAX_Q: {5:2.4f} / AVG_LOSS:
{6:.5f} / MODE: {7}'.format(self.episode + 1, self.t, self.duration, self.epsilon,
                    self.total_reward, self.total_q_max / float(self.duration),
                    self.total_loss    /    (float(self.duration)    /    float(TRAIN_INTERVAL)),
```

```
mode))

                self.total_reward = 0
                self.total_q_max = 0
                self.total_loss = 0
                self.duration = 0
                self.episode += 1

            self.t += 1
            return next_state

        def train_network(self):
            state_batch = []
            action_batch = []
            reward_batch = []
            next_state_batch = []
            terminal_batch = []
            y_batch = []

            # Random minibatch
            minibatch = random.sample(self.replay_memory, BATCH_SIZE)
            for data in minibatch:
                state_batch.append(data[0])
                action_batch.append(data[1])
                reward_batch.append(data[2])
                next_state_batch.append(data[3])
                terminal_batch.append(data[4])

            terminal_batch = np.array(terminal_batch) + 0
            # # DIFERENCIA entre DDQN Y DQN: target_q_values
            target_q_values_batch = self.target_q_values.eval(feed_dict={self.st:
np.float32(np.array(next_state_batch))})
            y_batch = reward_batch + (1 - terminal_batch) * GAMMA *
np.max(target_q_values_batch, axis=1)

            loss, _ = self.sess.run([self.loss, self.grads_update], feed_dict={
                self.s: np.float32(np.array(state_batch)),
                self.a: action_batch,
                self.y: y_batch
            })
            self.total_loss += loss

        def setup_summary(self):
            episode_total_reward = tf.Variable(0.)
```

```
        tf.summary.scalar(ENV_NAME + '/Total Reward/Episode', episode_total_reward)
        episode_avg_max_q = tf.Variable(0.)
        tf.summary.scalar(ENV_NAME + '/Average Max Q/Episode', episode_avg_max_q)
        episode_duration = tf.Variable(0.)
        tf.summary.scalar(ENV_NAME + '/Duration/Episode', episode_duration)
        episode_avg_loss = tf.Variable(0.)
        tf.summary.scalar(ENV_NAME + '/Average Loss/Episode', episode_avg_loss)
        summary_vars = [episode_total_reward, episode_avg_max_q, episode_duration, episode_avg_loss]
        summary_placeholders = [tf.placeholder(tf.float32) for _ in range(len(summary_vars))]
        update_ops = [summary_vars[i].assign(summary_placeholders[i]) for i in range(len(summary_vars))]
        summary_op = tf.summary.merge_all()
        return summary_placeholders, update_ops, summary_op

    def load_network(self):
        checkpoint = tf.train.get_checkpoint_state(SAVE_NETWORK_PATH)
        if checkpoint and checkpoint.model_checkpoint_path:
            self.saver.restore(self.sess, checkpoint.model_checkpoint_path)
            print('Successfully loaded: ' + checkpoint.model_checkpoint_path)
        else:
            print('Training new network...')

    def get_action_at_test(self, state):
        if random.random() <= 0.05:
            action = random.randrange(self.num_actions)
        else:
            action = np.argmax(self.q_values.eval(feed_dict={self.s: np.float32(state)}))
        self.t += 1
        return action


def preprocess(observation, last_observation):
    processed_observation = np.maximum(observation, last_observation)
    processed_observation = np.uint8(resize(rgb2gray(processed_observation), (FRAME_WIDTH, FRAME_HEIGHT)) )
    return np.reshape(processed_observation, (1, FRAME_WIDTH, FRAME_HEIGHT))

#DQN
```

```
class DQNAL:
    def __init__(self):
        self.algo = self
        self.name = "DQN"   #nombre del algoritmo

    def fit(self, trainSet):
        def makeTrainSetAvailable(trainSet):
            user_list = []
            item_list = []
            rating_list = []
            for (uid, iid, rating) in trainSet.all_ratings():
                outer_user = trainSet.to_raw_uid(uid)
                outer_item = trainSet.to_raw_iid(iid)
                outer_rating = rating
                user_list.append(float(outer_user))
                item_list.append(float(outer_item))
                rating_list.append(float(rating))
            train_dict = {'userId': user_list, 'itemId': item_list, 'rating':rating_list}
            data = pd.DataFrame(train_dict)

            #MovieLensSLD
            item = pd.read_csv(
                'data/ml-latest-small/movies.csv',
                header=0,
                delimiter=',',
                names=['itemId', 'itemName', 'genres'],
                encoding='latin-1'
            )

            """
            #MovieLens100K
            item = pd.read_csv(
                'data/ml-100k/u.item',
                header=0,
                delimiter='|',
                names=convert_header_to_camel_case(ITEM_HEADER),
                encoding='latin-1'
            )
            """

            kwargs = dict([(label, data) for label, data
                           in zip(['data', 'item'], [data, item])])
            return kwargs
```

```
            # Entorno
            env = gym.make(RecoEnvLatest.id, **makeTrainSetAvailable(trainSet)) # env
= gym.make(ENV_NAME)
            self.env = env
            num_states = env.observation_space.shape[0]
            num_actions = env.action_space.n
            agent = Agent(num_states, num_actions)
            self.agent = agent
            for _ in range(NUM_EPISODES):
                terminal = False
                state = env.reset()
                while not terminal:
                    action = agent.get_action(state)
                    next_state, reward, terminal, _ = env.step(action)
                    state = agent.run(state, action, reward, terminal, next_state)


        def test(self, testSet) -> list():
            predictions = list()
            state_size = self.env.observation_space.shape[0]
            state = self.env.reset()
            state = np.reshape(state, [1, state_size])
            for user, item, rating in testSet:
                state[0][state_size - 2] = float(user)
                state[0][state_size - 1] = float(item)
                est = self.agent.get_action_at_test(state)
                predictions.append(Prediction(uid=user,      iid=item,      r_ui=float(rating),
est=est, details={}))
            return predictions

        def run(self):
            def LoadMovieLensData():
                ml = MovieLens()
                data = ml.loadMovieLensLatestSmall() #MovieLensSLD
                #data = ml.loadMovieLens100k() #MovieLens100K
                rankings = ml.getPopularityRanks()
                return (ml, data, rankings)

            np.random.seed(0)
            random.seed(0)
            (ml, evaluationData, rankings) = LoadMovieLensData()
            # validación cruzada
            cross_validate(self.algo,  evaluationData,  measures=['RMSE',  'MAE'],  cv=5,
```

```
verbose=True)
        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
        evaluator.Evaluate(True)
```

(13) DoubleDQN

```
#Librerías
import os
import gym
import random
import numpy as np
import pandas as pd
import tensorflow as tf    # tensorflow para deep learning
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras import backend as K
from DDQNModule.gym_recommendation import RecoEnvLatest
from BaseModules.MovieLens import MovieLens
from BaseModules.Evaluator import Evaluator
from surprise.prediction_algorithms.predictions import Prediction
from surprise.model_selection import cross_validate

ENV_NAME = 'rec_ddqn'    # Environment name
NUM_EPISODES = 1000 # Número de episodes
GAMMA = 0.9    # Discount factor
EXPLORATION_STEPS = 100000
INITIAL_EPSILON = 1.0
FINAL_EPSILON = 0.1
INITIAL_REPLAY_SIZE = 200
NUM_REPLAY_MEMORY = 400000
BATCH_SIZE = 32    # Mini batch size
TARGET_UPDATE_INTERVAL = 10
TRAIN_INTERVAL = 4
LEARNING_RATE = 0.00025    # Learning rate
MOMENTUM = 0.95
MIN_GRAD = 0.01
SAVE_INTERVAL = 300000
NO_OP_STEPS = 30
LOAD_NETWORK = False
TRAIN = True
SAVE_NETWORK_PATH = 'saved_networks/' + ENV_NAME
SAVE_SUMMARY_PATH = 'summary/' + ENV_NAME
```

```
"""
#MovieLens100K
ITEM_HEADER = "movie id | movie title | release date | video release date | IMDb URL |
" \
                "unknown | Action | Adventure | Animation | Children's | Comedy | Crime
| " \
                "Documentary | Drama | Fantasy | Film-Noir | Horror | Musical | Mystery
| " \
                "Romance | Sci-Fi | Thriller | War | Western"
"""

class Agent():
    def __init__(self, num_states, num_actions):
        self.num_states = num_states
        self.num_actions = num_actions
        self.epsilon = INITIAL_EPSILON
        self.epsilon_step = (INITIAL_EPSILON - FINAL_EPSILON) /
EXPLORATION_STEPS
        self.t = 0
        # los parámetros
        self.total_reward = 0
        self.total_q_max = 0
        self.total_loss = 0
        self.duration = 0
        self.episode = 0

        # Experiencia Replay
        self.replay_memory = deque()

        # RED Q
        self.s, self.q_values, q_network = self.build_network()
        q_network_weights = q_network.trainable_weights

        # RED Target
        self.st, self.target_q_values, target_network = self.build_network()
        target_network_weights = target_network.trainable_weights
        self.update_target_network =
[target_network_weights[i].assign(q_network_weights[i]) for i in
range(len(target_network_weights))]
        self.a, self.y, self.loss, self.grads_update =
self.build_training_op(q_network_weights)
        self.sess = tf.InteractiveSession()
        self.saver = tf.train.Saver(q_network_weights)
```

```
            self.summary_placeholders, self.update_ops, self.summary_op = self.setup_summary()
            self.summary_writer = tf.summary.FileWriter(SAVE_SUMMARY_PATH, self.sess.graph)

            if not os.path.exists(SAVE_NETWORK_PATH):
                os.makedirs(SAVE_NETWORK_PATH)

            self.sess.run(tf.initialize_all_variables())

            # RED Q
            if LOAD_NETWORK:
                self.load_network()

            # Inicializar RED Target
            self.sess.run(self.update_target_network)

        #Huber Loss
        def _huber_loss(self, y_true, y_pred, clip_delta=1.0):
            error = y_true - y_pred
            cond  = K.abs(error) <= clip_delta
            squared_loss = 0.5 * K.square(error)   #mse
            quadratic_loss = clip_delta * (K.abs(error) - 0.5 * clip_delta)   #mae
            return K.mean(tf.where(cond, squared_loss, quadratic_loss))

        def build_network(self):
            model = Sequential()
            model.add(Dense(24, input_dim=self.num_states, activation='relu'))
            model.add(Dense(24, activation='relu'))
            model.add(Dense(self.num_actions, activation='linear'))
            s = tf.placeholder(tf.float32, [None, self.num_states])
            q_values = model(s)
            return s, q_values, model

        def build_training_op(self, q_network_weights):
            a = tf.placeholder(tf.int64, [None])
            y = tf.placeholder(tf.float32, [None])
            # Convert action to one hot vector
            a_one_hot = tf.one_hot(a, self.num_actions, 1.0, 0.0)
            q_value = tf.reduce_sum(tf.multiply(self.q_values, a_one_hot), reduction_indices=1)
            # Huber_loss para actualizar los parámetros
            loss = self._huber_loss(y,q_value)
            optimizer = tf.train.RMSPropOptimizer(LEARNING_RATE,
```

```
momentum=MOMENTUM, epsilon=MIN_GRAD)
        grads_update = optimizer.minimize(loss, var_list=q_network_weights)

        return a, y, loss, grads_update

    # Acción aleatoria con la probabilidad de decaimiento de épsilon
    def get_action(self, state):
        if self.epsilon >= random.random() or self.t < INITIAL_REPLAY_SIZE:
            action = random.randrange(self.num_actions)
        else:
            action = np.argmax(self.q_values.eval(feed_dict={self.s:
[np.float32(state)]}))

        if self.epsilon > FINAL_EPSILON and self.t >= INITIAL_REPLAY_SIZE:
            self.epsilon -= self.epsilon_step
        return action

    def run(self, state, action, reward, terminal, next_state):
        reward = np.clip(reward, -1, 1)
        self.replay_memory.append((state, action, reward, next_state, terminal))
        if len(self.replay_memory) > NUM_REPLAY_MEMORY:
            self.replay_memory.popleft()

        if self.t >= INITIAL_REPLAY_SIZE:
            # RED Q
            if self.t % TRAIN_INTERVAL == 0:
                self.train_network()
            # Actualizar RED Target
            if self.t % TARGET_UPDATE_INTERVAL == 0:
                self.sess.run(self.update_target_network)

            if self.t % SAVE_INTERVAL == 0:
                save_path = self.saver.save(self.sess, SAVE_NETWORK_PATH + '/'
+ ENV_NAME, global_step=self.t)
                print('Successfully saved: ' + save_path)

        self.total_reward += reward
        self.total_q_max += np.max(self.q_values.eval(feed_dict={self.s:
[np.float32(state)]}))
        self.duration += 1

        if terminal:
            if self.t >= INITIAL_REPLAY_SIZE:
                stats = [self.total_reward, self.total_q_max / float(self.duration),
```

```
                        self.duration, self.total_loss / (float(self.duration) /
float(TRAIN_INTERVAL))]
                    for i in range(len(stats)):
                        self.sess.run(self.update_ops[i], feed_dict={
                            self.summary_placeholders[i]: float(stats[i])
                        })
                    summary_str = self.sess.run(self.summary_op)
                    self.summary_writer.add_summary(summary_str, self.episode + 1)

                # Debug
                if self.t < INITIAL_REPLAY_SIZE:
                    mode = 'random'
                elif INITIAL_REPLAY_SIZE <= self.t < INITIAL_REPLAY_SIZE +
EXPLORATION_STEPS:
                    mode = 'explore'
                else:
                    mode = 'exploit'
                print('EPISODE: {0:6d} / TIMESTEP: {1:8d} / DURATION: {2:5d} /
EPSILON: {3:.5f} / TOTAL_REWARD: {4:3.0f} / AVG_MAX_Q: {5:2.4f} / AVG_LOSS:
{6:.5f} / MODE: {7}'.format(self.episode + 1, self.t, self.duration, self.epsilon,
                    self.total_reward, self.total_q_max / float(self.duration),
                    self.total_loss / (float(self.duration) / float(TRAIN_INTERVAL)),
mode))

                self.total_reward = 0
                self.total_q_max = 0
                self.total_loss = 0
                self.duration = 0
                self.episode += 1

            self.t += 1
            return next_state

        def train_network(self):
            state_batch = []
            action_batch = []
            reward_batch = []
            next_state_batch = []
            terminal_batch = []
            y_batch = []

            # Random minibatch
            minibatch = random.sample(self.replay_memory, BATCH_SIZE)
            for data in minibatch:
```

```
                state_batch.append(data[0])
                action_batch.append(data[1])
                reward_batch.append(data[2])
                next_state_batch.append(data[3])
                terminal_batch.append(data[4])

            terminal_batch = np.array(terminal_batch) + 0
            # DIFERENCIA entre DDQN Y DQN: next_action_batch
            next_action_batch = np.argmax(self.q_values.eval(feed_dict={self.s:
next_state_batch}), axis=1)
            target_q_values_batch = self.target_q_values.eval(feed_dict={self.st:
next_state_batch})
            for i in range(len(minibatch)):
                y_batch.append(reward_batch[i] + (1 - terminal_batch[i]) * GAMMA *
target_q_values_batch[i][next_action_batch[i]])
            loss, _ = self.sess.run([self.loss, self.grads_update], feed_dict={
                self.s: np.float32(np.array(state_batch)),
                self.a: action_batch,
                self.y: y_batch
            })
            self.total_loss += loss


    def setup_summary(self):
            episode_total_reward = tf.Variable(0.)
            tf.summary.scalar(ENV_NAME + '/Total Reward/Episode',
episode_total_reward)
            episode_avg_max_q = tf.Variable(0.)
            tf.summary.scalar(ENV_NAME + '/Average Max Q/Episode',
episode_avg_max_q)
            episode_duration = tf.Variable(0.)
            tf.summary.scalar(ENV_NAME + '/Duration/Episode', episode_duration)
            episode_avg_loss = tf.Variable(0.)
            tf.summary.scalar(ENV_NAME + '/Average Loss/Episode', episode_avg_loss)
            summary_vars = [episode_total_reward, episode_avg_max_q,
episode_duration, episode_avg_loss]
            summary_placeholders = [tf.placeholder(tf.float32) for _ in
range(len(summary_vars))]
            update_ops = [summary_vars[i].assign(summary_placeholders[i]) for i in
range(len(summary_vars))]
            summary_op = tf.summary.merge_all()
            return summary_placeholders, update_ops, summary_op

    def load_network(self):
```

```
            checkpoint = tf.train.get_checkpoint_state(SAVE_NETWORK_PATH)
            if checkpoint and checkpoint.model_checkpoint_path:
                self.saver.restore(self.sess, checkpoint.model_checkpoint_path)
                print('Successfully loaded: ' + checkpoint.model_checkpoint_path)
            else:
                print('Training new network...')

        def get_action_at_test(self, state):
            if random.random() <= 0.05:
                action = random.randrange(self.num_actions)
            else:
                action = np.argmax(self.q_values.eval(feed_dict={self.s:
np.float32(state)}))
            self.t += 1
            return action

    # Double DQN
    class DDQNAL:
        def __init__(self):
            self.algo = self
            self.name = "DoubleDQN"    #nombre del algoritmo


        def fit(self, trainSet):
            def makeTrainSetAvailable(trainSet):
                user_list = []
                item_list = []
                rating_list = []
                for (uid, iid, rating) in trainSet.all_ratings():
                    outer_user = trainSet.to_raw_uid(uid)
                    outer_item = trainSet.to_raw_iid(iid)
                    outer_rating = rating
                    user_list.append(float(outer_user))
                    item_list.append(float(outer_item))
                    rating_list.append(float(rating))
                train_dict = {'userId': user_list, 'itemId': item_list, 'rating':rating_list}
                data = pd.DataFrame(train_dict)

                #MovieLensSLD
                item = pd.read_csv(
                    'data/ml-latest-small/movies.csv',
                    header=0,
                    delimiter=',',
                    names=['itemId', 'itemName', 'genres'],
```

```
                    encoding='latin-1'
                )
            """
                #MovieLens100K
                item = pd.read_csv(
                    'data/ml-100k/u.item',
                    header=0,
                    delimiter='|',
                    names=convert_header_to_camel_case(ITEM_HEADER),
                    encoding='latin-1'
                )
                """

                kwargs = dict([(label, data) for label, data in zip(['data', 'item'], [data,
item])])
                return kwargs

            #Entorno
            env = gym.make(RecoEnvLatest.id, **makeTrainSetAvailable(trainSet))
            self.env = env
            num_states = env.observation_space.shape[0]
            num_actions = env.action_space.n
            agent = Agent(num_states, num_actions)
            self.agent = agent
            for _ in range(NUM_EPISODES):
                terminal = False
                state = env.reset()
                while not terminal:
                    action = agent.get_action(state)
                    next_state, reward, terminal, _ = env.step(action)
                    state = agent.run(state, action, reward, terminal, next_state)


        def test(self, testSet) -> list():
            predictions = list()
            state_size = self.env.observation_space.shape[0]
            state = self.env.reset()
            state = np.reshape(state, [1, state_size])
            for user, item, rating in testSet:
                state[0][state_size - 2] = float(user)
                state[0][state_size - 1] = float(item)
                est = self.agent.get_action_at_test(state)
                predictions.append(Prediction(uid=user,      iid=item,      r_ui=float(rating),
est=est, details={}))
```

```
        return predictions

    def run(self):
        def LoadMovieLensData():
            ml = MovieLens()
            data = ml.loadMovieLensLatestSmall() #MovieLensSLD
            #data = ml.loadMovieLens100k() #MovieLens100K
            rankings = ml.getPopularityRanks()
            return (ml, data, rankings)

        np.random.seed(0)
        random.seed(0)
        (ml, evaluationData, rankings) = LoadMovieLensData()
        # validación cruzada
        cross_validate(self.algo, evaluationData, measures=['RMSE', 'MAE'], cv=5,
verbose=True)
        # evaluación
        evaluator = Evaluator(evaluationData, rankings)
        evaluator.AddAlgorithm(self.algo, self.name)
        evaluator.Evaluate(True)
```

4. Entorno de los algoritmos de Reinforcement Learning (RL)

Conforme a lo mencionado en el apartado 5.2.3, hay diferencia entre las características del conjunto MovieLensSLD y las del MovieLens100K, por lo que se han creado 2 entornos de RL por separado para cada dataset y se han guardado en la carpeta *gym_recommendation* dentro de la carpeta *DDQNModule*.

(1) MovieLensSLD

Archivo: __init__.py

```
from DDQNModule.gym_recommendation.envs.reco_env import RecoEnvLatest
```

Archivo: reco_env.py

```
# Librerías
from typing import Dict, List, Tuple, Union
import numpy as np
import pandas as pd
from gym import Env
from gym import spaces

#MovieLensSLD
class RecoEnvLatest(Env):
    metadata = {'render.modes': ['human', 'logger']}
    id = 'reco-v1'
```

```
    actions = np.eye(5)   # Matriz 5x5

    def __init__(self,
                 data: pd.DataFrame,
                 item: pd.DataFrame,
                 seed: int = 1):

        self.data = data   # ratings
        self.item = item   # items
        self.movie_genre = self._get_movie_genre(item=self.item)
        self.user_mean = self.data.groupby('userId').mean().to_dict()['rating']
        self.movie_mean = self.data.groupby('itemId').mean().to_dict()['rating']
        # MDP variables
        self.reward = 0.0   #al inicio, reward=0
        self.done = False
        self.observation = None
        self.action = 0

        self.local_step_number = 0
        self._seed = seed
        self._random_state = np.random.RandomState(seed=self._seed)    #Random
state
        self.max_step = self.data.shape[0] - 2
        self.total_correct_predictions = 0

        self.data = self.data.values

        self.action_space = spaces.Discrete(len(RecoEnvLatest.actions))
        self.observation_space = spaces.Box(low=-1., high=5.0,
                                            shape=self._get_observation(
                                                step_number=0).shape,
                                            dtype=np.float32)

    def step(self, action: int = 0) -> Tuple[np.ndarray, float, bool, dict]:
        if self.done:
            self.observation = self.reset()
            return self.observation, self.reward, self.done, {}
        self.action = action
        self.reward                    =                    self._get_reward(action=action,
step_number=self.local_step_number)
        self.observation                                                              =
self._get_observation(step_number=self.local_step_number)
        if self.reward > 0:
            self.total_correct_predictions += 1
```

```
            if self.local_step_number >= self.max_step:
                self.done = True
            self.local_step_number += 1
            return self.observation, self.reward, self.done, {}

        # reset state
        def reset(self) -> np.ndarray:
            self.local_step_number = 0
            self.reward = 0.0
            self.done = False
            # print(f"Reco is being reset() --> "
            #       f"first step = {self.local_step_number} | "
            #       f"Total_correct = {self.total_correct_predictions}")
            self.total_correct_predictions = 0
            return self._get_observation(step_number=self.local_step_number)

        def render(self, mode: str = 'human') -> None:
            if mode == 'logger':
                print(f"Env observation at step {self.local_step_number} is
\n{self.observation}")


        # semilla aleatoria
        def seed(self, seed: int = 1) -> List[int]:
            self._random_state = np.random.RandomState(seed=seed)
            self._seed = seed
            return [seed]

        def __str__(self) -> str:
            return f'GymID={RecoEnv.id} | seed={self._seed}'

        @staticmethod
#este decorator sirve para usar el method sin necesidad de crear un objeto
        def _one_hot(num: int, selection: int) -> np.ndarray:
            return np.eye(num, dtype=np.float32)[selection]

        @staticmethod
        def _get_movie_genre(item: pd.DataFrame) -> Dict[int, np.ndarray]:
            ITEM_CLASS_LATEST = { \
                'Action': 0, 'Adventure': 1, 'Animation': 2, "Children": 3, 'Comedy': 4, 'Crime':
5, \
                'Documentary': 6, 'Drama': 7, 'Fantasy': 8, 'Film-Noir': 9, 'Horror': 10,
'Musical': 11, \
                'Mystery': 12, 'Romance': 13, 'Sci-Fi': 14, 'Thriller': 15, 'War': 16, 'Western':
```

```
17, \
                'IMAX': 18, '(no genres listed)': 19}

            movie_genre = dict([(movie_id, np.zeros(len(ITEM_CLASS_LATEST),
dtype=np.float32))
                                for movie_id in item['itemId'].tolist()])

            for raw_index in item.index:
                movie_id = item.iloc[raw_index, 0]
                for genre in item.iloc[raw_index, 2].split('|'):
                    if genre not in ITEM_CLASS_LATEST.keys():
                        #en toal 20 géneros cinematográficos, index se conta desde
0-19
                        index = 19
                    else:
                        index = int(ITEM_CLASS_LATEST[genre])
                    movie_genre[movie_id][index] = 1.0
            return movie_genre

        @staticmethod
        def _get_user_data(user: pd.DataFrame) -> Dict[int, Dict[str, Union[int, str]]]:
            tmp_user = user.drop(['zip_code'], axis=1)
            tmp_user.index = tmp_user.user_id
            tmp_user = tmp_user.drop(['user_id'], axis=1)
            return tmp_user.to_dict(orient='index')

        def _get_movie_genre_buckets(self, movie_id: int = 1) -> np.ndarray:
            return self.movie_genre.get(movie_id, np.empty(20, dtype=np.float32))

        def _get_observation(self, step_number: int = 0) -> np.ndarray:
            user_id = self.data[step_number, 0]
            movie_id = self.data[step_number, 1]
            user_mean = np.array([self.user_mean.get(user_id, 3.) / 5.], dtype=np.float32)
            movie_mean = np.array([self.movie_mean.get(movie_id, 3.) / 5.],
dtype=np.float32)
            movie_genre_bucket = self._get_movie_genre_buckets(movie_id=movie_id)

            return np.concatenate((user_mean, movie_mean, movie_genre_bucket,
[user_id], [movie_id]))

        # reward
        def _get_reward(self, action: int, step_number: int) -> float:
            users_rating = float(self.data[step_number, 2])
            predicted_rating = int(action) + 1
```

```
        prediction_difference = abs(predicted_rating - users_rating)
        reward = 0.
        if prediction_difference <= 0.5:
            reward += 1.
        elif prediction_difference == 5:
            return -10
        else:
            reward += np.log(1. - prediction_difference / 5)
        return reward
```

(2) MovieLens100K

Archivo: __init__.py

```
from DDQNModule.gym_recommendation.envs.reco_env import RecoEnvLatest
```

Archivo: reco_env.py

```
# Librerías
from typing import Dict, List, Tuple, Union
import numpy as np
import pandas as pd
from gym import Env
from gym import spaces

#MovieLens100K
class RecoEnvLatest(Env):
    metadata = {'render.modes': ['human', 'logger']}
    id = 'reco-v1'
    actions = np.eye(5)   # Matriz 5x5

    def __init__(self,
                 data: pd.DataFrame,
                 item: pd.DataFrame,
                 seed: int = 1):

        self.data = data   # ratings
        self.item = item   # items
        self.movie_genre = self._get_movie_genre(item=self.item)
        self.user_mean = self.data.groupby('userId').mean().to_dict()['rating']
        self.movie_mean = self.data.groupby('itemId').mean().to_dict()['rating']
        # MDP variables
        self.reward = 0.0   #al inicio, reward=0
        self.done = False
        self.observation = None
        self.action = 0
```

```
            self.local_step_number = 0
            self._seed = seed
            self._random_state = np.random.RandomState(seed=self._seed)
            self.max_step = self.data.shape[0] - 2
            self.total_correct_predictions = 0
            self.data = self.data.values
            self.action_space = spaces.Discrete(len(RecoEnvLatest.actions))
            self.observation_space = spaces.Box(low=-1., high=5.0,
                                                shape=self._get_observation(
                                                    step_number=0).shape,
                                                dtype=np.float32)

        def step(self, action: int = 0) -> Tuple[np.ndarray, float, bool, dict]:
            if self.done:
                self.observation = self.reset()
                return self.observation, self.reward, self.done, {}
            self.action = action
            self.reward = self._get_reward(action=action,
step_number=self.local_step_number)
            self.observation =
self._get_observation(step_number=self.local_step_number)
            if self.reward > 0:
                self.total_correct_predictions += 1
            if self.local_step_number >= self.max_step:
                self.done = True
            self.local_step_number += 1
            return self.observation, self.reward, self.done, {}

        def reset(self) -> np.ndarray:
            self.local_step_number = 0
            self.reward = 0.0
            self.done = False
            # print(f"Reco is being reset() --> "
            #       f"first step = {self.local_step_number} | "
            #       f"Total_correct = {self.total_correct_predictions}")
            self.total_correct_predictions = 0
            return self._get_observation(step_number=self.local_step_number)

        def render(self, mode: str = 'human') -> None:
            if mode == 'logger':
                print(f"Env observation at step {self.local_step_number} is
\n{self.observation}")

        # semilla aleatoria
```

```
    def seed(self, seed: int = 1) -> List[int]:
        self._random_state = np.random.RandomState(seed=seed)
        self._seed = seed
        return [seed]

    def __str__(self) -> str:
        return f'GymID={RecoEnv.id} | seed={self._seed}'

    @staticmethod
    def _one_hot(num: int, selection: int) -> np.ndarray:
        return np.eye(num, dtype=np.float32)[selection]

    @staticmethod
    # MovieLens100k solo tiene 19 géneros cinematográfico
    def _get_movie_genre(item: pd.DataFrame) -> Dict[int, np.ndarray]:
        movie_genre = dict([(movie_id, np.empty(19, dtype=np.float32))
                                    for movie_id in item['movie_id'].tolist()])
        for movie_id in range(1, len(movie_genre)):
            movie_genre[movie_id] = item.iloc[movie_id, 5:].values.astype(np.float32)
        return movie_genre
        # ITEM_CLASS_LATEST = { \
        #          'Action': 0, 'Adventure': 1, 'Animation': 2, "Children": 3, 'Comedy': 4,
'Crime': 5, \
        #          'Documentary': 6, 'Drama': 7, 'Fantasy': 8, 'Film-Noir': 9, 'Horror': 10,
'Musical': 11, \
        #           'Mystery': 12, 'Romance': 13, 'Sci-Fi': 14, 'Thriller': 15, 'War': 16,
'Western': 17, \
        #        'IMAX': 18, '(no genres listed)': 19}
        #
        #   movie_genre   =   dict([(movie_id,   np.zeros(len(ITEM_CLASS_LATEST),
dtype=np.float32))
        #                                for movie_id in item['itemId'].tolist()])
        #
        # for raw_index in item.index:
        #        movie_id = item.iloc[raw_index, 0]
        #        for genre in item.iloc[raw_index, 2].split('|'):
        #            if genre not in ITEM_CLASS_LATEST.keys():
        #                index = 19
        #            else:
        #                index = int(ITEM_CLASS_LATEST[genre])
        #            movie_genre[movie_id][index] = 1.0
        # return movie_genre

    @staticmethod
```

```
    def _get_user_data(user: pd.DataFrame) -> Dict[int, Dict[str, Union[int, str]]]:
        tmp_user = user.drop(['zip_code'], axis=1)
        tmp_user.index = tmp_user.user_id
        tmp_user = tmp_user.drop(['user_id'], axis=1)
        return tmp_user.to_dict(orient='index')

    def _get_movie_genre_buckets(self, movie_id: int = 1) -> np.ndarray:
        return self.movie_genre.get(movie_id, np.empty(20, dtype=np.float32))

    def _get_observation(self, step_number: int = 0) -> np.ndarray:
        user_id = self.data[step_number, 0]
        movie_id = self.data[step_number, 1]
        user_mean = np.array([self.user_mean.get(user_id, 3.) / 5.], dtype=np.float32)
        movie_mean = np.array([self.movie_mean.get(movie_id, 3.) / 5.],
dtype=np.float32)
        movie_genre_bucket = self._get_movie_genre_buckets(movie_id=movie_id)

        return np.concatenate((user_mean, movie_mean, movie_genre_bucket,
[user_id], [movie_id]))

    # reward
    def _get_reward(self, action: int, step_number: int) -> float:
        users_rating = float(self.data[step_number, 2])
        predicted_rating = int(action) + 1
        prediction_difference = abs(predicted_rating - users_rating)
        reward = 0.
        if prediction_difference <= 0.5:   #rating range：0-5
            reward += 1.
        elif prediction_difference == 5:
            return -10
        else:
            reward += np.log(1. - prediction_difference / 5)
        return reward
```

5. Métodos de evaluación de rendimiento

Los métodos de evaluación de rendimiento se guardan en un archivo *RecommenderMetrics.py* dentro de la carpeta *BaseModules.* De hecho, en dicha carpeta hay en total 6 archivos para llevar a cabo una serie de acciones: la importación de datos (*MovieLens.py*), descripción de datos (*DataCheck.py*), análisis de datos (EvaluationData.py) y evaluación de distintos modelos (*RecommenderMetrics.py, Evaluator.py* y *EvaluatedAlgorithm.py*).

Archivo: DataCheck.py

```
# Librerías
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)
import os
import numpy as np
import random
import pandas as pd
from surprise import Dataset
from surprise import Reader
from surprise.model_selection import PredefinedKFold

# Importar los datos ya depurados
def checkData():
    movieReader = Reader(line_format='user item rating timestamp', sep=',',
skip_lines=1)#MovieLensSLD
    #movieReader = Reader(line_format='user item rating timestamp', sep='\t',
skip_lines=0)    #MovieLens100K

    current_dir = os.path.dirname(os.path.realpath(__file__))
    folds_files = [(current_dir.split('BaseModules')[0] + 'data/ml-latest-small/ratings.csv',
                    current_dir.split('BaseModules')[0]                              +
'data/ml-latest-small/ratings.csv')]
    """
    #MovieLens100K
    folds_files = [(current_dir.split('BaseModules')[0] + 'data/ml-100k/u.data',
                    current_dir.split('BaseModules')[0] + 'data/ml-100k/u.data')]
    """

    data = Dataset.load_from_folds(folds_files=folds_files,
                                   reader=movieReader)

    pkf = PredefinedKFold()
    trainset, testset = next(pkf.split(data))

    #MovieLensSLD
    assert trainset.n_items == 9724
    assert trainset.n_ratings == 100836
    assert trainset.n_users == 610
    assert trainset.rating_scale == (1, 5)
    """
    #MovieLens100K
    assert trainset.n_items == 1682
    assert trainset.n_ratings == 100000
    assert trainset.n_users == 943
```

```
    assert trainset.rating_scale == (1, 5)
    """
    print("OK")

def viewData():
    def load_data():
        #MovieLensSLD
        #Ratings
        ratings = pd.read_csv("......directorio/data/ml-latest-small/ratings.csv")
        data = ratings.groupby("userId", as_index=False).agg({"movieId": 'count'})
        #Movies
        movies_table = pd.read_csv("......directorio/data/ml-latest-small/movies.csv")

        """
        #MovieLens100K
        # Ratings
        rating_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
        ratings = pd.read_csv("......directorio/data/ml-100k/u.data",
sep='\t',names=rating_cols)
        data = ratings.groupby("user_id", as_index=False).agg({"movie_id": 'count'})
        # Movies
        movie_cols = ['movie_id', 'title', 'release_date', 'video_release_date', 'imdb_url']
        movies_table = pd.read_csv('......directorio/data/ml-100k/u.item', sep='|',
                                   names=movie_cols, usecols=range(5),
encoding='latin-1')
        """

        return ratings,data,movies_table

    (ratings,data,movies_table)=load_data()
    print("##########TABLA DE RATING##########")
    print(ratings.head(10))
    print("")
    print("##########NUMBER OF MOVIES RATED PER USER##########")
    print(data.movieId.describe())
    print("")
    print("##########TABLA DE MOVIES##########")
    print(movies_table.head(10))
```

Archivo: MovieLens.py

```
# Librerías
import os
```

```
import csv
import sys
import random
import numpy as np
from surprise import Dataset
from surprise import Reader
from collections import defaultdict

class MovieLens:
    movieID_to_name = {}
    name_to_movieID = {}
    #MovieLensSLD
    ratingsPath = 'data/ml-latest-small/ratings.csv'
    moviesPath = 'data/ml-latest-small/movies.csv'
    """
    #MovieLens100K
    ratingsPath = 'data/ml-100k/u.data'
    moviesPath = 'data/ml-100k/u.item'
    """

    #MovieLensSLD
    def loadMovieLensLatestSmall(self):
        os.chdir(os.path.dirname(sys.argv[0]))
        ratingsDataset = 0
        self.movieID_to_name = {}
        self.name_to_movieID = {}
        reader = Reader(line_format='user item rating timestamp', sep=',',
skip_lines=1)
        ratingsDataset = Dataset.load_from_file(self.ratingsPath, reader=reader)

        with open(self.moviesPath, newline=' ', encoding='ISO-8859-1') as csvfile:
                movieReader = csv.reader(csvfile)
                next(movieReader)
                for row in movieReader:
                    movieID = int(row[0])
                    movieName = row[1]
                    self.movieID_to_name[movieID] = movieName
                    self.name_to_movieID[movieName] = movieID

        return ratingsDataset

    #MovieLens100K
    def loadMovieLens100k(self):
        os.chdir(os.path.dirname(sys.argv[0]))
```

```
        ratingsDataset = 0
        self.movieID_to_name = {}
        self.name_to_movieID = {}
        reader = Reader(line_format='user item rating timestamp', sep='\t',
skip_lines=0)
        ratingsDataset = Dataset.load_from_file(self.ratingsPath, reader=reader)
        with open(self.moviesPath, newline=' ', encoding='ISO-8859-1') as csvfile:
                for line in csvfile:
                        row = line.split("|")
                        movieID = int(row[0])
                        movieName = row[1]
                        self.movieID_to_name[movieID] = movieName
                        self.name_to_movieID[movieName] = movieID

        return ratingsDataset

    def getUserRatings(self, user):
        userRatings = []
        hitUser = False
        with open(self.ratingsPath, newline=' ') as csvfile:
            ratingReader = csv.reader(csvfile)
            next(ratingReader)
            for row in ratingReader:
                userID = int(row[0])
                if (user == userID):
                    movieID = int(row[1])
                    rating = float(row[2])
                    userRatings.append((movieID, rating))
                    hitUser = True
                if (hitUser and (user != userID)):
                    break

        return userRatings


    def getPopularityRanks(self):
        ratings = defaultdict(int)
        rankings = defaultdict(int)
        with open(self.ratingsPath, newline=' ') as csvfile:
                ratingReader = csv.reader(csvfile) #MovieLensSLD
                #ratingReader = csv.reader(csvfile, delimiter='\t') #MovieLens100K
            next(ratingReader)
            for row in ratingReader:
                movieID = int(row[1])
```

```
                ratings[movieID] += 1
        rank = 1
        for movieID, ratingCount in sorted(ratings.items(), key=lambda x: x[1],
reverse=True):
            rankings[movieID] = rank
            rank += 1
        return rankings

    def getGenres(self):
        genres = defaultdict(list)
        genreIDs = {}
        maxGenreID = 0
        #MovieLensSLD
        with open(self.moviesPath, newline=' ', encoding='ISO-8859-1') as csvfile:
            movieReader = csv.reader(csvfile)
            next(movieReader)
            for row in movieReader:
                movieID = int(row[0])
                genreList = row[2].split('|')
                genreIDList = []
                for genre in genreList:
                    if genre in genreIDs:
                        genreID = genreIDs[genre]
                    else:
                        genreID = maxGenreID
                        genreIDs[genre] = genreID
                        maxGenreID += 1
                    genreIDList.append(genreID)
                genres[movieID] = genreIDList
        for (movieID, genreIDList) in genres.items():
            bitfield = [0] * maxGenreID
            for genreID in genreIDList:
                bitfield[genreID] = 1
            genres[movieID] = bitfield

        """
        #MovieLens100K
        with open(self.moviesPath, newline=' ', encoding='ISO-8859-1') as csvfile:
            movieReader = csv.reader(csvfile, delimiter='|')
            for row in movieReader:
                movieID = int(row[0])
                genreList = [int(x) for x in row[5:24]]
                genres[movieID] = genreList
        """
```

```
        return genres

    def getYears(self):
        p = re.compile(r"(?:\((\d{4})\))?\s*$")
        years = defaultdict(int)
        with open(self.moviesPath, newline=' ', encoding='ISO-8859-1') as csvfile:
            movieReader = csv.reader(csvfile)   #MovieLensSLD
            #movieReader = csv.reader(csvfile, delimiter='|')   #MovieLens100K
            next(movieReader)
            for row in movieReader:
                movieID = int(row[0])
                title = row[1]
                m = p.search(title)
                year = m.group(1)
                if year:
                    years[movieID] = int(year)
        return years

    def getMiseEnScene(self):
        mes = defaultdict(list)
        with open("data/LLVisualFeatures13K_Log.csv", newline=' ') as csvfile:
            mesReader = csv.reader(csvfile)
            next(mesReader)
            for row in mesReader:
                movieID = int(row[0])
                avgShotLength = float(row[1])
                meanColorVariance = float(row[2])
                stddevColorVariance = float(row[3])
                meanMotion = float(row[4])
                stddevMotion = float(row[5])
                meanLightingKey = float(row[6])
                numShots = float(row[7])
                mes[movieID] = [avgShotLength, meanColorVariance,
stddevColorVariance,
                    meanMotion, stddevMotion, meanLightingKey, numShots]
        return mes

    def getMovieName(self, movieID):
        if movieID in self.movieID_to_name:
            return self.movieID_to_name[movieID]
        else:
            return " "

    def getMovieID(self, movieName):
```

```
        if movieName in self.name_to_movieID:
            return self.name_to_movieID[movieName]
        else:
            return 0
```

Archivo: EvaluationData.py

```
# Librerías
from surprise.model_selection import train_test_split
from surprise.model_selection import LeaveOneOut
from surprise import KNNBaseline


class EvaluationData:
    def __init__(self, data, popularityRankings):
        self.rankings = popularityRankings
        self.fullTrainSet = data.build_full_trainset()
        self.fullAntiTestSet = self.fullTrainSet.build_anti_testset()
        self.trainSet, self.testSet = train_test_split(data, test_size=.25,
random_state=1)
        LOOCV = LeaveOneOut(n_splits=1, random_state=1) #leave one out(LOO)
        for train, test in LOOCV.split(data):
            self.LOOCVTrain = train
            self.LOOCVTest = test

        self.LOOCVAntiTestSet = self.LOOCVTrain.build_anti_testset()
        sim_options = {'name': 'cosine', 'user_based': False}
        self.simsAlgo = KNNBaseline(sim_options=sim_options)
        self.simsAlgo.fit(self.fullTrainSet)

    def GetFullTrainSet(self):
        return self.fullTrainSet

    def GetFullAntiTestSet(self):
        return self.fullAntiTestSet

    def GetAntiTestSetForUser(self, testSubject):
        trainset = self.fullTrainSet
        fill = trainset.global_mean
        anti_testset = []
        u = trainset.to_inner_uid(str(testSubject))
        user_items = set([j for (j, _) in trainset.ur[u]])
        anti_testset += [(trainset.to_raw_uid(u), trainset.to_raw_iid(i), fill) for
                                    i in trainset.all_items() if
                                    i not in user_items]
```

```
        return anti_testset

    def GetTrainSet(self):
        return self.trainSet

    def GetTestSet(self):
        return self.testSet

    def GetLOOCVTrainSet(self):
        return self.LOOCVTrain

    def GetLOOCVTestSet(self):
        return self.LOOCVTest

    def GetLOOCVAntiTestSet(self):
        return self.LOOCVAntiTestSet

    def GetSimilarities(self):
        return self.simsAlgo

    def GetPopularityRankings(self):
        return self.rankings
```

Archivo: RecommenderMetrics.py

```
# Librerías
import itertools
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from surprise import accuracy
from collections import defaultdict

#las medidas de evaluación:
class RecommenderMetrics:
    #MAE
    def MAE(predictions):
        return accuracy.mae(predictions, verbose=False)
    #RMSE
    def RMSE(predictions):
        return accuracy.rmse(predictions, verbose=False)

    def GetTopN(predictions, n=10, minimumRating=4):
        topN = defaultdict(list)
```

```
        for userID, movieID, actualRating, estimatedRating, _ in predictions:
            if (estimatedRating >= minimumRating):
                topN[int(userID)].append((int(movieID), estimatedRating))
        for userID, ratings in topN.items():
            ratings.sort(key=lambda x: x[1], reverse=True)
            topN[int(userID)] = ratings[:n]
        return topN

    # HR
    def HitRate(topNPredicted, leftOutPredictions):
        hits = 0
        total = 0
        for leftOut in leftOutPredictions:
            userID = leftOut[0]
            leftOutMovieID = leftOut[1]
            hit = False
            for movieID, predictedRating in topNPredicted[int(userID)]:
                if (int(leftOutMovieID) == int(movieID)):
                    hit = True
                    break
            if (hit) :
                hits += 1

            total += 1

        return hits/total

    # cHR
    def CumulativeHitRate(topNPredicted, leftOutPredictions, ratingCutoff=4):
        hits = 0
        total = 0
        for userID, leftOutMovieID, actualRating, estimatedRating, _ in
leftOutPredictions:
            if (actualRating >= ratingCutoff):
                hit = False
                for movieID, predictedRating in topNPredicted[int(userID)]:
                    if (int(leftOutMovieID) == movieID):
                        hit = True
                        break
                if (hit) :
                    hits += 1

                total += 1
        return hits/total
```

```
    def RatingHitRate(topNPredicted, leftOutPredictions):
        hits = defaultdict(float)
        total = defaultdict(float)

        for userID, leftOutMovieID, actualRating, estimatedRating, _ in
leftOutPredictions:
            # Is it in the predicted top N for this user?
            hit = False
            for movieID, predictedRating in topNPredicted[int(userID)]:
                if (int(leftOutMovieID) == movieID):
                    hit = True
                    break
            if (hit) :
                hits[actualRating] += 1

            total[actualRating] += 1

        for rating in sorted(hits.keys()):
            print (rating, hits[rating] / total[rating])


    # AverageReciprocalHitRank(ARHR)
    def AverageReciprocalHitRank(topNPredicted, leftOutPredictions):
        summation = 0
        total = 0
        for userID, leftOutMovieID, actualRating, estimatedRating, _ in
leftOutPredictions:
            hitRank = 0
            rank = 0
            for movieID, predictedRating in topNPredicted[int(userID)]:
                rank = rank + 1
                if (int(leftOutMovieID) == movieID):
                    hitRank = rank
                    break
            if (hitRank > 0) :
                summation += 1.0 / hitRank

            total += 1
        return summation / total

    # Diversidad
    def Diversity(topNPredicted, simsAlgo):
        n = 0
```

```
    total = 0
    simsMatrix = simsAlgo.compute_similarities()
    for userID in topNPredicted.keys():
        pairs = itertools.combinations(topNPredicted[userID], 2)
        for pair in pairs:
            movie1 = pair[0][0]
            movie2 = pair[1][0]
            innerID1 = simsAlgo.trainset.to_inner_iid(str(movie1))
            innerID2 = simsAlgo.trainset.to_inner_iid(str(movie2))
            similarity = simsMatrix[innerID1][innerID2]
            total += similarity
            n += 1

    if n == 0:
        return 555
    else:
        return (1- total / n)

# Novedad
def Novelty(topNPredicted, rankings):
    n = 0
    total = 0
    for userID in topNPredicted.keys():
        for rating in topNPredicted[userID]:
            movieID = rating[0]
            rank = rankings[movieID]
            total += rank
            n += 1
    if n == 0:
        return 555
    else:
        return total / n

def getY(topNPredicted, testSet, cutoff=3):
    y = []     #valor real
    y_pred=[] #valor predictivo
    y_hat_prob = []
    for userID, movieID, rate in testSet:
        if rate > cutoff:
            y.append(1)
        else:
            y.append(0)

        if int(userID) in topNPredicted.keys():
```

```
                find = False
                for movieID2, rate2 in topNPredicted[int(userID)]:
                    if str(movieID2) == str(movieID):
                        y_pred.append(1)
                        y_hat_prob.append(rate2)
                        find = True
                        break
                if not find:
                    y_pred.append(0)
                    y_hat_prob.append(0)
            else:
                y_pred.append(0)
                y_hat_prob.append(0)

    return y, y_hat_prob, y_pred


def get_acc(y, y_hat):
    # Accuracy= (TP+TN)/TOTAL, siendo TOTAL=TP+FP+FN+TN
    true_positive = np.sum(np.logical_and(np.equal(y, 1), np.equal(y_hat, 1)))
    false_positive = np.sum(np.logical_and(np.equal(y, 0), np.equal(y_hat, 1)))
    true_negative= np.sum(np.logical_and(np.equal(y, 1), np.equal(y_hat, 0)))
    false_negative=np.sum(np.logical_and(np.equal(y, 0), np.equal(y_hat, 0)))
    TOTAL=true_positive+false_positive+true_negative+false_negative
    return (true_positive+false_positive)/TOTAL

def get_precision(y, y_hat):    #TP/(TP+FP)
    true_positive=np.sum(np.logical_and(np.equal(y, 1), np.equal(y_hat, 1)))
    false_positive= np.sum(np.logical_and(np.equal(y, 0), np.equal(y_hat, 1)))
    predicted_positive=true_positive+false_positive
    return true_positive / predicted_positive

def get_Inverse_precision(y, y_hat): #TN/TN+FN
    true_negative = np.sum(np.logical_and(np.equal(y, 1), np.equal(y_hat, 0)))
    false_negative = np.sum(np.logical_and(np.equal(y, 0), np.equal(y_hat, 0)))
    predicted_negative=true_negative+false_negative
    return true_negative / predicted_negative

def get_tpr(y, y_hat):    #TP/(TP+FN)
    true_positive = np.sum(np.logical_and(np.equal(y, 1), np.equal(y_hat, 1)))
    false_negative = np.sum(np.logical_and(np.equal(y, 0), np.equal(y_hat, 0)))
    actual_positive=true_positive+false_negative
    return true_positive / actual_positive
```

```
    def get_tnr(y, y_hat):     # TN/ (TN+FP)
        true_negative = np.sum(np.logical_and(np.equal(y, 1), np.equal(y_hat, 0)))
        false_positive = np.sum(np.logical_and(np.equal(y, 0), np.equal(y_hat, 1)))
        actual_negative = true_negative + false_positive
        return true_negative / actual_negative

    # Curva ROC
    def get_roc(y, y_hat_prob):
        thresholds = sorted(set(y_hat_prob), reverse=True)
        ret = [[0, 0]]
        for threshold in thresholds:
            y_hat = [int(yi_hat_prob >= threshold) for yi_hat_prob in y_hat_prob]
            ret.append([RecommenderMetrics.get_tpr(y, y_hat), 1 -
RecommenderMetrics.get_tnr(y, y_hat)])    #Recall, 1-Especificidad
        return ret

    # AUC
    def get_auc(y, y_hat_prob, points):
        roc = iter(points)
        tpr_pre, fpr_pre = next(roc)
        auc = 0
        for tpr, fpr in roc:
            auc += (tpr + tpr_pre) * (fpr - fpr_pre) / 2
            tpr_pre = tpr
            fpr_pre = fpr
        return auc

    # Dibujar Curva ROC
    def paintROC(topNPredicted, testSet):
        y,y_hat_prob,y_pred = RecommenderMetrics.getY(topNPredicted, testSet)
        points = RecommenderMetrics.get_roc(y, y_hat_prob)
        df = pd.DataFrame(points, columns=["tpr", "fpr"])
        fig = plt.figure()
        df.plot(x="fpr", y="tpr", label="roc")
        plt.title('Curva ROC', fontdict={'weight': 'normal', 'size': 20})
        plt.xlabel('1-Especificidad', fontdict={'weight': 'normal', 'size': 13})
        plt.ylabel('Sensibilidad ', fontdict={'weight': 'normal', 'size': 13})
        plt.show()
        return RecommenderMetrics.get_auc(y, y_hat_prob, points)

    #Precisión
    def Precision(topNPredicted, testSet):
        y,y_hat_prob,y_pred = RecommenderMetrics.getY(topNPredicted, testSet)
        precision = RecommenderMetrics.get_precision(y, y_pred)
```

```
        return precision

    # Recall
    def Recall(topNPredicted, testSet):
        y,y_hat_prob,y_pred = RecommenderMetrics.getY(topNPredicted, testSet)
        recall = RecommenderMetrics.get_tpr(y, y_pred)
        return recall

    # Especificidad
    def FPR(topNPredicted, testSet):
        y,y_hat_prob,y_pred = RecommenderMetrics.getY(topNPredicted, testSet)
        fpr = 1 - RecommenderMetrics.get_tnr(y, y_pred) l
        return fpr

    # F1_score
    def F1_score(topNPredicted, testSet):
        p = RecommenderMetrics.Precision(topNPredicted, testSet)
        r = RecommenderMetrics.Recall(topNPredicted, testSet)
        return 2*p*r/(p+r)

    def InversePrecision(topNPredicted, testSet):
        y,y_hat_prob,y_pred = RecommenderMetrics.getY(topNPredicted, testSet)
        Inver_precision = RecommenderMetrics.get_Inverse_precision(y, y_pred)
        return Inver_precision

    def InverseRecall(topNPredicted, testSet):
        y,y_hat_prob,y_pred = RecommenderMetrics.getY(topNPredicted, testSet)
        Inver_recall = RecommenderMetrics.get_tnr(y, y_pred)
        return Inver_recall

    # Markedness
    def Markedness(topNPredicted, testSet):
        p = RecommenderMetrics.Precision(topNPredicted, testSet)
        ip = RecommenderMetrics.InversePrecision(topNPredicted, testSet)
        return p+ip-1

    #Informedness
    def Informedness(topNPredicted, testSet):
        r = RecommenderMetrics.Precision(topNPredicted, testSet)
        ir = RecommenderMetrics.InverseRecall(topNPredicted, testSet)
        return r+ir-1
```

Archivo: Evaluator.py

```
# Librerías
from BaseModules.EvaluationData import EvaluationData
from BaseModules.EvaluatedAlgorithm import EvaluatedAlgorithm
from surprise.model_selection import cross_validate

# Evaluación para modelo simple
class Evaluator:
    algorithms = []

    def __init__(self, dataset, rankings):
        ed = EvaluationData(dataset, rankings)
        self.dataset = ed

    def AddAlgorithm(self, algorithm, name):
        alg = EvaluatedAlgorithm(algorithm, name)
        self.algorithms.append(alg)

    def Evaluate(self, doTopN):
        results = {}
        i = 1
        for algorithm in self.algorithms:
            print("************Algoritmo {} ******************".format(i))
            results[algorithm.GetName()] = algorithm.Evaluate(self.dataset, doTopN)
            i += 1
            if (doTopN):
                for (name, metrics) in results.items():
                    print("*********************************")
                    print("Algorithm : " + name)
                    print("-------------------------------")
                    for key in metrics.keys():
                        print("{:>15}".format(key)+"                :                "                +
"{:<10.4f}".format(metrics[key]))
                    print("*********************************")

    def SampleTopNRecs(self, ml, testSubject=85, k=10):
        for algo in self.algorithms:
            trainSet = self.dataset.GetFullTrainSet()
            algo.GetAlgorithm().fit(trainSet)
            testSet = self.dataset.GetAntiTestSetForUser(testSubject)
            predictions = algo.GetAlgorithm().test(testSet)
            recommendations = []
            for userID, movieID, actualRating, estimatedRating, _ in predictions:
                intMovieID = int(movieID)
                recommendations.append((intMovieID, estimatedRating))
```

```
                recommendations.sort(key=lambda x: x[1], reverse=True)
                for ratings in recommendations[:10]:
                    print(ml.getMovieName(ratings[0]), ratings[1])

# Evaluación para los modelos ensamblados
class EvaluatorForStacking:
    algorithms = []

    def __init__(self, dataset, rankings):
        ed = EvaluationData(dataset, rankings)
        self.dataset = ed

    def AddAlgorithm(self, algorithm, name):
        alg = EvaluatedAlgorithm(algorithm, name)
        self.algorithms.append(alg)

    def Evaluate(self, doTopN):
        results = {}
        for algorithm in self.algorithms:
            results[algorithm.GetName()]                                    =
algorithm.EvaluateForStacking(self.dataset, doTopN)
        if (doTopN):
            for (name, metrics) in results.items():
                print("********************************")
                print("Algorithm : " + name)
                print("--------------------------------")
                for key in metrics.keys():
                    print("{:>15}".format(key) + " : " + "{:<10.4f}".format(metrics[key]))
                print("********************************")
 def SampleTopNRecs(self, ml, testSubject=85, k=10):

        for algo in self.algorithms:
            trainSet = self.dataset.GetFullTrainSet()
            algo.GetAlgorithm().fit(trainSet)
            testSet = self.dataset.GetAntiTestSetForUser(testSubject)
            predictions = algo.GetAlgorithm().test(testSet)
            recommendations = []
            for userID, movieID, actualRating, estimatedRating, _ in predictions:
                intMovieID = int(movieID)
                recommendations.append((intMovieID, estimatedRating))

            recommendations.sort(key=lambda x: x[1], reverse=True)
            for ratings in recommendations[:10]:
                print(ml.getMovieName(ratings[0]), ratings[1])
```

Archivo: EvaluatedAlgorithm.py

```
# Librerías
import time
from BaseModules.RecommenderMetrics import RecommenderMetrics
from BaseModules.EvaluationData import EvaluationData

class EvaluatedAlgorithm:
    def __init__(self, algorithm, name):
        self.algorithm = algorithm
        self.name = name

    # Evaluación para modelo simple
    def Evaluate(self, evaluationData, doTopN, n=10, verbose=False):
        metrics = {}
        if (verbose):
            print("Evaluating RMSE and MAE")
        timeStart = time.time()
        self.algorithm.fit(evaluationData.GetTrainSet())
        print("Fit time:{:.2f}s".format(time.time()-timeStart))
        timeStart = time.time()
        predictions = self.algorithm.test(evaluationData.GetTestSet())
        print("Test time:{:.2f}s".format(time.time()-timeStart))
        metrics["RMSE"] = RecommenderMetrics.RMSE(predictions)
        metrics["MAE"] = RecommenderMetrics.MAE(predictions)


        if (doTopN):
            if (verbose):
                print("Evaluating top-N with leave-one-out...")
            topNPredictedPart = RecommenderMetrics.GetTopN(predictions, n)
            metrics["Precision"] = RecommenderMetrics.Precision(topNPredictedPart,
evaluationData.GetTestSet())
            metrics["Recall"]    =    RecommenderMetrics.Recall(topNPredictedPart,
evaluationData.GetTestSet())
            metrics["F1_score"] = RecommenderMetrics.F1_score(topNPredictedPart,
evaluationData.GetTestSet())
            metrics["AUC"]   =   RecommenderMetrics.paintROC(topNPredictedPart,
evaluationData.GetTestSet())
            metrics["Markedness"]                                                =
RecommenderMetrics.Markedness(topNPredictedPart, evaluationData.GetTestSet())
            metrics["Informedness"]                                              =
RecommenderMetrics.Informedness(topNPredictedPart, evaluationData.GetTestSet())
```

```
            self.algorithm.fit(evaluationData.GetLOOCVTrainSet())
            leftOutPredictions = self.algorithm.test(evaluationData.GetLOOCVTestSet())
            allPredictions = self.algorithm.test(evaluationData.GetLOOCVAntiTestSet())
            topNPredicted = RecommenderMetrics.GetTopN(allPredictions, n)

            if (verbose):
                print("Computing hit-rate and rank metrics...")
            metrics["HR"] = RecommenderMetrics.HitRate(topNPredicted, leftOutPredictions)
            metrics["cHR"] = RecommenderMetrics.CumulativeHitRate(topNPredicted, leftOutPredictions)
            metrics["ARHR"] = RecommenderMetrics.AverageReciprocalHitRank(topNPredicted, leftOutPredictions)

            if (verbose):
                print("Computing recommendations with full data set...")
            self.algorithm.fit(evaluationData.GetFullTrainSet())
            allPredictions = self.algorithm.test(evaluationData.GetFullAntiTestSet())
            topNPredicted = RecommenderMetrics.GetTopN(allPredictions, n)
            if (verbose):
                print("Analyzing coverage, diversity, and novelty...")
            metrics["Diversity"] = RecommenderMetrics.Diversity(topNPredicted, evaluationData.GetSimilarities())
            metrics["Novelty"] = RecommenderMetrics.Novelty(topNPredicted, evaluationData.GetPopularityRankings())
        if (verbose):
            print("Analysis complete.")

        return metrics

    # Evaluación para los modelos ensamblados
    def EvaluateForStacking(self, evaluationData, doTopN, n=10, verbose=False):
        metrics = {}
        if (verbose):
            print("Evaluating RMSE and MAE")
        timeStart = time.time()
        self.algorithm.fit()
        print("Fit time:{:.2f}s".format(time.time() - timeStart))
        timeStart = time.time()
        predictions = self.algorithm.test()
        print("Test time:{:.2f}s".format(time.time() - timeStart))
```

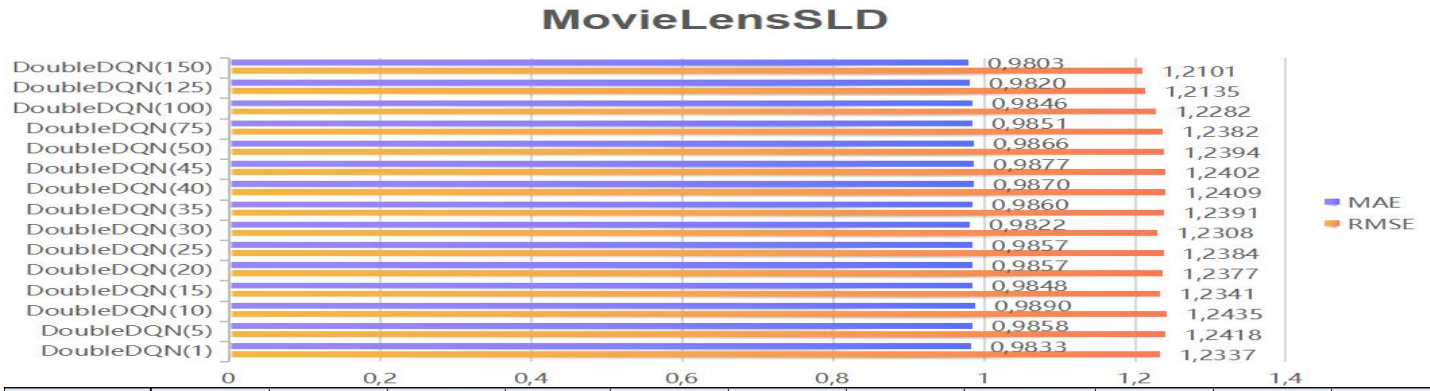
```
        metrics["RMSE"] = RecommenderMetrics.RMSE(predictions)
        metrics["MAE"] = RecommenderMetrics.MAE(predictions)

        if (doTopN):
            if (verbose):
                print("Evaluating top-N with leave-one-out...")
            topNPredictedPart = RecommenderMetrics.GetTopN(predictions, n)
            metrics["Precision"] = RecommenderMetrics.Precision(topNPredictedPart,
evaluationData.GetTestSet())
            metrics["Recall"]    =    RecommenderMetrics.Recall(topNPredictedPart,
evaluationData.GetTestSet())
            metrics["F1_score"] = RecommenderMetrics.F1_score(topNPredictedPart,
evaluationData.GetTestSet())
            #metrics["AUC"]     =     RecommenderMetrics.AUC(topNPredictedPart,
evaluationData.GetTestSet())
            metrics["AUC"]   =   RecommenderMetrics.paintROC(topNPredictedPart,
evaluationData.GetTestSet())
            metrics["Markedness"]                                                =
RecommenderMetrics.Markedness(topNPredictedPart, evaluationData.GetTestSet())
            metrics["Informedness"]                                              =
RecommenderMetrics.Informedness(topNPredictedPart, evaluationData.GetTestSet())


            leftOutPredictions                                                   =
self.algorithm.testWithSet(evaluationData.GetLOOCVTestSet())
            allPredictions                                                       =
self.algorithm.testWithSet(evaluationData.GetLOOCVAntiTestSet())
            topNPredicted = RecommenderMetrics.GetTopN(allPredictions, n)
            if (verbose):
                print("Computing hit-rate and rank metrics...")
            metrics["HR"]        =        RecommenderMetrics.HitRate(topNPredicted,
leftOutPredictions)
            metrics["cHR"] = RecommenderMetrics.CumulativeHitRate(topNPredicted,
leftOutPredictions)
            metrics["ARHR"]                                                      =
RecommenderMetrics.AverageReciprocalHitRank(topNPredicted, leftOutPredictions)

            if (verbose):
                print("Computing recommendations with full data set...")
            allPredictions                                                       =
self.algorithm.testWithSet(evaluationData.GetFullAntiTestSet())
            topNPredicted = RecommenderMetrics.GetTopN(allPredictions, n)
            if (verbose):
                print("Analyzing coverage, diversity, and novelty...")
```

```
            metrics["Diversity"] = RecommenderMetrics.Diversity(topNPredicted,
evaluationData.GetSimilarities())
            metrics["Novelty"] = RecommenderMetrics.Novelty(topNPredicted,
evaluationData.GetPopularityRankings())
        if (verbose):
            print("Analysis complete.")

        return metrics

    def GetName(self):
        return self.name

    def GetAlgorithm(self):
        return self.algorithm
```

Anexo III Resultados experimentales

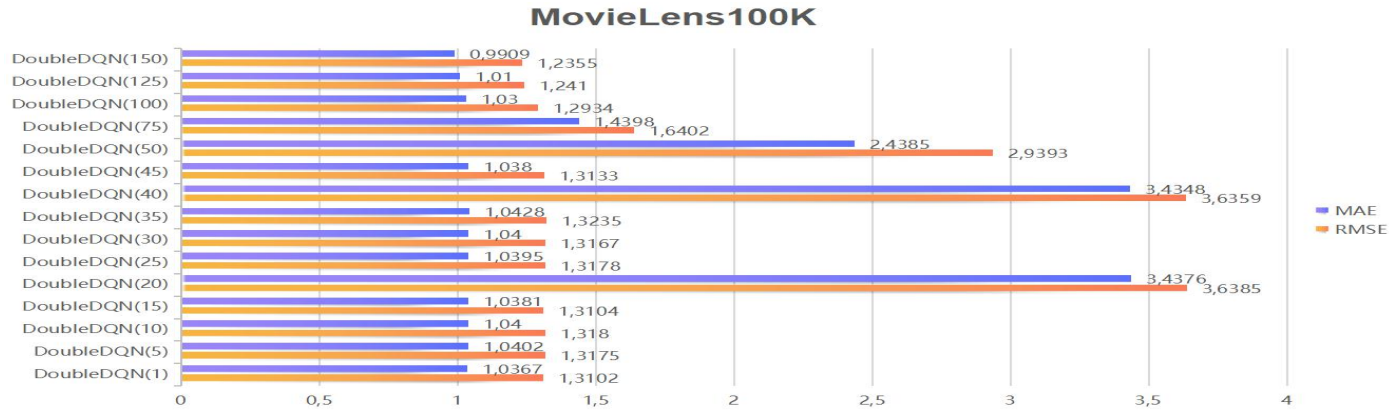
1. DoubleDQN

(1) MovieLensSLD



Episodes	1	5	10	15	20	25	30	35	40	45	50	75	100	125	150
RMSE	1,2337	1,2418	1,2435	1,2341	1,2377	1,2384	1,2308	1,2391	1,2409	1,2402	1,2394	1,2382	1,2282	1,2135	1,2101
MAE	0,9833	0,9858	0,9890	0,9848	0,9857	0,9857	0,9822	0,9860	0,9870	0,9877	0,9866	0,9851	0,9846	0,9820	0,9803
Precisión	63,31%	61,79%	54,89%	54,20%	55,88%	60,21%	64,99%	63,84%	69,34%	63,53%	65,58%	61,82%	65,19%	66,32%	67,00%
Recall	15,90%	15,40%	13,10%	10,23%	9,30%	16,82%	17,8%	16,84%	16,93%	15,93%	16,90%	16,93%	17,75%	18,06%	17,81%
F1_score	0,2542	0,2466	0,2115	0,1721	0,1595	0,2629	0,2795	0,2665	0,2721	0,2547	0,2687	0,2658	0,2790	0,2839	0,2814
AUC	0,5050	0,5046	0,5031	0,5249	0,5268	0,5283	0,5810	0,5795	0,5572	0,5488	0,5499	0,5477	0,5828	0,5853	0,5928
Markedness	0,2437	0,2286	0,1603	0,2813	0,2797	0,2970	0,2885	0,2888	0,2765	0,2895	0,2849	0,2934	0,3035	0,2919	0,2823
Informedness	0,6271	0,6118	0,5420	0,6249	0,6480	0,5413	0,5950	0,5831	0,6725	0,5685	0,6069	0,5352	0,4706	0,5624	0,6436
HR	0,0049	0,0033	0,0033	0,0016	0,0066	0,0066	0,0049	0,0066	0,0066	0,0016	0,0098	0	0,0049	0,0066	0,0066
cHR	0,0056	0,0056	0	0,0028	0,0028	0,0056	0,0056	0,0055	0,0028	0,0028	0,0169	0	0,0056	0,0056	0,0056
ARHR	0,0005	0,002	0,0009	0,0008	0,0011	0,0021	0,001	0,0042	0,0042	0,0008	0,0048	0	0,0035	0,0037	0,0014
Diversidad	0,2266	0,2218	0,2276	0,2267	0,2236	0,2174	0,2124	0,2245	0,2133	0,216	0,2265	0,2225	0,219	0,2314	0,2153
Novedad	1535,3508	1517,008	1549,6995	1559,019	1561,1644	1512,6098	1524,6525	1563,8785	1516,7256	1540,2775	1568,9231	1567,832	1548,4616	1584,6244	1541,1133
Fit time(s)	49,83	248,6	534,32	946,22	1201,52	1634,91	1818,46	2105,15	3071,84	3833,38	4387,38	6672,46	8443,77	17180,16	12876,48
Test time(s)	6,35	4,21	3,96	4,04	4	4,83	4,68	4,27	5,58	7,34	6,62	7,01	5,72	8,96	5,75

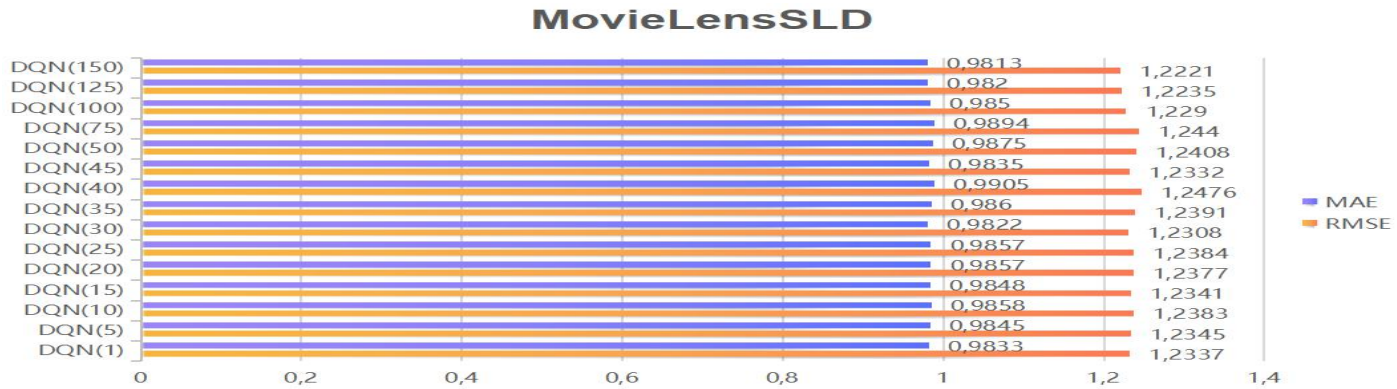
(2) MovieLens100K



Episodes	1	5	10	15	20	25	30	35	40	45	50	75	100	125	150
RMSE	1,3102	1,3175	1,318	1,3104	3,6385	1,3178	1,3167	1,3235	3,6359	1,3133	2,9393	1,6402	1,2934	1,241	1,2355
MAE	1,0367	1,0402	1,04	1,0381	3,4376	1,0395	1,0400	1,0428	3,4348	1,038	2,4385	1,4398	1,03	1,01	0,9909
Precisión	54,66%	60,08%	53,10%	58,36%	66,03%	64,00%	75,41%	74,59%	72,14%	65,34%	73,29%	67,44%	75,39%	72,94%	78,59%
Recall	12,40%	11,31%	11,06%	12,21%	10,50%	11,36%	15,51%	15,34%	10,48%	11,34%	12,51%	12,87%	15,50%	15,59%	16,16%
F1_score	0,2021	0,1904	0,1831	0,2019	0,1811	0,1929	0,2572	0,2544	0,1831	0,1933	0,2137	0,2161	0,2571	0,2569	0,2681
AUC	0,5018	0,5030	0,5014	0,5026	0,5250	0,5197	0,5746	0,5528	0,5374	0,5626	0,5300	0,5472	0,5781	0,5819	0,5842
Markedness	0,0952	0,1489	0,0797	0,1046	0,1636	0,0876	0,1150	0,1799	0,0963	0,1264	0,1977	0,1058	0,1389	0,2173	0,1163
Informedness	0,5388	0,5936	0,5232	0,5922	0,6524	0,5750	0,6508	0,7170	0,6320	0,7153	0,7881	0,6946	0,7862	0,8661	0,7634
HR	0,0127	0,0042	0,0095	0,0140	0,0046	0,0104	0,0153	0,0051	0,0115	0,0169	0,0056	0,0126	0,0185	0,0061	0,0139
cHR	0,0093	0,0019	0,0074	0,0102	0,0021	0,0081	0,0112	0,0023	0,0089	0,0123	0,0025	0,0098	0,0136	0,0028	0,0108
ARHR	0,0066	0,0017	0,0029	0,0073	0,0019	0,0032	0,0080	0,0021	0,0035	0,0088	0,0023	0,0038	0,0096	0,0025	0,0042
Diversidad	0,1963	0,1982	0,1956	0,1922	0,1862	0,1955	0,192	0,198	0,1952	0,1933	0,1932	0,1942	0,1937	0,1975	0,1941
Novedad	671,0563	670,1847	664,4925	675,06	673,3	680,3097	677,1022	682,2451	675,5386	675,2438	668,7127	672,1419	676,2818	680,4666	684,6232
Fit time(s)	46,16	258,51	618,52	891,79	1544,49	1426,77	2377,07	2929,37	3227,43	3600,31	4205,73	6310,85	8161,29	8743,03	9647,39
Test time(s)	3,8	3,91	4,27	4,56	4,67	3,77	5,27	5,47	5,31	5,55	5,22	5,18	5,52	5,48	5,84

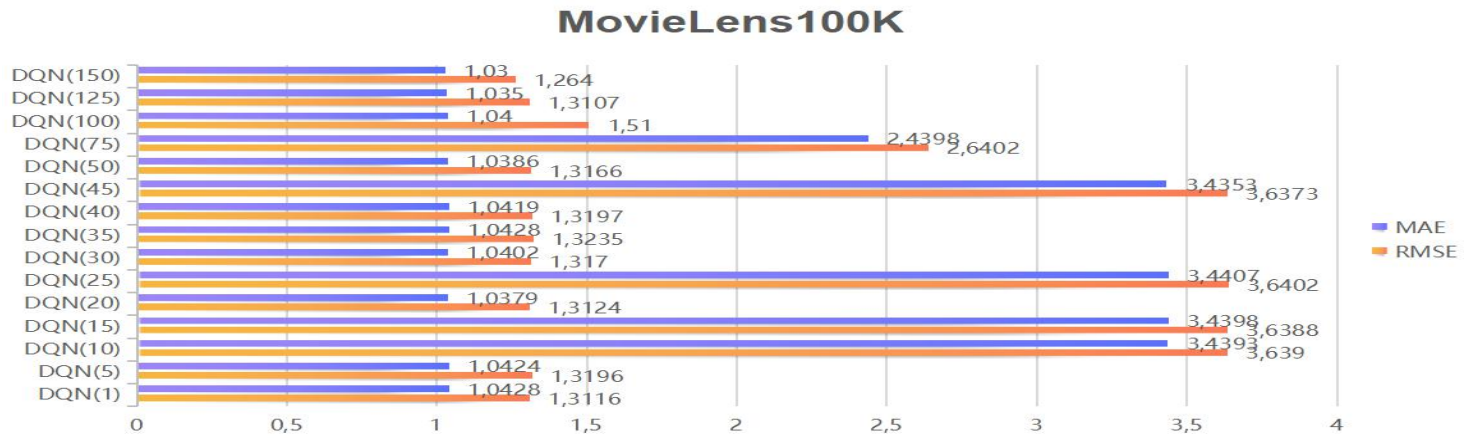
2. DQN

(1) MovieLensSLD



Episodes	1	5	10	15	20	25	30	35	40	45	50	75	100	125	150
RMSE	1,2337	1,2345	1,2383	1,2341	1,2377	1,2384	1,2308	1,2391	1,2476	1,2332	1,2408	1,244	1,229	1,2235	1,2221
MAE	0,9833	0,9845	0,9858	0,9848	0,9857	0,9857	0,9822	0,986	0,9905	0,9835	0,9875	0,9894	0,985	0,982	0,9813
Precisión	63,31%	61,80%	55,02%	53,33%	55,01%	59,34%	64,12%	62,97%	68,47%	62,66%	64,71%	60,95%	64,77%	65,43%	65,77%
Recall	14,90%	14,50%	15,10%	15,77%	15,22%	15,08%	15,87%	15,83%	14,92%	15,82%	15,9%	15,82%	16,07%	16,80%	16,98%
F1_score	0,2412	0,2349	0,2370	0,2434	0,2384	0,2404	0,2544	0,2530	0,2450	0,2526	0,2553	0,2512	0,2575	0,2673	0,2699
AUC	0,5050	0,5047	0,5032	0,5239	0,5258	0,5273	0,5800	0,5785	0,5562	0,5478	0,5689	0,5467	0,5818	0,5843	0,5917
Markedness	0,2537	0,2287	0,1604	0,2814	0,2798	0,2971	0,2886	0,2889	0,2766	0,2896	0,2850	0,2935	0,3036	0,2920	0,2824
Informedness	0,6271	0,6119	0,5421	0,6250	0,6481	0,5414	0,5951	0,5832	0,6726	0,5686	0,6070	0,5353	0,4707	0,5625	0,6437
HR	0,0049	0,0034	0,0016	0,0016	0,0066	0,0066	0,0049	0,0066	0,0082	0,0033	0,0016	0,0131	0,0049	0,0066	0,0066
cHR	0,0056	0,0057	0,0028	0,0028	0,0028	0,0056	0,0056	0,0055	0,0112	0,0056	0,0028	0,0169	0,0056	0,0056	0,0056
ARHR	0,0035	0,0021	0,0016	0,0008	0,0011	0,0021	0,001	0,0042	0,0036	0,0018	0,0002	0,0077	0,0035	0,0037	0,0014
Diversidad	0,223	0,2183	0,2171	0,2267	0,2236	0,2174	0,2124	0,2245	0,2144	0,2204	0,222	0,2168	0,219	0,2314	0,2153
Novedad	1515,35	1516,008	1550,6995	1559,019	1561,1644	1512,6098	1524,6525	1563,8785	1516,7256	1540,2775	1568,9231	1567,832	1548,4616	1584,6244	1541,1133
Fit time(s)	48,28	225,63	536,75	954	1048,43	1318,86	1706,12	1842,25	2157,17	2503,05	2679,34	3960,09	6626,18	6708,8	11958,85
Test time(s)	4,44	3,81	4,16	4,75	4,09	4,03	4,22	3,89	4,11	4,2	4,13	3,93	4,24	4,45	6,34

(2) MovieLens100K



Episodes	1	5	10	15	20	25	30	35	40	45	50	75	100	125	150
RMSE	1,3116	1,3196	3,639	3,6388	1,3124	3,6402	1,317	1,3235	1,3197	3,6373	1,3166	2,6402	1,51	1,3107	1,264
MAE	1,0428	1,0424	3,4393	3,4398	1,0379	3,4407	1,0402	1,0428	1,0419	3,4353	1,0386	2,4398	1,04	1,035	1,03
Precisión	54,20%	60,08%	52,22%	52,51%	61,81%	60,91%	73,89%	71,23%	70,11%	72,25%	68,16%	72,22%	71,23%	70,99%	76,11%
Recall	10,14%	9,91%	12,03%	12,02%	11,90%	10,91%	10,79%	10,81%	10,82%	10,80%	10,83%	10,80%	10,81%	10,81%	10,77%
F1_score	0,1708	0,1701	0,1955	0,1957	0,1996	0,1851	0,1882	0,1877	0,1874	0,1879	0,1870	0,1879	0,1877	0,1876	0,1887
AUC	0,5017	0,5030	0,5023	0,5023	0,5184	0,5183	0,5501	0,5397	0,5395	0,5295	0,5393	0,5199	0,5671	0,5681	0,5740
Markedness	0,0907	0,1489	0,0997	0,1278	0,1096	0,1163	0,1527	0,1278	0,1204	0,1278	0,1527	0,1405	0,1323	0,1405	0,1678
Informedness	0,5341	0,5936	0,5870	0,6524	0,6452	0,7170	0,8390	0,7634	0,6946	0,7390	0,7634	0,7122	0,7390	0,7828	0,8122
HR	0,0074	0,0042	0,0081	0,0046	0,0089	0,0051	0,0098	0,0056	0,0108	0,0061	0,0119	0,0067	0,0130	0,0074	0,0143
cHR	0,0037	0,0019	0,0041	0,0021	0,0045	0,0023	0,0049	0,0025	0,0054	0,0028	0,0059	0,0030	0,0065	0,0033	0,0072
ARHR	0,0044	0,0017	0,0048	0,0019	0,0053	0,0021	0,0058	0,0023	0,0064	0,0025	0,0071	0,0027	0,0078	0,0030	0,0085
Diversidad	0,1927	0,1982	0,1877	0,1922	0,1862	0,1955	0,192	0,198	0,1952	0,1933	0,1932	0,1942	0,1937	0,1975	0,1941
Novedad	660,4	670,1847	671,2147	667,0463	663,3543	665,3097	671,1022	672,2451	675,5386	673,2438	668,7127	672,1419	675,2818	680,4666	684,6232
Fit time(s)	36,42	234,32	496,83	795,4	1043,38	1304,97	1550,81	2508,06	2837,63	3290,48	3495,37	5880,64	6646,83	8831,94	10277,86
Test time(s)	3,3	3,7	3,68	3,74	3,7	3,8	3,85	5,11	5,25	5,56	4,98	4,92	4,28	4,68	4,88

3. Algoritmos clásicos de RS

(1) MovieLensSLD

Algoritmo	RMSE	MAE	Precisión	Recall	F1_score	AUC	Markedness	Informedness	HR	cHR	ARHR	Diversidad	Novedad	Fit time(s)	Test time(s)
MostPopular	16,3967	16,3632	66,43%	8,24%	0,1466	0,5267	0,2469	0,5051	0,0197	0,0281	0,0110	0,0489	502,4561	0,57	0,25
RecentPopular	15,7997	15,7649	67,89%	8,46%	0,1505	0,5284	0,2469	0,5051	0,0197	0,0281	0,0110	0,0489	502,4561	0,51	0,24
Random	1,4185	1,135	54,51%	14,69%	0,2314	0,5021	0,2588	0,5675	0,0148	0,014	0,0068	0,0482	814,7995	0,1	0,1
UserCF	0,9787	0,7547	83,18%	22,94%	0,3596	0,5947	0,4092	0,7896	0	0	0	0,8882	6126,9782	0,47	1,3
ItemCF	0,9788	0,761	85,07%	14,60%	0,2493	0,5633	0,4417	0,1058	0	0	0	0,7204	6933,13	14,58	7,67
ItemCFNorm	0,919	0,7088	90,07%	13,86%	0,2402	0,5634	0,4903	0,8885	0	0	0	0,7184	7142,8865	45,26	17,84
TItemCF	0,8688	0,6619	88,31%	20,04%	0,3267	0,5891	0,4613	0,8595	0	0	0	0,7514	7041,8294	53,76	14,38
ContentKNN	0,9055	0,6983	88,34%	17,93%	0,2981	0,5800	0,4661	0,8631	0	0	0	0,6239	4892,5833	293,64	10,42
SVD	0,878	0,6739	88,14%	20,50%	0,3326	0,5910	0,4588	0,8566	0,0148	0,0225	0,0075	0,0305	519,7967	3,26	0,12
SVD++	0,8698	0,6658	88,27%	21,22%	0,3421	0,5941	0,4583	0,857	0,0246	0,0421	0,0142	0,0740	896,987	472,05	9,99
TimeSVD++	1,0475	0,8282	88,21%	20,86%	0,3374	0,5943	0,4586	0,8568	0,0197	0,0293	0,0109	0,0523	708,3919	409,38	0,67
DoubleDQN*	1,2308	0,9822	64,99%	17,8%	0,2795	0,5810	0,2885	0,595	0,0049	0,0056	0,001	0,2124	1524,6525	1818,46	4,68
DQN*	1,2408	0,9875	64,71%	15,9%	0,2553	0,5689	0,2850	0,6070	0,0016	0,0028	0,0002	0,2220	1568,9231	2679,34	4,13

*Se ha sacado el mejor resultado según el número de episodio, considerando todas las medidas de evaluación.

(2) MovieLens100K

Algoritmo	RMSE	MAE	Precisión	Recall	F1_score	AUC	Markedness	Informedness	HR	cHR	ARHR	Diversidad	Novedad	Fit time(s)	Test time(s)
MostPopular	22,7116	22,6837	58,59%	9,83%	0,1683	0,5191	0,1104	0,2909	0,0244	0,0241	0,0066	0,0631	320,0398	0,54	0,03
RecentPopular	22,7119	22,684	60,25%	10,21%	0,1748	0,5219	0,1104	0,2909	0,0244	0,0241	0,0066	0,0631	320,0398	0,62	0,04
Random	1,5157	1,2214	57,43%	9,72%	0,1662	0,5175	0,1139	0,3858	0,0329	0,037	0,0078	0,0732	355,3353	0,14	0,27
UserCF	1,0212	0,8074	78,46%	26,89%	0,4005	0,5911	0,277	0,6903	0	0	0	0,9934	1541,8743	2	4,52
ItemCF	1,0329	0,8163	82,75%	16,33%	0,2727	0,5641	0,3444	0,7909	0,0042	0,0037	0,0006	0,5472	1282,7749	2,49	4,84
ItemCFNorm	1,0143	0,8081	85,62%	12,27%	0,2146	0,5512	0,3809	0,8354	0	0	0	0,6970	1573,3739	9,45	5,09
TItemCF	0,961	0,7572	84,74%	20,03%	0,324	0,5803	0,3523	0,805	0	0	0	0,7700	1509,5451	19,19	5,02
ContentKNN	1,0121	0,8038	80,75%	19,30%	0,3116	0,5721	0,3186	0,7557	0,0032	0,0056	0,0011	0,2558	808,4103	8,69	4,21
SVD	0,9437	0,742	84,05%	24,51%	0,3795	0,5952	0,3317	0,7805	0,0445	0,0722	0,0161	0,0282	213,1181	3,89	0,27
SVD++	0,9216	0,7233	85,96%	25,42%	0,3924	0,6021	0,3447	0,8039	0,0445	0,0704	0,0142	0,0302	230,4864	148,72	3,85
TimeSVD++	1,1261	0,9433	85,01%	25,07%	0,3872	0,5950	0,3335	0,7922	0,0445	0,0720	0,0152	0,0292	221,9800	158,03	0,22
DoubleDQN*	1,3167	1,04	75,41%	15,51%	0,2572	0,5746	0,1150	0,6508	0,0153	0,0112	0,0080	0,1920	677,1022	2377,07	5,27
DQN*	1,3170	1,0402	73,89%	10,79%	0,1882	0,5501	0,1527	0,8390	0,0098	0,0049	0,0058	0,1920	671,1022	1550,81	3,85

*Se ha sacado el mejor resultado según el número de episodio, considerando todas las medidas de evaluación.

4. Modelos ensamblados

(1) MovieLensSLD

Algoritmo	RMSE	MAE	Precisión	Recall	F1_score	AUC(media)	Markedness	Informedness	HR	cHR	ARHR	Diversidad	Novedad	Fit time(s)	Test time(s)
MostPopular&RecentPopular	16,0851	15,7606	67,89%	8,46%	0,1505	0,5094	0,2861	0,6514	0,3623	0,3652	0,3574	0,0489	502,4561	606,89	0,05
MostPopular&Random	1,8242	1,4979	67,81%	8,26%	0,1472	0,5290	0,2854	0,6513	0,0197	0,0281	0,011	0,0489	502,4561	744,63	0,1
MostPopular&UserCF	1,8303	1,5033	65,41%	8,19%	0,1456	0,6788	0,2626	0,6247	0,0197	0,0281	0,011	0,0489	502,4561	729,49	2,2
MostPopular&ItemCF	1,8218	1,4913	65,81%	8,02%	0,1429	0,6784	0,2664	0,6298	0,0197	0,0281	0,011	0,0489	502,4561	742,51	8,2
MostPopular&ItemCFNorm	16,3782	16,3448	66,79%	8,59%	0,1523	0,6816	0,2756	0,6386	0	0	0	0,7101	7294,9711	811,1	19,47
MostPopular&TItemCF	16,3884	16,3549	62,98%	7,97%	0,1414	0,6760	0,2396	0,5982	0	0	0	0,7546	7221,7352	730,11	21,38
MostPopular&ContentKNN	1,8337	1,5062	65,13%	7,90%	0,1410	0,6775	0,2600	0,6226	0,0197	0,0281	0,011	0,0489	502,4561	961,77	7,23
MostPopular&SVD	1,8345	1,5085	65,06%	8,09%	0,1440	0,6782	0,2593	0,6211	0,0197	0,0281	0,011	0,0489	502,4561	907,2	0,19
MostPopular&SVD++	1,8197	1,4953	68,23%	8,35%	0,1487	0,6816	0,2894	0,6556	0,0197	0,0281	0,011	0,0489	502,4561	1895,42	15,56
MostPopular&TimeSVD++	16,3798	16,3467	67,24%	8,43%	0,1498	0,6812	0,2799	0,6442	0,0197	0,0281	0,011	0,0489	502,4561	1539,07	0,55
MostPopular&DoubleDQN	3,5915	3,4142	68,08%	8,20%	0,1464	0,6591	0,0919	0,479	0,0066	0,0112	0,0026	0,2155	1492,9726	787,24	4,18
MostPopular&DQN	3,5963	3,4186	64,29%	5,70%	0,1047	0,6584	0,2536	0,6419	0,0148	0,014	0,0052	0,2224	1511,8069	874,72	6,88
RecentPopular&MostPopular	16,805	16,3619	66,43%	8,24%	0,1466	0,5077	0,2723	0,6359	0,4967	0,4747	0,4967	0,0489	502,4561	682,46	0,06
RecentPopular&Random	1,825	1,4963	64,77%	7,93%	0,1413	0,5261	0,2566	0,6185	0,0197	0,0281	0,011	0,0489	502,4561	731,23	0,13
RecentPopular&UserCF	1,8365	1,5092	65,69%	8,12%	0,1446	0,6788	0,2653	0,628	0,0197	0,0281	0,011	0,0489	502,4561	839,16	1,62
RecentPopular&ItemCF	1,8307	1,5032	66,26%	8,42%	0,1495	0,6805	0,2706	0,6333	0,0197	0,0281	0,011	0,0489	502,4561	881,39	8,42
RecentPopular&ItemCFNorm	15,7844	15,7501	63,37%	8,39%	0,1481	0,6522	0,2433	0,6009	0	0	0	0,7052	7273,8643	875	22,27
RecentPopular&TItemCF	15,7902	15,7559	63,36%	7,87%	0,1401	0,6502	0,2432	0,6029	0	0	0	0,7493	7221,8702	808,42	14,79
RecentPopular&ContentKNN	1,8262	1,4979	63,31%	7,99%	0,1419	0,6763	0,2427	0,6019	0,0197	0,0281	0,011	0,0489	502,4561	989,67	8,19
RecentPopular&SVD	1,8262	1,5012	64,09%	7,83%	0,1396	0,6765	0,2502	0,6113	0,0197	0,0281	0,011	0,0489	502,4561	737,46	0,15
RecentPopular&SVD++	1,8273	1,5044	66,11%	8,12%	0,1446	0,6791	0,2693	0,6327	0,0197	0,0281	0,011	0,0489	502,4561	1504,99	14,05
RecentPopular&TimeSVD++	15,798	15,7634	62,75%	7,96%	0,1413	0,6500	0,2374	0,5957	0,0197	0,0281	0,011	0,0489	502,4561	1647,93	0,57
RecentPopular&DoubleDQN	3,6002	3,4232	66,00%	7,40%	0,1331	0,6598	0,2707	0,6589	0,0098	0,0169	0,005	0,2155	1471,2548	901,53	3,96
RecentPopular&DQN	3,5918	3,4129	61,22%	6,10%	0,1109	0,6593	0,2231	0,611	0,0082	0,0084	0,0038	0,2009	1529,1256	932,02	4,93
Random&MostPopular	16,7869	16,3388	66,33%	8,34%	0,1481	0,5079	0,2713	0,6344	0,4967	0,4747	0,4967	0,0489	502,4561	1560,97	0,33

Random&RecentPopular	16,0607	15,7371	68,74%	8,38%	0,1495	0,5097	0,2942	0,6611	0,3623	0,3652	0,3574	0,0489	502,4561	1510,47	0,05
Random&UserCF	1,0911	0,8669	51,52%	0,87%	0,0170	0,6472	0,1266	0,51	0	0	0	0,8792	6244,5574	1501,95	1,4
Random&ItemCF	1,0641	0,8495	58,72%	0,65%	0,0129	0,6473	0,1981	0,5842	0,0066	0,0056	0,0026	0,6826	7404,627	1674,84	9,27
Random&ItemCFNorm	1,0398	0,8324	80,00%	0,02%	0,0004	0,6451	0,6108	1	0	0	0	0,6915	7429,3434	1557,22	22,16
Random&TItemCF	1,0587	0,846	67,50%	0,27%	0,0055	0,6463	0,2857	0,6742	0	0	0	0,7172	7013,4154	1558,21	15,31
Random&ContentKNN	1,0622	0,851	59,26%	0,16%	0,0033	0,6456	0,2034	0,5919	0	0	0	0,646	5012,4486	1720,32	8,05
Random&SVD	1,1063	0,8842	61,22%	1,63%	0,0318	0,6512	0,2223	0,6055	0,0082	0,0056	0,0027	0,3596	2373,0658	1424,64	0,16
Random&SVD++	1,1074	0,8848	56,32%	1,09%	0,0213	0,6485	0,1743	0,5578	0,0033	0,0056	0,0005	0,294	2109,843	1833,09	14,88
Random&TimeSVD++	1,0423	0,8326	58,77%	1,36%	0,0266	0,6489	0,1983	0,5817	0,0058	0,0026	0,0016	0,3268	2241,4544	2574,4	0,5
Random&DoubleDQN	1,864	1,6369	65,66%	5,31%	0,0983	0,6462	0,2108	0,5987	0,0049	0,0056	0,0009	0,2207	1510,5018	1709,89	3,86
Random&DQN	1,2448	0,9938	63,16%	3,37%	0,0640	0,6464	0,2423	0,6302	0,0066	0,0056	0,002	0,2183	1521,5667	1518,84	5,2
UserCF&MostPopular	16,776	16,3326	67,09%	8,32%	0,1480	0,5683	0,2785	0,6429	0,4967	0,4747	0,4967	0,0489	502,4561	733,32	0,07
UserCF&RecentPopular	16,0707	15,747	66,05%	8,31%	0,1477	0,5676	0,2686	0,6313	0,3623	0,3652	0,3574	0,0489	502,4561	744,75	0,06
UserCF&Random	1,1928	0,9454	62,44%	4,20%	0,0787	0,6101	0,2349	0,6079	0,0016	0,0028	0,0016	0,2792	1782,7339	732,98	0,13
UserCF&ItemCF	1,0299	0,8147	91,14%	0,73%	0,0154	0,6494	0,5213	0,9109	0	0	0	0,7624	7425,7836	748,66	8,02
UserCF&ItemCFNorm	1,0064	0,7898	84,00%	0,64%	0,0127	0,6486	0,4501	0,9382	0	0	0	0,7572	7421,4318	772,17	22,12
UserCF&TItemCF	0,9687	0,7507	88,08%	2,65%	0,0514	0,6606	0,4883	0,8784	0	0	0	0,7315	6868,0673	794,63	23,44
UserCF&ContentKNN	1,0064	0,7887	82,61%	1,34%	0,0264	0,6525	0,4355	0,8243	0	0	0	0,6436	5333,276	1082,69	9,82
UserCF&SVD	0,9314	0,7227	85,48%	5,66%	0,1062	0,6773	0,4589	0,8482	0,0541	0,0815	0,0208	0,0242	192,4018	796,93	0,16
UserCF&SVD++	0,9181	0,7112	84,40%	5,63%	0,1056	0,6767	0,4483	0,8369	0,0377	0,059	0,0176	0,036	660,9366	1684,25	13
UserCF&TimeSVD++	1,0482	0,8301	84,94%	5,65%	0,1059	0,6944	0,4082	0,8426	0,0459	0,0673	0,0192	0,0301	426,6692	1872,7	0,71
UserCF&DoubleDQN	1,237	0,9819	68,62%	0,32%	0,0064	0,6591	0,2070	0,5948	0,0049	0,0084	0,0038	0,2202	1513,2889	1030,3	5,03
UserCF&DQN	1,2435	0,9886	64,71%	0,34%	0,0068	0,6464	0,2578	0,6459	0,0066	0,0056	0,0018	0,2249	1516,7762	1060,1	3,44
ItemCF&MostPopular	16,7754	16,3291	64,98%	7,94%	0,1416	0,5655	0,2586	0,6208	0,4967	0,4747	0,4967	0,0489	502,4561	912,57	0,1
ItemCF&RecentPopular	16,0741	15,7478	68,85%	8,96%	0,1585	0,5607	0,2950	0,6605	0,3623	0,3652	0,3574	0,0489	502,4561	785,62	0,05
ItemCF&Random	1,1637	0,9287	62,57%	3,48%	0,0659	0,6076	0,2362	0,6121	0,0016	0,0028	0,0016	0,6182	3737,8566	751,88	0,14
ItemCF&UserCF	0,9858	0,7735	90,83%	2,18%	0,0425	0,6582	0,5163	0,9069	0,0033	0,0056	0,0005	0,9175	5092,5502	806,45	1,74
ItemCF&ItemCFNorm	0,9606	0,7508	87,04%	3,07%	0,0593	0,6628	0,4775	0,8674	0	0	0	0,6486	7484,3938	1053,15	26,3
ItemCF&TItemCF	0,9515	0,7434	89,05%	3,06%	0,0593	0,6632	0,4974	0,888	0	0	0	0,6227	6731,0823	1080,19	22,66

ItemCF&ContentKNN	0,9515	0,7423	86,89%	3,03%	0,0585	0,6625	0,4761	0,8659	0,0016	0	0,0003	0,6357	4980,0257	1465,47	9,66
ItemCF&SVD	0,9562	0,7434	81,31%	2,61%	0,0505	0,6593	0,4213	0,8091	0,0016	0	0,0005	0,862	5238,9858	884,85	0,13
ItemCF&SVD++	0,9348	0,7303	89,14%	2,77%	0,0538	0,6615	0,4987	0,8891	0,0016	0,0028	0,0002	0,8081	4810,2498	1808,7	9,69
ItemCF&TimeSVD++	1,0435	0,8318	85,23%	2,69%	0,0522	0,6624	0,4718	0,8491	0,0016	0,0014	0,00035	0,83505	5024,6178	1793,33	0,5
ItemCF&DoubleDQN	1,244	0,9864	60,91%	9,29%	0,1612	0,6716	0,1201	0,5073	0,0098	0,0084	0,0027	0,22	1529,8389	1147,55	7,76
ItemCF&DQN	1,2413	0,9808	58,00%	8,30%	0,1452	0,6718	0,1909	0,5786	0,0016	0,0028	0,0008	0,2114	1475,3328	1071,79	4,07
ItemCFNorm&MostPopular	16,7719	16,3213	67,10%	8,53%	0,1514	0,5691	0,2785	0,6423	0,4967	0,4747	0,4967	0,0489	502,4561	1079,21	0,06
ItemCFNorm&RecentPopular	16,0939	15,7712	67,82%	8,22%	0,1466	0,5685	0,2855	0,6515	0,3623	0,3652	0,3574	0,0489	502,4561	1041,25	0,05
ItemCFNorm&Random	1,14	0,903	63,86%	3,35%	0,0637	0,6076	0,2488	0,6262	0	0	0	0,7365	4859,5007	1071,53	0,12
ItemCFNorm&UserCF	0,9755	0,763	90,91%	0,61%	0,0121	0,7260	0,5191	0,9087	0	0	0	0,8827	6842,9775	1097,16	1,57
ItemCFNorm&ItemCF	0,9403	0,732	84,30%	2,89%	0,0558	0,7130	0,4506	0,8395	0,0016	0,0028	0,0005	0,6521	7179,4951	1186,35	8,75
ItemCFNorm&TItemCF	0,9198	0,7115	90,14%	3,17%	0,0612	0,7027	0,5081	0,8991	0	0	0	0,597	6666,7251	1322,72	26,96
ItemCFNorm&ContentKNN	0,9313	0,7225	88,92%	23,18%	0,0614	0,8041	0,4960	0,8866	0	0	0	0,6447	5084,891	1292,92	8,22
ItemCFNorm&SVD	0,9333	0,7238	89,39%	3,17%	0,0612	0,7025	0,5006	0,8913	0	0	0	0,5836	3510,7256	920,58	0,14
ItemCFNorm&SVD++	0,9256	0,7145	90,23%	3,11%	0,0602	0,6894	0,5090	0,9	0,0016	0,0028	0,0002	0,7545	4154,7766	2152,88	11,95
ItemCFNorm&TimeSVD++	1,0495	0,8339	89,81%	3,14%	0,0607	0,7024	0,4846	0,8957	0,0006	0,0014	0,0001	0,66905	3832,7511	2293,96	0,6
ItemCFNorm&DoubleDQN	1,2423	0,9859	69,32%	10,36%	0,1803	0,7354	0,2041	0,5917	0,0049	0,0084	0,0027	0,2117	1496,1259	1592,87	5,24
ItemCFNorm&DQN	1,2444	0,9909	68,25%	9,44%	0,1659	0,7114	0,2932	0,6812	0,0049	0,0028	0,0036	0,2283	1542,7082	1138,6	3,66
TItemCF&MostPopular	16,7891	16,3449	67,27%	8,40%	0,1493	0,5686	0,2802	0,6446	0,4967	0,4747	0,4967	0,0489	502,4561	1923,65	0,11
TItemCF&RecentPopular	16,062	15,7389	67,73%	8,31%	0,1480	0,5687	0,2846	0,6501	0,3623	0,3652	0,3574	0,0489	502,4561	1084,42	0,05
TItemCF&Random	1,185	0,9407	65,78%	4,45%	0,0833	0,6124	0,2673	0,6425	0,0049	0,0028	0,0034	0,2841	1831,1823	1159,57	0,54
TItemCF&UserCF	0,944	0,7285	89,17%	4,44%	0,0845	0,6713	0,4968	0,8881	0	0	0	0,9815	7376,5962	1152,98	2,28
TItemCF&ItemCF	0,9395	0,7276	88,50%	3,72%	0,0714	0,6669	0,4911	0,8817	0,0016	0,0028	0,0016	0,6803	7190,124	1229,65	9
TItemCF&ItemCFNorm	0,9273	0,7153	90,43%	3,54%	0,0682	0,6663	0,5104	0,9018	0	0	0	0,6652	7315,971	1216,84	14,61
TItemCF&ContentKNN	0,9016	0,6958	90,18%	3,89%	0,0745	0,6682	0,5075	0,899	0	0	0	0,6514	5305,9779	1378,67	9,55
TItemCF&SVD	0,8882	0,6854	90,83%	4,27%	0,0816	0,6707	0,5134	0,9054	0,0098	0,0169	0,0036	0,0429	914,7285	1089,32	0,47
TItemCF&SVD++	0,8895	0,6805	91,32%	4,33%	0,0827	0,6712	0,5181	0,9104	0,0033	0,0056	0,0004	0,0568	1185,9842	1758,07	11,8
TItemCF&TimeSVD++	1,0438	0,83	91,08%	4,30%	0,0821	0,7372	0,5106	0,9079	0,00635	0,01125	0,0016	0,04985	1050,35635	1840,77	0,53

TItemCF&DoubleDQN	1,2429	0,9863	76,67%	11,43%	0,1989	0,6985	0,2773	0,6653	0,0049	0,0056	0,0027	0,2197	1521,7705	1556,21	3,6
TItemCF&DQN	1,2424	0,9881	76,13%	10,42%	0,1833	0,6855	0,272	0,6599	0,0033	0,0028	0,0016	0,2157	1504,4149	1134,79	3,75
ContentKNN&MostPopular	16,7944	16,3457	68,00%	8,44%	0,1502	0,5805	0,2871	0,6527	0,4967	0,4747	0,4967	0,0489	502,4561	2174,34	0,09
ContentKNN&RecentPopular	16,088	15,7629	68,50%	8,48%	0,1509	0,5811	0,2919	0,6581	0,3623	0,3652	0,3574	0,0489	502,4561	2052,89	0,05
ContentKNN&Random	1,1674	0,9304	62,99%	3,56%	0,0673	0,6080	0,2403	0,6163	0,0033	0,0056	0,0018	0,4258	2471,6352	2014,08	0,28
ContentKNN&UserCF	0,9731	0,7618	89,27%	10,55%	0,1886	0,6574	0,5009	0,8911	0	0	0	0,8976	5237,3949	1972,2	1,21
ContentKNN&ItemCF	0,9289	0,7225	90,35%	9,18%	0,1667	0,6633	0,5103	0,9013	0	0	0	0,6431	7161,1651	2032,76	6,01
ContentKNN&ItemCFNorm	0,9321	0,7259	91,38%	14,96%	0,2571	0,6628	0,5207	0,9120	0	0	0	0,6283	7363,7827	2119,25	18,11
ContentKNN&TItemCF	0,9023	0,7038	91,11%	17,23%	0,2897	0,6646	0,5176	0,9090	0	0	0	0,6515	6778,1769	2160,32	19,36
ContentKNN&SVD	0,915	0,7106	89,97%	13,29%	0,2316	0,6646	0,5063	0,8973	0,0033	0,0056	0,0005	0,0222	1161,7791	2129,14	8,96
ContentKNN&SVD++	0,9053	0,7021	90,65%	13,45%	0,2342	0,6658	0,5127	0,9041	0,0016	0,0028	0,0002	0,0152	1252,3589	2920,35	10,61
ContentKNN&TimeSVD++	1,0361	0,82	90,31%	13,37%	0,2329	0,6837	0,5044	0,9007	0,00225	0,0042	0,00035	0,0187	1207,069	2717,22	0,43
ContentKNN&DoubleDQN	1,2465	0,9905	87,41%	15,20%	0,2590	0,6718	0,185	0,5726	0,0082	0,0056	0,0022	0,2164	1493,8018	2786,68	5,4
ContentKNN&DQN	1,24	0,9875	84,29%	14,27%	0,2441	0,6591	0,2536	0,6419	0,0033	0	0,0011	0,2235	1519,9757	2461,93	4,75
SVD&MostPopular	16,8022	16,3571	64,95%	8,30%	0,1472	0,5667	0,2582	0,6191	0,4967	0,4747	0,4967	0,0489	502,4561	637,49	0,05
SVD&RecentPopular	16,0449	15,7197	68,16%	8,42%	0,1499	0,5694	0,2887	0,6546	0,3623	0,3652	0,3574	0,0489	502,4561	632,06	0,05
SVD&Random	1,1969	0,9523	66,06%	3,99%	0,0752	0,6107	0,2702	0,6471	0,0066	0,0112	0,0026	0,2953	1878,0915	577,59	0,09
SVD&UserCF	0,9213	0,7141	84,92%	15,64%	0,2642	0,6770	0,4534	0,8423	0,0607	0,0927	0,0289	0,1686	774,1018	636,35	1,26
SVD&ItemCF	0,9339	0,7307	90,85%	22,96%	0,2268	0,7918	0,5154	0,9066	0	0	0	0,7571	7412,6379	646,37	5,94
SVD&ItemCFNorm	0,9311	0,7244	90,54%	12,67%	0,2223	0,6610	0,5127	0,9036	0	0	0	0,7558	7427,9514	672,48	16,41
SVD&TItemCF	0,895	0,6865	92,25%	13,75%	0,2393	0,6678	0,5281	0,9204	0	0	0	0,78	6788,1803	694,26	14,65
SVD&ContentKNN	0,903	0,7072	90,06%	12,88%	0,2254	0,6622	0,5077	0,8985	0,0016	0	0,0016	0,6541	5376,3015	922,45	8,62
SVD&SVD++	0,828	0,6523	90,51%	15,54%	0,2653	0,6782	0,5083	0,9010	0,0344	0,0506	0,0134	0,0208	221,8673	1294,86	8,11
SVD&TimeSVD++	1,0428	0,8266	89,56%	17,18%	0,2882	0,6648	0,5039	0,8926	0,031	0,017	0,009	0,33745	2799,0844	1374,7	0,58
SVD&DoubleDQN	1,2476	0,9907	64,29%	20,27%	0,3082	0,6695	0,2536	0,6419	0,0049	0,0056	0,0011	0,2226	1511,3093	1123,3	4,24
SVD&DQN	1,2357	0,9776	52,08%	17,25%	0,2592	0,6585	0,1318	0,5193	0,0049	0,0084	0,0022	0,216	1498,7393	1018,26	4,28
SVD++ &MostPopular	16,8105	16,3596	65,86%	8,29%	0,1472	0,5785	0,2669	0,6293	0,4967	0,4747	0,4967	0,0489	502,4561	2750,26	0,06
SVD++ &RecentPopular	16,0387	15,7117	66,35%	8,10%	0,1444	0,5784	0,2715	0,6354	0,3623	0,3652	0,3574	0,0489	502,4561	2987,25	0,05
SVD++ &Random	1,2067	0,9607	64,03%	4,29%	0,0804	0,6129	0,2503	0,6245	0,0016	0,0028	0,0003	0,2414	1606,1507	2798,13	0,11

SVD++ &UserCF	0,902	0,6973	86,36%	15,95%	0,2693	0,6792	0,4671	0,8571	0,0492	0,0758	0,024	0,2952	1564,8895	2707,58	1,29
SVD++ &ItemCF	0,9278	0,7284	89,16%	13,26%	0,2309	0,6644	0,4982	0,8890	0	0	0	0,7344	7422,9655	2716,23	5,68
SVD++ &ItemCFNorm	0,9143	0,7161	91,38%	12,64%	0,2221	0,6610	0,5211	0,9121	0	0	0	0,7356	7411,9173	2807,07	20,3
SVD++ &TItemCF	0,8788	0,677	89,10%	13,63%	0,2364	0,6665	0,4972	0,8881	0	0	0	0,7725	6809,9036	2834,49	18,35
SVD++ &ContentKNN	0,9076	0,7058	93,18%	13,50%	0,2358	0,6665	0,5376	0,9300	0,0033	0,0028	0,0033	0,6546	5369,1995	3233,86	7,95
SVD++ &SVD	0,846	0,6519	90,20%	14,85%	0,2550	0,6739	0,5063	0,8984	0,023	0,0365	0,009	0,0204	379,2602	2774,18	0,11
SVD++ &TimeSVD++	1,0458	0,8313	89,73%	13,03%	0,2276	0,6566	0,6603	0,0721	0,0132	0,0196	0,0062	0,3375	2874,22985	3805,01	0,47
SVD++ &DoubleDQN	1,2407	0,9837	70,39%	16,35%	0,2653	0,7045	0,2797	0,7027	0,0049	0,0084	0,0021	0,2218	1527,1467	3007,49	5,43
SVD++ &DQN	1,2424	0,9854	64,58%	13,32%	0,2208	0,6579	0,2566	0,6447	0,0049	0,0084	0,0021	0,2115	1473,3793	2701,71	2,61
TimeSVD++ &MostPopular	16,7413	16,2965	66,85%	8,27%	0,1472	0,5904	0,2763	0,6405	0,4967	0,4747	0,4967	0,0489	502,4561	3337,46	0,05
TimeSVD++ &RecentPopular	16,1074	15,7824	66,64%	8,35%	0,1484	0,5906	0,2742	0,6378	0,3623	0,3652	0,3574	0,0489	502,4561	3075,18	0,05
TimeSVD++ &Random	1,0408	0,8235	58,79%	3,55%	0,0670	0,6030	0,2405	0,5759	0,0059	0,0100	0,0023	0,262817	1671,501435	2961,07	0,1
TimeSVD++ &UserCF	1,0375	0,8234	88,95%	16,13%	0,2731	0,6996	0,4811	0,8828	0,0507	0,0781	0,0247	0,304056	1611,836185	2703,53	1,32
TimeSVD++ &ItemCF	1,0473	0,8315	71,16%	24,90%	0,3689	0,6886	0,3849	0,7063	0,0405	0,0625	0,0198	0,2432448	1289,468948	2616,85	5,92
TimeSVD++ &ItemCFNorm	1,0487	0,8303	75,61%	20,21%	0,3189	0,7236	0,4089	0,7504	0,0431	0,0664	0,0210	0,2584476	1370,060757	2747,5	17,16
TimeSVD++ &TItemCF	1,0421	0,8272	80,06%	15,52%	0,2599	0,7786	0,4330	0,7945	0,0456	0,0703	0,0222	0,2736504	1450,652567	3036,61	17,58
TimeSVD++ &ContentKNN	1,04374	0,81167	82,72%	25,70%	0,3922	0,6506	0,4474	0,8210	0,0471	0,0726	0,0230	0,28277208	1499,007652	3433,86	10,95
TimeSVD++ &SVD	0,9729	0,749685	92,51%	16,37%	0,2782	0,7275	0,5004	0,9181	0,0527	0,0812	0,0257	0,31621824	1676,309632	2974,18	0,43
TimeSVD++ &SVD++	1,20267	0,955995	86,28%	25,94%	0,3989	0,6786	0,4667	0,8563	0,0492	0,0757	0,0240	0,29493432	1563,481099	3965,01	0,47
TimeSVD++ &DoubleDQN	1,2483	0,9937	64,71%	23,34%	0,3431	0,6850	0,2578	0,6459	0,0033	0,0056	0,0006	0,2162	1501,4349	3017,69	3,97
TimeSVD++ &DQN	1,2466	0,9955	57,14%	20,33%	0,2999	0,6747	0,1823	0,5699	0,0033	0,0028	0,0008	0,2214	1518,8682	2749,59	2,66
DQN&MostPopular	16,7894	16,3444	66,64%	8,07%	0,1440	0,5895	0,2743	0,6388	0,4967	0,4747	0,4967	0,0489	502,4561	1843,42	0,29
DQN&RecentPopular	16,0987	15,7708	67,70%	8,71%	0,1543	0,5926	0,2842	0,6484	0,3623	0,3652	0,3574	0,0489	502,4561	1996,64	0,32
DQN&Random	1,2374	1,0267	50,00%	0,07%	0,0014	0,5951	0,1109	0,4995	0,0049	0,0056	0,001	0,7374	4927,8379	1885,79	0,1
DQN&UserCF	0,5002	0,97	83,33%	10,05%	0,1794	0,6453	0,4441	0,8333	0	0	0	0,7433	6518,9905	1930,62	1,77
DQN&ItemCF	1,1595	0,9432	65,65%	23,78%	0,3492	0,7571	0,2914	0,6010	0,0049	0,0057	0,0010	0,4765	7382,7032	1818,23	5,37

DQN&ItemCFNorm	1,18	0,9707	66,69%	20,30%	0,3113	0,7181	0,2343	0,6656	0,0054	0,0093	0,0010	0,421	7053,3633	1341,87	15,16
DQN&TItemCF	1,1765	0,9672	67,36%	18,30%	0,2878	0,7252	0,2366	0,6722	0,0055	0,0094	0,0010	0,421	7053,3633	1456,73	15,74
DQN&ContentKNN	1,1836	0,9736	93,80%	7,01%	0,1304	0,7099	0,5729	0,938	0	0	0	0,5534	3784,1111	1599,24	6,63
DQN&SVD	1,1802	0,9725	90,25%	10,01%	0,1802	0,6623	0,5512	0,9025	0	0	0	0,332842	2548,734186	1401,6	0,19
DQN&SVD++	1,184	0,9761	90,25%	10,01%	0,1802	0,6725	0,5512	0,9025	0	0	0	0,332842	2548,734186	2017,22	8,42
DQN&TimeSVD++	1,1795	0,9687	69,15%	10,82%	0,1872	0,7675	0,3069	0,6331	0,0052	0,0060	0,0011	0,2259936	2736,535652	1981,73	0,17
DQN&DoubleDQN	1,8462	1,6224	56,25%	7,28%	0,1289	0,6846	0,1734	0,5611	0,0049	0,0056	0,0013	0,2181	1491,098	1577,64	3,09
DoubleDQN&MostPopular	16,7721	16,3251	65,49%	8,02%	0,1429	0,5886	0,2634	0,6262	0,5967	0,4747	0,4967	0,0489	502,4561	2462,39	0,27
DoubleDQN&RecentPopular	16,1114	15,7812	68,91%	8,36%	0,1492	0,5920	0,2958	0,6631	0,3623	0,3652	0,3574	0,0489	502,4561	2190,45	0,36
DoubleDQN&Random	1,2413	1,0284	42,86%	10,09%	0,0018	0,5951	0,0395	0,4278	0	0	0	0,7439	4970,6702	2498,99	0,09
DoubleDQN&UserCF	1,1725	0,9605	99,00%	30,02%	0,4607	0,6451	0,6108	1	0	0	0	0,7433	6518,9905	2392,59	1,36
DoubleDQN&ItemCF	1,162	0,9584	68,24%	10,81%	0,1867	0,7870	0,3029	0,6248	0,0051	0,0059	0,0011	0,4765	7382,703	2160,53	6,37
DoubleDQN&ItemCFNorm	1,185	0,9765	70,20%	25,32%	0,3721	0,7559	0,2466	0,7006	0,0057	0,0098	0,0011	0,4893	7454,6032	2052,81	17,81
DoubleDQN&TItemCF	1,1753	0,9635	71,60%	28,32%	0,4059	0,7710	0,2516	0,7146	0,0058	0,0100	0,0011	0,4893	7454,6032	2246,59	19,29
DoubleDQN&ContentKNN	1,1812	0,9694	99,00%	35,01%	0,5173	0,7237	0,6108	1	0	0	0	0,3688	2824,0822	2724,58	10,26
DoubleDQN&SVD	1,1862	0,9785	95,00%	30,01%	0,4561	0,7419	0,5803	0,95	0	0	0	0,35036	2682,87809	2179,64	0,18
DoubleDQN&SVD++	1,1777	0,9676	72,79%	15,87%	0,2606	0,7437	0,3231	0,6664	0,0055	0,0063	0,0011	0,237888	2880,563844	5353,97	48,85
DoubleDQN&TimeSVD++	1,1786	0,9689	69,88%	10,83%	0,1876	0,7061	0,3101	0,6397	0,0053	0,0060	0,0011	0,22837248	2765,34129	3210,5	0,15
DoubleDQN&DQN	1,8614	1,6357	60,00%	9,27%	0,1606	0,6718	0,2108	0,5988	0,0049	0,0084	0,0009	0,2235	1522,478	2469,71	3,54
Es posible tener una pequeña discrepancia sobre la conclusión del mejor modelo entre la tasa de fallos y el AUC, porque se obtienen por el promedio de validación cruzada repetida.															

(2) MovieLens100K

Algoritmo	RMSE	MAE	Precisión	Recall	F1_score	AUC(media)	Markedness	Informedness	HR	cHR	ARHR	Diversidad	Novedad	Fit time(s)	Test time(s)
MostPopular&RecentPopular	22,7078	22,6794	58,59%	9,83%	0,1684	0,5101	0,1314	0,5256	0,5069	0,5111	0,5069	0,0631	320,0398	674,14	0,09
MostPopular&Random	1,8581	1,4779	57,43%	9,72%	0,1663	0,5085	0,1207	0,5121	0,0244	0,0241	0,0066	0,0631	320,0398	623,84	0,13
MostPopular&UserCF	1,8584	1,4734	56,35%	9,50%	0,1626	0,6656	0,1108	0,5005	0,0244	0,0241	0,0066	0,0631	320,0398	706,26	4,01
MostPopular&ItemCF	1,8475	1,4654	55,35%	9,40%	0,1607	0,6640	0,1017	0,4889	0,0244	0,0241	0,0066	0,0631	320,0398	680,87	5,12
MostPopular&ItemCFNorm	22,6977	22,6695	54,88%	8,06%	0,1406	0,6609	0,0973	0,4927	0	0	0	0,7541	1568,2543	766,41	6,23
MostPopular&TItemCF	22,6923	22,6646	54,62%	9,28%	0,1586	0,6625	0,095	0,4809	0	0	0	0,8143	1525,542	641,03	4,61
MostPopular&ContentKNN	1,8614	1,4794	55,87%	9,30%	0,1595	0,6646	0,1064	0,496	0,0244	0,0241	0,0066	0,0631	320,0398	825,95	6,01
MostPopular&SVD	1,8498	1,4684	56,59%	9,63%	0,1646	0,6663	0,113	0,5025	0,0244	0,0241	0,0066	0,0631	320,0398	701	0,12
MostPopular&SVD++	1,8635	1,4827	55,47%	9,30%	0,1593	0,6640	0,1028	0,4912	0,0244	0,0241	0,0066	0,0631	320,0398	1180,77	5,74
MostPopular&TimeSVD++	22,6962	22,6717	56,45%	9,64%	0,1647	0,6660	0,1117	0,5007	0,0244	0,0241	0,0066	0,0631	320,0398	1294,53	0,49
MostPopular&DoubleDQN	3,6538	3,4527	55,65%	9,39%	0,1607	0,6579	0,0053	0,4547	0,0074	0,0093	0,0041	0,1966	666,0881	804,29	4,09
MostPopular&DQN	3,6479	3,4485	54,84%	9,30%	0,1590	0,6456	0,097	0,5463	0,0117	0,013	0,0058	0,1975	660,9894	864,12	4,52
RecentPopular&MostPopular	22,7012	22,6725	60,25%	8,21%	0,1445	0,5129	0,1491	0,5468	0,5069	0,5111	0,5069	0,0631	320,0398	693,78	0,07
RecentPopular&Random	1,8547	1,4746	57,60%	9,71%	0,1662	0,5187	0,1223	0,5143	0,0244	0,0241	0,0066	0,0631	320,0398	762,5	0,14
RecentPopular&UserCF	1,8635	1,4811	56,65%	9,41%	0,1614	0,6659	0,1136	0,5048	0,0244	0,0241	0,0066	0,0631	320,0398	780,38	3,89
RecentPopular&ItemCF	1,8547	1,4781	57,06%	9,81%	0,1674	0,6673	0,1173	0,5071	0,0244	0,0241	0,0066	0,0631	320,0398	671,7	5,82
RecentPopular&ItemCFNorm	22,6958	22,6678	56,28%	8,22%	0,1434	0,6629	0,1104	0,5085	0	0	0	0,7523	1568,4075	757,82	7,03
RecentPopular&TItemCF	22,7025	22,6744	55,16%	9,12%	0,1565	0,6629	0,1	0,4887	0	0	0	0,8146	1525,4285	728,97	6,83
RecentPopular&ContentKNN	1,8435	1,4615	58,32%	9,58%	0,1646	0,6685	0,1289	0,5239	0,0244	0,0241	0,0066	0,0631	320,0398	730,42	4,88
RecentPopular&SVD	1,8418	1,4623	56,21%	9,54%	0,1631	0,6655	0,1095	0,4985	0,0244	0,0241	0,0066	0,0631	320,0398	721,43	0,15
RecentPopular&SVD++	1,8588	1,4757	57,80%	9,84%	0,1682	0,6683	0,1241	0,5159	0,0244	0,0241	0,0066	0,0631	320,0398	979,06	4,69
RecentPopular&TimeSVD++	22,6938	22,6659	57,31%	10,02%	0,1706	0,6681	0,1196	0,5087	0,0244	0,0241	0,0066	0,0631	320,0398	1103,65	0,5
RecentPopular&DoubleDQN	3,6476	3,4469	60,42%	10,26%	0,1754	0,6716	0,1527	0,6028	0,0106	0,013	0,0037	0,1938	664,2436	948,51	4,73
RecentPopular&DQN	3,66	3,4585	58,49%	9,27%	0,1600	0,6656	0,1334	0,5833	0,0053	0,0074	0,0021	0,1968	668,6159	667,95	2,85
Random&MostPopular	22,6936	22,6653	60,08%	9,70%	0,1671	0,5214	0,1452	0,5445	0,5069	0,05111	0,5069	0,0631	320,0398	1359,86	0,06

Random&RecentPopular	22,6889	22,6607	60,17%	10,05%	0,1722	0,5222	0,1458	0,5434	0,5069	0,05111	0,5069	0,0631	320,0398	1377,29	0,06
Random&UserCF	1,1464	0,941	54,71%	9,82%	0,1665	0,6854	0,0957	0,5414	0	0	0	0,9867	1593,5097	1556,06	4,29
Random&ItemCF	1,153	0,9451	45,39%	10,61%	0,1720	0,6450	0,0031	0,4479	0,0159	0,0204	0,0159	0,6669	1533,6291	1422,27	4,33
Random&ItemCFNorm	1,1317	0,9501	33,33%	0,01%	0,0002	0,6450	-0,118	0,3332	0	0	0	0,6791	1575,7122	1521,36	6,84
Random&TItemCF	1,1476	0,9597	44,44%	0,07%	0,0014	0,6450	-0,0069	0,4437	0	0	0	0,7837	1478,7213	1525,7	6,87
Random&ContentKNN	1,1533	0,9564	58,33%	10,50%	0,1780	0,6837	0,1318	0,5804	0,0074	0,0093	0,0049	0,241	823,4277	1502,38	5,02
Random&SVD	1,188	0,9664	55,09%	1,86%	0,0360	0,6489	0,0995	0,5383	0,035	0,0222	0,0129	0,1956	702,6871	1505,29	0,11
Random&SVD++	1,1627	0,9526	59,77%	1,38%	0,0270	0,6489	0,1458	0,59	0,0127	0,013	0,0048	0,2469	748,0588	1700,61	4,02
Random&TimeSVD++	1,1305	0,9532	57,43%	1,62%	0,0315	0,6502	0,1227	0,5642	0,0239	0,0146	0,0089	0,22125	725,37295	1735,1	0,4
Random&DoubleDQN	3,6592	3,4577	55,26%	5,19%	0,0949	0,6489	0,1012	0,5514	0,0085	0,0037	0,0032	0,1968	663,9244	1343,43	2,74
Random&DQN	1,3307	1,0525	48,94%	5,20%	0,0940	0,6463	0,0381	0,4876	0,0085	0,0093	0,0033	0,1923	666,4724	1452,1	3,68
UserCF&MostPopular	22,7045	22,6759	58,65%	10,03%	0,1713	0,5817	0,1318	0,525	0,5069	0,5111	0,5069	0,0631	320,0398	769,55	0,22
UserCF&RecentPopular	22,7048	22,6762	58,65%	10,03%	0,1713	0,5817	0,1318	0,525	0,5069	0,5111	0,5069	0,0631	320,0398	1238,84	0,19
UserCF&Random	1,2666	1,0176	54,04%	4,94%	0,0905	0,6036	0,0893	0,5053	0,0074	0,0074	0,0031	0,2804	776,9109	726	0,28
UserCF&ItemCF	1,1047	0,89	73,61%	12,19%	0,2092	0,6548	0,2821	0,7294	0,0011	0,0019	0,0004	0,6308	1472,3651	700,89	4,05
UserCF&ItemCFNorm	1,0993	0,8897	74,60%	13,24%	0,2249	0,6507	0,2931	0,7425	0	0	0	0,6718	1577,4543	704,78	4,66
UserCF&TItemCF	1,0608	0,8384	76,26%	14,55%	0,2444	0,6664	0,3051	0,7502	0	0	0	0,8158	1506,7295	785,445	6,83
UserCF&ContentKNN	1,0889	0,881	70,95%	13,12%	0,2214	0,6582	0,2548	0,6986	0,0053	0,0093	0,0027	0,2885	852,4131	655,65	5,14
UserCF&SVD	1,0029	0,7975	78,15%	16,84%	0,2771	0,6782	0,32	0,7642	0,0361	0,0537	0,0087	0,0242	177,712	749,82	0,47
UserCF&SVD++	1,0136	0,803	77,31%	15,84%	0,2629	0,6776	0,3119	0,755	0,0308	0,0444	0,0085	0,0233	189,4086	995,82	7,56
UserCF&TimeSVD++	1,1373	0,9409	77,73%	16,54%	0,2728	0,6827	0,3326	0,7596	0,0335	0,0461	0,0086	0,02375	183,5603	1064,58	0,73
UserCF&DoubleDQN	1,3296	1,0499	63,41%	10,23%	0,1762	0,6729	0,1826	0,6331	0,018	0,0148	0,0081	0,1964	665,339	960,21	4,75
UserCF&DQN	1,3212	1,0483	57,14%	9,25%	0,1592	0,6714	0,12	0,5699	0,0074	0,0111	0,0028	0,187	656,8627	811,68	4,3
ItemCF&MostPopular	22,7001	22,6716	61,23%	10,32%	0,1766	0,5743	0,1555	0,5546	0,5069	0,5111	0,5069	0,0631	320,0398	632,95	0,06
ItemCF&RecentPopular	22,6729	22,6447	61,18%	10,40%	0,1778	0,5747	0,155	0,53535	0,5069	0,5111	0,5069	0,0631	320,0398	881,11	0,08
ItemCF&Random	1,2404	1,001	50,28%	3,99%	0,0739	0,5994	0,0531	0,4703	0,0085	0,0111	0,0027	0,3654	881,7265	795,56	0,34
ItemCF&UserCF	1,0804	0,8542	79,64%	13,08%	0,2247	0,6734	0,3405	0,7897	0,0159	0,0222	0,0025	0,4184	843,6733	940,25	4,42
ItemCF&ItemCFNorm	1,0301	0,8163	84,90%	12,82%	0,2228	0,6863	0,3929	0,8446	0	0	0	0,791	1567,5451	804,09	7,62
ItemCF&TItemCF	1,0249	0,8133	85,33%	11,26%	0,1989	0,6953	0,3964	0,8484	0	0	0	0,7926	1494,0431	850,78	6,47

ItemCF&ContentKNN	1,0354	0,8241	79,82%	18,12%	0,2954	0,7136	0,3422	0,7915	0,0032	0,0056	0,0011	0,2618	834,109	704,71	4,49
ItemCF&SVD	1,0142	0,8075	84,73%	12,49%	0,2177	0,6640	0,3901	0,8418	0,0011	0	0,0001	0,4373	1335,6129	699,74	0,39
ItemCF&SVD++	1,0211	0,8111	83,51%	13,40%	0,2309	0,6632	0,3782	0,8293	0	0	0	0,3641	1525,9355	1104,84	5,35
ItemCF&TimeSVD++	1,1327	0,9459	84,12%	14,45%	0,2466	0,6672	0,3611	0,8356	0,0006	0	0,0001	0,4007	1430,7742	1118,79	7,9
ItemCF&DoubleDQN	1,3271	1,0529	75,56%	15,27%	0,2541	0,6972	0,1041	0,5538	0,0085	0,0093	0,004	0,2004	669,9732	990,99	3,73
ItemCF&DQN	1,3262	1,0503	69,81%	12,31%	0,2093	0,6913	0,1965	0,6468	0,0138	0,0167	0,0059	0,1958	663,8723	1044,38	3,47
ItemCFNorm&MostPopular	22,7059	22,6772	60,46%	9,78%	0,1683	0,5834	0,1486	0,5485	0,5069	0,5111	0,5069	0,0631	320,0398	1030,31	0,11
ItemCFNorm&RecentPopular	22,7025	22,6741	59,55%	10,00%	0,1713	0,5830	0,1401	0,5361	0,5069	0,5111	0,5069	0,0631	320,0398	728,75	0,07
ItemCFNorm&Random	1,2045	0,9761	56,55%	3,72%	0,0698	0,6029	0,1136	0,5416	0,0085	0,0111	0,0023	0,3629	883,6845	769,22	0,13
ItemCFNorm&UserCF	1,0749	0,8727	97,00%	20,02%	0,3319	0,6451	0,5486	1	0,0032	0,0037	0,0004	0,709	1372,4076	757,002	3,88
ItemCFNorm&ItemCF	1,0213	0,8144	83,67%	14,84%	0,2521	0,6602	0,3807	0,832	0,0042	0,0074	0,002	0,4817	1203,3789	931,53	6,48
ItemCFNorm&TItemCF	1,0172	0,8129	88,67%	16,78%	0,2822	0,6610	0,4304	0,8837	0	0	0	0,6828	1479,6199	949,79	7,25
ItemCFNorm&ContentKNN	1,0239	0,8197	87,06%	15,79%	0,2673	0,7123	0,4144	0,867	0,0042	0,0074	0,0033	0,2705	859,7301	945,56	5,84
ItemCFNorm&SVD	1,0165	0,8112	86,07%	12,73%	0,2218	0,6601	0,4046	0,8569	0	0	0	0,2374	843,5228	829,18	0,14
ItemCFNorm&SVD++	1,0161	0,8148	86,25%	12,77%	0,2225	0,6605	0,4064	0,8587	0,0011	0,0019	0,0001	0,5836	1160,2404	1131,32	4,12
ItemCFNorm&TimeSVD++	1,1311	0,9455	86,16%	12,75%	0,2221	0,6603	0,4055	0,8578	0,00055	0,00095	0,00005	0,4105	1001,8816	1034,22	0,38
ItemCFNorm&DoubleDQN	1,3195	1,0411	72,94%	16,24%	0,2657	0,7098	0,0581	0,5075	0,0042	0,0056	0,002	0,1953	662,8486	1126,57	4,2
ItemCFNorm&DQN	1,3185	1,0429	70,05%	15,20%	0,2498	0,7033	0,0487	0,4983	0,0085	0,0093	0,0035	0,1894	662,2883	889,22	5,89
TItemCF&MostPopular	22,6984	22,6703	60,89%	10,29%	0,1760	0,5851	0,1523	0,5506	0,5069	0,5111	0,5069	0,0631	320,0398	846,25	0,07
TItemCF&RecentPopular	22,6977	22,6691	58,73%	10,12%	0,1726	0,5821	0,1326	0,5254	0,5069	0,5111	0,5069	0,0631	320,0398	810,72	0,09
TItemCF&Random	1,2439	1,0014	56,07%	4,86%	0,0894	0,6049	0,1088	0,5288	0,0075	0,0074	0,0042	0,3511	867,5959	842,33	0,13
TItemCF&UserCF	1,0403	0,8171	77,68%	14,77%	0,2482	0,6681	0,3187	0,7648	0,0042	0,0074	0,0005	0,8857	1434,3385	834,71	4,69
TItemCF&ItemCF	0,99	0,7813	86,15%	24,76%	0,3846	0,6714	0,4018	0,8547	0,0117	0,0167	0,0018	0,3791	1084,975	901,37	5,61
TItemCF&ItemCFNorm	1,0294	0,8169	85,17%	23,46%	0,3679	0,6640	0,3945	0,8465	0	0	0	0,6114	1567,3534	890,53	8,36
TItemCF&ContentKNN	1,0022	0,7977	83,37%	20,46%	0,3286	0,7150	0,3767	0,8277	0,0011	0,0019	0,0011	0,2871	869,4297	752,03	4,95
TItemCF&SVD	0,9715	0,7689	86,06%	24,53%	0,3818	0,6700	0,4013	0,8541	0,0053	0,0093	0,0014	0,0384	492,9589	818,17	0,17
TItemCF&SVD++	0,968	0,7656	87,05%	25,52%	0,3947	0,6704	0,411	0,8645	0,0042	0,0074	0,0008	0,055	568,5435	995,27	3,84
TItemCF&TimeSVD++	1,1296	0,9369	86,56%	24,53%	0,3822	0,6716	0,4062	0,8593	0,0048	0,0084	0,0011	0,0467	530,7512	1087,57	0,5
TItemCF&DoubleDQN	1,3283	1,0556	79,57%	25,25%	0,3834	0,7101	0,1443	0,5944	0,0064	0,0074	0,0024	0,1964	662,0155	1139,92	3,99

TItemCF&DQN	1,2373	0,9843	76,90%	23,34%	0,3581	0,6978	0,1799	0,5673	0,0033	0,0028	0,0008	0,2316	1542,1256	1385,77	4,22
ContentKNN&MostPopular	22,6966	22,6684	58,43%	9,87%	0,1689	0,5812	0,1299	0,5233	0,5069	0,5111	0,5069	0,0631	320,0398	908,8	0,11
ContentKNN&RecentPopular	22,7065	22,678	58,87%	9,85%	0,1687	0,5818	0,1339	0,5289	0,5069	0,5111	0,5069	0,0631	320,0398	775,72	0,06
ContentKNN&Random	1,2661	1,0262	53,94%	4,04%	0,0752	0,6019	0,0883	0,5106	0,0106	0,0111	0,0045	0,3555	868,2276	790,89	0,13
ContentKNN&UserCF	1,0541	0,8552	79,40%	27,51%	0,4086	0,6576	0,3389	0,7884	0,0074	0,013	0,0013	0,0764	765,26	700,79	4,06
ContentKNN&ItemCF	1,0051	0,8056	85,00%	33,23%	0,4778	0,7056	0,3932	0,8451	0,0085	0,013	0,003	0,4769	1108,2748	770,24	3,93
ContentKNN&ItemCFNorm	1,0213	0,8209	81,76%	29,19%	0,4302	0,8310	0,3628	0,8134	0	0	0	0,7324	1558,9438	809,67	7,59
ContentKNN&TItemCF	1,0012	0,802	90,08%	39,80%	0,5521	0,8564	0,4439	0,8979	0	0	0	0,7406	1461,6355	812,17	6,94
ContentKNN&SVD	0,9955	0,7997	85,12%	33,69%	0,4827	0,6651	0,3936	0,8456	0,0053	0,0093	0,0011	0,0388	601,672	859,17	0,18
ContentKNN&SVD++	0,9999	0,8034	86,75%	34,90%	0,4978	0,6646	0,41	0,8629	0,0032	0,0056	0,0006	0,0714	690,6912	1618,17	7,69
ContentKNN&TimeSVD++	1,1238	0,9437	85,94%	33,59%	0,4830	0,6649	0,402	0,85425	0,0033	0,0065	0,0009	0,0551	646,1816	983,26	0,41
ContentKNN&DoubleDQN	1,3235	1,0498	75,11%	25,29%	0,3784	0,7104	0,1596	0,6096	0,0053	0,0056	0,0030	0,19	670,0045	876,48	3,52
ContentKNN&DQN	1,3136	1,0403	70,00%	20,22%	0,3138	0,7085	0,0487	0,4982	0,0042	0,0074	0,0020	0,1925	662,997	717,63	3,44
SVD&MostPopular	22,707	22,6786	58,53%	10,12%	0,1726	0,5820	0,1307	0,5229	0,5069	0,5111	0,5069	0,0631	320,0398	634,39	0,06
SVD&RecentPopular	22,7079	22,6793	60,92%	10,37%	0,1773	0,5853	0,1526	0,5505	0,5069	0,5111	0,5069	0,0631	320,0398	697,51	0,05
SVD&Random	1,2847	1,0345	58,50%	5,25%	0,0964	0,6075	0,1321	0,5535	0,0074	0,0056	0,0037	0,169	616,7233	560,04	0,11
SVD&UserCF	0,996	0,7884	77,34%	16,57%	0,2729	0,7279	0,3125	0,756	0,0403	0,063	0,0117	0,0322	210,4195	689,12	2,9
SVD&ItemCF	0,9944	0,7918	86,54%	23,86%	0,3741	0,7181	0,4073	0,8601	0,0127	0,0185	0,0053	0,5267	1337,2407	645,84	4,4
SVD&ItemCFNorm	1,0168	0,8141	87,46%	24,38%	0,3813	0,7229	0,4171	0,8696	0	0	0	0,6698	1570,7128	651,27	4,61
SVD&TItemCF	0,9657	0,7657	83,54%	20,01%	0,3229	0,7309	0,3775	0,8285	0	0	0	0,787	1501,6859	710,72	7,23
SVD&ContentKNN	0,9986	0,8036	86,49%	23,58%	0,3706	0,7166	0,4073	0,86	0,0064	0,0111	0,0038	0,2929	854,2148	628,64	3,73
SVD&SVD++	0,9183	0,726	87,30%	26,01%	0,4008	0,6788	0,4106	0,8651	0,0265	0,0426	0,0066	0,0231	207,2375	898,15	4,07
SVD&TimeSVD++	1,1126	0,9308	86,90%	24,80%	0,3858	0,6759	0,4057	0,8626	0,0155	0,0259	0,0052	0,158	327,72615	1014,03	0,32
SVD&DoubleDQN	1,3223	1,0467	75,92%	15,24%	0,2538	0,6837	0,0679	0,5174	0,0085	0,0130	0,0044	0,1976	671,3504	899,5	3,64
SVD&DQN	1,3129	1,0394	73,49%	14,20%	0,2380	0,6828	0,0835	0,5334	0,0095	0,0074	0,0042	0,1953	667,2612	817,87	3,98
SVD++ &MostPopular	22,6919	22,6637	58,55%	9,70%	0,1665	0,5811	0,131	0,5258	0,5069	0,5111	0,5069	0,0631	320,0398	1521,39	0,06
SVD++ &RecentPopular	22,7037	22,675	60,39%	10,31%	0,1761	0,5846	0,1477	0,5444	0,5069	0,5111	0,5069	0,0631	320,0398	1334,42	0,06
SVD++ &Random	1,3237	1,0692	54,22%	4,97%	0,0911	0,6037	0,0911	0,5072	0,0180	0,0222	0,0069	0,1094	495,858	1445,35	0,11
SVD++ &UserCF	0,9809	0,7764	81,26%	17,25%	0,2846	0,7080	0,3494	0,7972	0,0414	0,0648	0,0118	0,0613	237,6351	1473,22	2,9

SVD++ &ItemCF	0,9552	0,7555	87,32%	24,69%	0,3850	0,7101	0,4134	0,8672	0,0159	0,0259	0,0057	0,4512	1196,8855	1597,96	4,03
SVD++ &ItemCFNorm	1,0069	0,8064	85,48%	22,75%	0,3594	0,6731	0,3988	0,8508	0	0	0	0,6828	1568,2228	1421,63	5,51
SVD++ &TItemCF	0,9464	0,7501	86,75%	23,30%	0,3673	0,6690	0,4085	0,8617	0	0	0	0,7881	1496,3394	1692,06	5,02
SVD++ &ContentKNN	0,9948	0,7958	85,11%	22,64%	0,3577	0,6649	0,3936	0,8456	0,0021	0,0037	0,0021	0,2914	854,9237	1548,29	4,09
SVD++ &SVD	0,8995	0,7088	88,32%	26,45%	0,4071	0,6816	0,4197	0,8753	0,0308	0,0537	0,0096	0,0245	214,2696	1191,72	0,244
SVD++ &TimeSVD++	1,1276	0,9449	86,72%	23,05%	0,3641	0,6762	0,4042	0,8605	0,0155	0,0277	0,0059	0,15795	331,59665	1888,65	0,39
SVD++ &DoubleDQN	1,3149	1,0439	63,90%	10,16%	0,1753	0,6708	-0,0122	0,4373	0,0074	0,0074	0,0029	0,192	659,232	2036,19	4,33
SVD++ &DQN	1,3133	1,041	61,85%	10,25%	0,1759	0,6699	0,0672	0,5166	0,0138	0,0185	0,0059	0,1929	665,4717	2005,59	3,79
TimeSVD++ &MostPopular	22,7046	22,6764	58,49%	97,90%	0,1677	0,5904	0,1304	0,5246	0,5069	0,5111	0,5069	0,0631	320,0398	1316,86	0,05
TimeSVD++ &RecentPopular	22,6893	22,6612	59,27%	10,18%	0,1737	0,5906	0,1375	0,5316	0,5069	0,5111	0,5069	0,0631	320,0398	1616,95	0,06
TimeSVD++ &Random	1,1182	0,939	53,22%	1,38%	0,0269	0,5971	0,086	0,5084	0,0138	0,0167	0,0038	0,1092	492,5269	1595,98	0,27
TimeSVD++ &UserCF	1,1301	0,9482	78,82%	17,03%	0,2801	0,6617	0,3389	0,7733	0,0402	0,0629	0,0114	0,0528	224,7854	1651,74	4,15
TimeSVD++ &ItemCF	1,1307	0,9476	78,52%	16,73%	0,2759	0,6578	0,3359	0,7703	0,0372	0,0599	0,0084	0,4489	1188,4625	1610,98	3,28
TimeSVD++ &ItemCFNorm	1,1341	0,9513	80,88%	16,93%	0,2801	0,6776	0,3460	0,7934	0,0383	0,0617	0,0087	0,6801	1433,3014	1606,81	5,08
TimeSVD++ &TItemCF	1,1234	0,9422	81,19%	17,24%	0,2845	0,6815	0,3491	0,7965	0,0414	0,0647	0,0118	0,7765	1357,8765	1561,68	5,45
TimeSVD++ &ContentKNN	1,118	0,9396	73,27%	11,38%	0,1970	0,6480	0,0861	0,5089	0,0138	0,0167	0,0038	0,2874	760,3921	1483,17	4,12
TimeSVD++ &SVD	1,135	0,951	80,72%	16,92%	0,2798	0,6762	0,3453	0,7918	0,0382	0,0615	0,0087	0,1453	897,0243	1414,29	0,28
TimeSVD++ &SVD++	1,1241	0,9438	84,92%	7,28%	0,1341	0,7114	0,3633	0,8331	0,0402	0,0647	0,0091	0,1509	875,0853	1813,99	3,6
TimeSVD++ &DoubleDQN	1,3254	1,0486	70,00%	10,24%	0,1787	0,7038	0,0487	0,498	0,0074	0,0019	0,0046	0,1929	666,6376	1567,51	3,96
TimeSVD++ &DQN	1,3196	1,0426	65,00%	10,27%	0,1774	0,6914	0,1485	0,5985	0,0085	0,0074	0,0055	0,1919	659,9138	1435,2	2,64
DQN&MostPopular	22,6957	22,6672	60,25%	10,18%	0,1742	0,5841	0,1465	0,5436	0,5069	0,5111	0,5069	0,0631	320,0398	1628,01	0,07
DQN&RecentPopular	22,6895	22,6612	59,79%	9,84%	0,1691	0,5828	0,1424	0,54	0,5069	0,5111	0,5069	0,0631	320,0398	1655,15	0,05
DQN&Random	2,5309	2,2563	42,86%	6,08%	0,1065	0,5950	-0,0227	0,4277	0,0021	0,0037	0,0007	0,3596	867,1183	1498,7	0,12
DQN&UserCF	1,2477	1,0089	84,97%	20,90%	0,3355	0,7013	0,1756	0,9649	0,0113	0,0056	0,0067	0,896	1568,9612	2851,13	3,56
DQN&ItemCF	1,594	1,3532	76,11%	15,81%	0,2618	0,7177	0,1572	0,8642	0,0135	0,1230	0,0036	0,0451	1523,5714	1815,61	4,89
DQN&ItemCFNorm	1,2695	1,0395	81,28%	18,86%	0,3062	0,7664	0,1679	0,9229	0,0108	0,0054	0,0064	0,17	1573,9375	1634,49	3,79
DQN&TItemCF	1,267	1,0344	77,73%	11,83%	0,2053	0,7330	0,1606	0,8826	0,0066	0,0072	0,0027	0,17	1573,9375	1583,05	5,15
DQN&ContentKNN	2,0905	1,837	72,78%	10,77%	0,1877	0,6863	0,1504	0,8264	0,0097	0,0048	0,0058	0,1353	701,2426	1561,6	4,18
DQN&SVD	2,1354	1,8394	53,33%	10,01%	0,1686	0,6902	-0,1180	0,3332	0	0	0	0,0618	138,8065	1880,3	0,26

DQN&SVD++	1,5398	1,2985	66,67%	10,02%	0,1742	0,6957	0,2153	0,6666	0	0	0	0,066	182,7213	2694,17	4,29
DQN&TimeSVD++	3,0738	2,8582	63,33%	10,01%	0,1729	0,6967	-0,1357	0,3832	0	0	0	0,07107	159,627475	3340,28	0,51
DQN&DoubleDQN	1,9253	1,6645	57,45%	10,24%	0,1738	0,6976	0,1230	0,5730	0,0074	0,0056	0,0035	0,1991	671,057	2479,75	3,72
DoubleDQN&MostPopular	22,7121	22,6838	60,85%	10,34%	0,1767	0,5853	0,1520	0,5499	0,5069	0,5111	0,5069	0,0631	320,0398	1730,02	0,08
DoubleDQN&RecentPopular	22,6941	22,666	59,99%	10,10%	0,1730	0,5837	0,1441	0,5408	0,5565	0,5111	0,5069	0,0631	320,0398	1712,32	0,06
DoubleDQN&Random	1,3179	1,0989	50,00%	6,08%	0,1084	0,5951	0,0486	0,4993	0,0011	0,0019	0,0002	0,3673	879,2716	1570,17	0,1
DoubleDQN&UserCF	2,7482	2,5029	90,91%	30,09%	0,4521	0,6842	0,4575	0,9090	0,0171	0,0171	0,0006	0,2608	852,0436	2375,86	3,36
DoubleDQN&ItemCF	2,4637	2,1958	97,00%	35,01%	0,5145	0,7224	0,5486	1,0000	0,0011	0,0019	0,0011	0,4808	1412,2692	2262,39	4,33
DoubleDQN&ItemCFNorm	1,265	1,0312	82,20%	21,69%	0,3432	0,8079	0,1253	0,7094	0,0167	0,0122	0,0087	0,6448	1573,9375	2482,76	6,08
DoubleDQN&TItemCF	1,6219	1,409	81,05%	20,67%	0,3294	0,7966	0,1236	0,6995	0,0165	0,0121	0,0086	0,6448	1573,9375	2286,64	4,53
DoubleDQN&ContentKNN	1,2645	1,0318	83,73%	22,72%	0,3574	0,8230	0,1277	0,7226	0,0170	0,0125	0,0089	0,1353	701,2426	2419,5	4,74
DoubleDQN&SVD	1,6536	1,3927	81,45%	21,67%	0,3424	0,8005	0,1242	0,7029	0,0146	0,0321	0,0076	0,2653	105	2451,33	0,16
DoubleDQN&SVD++	1,2667	1,0359	80,56%	19,66%	0,3160	0,7418	0,1228	0,6952	0,0156	0,0245	0,0117	0,0671	132,9211	1797,23	3,78
DoubleDQN&TimeSVD++	1,2655	1,0338	80,52%	19,66%	0,3160	0,7914	0,1228	0,6949	0,0150	0,0281	0,0096	0,1652028	118,2467867	2490,31	0,48
DoubleDQN&DQN	1,9281	1,6651	77,89%	15,29%	0,2556	0,6845	0,1275	0,5772	0,0106	0,0111	0,0044	0,1901	661,6044	2695,63	3,92
Es posible tener una pequeña discrepancia sobre la conclusión del mejor modelo entre la tasa de fallos y el AUC, porque se obtienen por el promedio de validación cruzada repetida.															