Sistemas de recomendación aplicados a jueces en línea

Trabajo de Fin de Grado

Pedro Pablo Doménech Arellano
Alfonso Soria Muñoz

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Mayo 2019

Documento maquetado con TEXIS v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

Sistemas de recomendación aplicados a jueces en línea

Memoria que presentan para optar al título de Grado en Ingeniería Informática

Sistemas de recomendación aplicados a jueces en línea

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Mayo 2019

Resumen

Un *sistema de recomendación* aplicado a *jueces en línea*, como bien muestra el título de este documento, aúna dos conceptos que deben entenderse previamente.

El primer concepto de ellos es el de *recomendador*. Un recomendador es un sistema de filtrado de información que busca predecir la preferencia de un usuario respecto a un conjunto de elementos. El objetivo de los recomendadores, por lo tanto, es facilitar las búsquedas del usuario dentro de una plataforma, de manera que al usuario se le faciliten elementos personalizados que tienen relación con el comportamiento que este genera bajo la plataforma.

El segundo concepto es el de *juez en línea*. Un juez en línea funciona como un repositorio de ejercicios que un usuario puede hacer y enviar, para así obtener un veredicto. Los jueces en línea son, por lo tanto, un tipo de plataformas orientadas a la resolución de ejercicios de programación, con la finalidad de que los usuarios puedan intentar las propuestas de problemas que cada plataforma tenga y comprobar el estado de sus envíos. Este estado indica si el envío es correcto o incorrecto y a su vez incorpora información extra como el tiempo de ejecución o el consumo de memoria.

Este trabajo, por lo tanto, consiste en el estudio de recomendadores y jueces en línea con alto detalle, para así poder aunar el mundo de los recomendadores y centrarlos en estas plataformas, llegando a redactar varios modelos de recomendación y evaluarlos sobre un caso real de juez en línea, como es el de la plataforma *¡Acepta el reto!*.

Palabras clave: sistema de recomendación, teorema de bayes, k-vecinos más cercanos, juez en línea, recomendadores, *¡Acepta el reto!*.

Abstract

A recommender system applied to online judges, as the title of this document says, join two concepts that must be understood previously.

The first concept of them is *recommender*. A *recommender* is an information filtering system that seeks to predict the preference of an user regarding a set of elements. The goal of the reccomenders therefore, is to facilitate user searches within a platform, so that through the recommender, the user is provided with customized elements that are related to the behavior that is generated under the platform.

The second concept is *online judge*. An *online judge* works as a repository of exercises that an user can do and send, in order to obtain a result that is a verdict. *Online judges* are therefore a type of platforms oriented to the resolution of programming exercises, in order that users can try the proposals of problems that each platform has and check the status of their submissions with extra information in many cases like the time of execution and the memory that the sent program has consumed, as well as the main information on whether the submission is valid or dddoes not work correctly.

This work, therefore, consists of the study of *recommenders* and *online judges* with high detail, so we can bring together the world of recommenders and center them on these platform, *¡Acepta el reto!*.

Keywords: recommender system, bayes theorem, K-Nearest neighbors, online judges, recommenders, *¡Acepta el reto!*.

Notas de los autores

En el presente documento se pone nuestro esfuerzo al servicio del conocimiento, intentando aunar conceptos aprendidos a lo largo de estos años, para aplicarlos sobre un mundo desconocido hasta ahora para quienes elaboramos esta memoria.

Este es el mundo de los recomendadores, cada día más usados en todos los aspectos de nuestra vida cotidiana, como interesantes por todo el potencial que tienen y que les queda por tener.

Este trabajo de fin de grado es, por lo tanto, un intento de documentarnos y aprender sobre algo nuevo, así como ponerlo en práctica y llevarlo a cabo en el ámbito de los jueces en línea, plataformas que tienen poca investigación de trasfondo en lo que al mundo de la recomendación se refiere.

Es además interesante poder estudiar la multitud de modelos de recomendación con ideas tan diferentes entre ellos, pero cuyo objetivo siempre es el mismo: generar una solución correcta a las necesidades del usuario.

Por otro lado tenemos las plataformas de jueces en línea, las cuales son cada día más populares entre matemáticos, ingenieros, y personas cercanas a estos sectores, ya que proponen una serie de ejercicios que favorecen y ejercitan la agilidad mental, y mejoran los conocimientos del campo de la algoritmia.

Son además destacables los jueces en línea para los estudiantes de estas áreas, ya que mediante estas plataformas pueden aprender ciertas ramas de sus carreras de una manera entretenida y en algunos casos, incluso, competitiva, para aquellas que son capaces de integrar concursos. De esta manera, un juez en línea a la vez de ser una plataforma de ocio, favorece el aprendizaje de los usuarios que se aficionan a estas plataformas.

Con este documento no sólo nos aportamos a nosotros mismos nuevos conocimientos, adentrándonos en estos dos mundos que no se tocan prácticamente durante el transcurso de la carrera, sino que intentamos aportar nuestro grano de arena tanto a la investigación de nuevos modelos de recomendación, como a la investigación de la implementación de recomendadores en este modelo de plataformas, para que generen soluciones óptimas a unas necesidades concretas.

Sin más dilación, le dejamos con nuestro trabajo, y esperamos que lo

disfrute leyéndolo tanto como nosotros hemos disfrutado realizándolo.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

La revolución digital hace cada vez más fácil la vida de los usuarios, permitiéndoles ganar tiempo así como tener información personal y general siempre a mano. Vivimos en un momento donde hace unos años (y en ocasiones meses) no existían plataformas y aplicaciones que ahora son usadas de forma masiva, y cada día surgen nuevas ideas que tienen un éxito de uso inmediato.

Este proyecto parte de la idea de dotar con herramientas que facilitan la vida del usuario, a plataformas emergentes que son los *jueces en línea*. Estas herramientas se van a centrar en concreto en los *sistemas de recomendación* o SR, que permiten ahorrar el tiempo del usuario haciéndole propuestas y sugerencias basadas en el comportamiento que este genera bajo la plataforma.

Cuando escogimos este trabajo de fin de grado, lo hicimos porque vimos una idea prometedora para dotar de estas herramientas a un juez en línea en concreto que es popular entre la facultad, y gracias a todo el camino que se ha realizado a lo largo de este año, se ha logrado a hacer un estudio muy completo para poder llegar a aplicar dos recomendadores que se han realizado bajo la plataforma que es *¡Acepta el reto!*.

1.1. Objetivos

El presente trabajo de fin de grado tiene los siguientes objetivos:

- Estudiar diferentes modelos de recomendación.
- Realizar un estudio sobre las plataformas de jueces en línea.
- Estudiar casos ya existentes de implementaciones de recomendadores en plataformas de jueces en línea.
- Realizar un estudio sobre métodos de evaluación para analizar la eficacia de un recomendador.

- Analizar modelos de implementación como servicio externo para los sistemas de recomendación.
- Implementar un par de modelos de recomendación destacables y que funcionen bien sobre plataformas de jueces en línea y proponer una arquitectura de comunicación para que funcione como servicio externo.
- Evaluar sobre un juez en línea la eficacia de los modelos implementados y poder contrastar resultados para ver las diferencias de comportamiento de cada modelo de recomendación realizado.

1.2. Plan de trabajo

La planificación seguida en este trabajo ha comenzado por documentarnos en diferentes aspectos. Primero se ha realizado el estudio sobre sistemas de recomendación, empezando a ver lo que son, su funcionamiento, sus objetivos y acabando por estudiar diferentes técnicas existentes que pueden usarse para estos sistemas. De estas técnicas se focaliza el estudio con mayor profundidad en tres de ellas que han resultado las más interesantes.

Posteriormente y paralelo al estudio en profundidad de tres técnicas de recomendación concretas, se ha comenzado a documentar y recabar información sobre los jueces en línea, y en concreto sobre el funcionamiento de *¡Acepta el reto!* un juez en línea con el que se ha podido trabajar en este proyecto. Se ha hecho un análisis de sus datos, desde las entregas hasta la cantidad de problemas y usuarios que esta plataforma maneja.

Finalmente, se ha hecho un análisis de jueces en línea existentes que ya implementan sistemas de recomendación, y se han leído artículos de sistemas de recomendación propuestos para ser aplicados a jueces en línea.

Tras esta primera fase de analizar todo lo existente y documentarnos de información, se ha entrado en una segunda fase de implementación, donde se han realizado dos modelos de recomendación y posteriormente se han optimizado para reducir los cálculos a tiempos insignificantes.

Estos modelos de recomendación han pasado por una fase de evaluación individual sobre la base de datos de *¡Acepta el reto!* para analizar su comportamiento y posteriormente se ha realizado una evaluación en conjunto para contrastar resultados y sacar unas conclusiones para ambos recomendadores.

Finalmente se ha realizado una propuesta de API de comunicación, que permite a estos sistemas de recomendación poder ser usados como un servicio externo para los jueces en línea. Esta API de comunicación se ha podido poner a prueba con uno de los modelos implementados sobre *¡Acepta el reto!*.

Todo este plan de trabajo acaba en unas conclusiones interesantes sobre la multitud de análisis que se ha intentado hacer y por otro lado se proponen las líneas de trabajo futuro que puede tomar desde este punto el proyecto que se ha realizado.

Capítulo 2

Introduction

The Digital Revolution makes life easier for the public, allowing them to save time in addition to have general and personal information accessible. We live at a time where a few years ago (or even only months) we had no platforms or applications that are now extremely used, and every day come new ideas that have an immediate success.

This project arises from the idea of providing tools that make life easier for a user to the new platforms that are the *online judges*. These tools are going to focus on the *recommender systems* or RS, that save the user time making him proposals and suggestions based on the behavior that the user generates on the platform.

When we chose this final thesis, we saw that it was a promising idea to provide these tools to a popular online judge at the faculty, and due all the work made this year, we managed to make a full study to be able to apply two recommenders under the *¡Acepta el reto!* platform.

2.1. Objectives

This thesis has the following objectives:

- Study different recommendation models.
- Perform a study about online judge platforms.
- Investigate existing cases of implemented recommenders in online judges.
- Perform another study about evaluation methods to analyze a recommender effectiveness.
- Analyze implementation models as an external service to the recommendation systems.

- Implement a couple of remarkable implementation models that work satisfactorily on the online judges platforms and propose a communication architecture to work as an external service.
- Evaluate the effectiveness of implemented models on an online judge and compare results to see the behavior differences of each recommendation model.

2.2. Work plan

The work plan followed in this dissertation has begun by gathering information on different aspects. First, we have done a study on recommendation systems, starting to see what they are, how they work, their objectives and finally studying different existing techniques that can be used for these systems. Of these techniques, our study has focused in greater depth on three of them that have turned out to be the more interesting ones.

Subsequently, and parallel to the in-depth study of the three concrete recommendation techniques, we have begun to gather information about online judges, and specifically about how *¡Acepta el reto!* works, an online judge with which we have been able to work in this project. We have done an analysis of the data, from the user deliveries to the number of exercises and users that this platform manages.

Finally, we have carried out an analysis of existing online judges which already implement recommendation systems and we have read articles from proposed systems to be added to online judges.

After this first phase of analyzing all the previous information and collecting data, we have entered into a second phase of implementation, where we have made two recommendation models and then have been optimized to reduce the calculation time to a minimum.

These recommendation models have gone through an individual evaluation phase on the *¡Acepta el reto!* database to analyze their behavior and then, we have carried out an evaluation between both to compare results and obtain conclusions for the twain recommenders.

Finally, we have proposed a communication API, which allows these recommendation systems to be used as an external service for online judges. This communication API has been tested with one of the models implemented on *¡Acepta el reto!*.

All this work plan ends in some intriguing conclusions about the multitude of analysis that have been tried and additionally we have decided the future work paths that we can follow from this point the project.

Capítulo 3

Sistemas de recomendación

Resumen: En este capítulo se introducen los sistemas de recomendación, en adelante también llamados *SR*, entrando en detalle en la finalidad de estos, y explicando diferentes técnicas de recomendación que pueden ser aplicadas y se han considerado como las más destacables para este trabajo.

Con el avance de los sistemas informáticos, se han ido desarrollando sistemas orientados a usuarios que manejan información, como son las plataformas que gestionan diferentes tipos de contenido.

La gran problemática de ofrecer al usuario el contenido más indicado a su perfil fue surgiendo a medida que el contenido crecía masivamente dentro de estas plataformas. Fue por esto que aparecieron necesidades de implementar nuevas tecnologías que permitieran proporcionar al usuario ciertas facilidades, para obtener cómodamente lo que éste realmente necesitaba de entre toda la información existente.

Estas tecnologías, que facilitaban la selección de la información que más le interesaba al usuario, engloban el mundo de los buscadores, los predictores, los clasificadores de contenido y los sistemas de recomendación, entre otros.

Son estas últimas tecnologías las que, aplicando unas reglas internas o algoritmos, logran, en mayor o menor medida, encontrar similitudes entre elementos o en adelante, ítems, para facilitar al usuario el acceso a lo que éste más necesita o le pueda resultar más interesante y cercano a sus gustos, según las relaciones o el historial de interacciones que el usuario ha realizado.

Como se muestra en la figura 3.1, un sistema de recomendación genera una recomendación, basándose en relaciones de interacción entre usuarios e ítems, y haciendo uso de algoritmos internos que asignan valores a estas relaciones para la generación de las recomendaciones.

El recomendador se encarga por lo tanto, de encontrar estas relaciones generadas por el historial del usuario sobre la actividad de los ítems y hacer

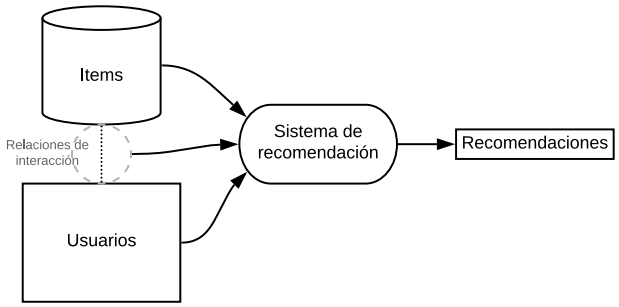

Figura 3.1: Funcionamiento en básico de un sistema de recomendación.

los cálculos pertinentes para asociar nuevos ítems en concordancia con estas relaciones y similitudes (Aggarwal (2016)).

3.1. Recomendadores

Los sistemas de recomendación proporcionan la información más adecuada a partir de las interacciones y acciones que el usuario realiza.

Un recomendador es por lo tanto un sistema capaz de generar información subjetiva y filtrada, con ciertas características, gracias a un algoritmo que analiza la actividad del usuario y otros datos (Aggarwal (2016)).

Estas recomendaciones generadas tienen la característica de ser adaptadas al usuario al que se le está generando la recomendación, ya que el propósito general de un recomendador es facilitar los datos necesarios de manera *personalizada*.

Actualmente un portal web con gran afluencia de usuarios y contenido dispone de sistemas de recomendación. Como ejemplos de casos populares y de éxito, que tienen importantes y destacables recomendadores debido a su complejidad y que prestan gran apoyo a las plataformas sobre las que actúan, podemos mencionar a Netflix[1] (series y películas), Amazon[2] (venta de productos), o YouTube[3] (contenido audiovisual).

Son estos sistemas de recomendación tan avanzados los que han favorecido la popularidad y el éxito creciente de estas plataformas.

Otros sistemas de recomendación son usados de manera menos directa para asociar a usuarios elementos específicos, como se da en el ámbito de la publicidad en internet, cuyo referente más representable en este caso es Google Adwords[4], que, a través de la información de un usuario que usa

[1] `https://www.netflix.com`
[2] `https://www.amazon.es`
[3] `https://www.youtube.com`
[4] `https://ads.google.com`

servicios de Google, refina y clasifica qué tipos de anuncios son recomendables para el usuario según lo que éste busca navegando por sus servicios.

Los sistemas de recomendación han ido evolucionando a causa de la necesidad de perfeccionar las recomendaciones, ya que un recomendador que hace mejores recomendaciones, implica un aumento significativo de los ingresos para muchas plataformas ya sea porque el SR (*Sistema de Recomendación*) se aplique en temas de publicidad, ventas, etc, o simplemente la mejora de imagen de la propia plataforma que la hacen más atractiva a nuevos usuarios. La evolución tecnológica ha hecho posible pasar de técnicas sencillas basadas en recomendar ítems con características similares, según atributos comunes, a recomendaciones avanzadas, basadas en técnicas de filtros colaborativos. En estas técnicas más avanzadas se aplican conjuntos de datos masivos permitiendo hacer análisis de datos con las interacciones específicas de un usuario, antes de generar la recomendación.

Dentro de estas técnicas existen también algunas técnicas de recomendación que usan métodos de clasificación y los adaptan al objetivo de una recomendación. Los métodos de clasificación se encargan de colocar elementos que no están aún en un lugar definido para situarlos en diferentes grupos existentes, es decir, los elementos que no tienen una clasificación previa, son asignados por un algoritmo en un grupo según características que se encuentren de este.

Estas formas de recomendar se explican con mayor detalle en las siguientes secciones de este capítulo.

3.2. Técnicas de recomendación informadas

Las técnicas de recomendación informadas engloban aquellos métodos de recomendación que no suponen un conocimiento global de información, es decir, métodos que solo se fijan en la información propia de un usuario respecto las interacciones con diferentes elementos, sin necesidad de conocer la del resto de usuarios.

Estas técnicas, utilizan propiedades que tienen los propios ítems con los que el usuario interactúa, para agruparlos en base a estas propiedades.

El ejemplo más entendible de las técnicas de recomendación informadas, es el caso de un recomendador basado en etiquetas y atributos. Este tipo de recomendador mira las interacciones que tiene un usuario con ciertos ítems, los cuales tienen ciertos atributos y el recomendador busca otros ítems con los mismos tipos de atributos para usarlos como opciones a recomendar.

Un recomendador basado en etiquetas y atributos necesita por lo tanto que los ítems sobre los que trabaja tengan internamente propiedades y etiquetas para poder ser clasificados de una manera directa por estas propiedades. De esta manera el recomendador hace uso de las interacciones que un usuario tiene con los distintos ítems, para así seleccionar los registros más

similares a los buscados Zheng et al. (2014). Esta similitud se obtiene de manera directa a través de propiedades que tengan en común los ítems.

Estos algoritmos también consideran qué tipos de ítems son con los que más interacúa el usuario, para buscar otros ítems de ese mismo tipo.

Los parámetros de configuración del algoritmo permiten ajustarse para generar recomendaciones más o menos refinadas según el caso práctico en el que éste corresponda ser aplicado.

Por ejemplo, el algoritmo puede ser modificado internamente para considerar casos en los que si el usuario empieza a tener interacciones con un nuevo tipo de ítem empiece a recomendar ítems de ese tipo, de tal forma que aunque la cantidad de interacciones anteriores del usuario sean de ítems con otros atributos, como en sus últimas interacciones están destacando otros tipos de ítems, el algoritmo considerará solo estos últimos.

Al igual que este tipo de ejemplo tiene esas consideraciones internas, se puede modificar internamente el algoritmo de muchas otras maneras para refinar otros casos.

A modo de ejemplo, vamos a suponer el caso en una plataforma de películas. Los ítems o el contenido de esta página se pueden considerar *"películas"*, con atributos en concreto como "Género", "año de publicación", "directores", etc. De esta forma, si un usuario comienza a ver películas con un género concreto, como puede ser "acción", un recomendador basado en etiquetas y atributos empezaría a recomendar películas de "acción".

Si el algoritmo se refina aún más, se pueden considerar también otras características, como usar adicionalmente el atributo "Director" priorizándolo, de manera que las nuevas recomendaciones mostrarán las películas de un director concreto que sean de "Acción".

También se pueden considerar las películas que tienen mejores puntuaciones dentro del género que el usuario ve con frecuencia.

Como se ha podido observar en el caso de uso, el recomendador puede aplicar muchas versiones de esta técnica, siempre basada en métodos directos donde las películas tienen una clasificación *directa* en base a puntos y atributos.

De esta forma un recomendador que usa las propiedades que los ítems ofrecen, se basa en estos métodos directos o *informados*, que no siempre pueden ser aplicados en todos los casos de uso, ya que no siempre los ítems tendrán atributos o serán suficiente para generar una recomendación adecuada. Así, este tipo de recomendadores son muy útiles y funcionan bien para casos donde los métodos informados se puedan aplicar, como es el ejemplo de una plataforma de películas cuyo contenido tenga internamente clasificaciones asignadas, sin embargo, el uso de estos métodos no pueden darse siempre, o no funcionan de manera tan precisa en plataformas cuyo contenido sea más difícil de catalogar.

3.3. Técnicas de recomendación no informadas

Al contrario que las técnicas anteriores, las técnicas de recomendación no informadas, no necesitan de un conocimiento previo y estático como punto de partida.

El punto de partida en este caso se forma a partir de las interacciones entre los usuarios con la aplicación, así como el resto de interacciones dentro del conjunto de todos los usuarios.

De estas interacciones se puede obtener información, aplicando conceptos de *big data* y análisis de datos, para conseguir agrupar perfiles de usuarios o perfiles de ítems. También se pueden aplicar otras muchas técnicas que tengan que ver con patrones para relacionar las interacciones que hace un usuario en particular, con las muchas interacciones que hacen el resto del conjunto de usuarios.

De esta manera, ahora, se tiene en cuenta la actividad del conjunto de usuarios a la hora de recomendar a un individuo en concreto.

Estas formas de recomendar son, a menudo, técnicas basadas *en filtros colaborativos*, de manera que a través de información colectiva ajena al individuo, se pueden encontrar patrones que nos ayuden a agrupar a este individuo. Estos patrones también sirven para contrastar con la información propia de cada usuario.

Dentro de las técnicas *no informadas* en los sistemas de recomendación, se han recopilado las más interesantes para este proyecto, y en alguno de los casos, las más exóticas. Vamos a ver con cierto detalle estas técnicas recopiladas en las siguientes subsecciones.

3.3.1. K-Nearest Neighbour

K-Nearest Neighbour es un método de clasificación supervisada, no paramétrico, que se encarga de estimar la probabilidad a posteriori de que un elemento x pertenezca a una clase Cj, a partir de información que saca de un conjunto de prototipos (Wikipedia (k-nearest neighbors)).

El algoritmo se aplica en ámbitos como reconocimiento de patrones, para clasificar objetos basándose en un entrenamiento mediante ejemplos cercanos a él en el espacio.

A modo de ejemplo gráfico para entender mejor el funcionamiento de *K-Nearest Neighbour* en la figura 3.2 se muestran elementos clasificados en dos grupos. Sobre estos dos grupos también se van a clasificar los elementos que llegan: elementos del grupo A y elementos del grupo B.

Estos elementos mostrados ya clasificados previamente en uno de esos dos grupos, se muestran a cierta distancia del nuevo elemento aún no clasificado representado como "??". La distancia a la que se encuentran implica la similitud de cada elemento ya clasificado respecto al elemento que se quiere

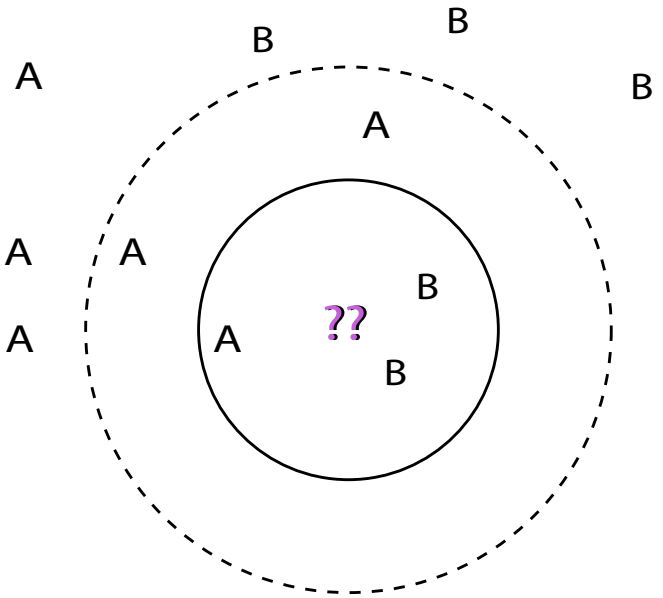

Figura 3.2: Ejemplo de clasificación de un elemento aplicando K-Nearest Neighbour.

clasificar.

Esta similitud se obtiene según características y propiedades comunes que tengan entre el elemento a clasificar y el resto. Estas características pueden ser etiquetas, interacciones, relaciones, propiedades o lo que se considere en cada caso práctico real.

En el ejemplo, se aplica un factor $K = 3$, es decir, se quieren buscar los tres vecinos más cercanos sobre el elemento "??", para clasificarlo en uno de los dos grupos. De esta forma, los tres vecinos más cercanos representan los que se encuentran en el área circular interna, con una distancia máxima que marca el tercer vecino más cercano. Esta distancia sirve también para ver los siguientes vecinos más cercanos calculando el doble de la distancia máxima obtenida de mis tres vecinos más cercanos, que en la figura se encuentran dentro del área circular marcada por una línea entrecortada.

De esta manera, se observa del nuevo elemento, que entre sus tres vecinos más cercanos, dos se encuentran clasificados como del grupo B y uno está clasificado en el grupo A, por lo que por similitudes se le clasifica al nuevo elemento como B, ya que es el tipo más repetido entre sus tres vecinos más cercanos.

Es importante también saber cómo encontrar el factor de K ya que la calidad de la recomendación se ve muy afectado por él. La selección de K puede obtenerse tras una evaluación con varios vecinos diferentes, para ver

el comportamiento de este según crece, de manera que se puedan encontrar en los resultados de evaluación, máximos y mínimos que interesen.

Es de destacar, el caso especial para $K = 1$, donde solo se utiliza el vecino más cercano, ya que el algoritmo de este caso es el llamado *Nearest Neightbor Algoritm*, predecesor o "padre" de *K-Nearest Neightbor*.

K-Nearest Neightbor tiene diferentes variaciones a la hora de hacer las clasificaciones, una de ellas destacable, considera el factor de la distancia para los K-Vecinos seleccionados.

Con este nuevo grado de libertad (el factor de la distancia) se puede pulir más la clasificación ya que para K elementos más cercanos, si n elementos son de un tipo A y m elementos de un tipo B[5], tal que $n < m$ y $n + m = K$, aplicando esta nueva variable a tener en cuenta, el nuevo elemento podría ser capaz de clasificarse en el tipo A si estos n vecinos tienen menores distancias respecto al nuevo elemento a clasificar, que los m vecinos del tipo B (Bellogín y de Vries (2013)).

K-Nearest Neighbour es también un algoritmo bastante ustilizado en sistemas de recomendación y rankings, ya que la idea conceptual es encontrar los vecinos más cercanos y usarlos para obtener información sobre el dato que se está midiendo (Hall et al. (2008)). En este tipo de sistemas, el algoritmo se modifica para obtener una lista ordenada basada en una clasificación, según la cantidad de vecinos que contengan esa similitud con el elemento de estudio en cuestión.

Este algoritmo es altamente modificable y adaptable a muchos casos, y tiene similitudes de concepto con uno de los SR realizados, del que se habla más profundamente en el capítulo (7).

3.3.2. Redes neuronales

Un recomendador basado en redes neuronales artificiales, en adelante mencionado también como RRNN, aplica un modelo de red neuronal que se entrena hasta empezar a obtener valores interesantes y suficientemente fiables para el objetivo que se tiene.

Este tipo de recomendador entra dentro del mundo del *deep learning*, es decir, algoritmos de inteligencia artificial subsimbólica que intentan modelar abstracciones de alto nivel en datos usando arquitecturas compuestas de transformaciones no lineales múltiples. Dicho de otra forma más coloquial, este tipo de inteligencia artificial subsimbólica, aplica un paradigma que no requiere de intervención humana previa pudiendo sacar los propios algoritmos conclusiones acerca de la semántica embebida de los datos (Wikipedia (Deep learning)).

De manera resumida, una red neuronal artificial es un modelo inspirado

[5] n y m representan una cantidad en el conjunto de los naturales, así como A y B, dos conjuntos tal que $A \neq B$

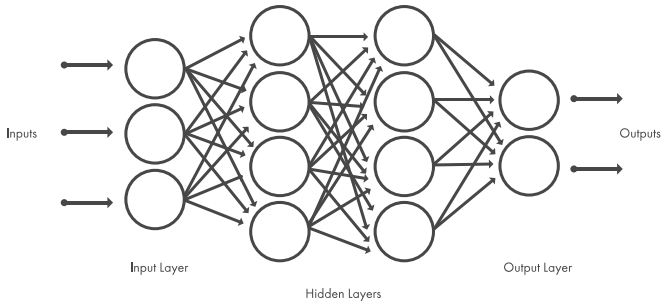

Figura 3.3: Ejemplo de red neuronal modelo perceptrón multicapa.

en el comportamiento de las neuronas de los seres vivos. Una neurona es una célula específica del sistema nervioso que desempeña una serie de intercambios de información con otras neuronas a través de las prolongaciones de su cuerpo. Estas prolongaciones son denominadas como dendritas (cortas y receptoras de información) y axón (larga y emisora de información).

Estos cambios de información entre neuronas provocan las respuestas ante estímulos externos y el auto aprendizaje del conjunto del sistema. De esta manera, en las RRNN (*Redes neuronales*), tenemos distintas capas con neuronas simuladas, que intercambian información entre ellas de diferente manera (entre capas o entre neuronas de la misma capa, de manera unidireccional o bidireccional). Estas capas son de entrada, ocultas y de salida, como se observa en la figura 3.3.

El modelo general de una neurona artificial, representa un dispositivo simple de cálculo que, a partir de un vector de entrada que viene del exterior o de otras neuronas, genera una única salida. Los elementos que contiene una neurona artificial son los siguientes:

- Conjunto de *entradas*.
- *Pesos sinápticos* que representan la fuerza de interacción entre las neuronas de las que recibe información.
- *Regla de propagación*, que proporciona el valor del potencial postsináptico de la neurona según sus pesos y sus entradas.
- *Función de activación*, que proporciona el estado de activación actual de la neurona en función de su estado anterior y de su potencial postsináptico actual.
- *Función de salida*, que proporciona la salida actual en función de su estado de activación.

Según la configuración de estas neuronas se introducen diferentes diseños de RRNN artificiales que no son objeto de este documento (Sanz Molina y Martin del Brio (2006)).

El intercambio de información que las neuronas hacen entre ellas acarrean la modificación de esa información, con técnicas de asignar pesos a las sinapsis (o conexiones entre neuronas), así como pesos internos de la propia neurona.

Estos pesos son los que hacen que una información de entrada se transforme en una información de salida esperada, al pasar por diversas neuronas.

Para calcular los pesos ideales para que la red neuronal comience a dar resultados correctos, ésta debe pasar por una fase de entrenamiento donde se la puede dirigir para que decida si la salida que genera está cerca de lo esperado o, por el contrario, se aleja.

En esta fase de entrenamiento se aplican algoritmos de corrección de pesos según la salida y el resultado esperado, para perfeccionar poco a poco la red neuronal hasta minimizar la tasa de fallos.

Las variables de entrada y salida en una red neuronal, pueden ser binarias (digitales), o continuas(analógicas), dependiendo del modelo y aplicación. Por ejemplo, un perceptrón multicapa, como el mostrado en la figura 3.3 admite ambos tipos de señales. Así, para tareas de clasificación o recomendación poseería salidas digitales.

De esta manera se pueden aplicar RRNN cuyos parámetros de entrada sean los elementos con los que un usuario ha interactuado, representándolos de manera binaria, y como salida devolver ítems recomendados. Para ello debe elegirse una configuración correcta de neuronas, capas y modelo de comunicación entre neuronas, además de la forma de entrenar y los algoritmos de corrección que va a aplicar la red neuronal durante el entrenamiento.

Con una buena selección de parámetros una red neuronal puede aprender y retroalimentarse, perfeccionando sus recomendaciones y minimizando la función de error.

Una ventaja de las RRNN es el aprendizaje que tienen, que al igual que la biología de donde son copiadas, se usan con la intención de "enseñarlas" a recomendar para que lo hagan sin necesidad de decirles cómo deben hacerlo. Además, la respuesta de las neuronas biológicas es de tipo *no lineal*, característica que es emulada en las RRNN ya que es el tratamiento de problemas altamente no lineales donde más destacan las aplicaciones de una RRNN.

Por otro lado, las redes neuronales no siempre son ventajosas, ya que para tareas complejas y grandes requieren un entrenamiento alto y costoso. En muchas ocasiones existen otra serie de algoritmos más eficientes que no requieren de ese coste, y pueden sustituirse por las redes neuronales ya que mantienen un indice de eficacia similar en cuanto a las respuestas generadas.

3.3.3. Recomendador basado en itinerarios de aprendizaje

Un recomendador basado en itinerarios de aprendizaje comprueba qué línea ha seguido un usuario a la hora de elegir ítems y analiza el resto de líneas seguidas por otros usuarios para encontrar las más similares y generar una recomendación basada en ítems que seguirían ese recorrido.

Estos algoritmos se pueden usar en plataformas donde los usuarios tienen intención de seguir un itinerario de aprendizaje, como es el caso de los *jueces en línea* (capítulo 4), donde existen ejercicios o problemas a resolver. En este caso se recomiendan los problemas más interesantes para la línea de aprendizaje del individuo.

De esta manera, se puede considerar que el recomendador solo se fija en las líneas más avanzadas que existen de usuarios que han seguido por la misma trayectoria del individuo a recomendar, para así poder recomendar líneas más precisas (Sánchez-Ruiz et al. (2017)).

A modo de ejemplo, en el caso de *jueces en línea*, este recomendador genera un itinerario de aprendizaje por cada usuario en base a los problemas que ha ido consiguiendo realizar de forma satisfactoria.

Estas secuencias de problemas que se les asigna a cada usuario, mantienen un orden que viene dado por la fecha desde el primer problema resuelto que tienen, hasta el último.

Dado un usuario U_i, al que se le asigna un itinerario L_i, para generar la recomendación, se aplica una consulta en base al L_i asociado a U_i. De esta consulta se obtiene una tupla indicando la similitud entre cada itinerario del resto de usuarios, con L_i, obteniendo los N más similares.

Utilizando estos nuevos itinerarios, el recomendador puede buscar problemas que no ha resuelto el individuo a ser recomendado, y clasificarlos de diferentes maneras, para generar la recomendación.

3.3.4. Recomendador basado en análisis de redes sociales

Un recomendador que aplica técnicas usadas en análisis de redes sociales, trata a los usuarios y a los ítems como un grafo de individuos y las relaciones entre ellos. De esta forma, por ejemplo, en el caso de un sistema de *jueces en línea*, los nodos representan problemas y usuarios, y las entregas, las relaciones que existen entre ellos.

Esta representación de la red de usuarios y problemas, permite usar métodos y métricas definidas en el campo de análisis de redes sociales.

Como se muestra en la figura 3.4, de este grafo se pueden obtener por separado dos grafos donde se tienen las relaciones entre usuarios y las relaciones entre problemas, de tal forma que del grafo bipartito de usuarios más problemas tratados como nodos se obtienen dos proyecciones, una que solo mantiene como nodo los usuarios y sus relaciones entre sí, y otra que solo mantiene a los problemas y sus relaciones entre sí (Furht (2010)).

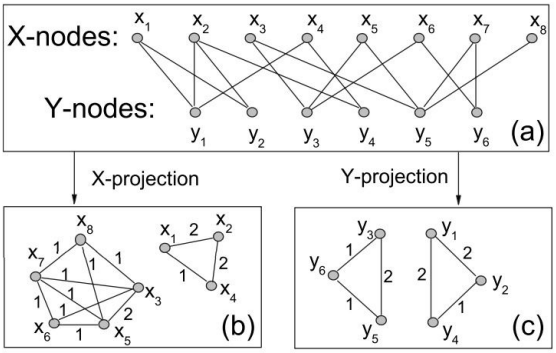

Figura 3.4: Ilustración de un grafo bipartito(a), donde se extraen las proyecciones de los nodos X (b) e Y(c).

Utilizando estas proyecciones, se pueden aplicar algoritmos de "*link prediction*". Estos algoritmos se usan para hacer predicciones de nuevos enlaces entre nodos, que puedan surgir en un futuro, o para hacer predicciones de enlaces entre nodos, que puedan desaparecer en un futuro (Jimenez-Diaz et al. (2017)).

3.3.5. Recomendador Bayesiano

Un recomendador bayesiano es un SR que aplica cálculos probabilísticos basados en el *teorema de Bayes* de la probabilidad condicionada, con el objetivo de obtener la probabilidad de que un ítem sea del agrado de un usuario. Una vez obtenidas todas las probabilidades se ordenan de mayor a menor, para así poder recomendar al usuario aquellos ítems que mejor encajen con su perfil.

La teoría básica de la probabilidad explica que la probabilidad de que ocurra un suceso se calcula dividiendo el número de casos favorables entre el número de casos posibles (sección 3.1). La probabilidad de que ocurra el suceso A se expresa como $P(A)$. Así en un dado de seis caras no cargado, la probabilidad de que salga el número 4 sería un sexto ya que solo una cara, la del cuatro, es favorable y hay 6 posibles caras equiprobables.

$$P(A) = \frac{n^o favorables}{n^o posibles} \tag{3.1}$$

Si ahora se tienen dos dados diferenciados, uno azul y otro rojo, tenemos un total de 36 casos. Entonces si queremos calcular la probabilidad de que tirando los dos dados salga en el azul un 4 y en el rojo un 6, seguimos teniendo un único caso favorable de los 36 posibles, lo que coincide con la probabilidad de que salga un 4 en el azul multiplicado por la probabilidad

de que salga un 6 en el rojo. Si nos abstraemos de casos concretos se puede demostrar que la probabilidad de que ocurran dos sucesos independientes simultáneamente es la multiplicación de la probabilidad de que ocurra cada uno por separado (fórmula 3.2). Cuando escribimos $P(A, B)$ nos referimos a la probabilidad de que ocurran los sucesos A y B a la vez,

$$P(A, B) = P(A) \cdot P(B) \tag{3.2}$$

Si ahora establecemos que sabemos que en el dado azul ha salido el 4, la probabilidad de que ocurran ambos sucesos será la multiplicación de la probabilidad que salga 6 en el rojo condicionado a que ha salido 4 en el azul, por la probabilidad de que ocurra 4 en el azul (3.3). La probabilidad de que ocurra un suceso A condicionado a que ha ocurrido B se denota como $P(A|B)$.

$$P(A, B) = P(A|B) \cdot P(B) \tag{3.3}$$

Y como que ocurran ambos sucesos simultáneamente es independiente del orden podemos establecer la igualdad que se muestra en la fórmula 3.4. Sustituyendo $P(A, B)$ por la parte derecha de la igualdad de la fórmula 3.3 y análogamente $P(B, A)$, y despejando $P(A|B)$ obtenemos la fórmula 3.5. Esta última fórmula es lo que se conoce como *teorema de Bayes*.

$$P(A, B) = P(B, A) \tag{3.4}$$

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{3.5}$$

Extendiendo el teorema, supongamos que sabemos que varios sucesos han ocurrido y queremos saber la probabilidad de que ocurra otro suceso. Para ello, en la fórmula 3.5, sustituimos donde pone B por la historia de sucesos que han ocurrido y obtenemos la formula 3.6. La historia de sucesos la escribiremos como $B_0, ..., B_n$. El elemento $P(B_0, ..., B_n|A)$ de la fórmula se refiere a la probabilidad de que ocurra la historia de sucesos en la hipótesis de que sabemos que ha ocurrido el suceso A.

$$P(A|B_0, ..., B_n) = \frac{P(B_0, ..., B_n|A) \cdot P(A)}{P(B_0, ..., B_n)} \tag{3.6}$$

Para aplicar el *teorema de Bayes* será necesario calcular los elementos de la parte derecha de la ecuación 3.6. Calcular $P(A)$, la probabilidad a priori de cada uno de los ítems a recomendar, suele ser un cálculo trivial. Sin embargo, $P(B_0, ..., B_n|A)$ no lo es tanto. Por ello se suele aplicar una aproximación naïve, asumiendo que todos los sucesos B_0 hasta B_n son independientes entre sí. Con lo que aplicando la fórmula 3.2, la expresión quedaría como $\prod_{i=0}^{n} P(B_i|A)$. En lenguaje natural es la multiplicación de las probabilidades

B_i condicionadas a que ha ocurrido A, donde i toma valores desde 0 hasta n, y B_i representa cada ítem de la historia de sucesos. Así mismo, esta asunción de independencia de sucesos hace que, aplicando una extensión de la fórmula 3.2, se transforme también en una multiplicación de probabilidades, quedando la fórmula final como en 3.7:.

$$P(A|B_0, ..., B_n) = \frac{P(A) \cdot \prod_{i=0}^{n} P(B_i|A)}{\prod_{i=0}^{n} P(B_i)} \tag{3.7}$$

Estas probabilidades $P(B_i|A)$ pueden calcularse como el número de veces que ocurren B_i y A a la vez, dividido por el número de veces que ocurre A independientemente de que haya ocurrido B_i.

$$P(B_i|A) = \frac{P(B_i, A)}{P(A)} \tag{3.8}$$

Un problema que suele aparecer es que $P(B_i|A)$ sea 0 para algún i, lo que provoca que el resultado final de aplicar la fórmula anterior sea siempre 0. La razón de que esto ocurra normalmente es porque no hay suficiente información sobre el dominio para determinar el valor de $P(B_i|A)$. Si se diera el caso, se podría estimar un valor de partida para dicha expresión.

Una forma de hacerlo es aplicar la estimación de *Laplace* (Clark y Boswell (1991)). Se propone que para solucionar la ausencia de suficientes datos se asuma que existe un elemento de cada miembro del dominio. Es decir, si para A existen dos estados, por ejemplo, A es un vaso que puede estar lleno o estar vacío, entonces tendríamos que asumir que existen al menos dos vasos uno lleno y otro vacío. Si aplicamos esta estimación a la fórmula 3.8 nos quedaría la fórmula 3.9, donde x representa el número de estados para A, en este ejemplo 2. En un caso con más estados, por ejemplo A es una bola quede ser de color verde, rojo, azul, blanco o negro, pues como hay 5 colores x tomaría el valor 5, asumiendo que existe al menos una bola de cada color.

$$P(B_i|A) = \frac{P(B_i, A) + 1}{P(A) + x} \tag{3.9}$$

Aunque la estimación ayuda siempre que tengamos casos en que $P(B_i, A)$ valga 0, se ha de aplicar a todos los casos incluso aquellos en los que $P(B_i, A)$ valga 1. De esta manera aunque las probabilidades que den 0 o valores cercanos a 0 se beneficien de la estimación, las que no también se beneficiarán aunque en menor medida.

Capítulo 4

Jueces en línea

Resumen: En este capítulo se explicará qué son los jueces en línea mencionando algunos ejemplos. También se entrará a fondo en la explicación sobre el juez en línea de *¡Acepta el reto!* y también en otro juez en línea que además tiene incorporado un SR.

4.1. ¿Qué son los jueces en línea?

Los jueces en línea son repositorios de problemas con un evaluador de soluciones para comprobar automáticamente si un programa es solución a un problema dado. Para ello el sistema compila el código del programa enviado, lo ejecuta con una serie de casos de prueba como entrada y comprueba si las salidas obtenidas coinciden con las que el sistema tiene asociadas como correctas. Además suelen medir parámetros extra, como pueden ser el tiempo de ejecución y la memoria utilizada. Una vez hecho esto, el juez proporciona un veredicto. Dos ejemplos de jueces en línea son *UVa Online Judge*[1] y *¡Acepta el reto!*[2].

Estos sistemas se utilizan principalmente en el ámbito de los concursos de programación, así como para practicar para estos, aunque también se utilizan en el ámbito académico para poner en práctica los conocimientos adquiridos en los aulas.

Algunos de ellos, como el *UVa Online Judge*, además de ser repositorios de problemas son capaces de alojar concursos, fomentando así la competitividad y el desarrollo de las habilidades de los usuarios.

En ocasiones, el repositorio de problemas añade etiquetas a los problemas para clasificarlos según los conceptos de programación necesarios para resolverlos, por la temática del enunciado, por su aparición en concursos, etc.

[1] `https://uva.onlinejudge.org`
[2] `https://www.aceptaelreto.com`

Es una manera de facilitar a los usuarios encontrar problemas que puedan ajustarse a lo que están buscando.

Por lo tanto, en estas plataformas existen usuarios que seleccionan problemas de un extenso repositorio y realizan envíos de código para solucionarlos. Se pueden establecer entonces relaciones entre usuarios en función de los envíos.

Los jueces en línea se hacen cada vez más populares gracias a los concursos de programación que promocionan empresas como Google con el *Hashcode*[3], o como la compañía de móvil Tuenti con el *Tuenti Challenge*[4].

4.2. El juez en línea *¡Acepta el reto!*

¡Acepta el reto! es un juez en línea desarrollado por algunos profesores de la Facultad de Informática de la Universidad Complutense de Madrid. Además de ser un extenso repositorio de problemas, *¡Acepta el reto!* es un corrector automático, es decir, se puede subir el código creado como solución a un problema del repositorio para que *¡Acepta el reto!* lo compile, ejecute y proporcione un veredicto.

Los veredictos de *¡Acepta el reto!* van más allá de un simple Bien o Mal, es más, existen hasta once veredictos distintos, de los cuales dos no dependen del resultado del problema, sino que son por errores internos o porque aún no se ha evaluado la solución, y uno es un rechazo del código por motivos de seguridad. Los otros ocho son los siguientes:

- *Accepted (AC)*: Aceptado. Es el veredicto más deseado, pues significa que la solución es correcta.
- *Presentation error (PE)*: Error de presentación. Se produce cuando la solución al problema es correcta, pero al mostrar la solución, los saltos de línea o espacios en blanco difieren de los que deberían ser.
- *Wrong Answer (WA)*: Respuesta incorrecta. El programa no produce las salidas esperadas, por tanto no es solución al problema dado.
- *Compilation Error (CE)*: Error de compilación. El sistema no ha podido compilar el código porque tiene errores.
- *Run Time Error (RTE)*: Error durante la ejecución. Se produce cuando la solución al ser ejecutada produce algún tipo de excepción, como divisiones por cero o índices fuera de rango.
- *Time Limit Exceeded (TLE)*: Tiempo límite superado. Se produce cuando la solución dada tarda en ejecutarse más tiempo del esperado, por ejemplo por culpa de bucles infinitos o soluciones no optimizadas.

[3] `https://codingcompetitions.withgoogle.com/hashcode`
[4] `https://contest.tuenti.net`

- *Memory Limit Exceeded (MLE)*: Memoria límite superada. Se produce cuando la solución al problema pide más memoria de la permitida.
- *Output Limit Exceeded (OLE)*: Salida límite superada. Se produce cuando el programa solución genera una salida más amplia de la permitida.

Cuando un usuario realiza un envío y recibe una respuesta errónea es habitual que revise el código y en poco tiempo realice un nuevo envío con la esperanza de obtener el veredicto *AC (*Accepted*)*. Esto no quiere decir que cuando un usuario ya tiene un *AC* en un problema no pueda realizar nuevos envíos. *¡Acepta el reto!* cuenta con un ranking para cada problema de modo que las soluciones más óptimas se encuentren en posiciones más elevadas. Por ello, a pesar de que un usuario haya recibido ya un *AC* para un problema no es de extrañar que por competitividad optimice su solución y realice otro envío para subir en el ranking.

Con esto ahora se pueden categorizar los problemas para un usuario en tres estados:

- Resueltos: el usuario ha realizado uno o varios envíos y al menos en uno de ellos ha obtenido *AC*.
- Intentados: el usuario ha realizado uno o varios envíos, pero en ninguno de ellos ha obtenido como veredicto un *AC*.
- No intentados: el usuario no ha realizado ningún envío, ya sea porque el problema le parece muy complicado o porque no lo ha leído todavía.

Por otro lado, *¡Acepta el reto!* incluye una categorización en sus problemas, una ventaja tanto para algunos modelos de recomendación, que se basan en etiquetas y métodos directos, como también para los usuarios, que pueden marcarse itinerarios en base a las categorías para practicar una rama específica.

El juez de *¡Acepta el reto!* es capaz de compilar soluciones en diferentes lenguajes de programación (C, C++ y Java) y ofrecer un conjunto limitado de uso de librerías estándar para limitar a los usuarios el uso de recursos para que sean ellos los que tengan que diseñarlos al resolver los problemas, así como evitar que el usuario pueda ejecutar código que no se quiere que se emplee para la resolución de los ejercicios.

En la sección de preguntas frecuentes de *¡Acepta el reto!* se encuentra con más detalle este tipo de información además de información extra, entre la que incluye el comportamiento del Juez, veredictos, las librerías aceptadas, y más información útil tanto para los usuarios para aquel que quiera entender con mayor profundidad ciertos aspectos del juez que aquí se hayan explicado sin entrar al detalle.

Se puede observar la apariecia de *¡Acepta el reto!* en la figura 4.1

Figura 4.1: Página de inicio de *¡Acepta el reto!*.

4.3. El juez en línea *Kattis*

Kattis[5] es un juez online que al igual que *¡Acepta el reto!* tiene un amplio repositorio de problemas con su evaluador correspondiente, y que al igual que *UVa Online Judge* es capaz de gestionar concursos. Además incluye un SR que recomienda varios problemas agrupados en cuatro categorías en función de su dificultad: *trivial*, *easy*, *medium* y *hard* (Revilla et al. (2008)).

Esta dificultad no está preestablecida, sino que por el contrario varía en función del número de envíos correctos y totales. Para calcular dicha dificultad utiliza una variante de *ELO rating system* (Wikipedia (Elo rating system)), que, resumidamente, lo que hace es dar menor dificultad a los problemas que han sido resueltos por muchos usuarios en pocos envíos, y mayor dificultad a aquellos que han sido resueltos por pocos pero intentados por muchos. Aquellos problemas que tienen pocos envíos se clasificarían con dificultad media, ya que no se dispone de información suficiente para determinarla.

Además este sistema también evalúa al usuario, teniendo en cuenta los problemas que ha resuelto, su puntuación total y la precisión de sus envíos. De esta manera para un usuario nuevo el problema que se le recomienda como difícil puede ser para otro usuario recomendado como fácil, obteniendo así una recomendación más acorde con el nivel del usuario.

Se puede observar la apariencia de *Kattis* en la figura 4.2

[5] `https://open.kattis.com/`

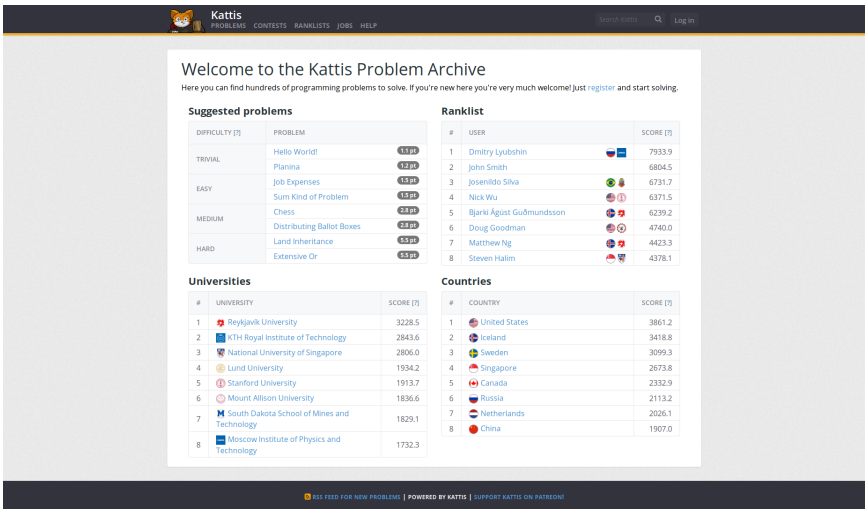

Figura 4.2: Página de inicio de *Kattis*.

Capítulo 5

Evaluación de recomendadores

Resumen: En este capítulo se introducen los métodos de entrenamiento y evaluación existentes a la hora de comprobar la eficacia de un recomendador. El capítulo también trata en profundidad del método de evaluación seleccionado para los recomendadores implementados.

Una vez que se ha aplicado cualquier técnica de recomendación, surge la pregunta de si dicha recomendación ha sido acertada. Para ello existen diferentes métodos de evaluación que ayudan a comprobarlo.

Tradicionalmente los sistemas de recomendación han sido evaluados con experimentos sobre un conjunto de interacciones, para así poder estimar el error en la predicción de las recomendaciones (Aggarwal (2016)). Es decir, se quiere comprobar si el recomendador piensa como el usuario, dado que la intención de la recomendación será guiar al usuario por donde este mismo se encaminaría.

5.1. Métricas de un recomendador

En el ámbito de la evaluación de los SR, o predictores, existen multitud de métricas y variantes que permiten dar cierto juego para entender diferentes aspectos del funcionamiento de estos. Las métricas permiten analizar ciertas relaciones de estos casos de test, donde un conjunto de recomendaciones puede haber sido correcto, incorrecto, o resultar inútil. Gracias a las métricas aplicadas se puede obtener con cierta fiabilidad el éxito de un recomendador durante la fase de pruebas o en su ciclo de vida de uso (Herlocker et al. (2004)).

De esta manera, se puede llevar a implementar un SR conociendo de antemano el comportamiento que vaya a tener durante su ciclo de utilización, y así hacer las estimaciones necesarias de los beneficios que acarreen sobre la plataforma en la que el SR vaya a ser lanzada.

De todas las métricas existentes, se han seleccionado 4, que son las más destacables y usadas para evaluar en sistemas de recomendación y predictores. Estas métricas se obtienen a través de una tabla de estados que representan diferentes situaciones que se pueden dar en una predicción (5.1).

Los estados de predicción representan si una recomendación ha tenido éxito o no, de manera que se categorizan los resultados de cada recomendación, para así sacar luego diferentes métricas.

Estos estados de predicción son los siguientes:

- TP (*True Positive*): Indica que la recomendación propuesta ha sido correcta ya que se ha dado en la realidad. Para el caso de recomendadores y jueces en línea, representa las situaciones en los que el usuario ha hecho tras el corte los problemas que el recomendador le ha recomendado en la fase de entrenamiento.

- FP (*False Positive*): Indica que la recomendación propuesta ha sido incorrecta ya que no se ha dado en la realidad. Para el caso de recomendadores y jueces en línea, representa, por contraposición a TP (*True Positive*), las situaciones en los que el usuario *no* ha hecho tras el corte uno de los problemas que el recomendador ha recomendado.

- TN (*True Negative*): Indica que la recomendación no se ha propuesto y no se ha dado en la realidad por lo que ha sido una omisión correcta. Para el caso de recomendadores y jueces en línea, representa las situaciones en los que el usuario *no* ha hecho tras el corte uno de los problemas que el usuario *no* ha recomendado. Para nuestras métricas no va a ser necesario utilizar los TN (*True Negative*) por lo que no se han calculado.

- FN (*False Negative*): Indica que la recomendación no se ha propuesto y se ha dado en la realidad por lo que ha sido una omisión incorrecta. Para el caso de recomendadores y jueces en línea, representa las situaciones en los que el usuario ha hecho tras el corte uno de los problemas que el recomendador *no* ha recomendado.

Con estos estados que representan una "tabla booleana" como se puede observar en la figura 5.1, se clasifica el resultado de cada recomendación en uno de estos cuatro estados mostrados, haciendo un sumatorio del total de cada estado, para así calcular después las métricas de evaluación.

Teniendo la cantidad total de los estados TP, TN y FN representados en la figura 5.1, se pueden sacar de ahí las siguientes métricas:

- *Precision*: La precisión viene dada por la siguiente fórmula:

$$precision = \frac{total.TP}{total.TP + total.FP} \tag{5.1}$$

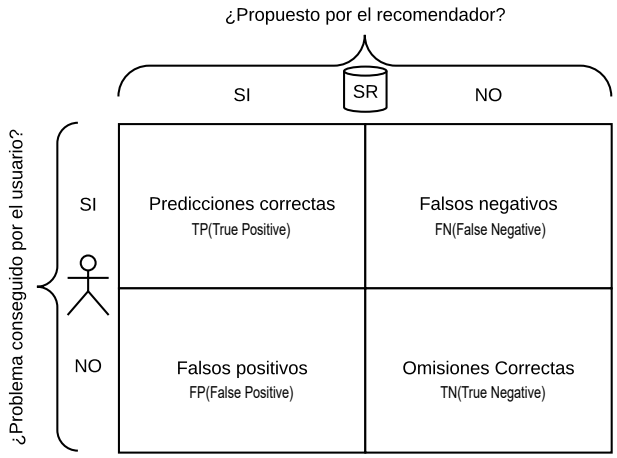

Figura 5.1: Estados de predicción de un problema durante el entrenamiento.

La ecuación anterior viene a mostrar qué fracción corresponde a los true positive para el conjunto donde el recomendador ha obtenido "éxito" o "fallo" de recomendación, obteniendo un "porcentaje" sobre 1 que indica una tasa de ese éxito. Dicho de otra manera, la precisión representa que cuando nuestro modelo predice un problema, acierta el *precision* por ciento de las veces (sobre 1).

Lo que se espera entonces de la precisión es que empiece en un valor alto y que a medida que se recomiendan más ítems vaya disminuyendo progresivamente. Esto es porque lo esperado es que los primeros ítems recomendados sean de mayor interés para el usuario y el resto sean cada vez de menor interés.

- *Recall*: Se define el recall con la siguiente expresión:

$$recall = \frac{total.TP}{total.TP + total.FN} \tag{5.2}$$

Esta expresión muestra la fracción de los éxitos sobre el conjunto total de los TP y los FN, que representan los éxitos mas los casos en los que el usuario ha preferido ítems que no se han recomendado. Este concepto es un poco más difícil de entender respecto a la precisión. Dicho de otra manera el *recall* o "exhaustividad" en español, intenta responder a la pregunta de "¿Qué proporción de positivos reales se calculó correctamente?".

En consecuencia, para nuestro caso a medida que se van recomendando más problemas, aquellos que eran resueltos pero no recomendados pueden ir pasando a resueltos y recomendados. Por ello, si un usuario

ha resuelto muchos más problemas de los que se le están recomendado se obtendrán valores de *recall* bastante bajos.

- *F1-Score*: Se define el *F1-Score* o *F-Score* mediante la siguiente expresión.

$$F.Score = 2 \times \frac{Recall \times Precision}{Recall + Precision} \tag{5.3}$$

  De esta manera, con el *F-Score* se mide la precisión que tiene un test, en este caso, el entrenamiento/evaluación de los recomendadores.

- *One-Hit*: El *One-Hit* sigue la misma fórmula que *precision*, pero los TP asociados se obtienen considerando los casos en los que el usuario ha preferido al menos uno de los ítems que el recomendador le ha propuesto en la fase de entrenamiento.

  Así, el *One-Hit* obtendrá los mismos valores que la precisión para N=1 ítmes propuestos[1], pero mejorará los valores para N>1, aproximándose rápidamente al 100 por ciento a medida que N incrementa, ya que es más fácil encontrar en un conjunto de ítems propuestos más grande, al menos un ítem que el usuario haya resuelto.

5.2. Evaluación de nuestros recomendadores

Para la fase de evaluación de los recomendadores seleccionados en este proyecto, se han decidido aplicar la siguientes técnicas explicadas a continuación.

Partiendo de una BD (*Base de datos*) anonimizada que representa un historial del juez *¡Acepta el reto!*, se la hace un tipo de corte. De este corte, la primera parte servirá para entrenar a ambos recomendadores y generar unos resultados de recomendación sobre esa información de entrenamiento, para después, contrastar esos resultados con la segunda parte del corte.

Es evidente que los resultados obtenidos variarán según qué consideraciones o filtros se utilicen al hacer el corte. De esta manera podemos seleccionar un corte por fecha, por usuarios... O podremos excluir casos innecesarios para el entrenamiento como aquellos usuarios que no tengan problemas resueltos o una mínima cantidad de problemas, ya que supone una información más fiable.

De esta manera, el corte seleccionado para evaluar estos recomendadores ha sido por fecha, dividiendo en dos partes la BD, donde la parte de entrenamiento será la BD desde su comienzo hasta 1 año previo a la copia con la que se está trabajando, y la parte usada para la evaluación será desde 1

[1] Se considera N como el cardinal del conjunto de ítems que devuelve el recomendador en su recomendación.

año previo a la versión actual de la copia, hasta su momento actual, es decir, hasta el último registro que esta tiene.

Para el entrenamiento no se ha considerado excluir ningún usuario a la hora de recomendar, pero en la evaluación solo se tendrán en cuenta aquellos usuarios que han realizado con éxito tras el corte, al menos un problema.

A la hora de calcular las diferentes métricas explicadas en la sección anterior, es importante tener en cuenta el factor comentado anteriormente, donde decíamos que se han tenido en consideración aquellos usuarios con al menos un problema resuelto a posteriori, excluyendo los que no han resuelto ninguno tras el corte. Para la evaluación se han considerado obtener las métricas para diferentes cardinales del conjunto de los mejores problemas que se han obtenido en la recomendación. De esta manera podemos ver cómo actúan las métricas para los N mejores problemas, desde $N = 1$ (el mejor problema) hasta $N = 10$ (los 10 mejores problemas).

Sacar la precisión de los N mejores problemas, considerando aquellos usuarios que han resuelto al menos un problema, provoca en parte una pérdida de precisión ya que tal como se ha medido, no se han descartado aquellos usuarios que han resuelto M problemas siendo $M < N$, haciendo que esos $N - M$ restantes problemas recomendados, se puedan tomar como "malos aciertos" y baje la precisión. Esto se podrá comprobar en las gráficas resultantes de las secciones 6.2 y 7.4 de los posteriores capítulos. Es importante por lo tanto esto, ya que varias veces se va a observar como conclusión que las métricas son objeto sobre todo para, bajo las mismas circunstancias de evaluación y entrenamiento entre recomendadores diferentes, poder usar estos datos como una forma de *compararlos* entre ellos, o de analizar el propio recomendador bajo la perspectiva de ver bajo qué circunstancias funcionan mejor, pero siempre serán datos que no sirven para evaluar de manera independiente de una manera exacta.

Cabe destacar de antemano, que este método de evaluación no es el más indicado, ya que el usuario no se ve directamente influido por el recomendador a la hora de elegir problemas a posteriori, por lo que muchas veces quizás no han elegido el problema que más les conviene por desconocimiento de este mismo.

Una evaluación más correcta y fiable sería bajo un caso de uso real durante un periodo de tiempo, que permita comprobar que su funcionamiento es correcto si el usuario por un lado intenta lo que recomienda el SR, y una vez intentado, además logra resolverlo.

Los resultados de entrenamiento de los recomendadores implementados se pueden ver en las secciones 6.2 y 7.4, respectivamente, así como en el capítulo 8 de comparación de resultados 8.

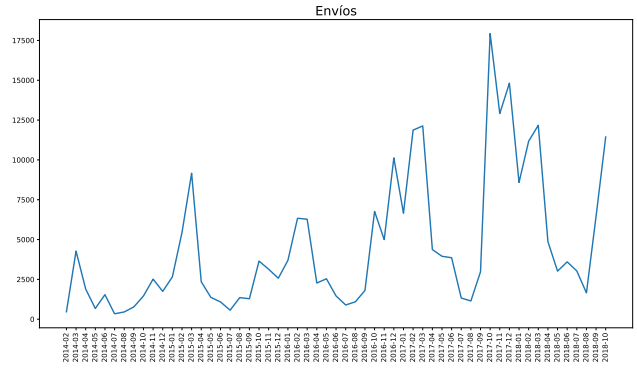

Figura 5.2: Gráfica de envíos desde la creación de *¡Acepta el reto!*.

5.2.1. Análisis del número de envíos

Esta sección trata de analizar los datos con los que se va a realizar la evaluación de nuestros recomendadores, para así poder entender de antemano posibles situaciones que se puedan encontrar en las recomendaciones durante la fase de entrenamiento y tenerlo en cuenta a la hora de realizar el análisis de los resultados.

En la gráfica de la figura 5.2, se muestra el número de envíos realizados en *¡Acepta el reto!* desde su puesta en marcha el 17-02-2014 hasta el 23-10-2018. Fecha hasta la cual tenemos datos para la realización de este proyecto. Se han realizado un total de 258.967 envíos, donde el 50 % de estos se realizó antes de finales de marzo de 2017, lo que significa que el otro 50 % realizados posteriormente se ha conseguido en la mitad de tiempo. El gran aumento del número de envíos se debe a que con el tiempo *¡Acepta el reto!* se ha popularizado más y más, sobre todo en el ámbito de la Facultad de Informática de la Universidad Complutense de Madrid, ya que los profesores que lo crearon han sabido llamar la atención de los alumnos mediante concursos de programación, y la de otros profesores proponiéndolo como una herramienta para que los alumnos puedan practicar los conocimientos adquiridos en clase.

Se observa en la gráfica que existen varios picos. Se repite siempre un pico alrededor de marzo y otro en diciembre. Es probable que el número de envíos aumente en esas fechas debido a tres concursos de programación organizados por los fundadores, *AdaByron*, *ProgramaMe* y *Las 12 UVas*. El pico que se suele repetir en octubre podría deberse a que tras un mes desde el inicio de curso en septiembre, los profesores proponen problemas a sus alumnos y, como es habitual, la mayoría empieza a resolverlos y luego gran parte de ellos abandonan la plataforma o disminuyen su actividad.

Capítulo 6

Recomendador Bayesiano

Resumen: Este capítulo describe la implementación de un recomendador probabilístico que aplica el teorema de Bayes, la evaluación de dicho recomendador en función de unas métricas y el análisis de los resultados obtenidos con dichas métricas.

En los capítulos anteriores se han descrito distintos tipos de SR, el juez online de *¡Acepta el reto!* y diferentes métricas que nos ayuden a comprobar si un recomendador está dando buenos resultados. En este, pasamos a detallar el modo en el que se han usado las ideas de los recomendadores bayesianos para aplicar este tipo de recomendación en *¡Acepta el reto!*.

Un recomendador bayesiano es un SR que utiliza el *teorema de Bayes* para averiguar la probabilidad que un elemento le guste a un usuario, basándose en el perfil de este. Obteniendo las probabilidades de todos los elementos se ordenan de mayor a menor probabilidad para recomendarle los mejores.

Dicho perfil del usuario se puede extraer del historial de envíos realizados en *¡Acepta el reto!*, ya que se nos ha proporcionado una base de datos que contiene esta información. A cada problema se le establecerá un estado para cada usuario.

Hemos considerado dos variantes a la hora de establecer los estados. Una de ellas tomará como estados posibles para un problema: resuelto, intentado (pero no resuelto) y no intentado. La otra variante contemplará solo dos estados: resuelto y no resuelto (la unión de los intentados y los no intentados). Estos estados están descritos en la sección 4.2.

Dentro de cada una de estas dos variantes se ha decidido realizar una aproximación aplicando la estimación de *Laplace* y otra aplicando una estimación arbitraria. En consecuencia obtendremos cuatro versiones de recomendadores bayesianos, las cuales se analizarán y se decidirá cual ha sido mejor para el caso concreto de *¡Acepta el reto!*.

6.1. Implementación

Como se recordará de la sección 3.3.5, los recomendadores bayesianos se basan en el *teorema de Bayes* cuya fórmula, repetida aquí para mayor claridad, es:

$$P(A|B_0, ..., B_n) = \frac{P(B_0, ..., B_n|A) \cdot P(A)}{P(B_0, ..., B_n)} \tag{6.1}$$

También recordamos que el cálculo de $P(B_0, ..., B_n|A)$ de la fórmula anterior es bastante complejo. Por este motivo se ha tomado la decisión de asumir que los sucesos $B_0, ..., B_n$ son independientes entre sí. En nuestro contexto, asumimos que la probabilidad de que un usuario resuelva un problema X es independiente de que resuelva el problema Y. Esto no es del todo cierto ya que problemas parecidos tendrán mayor probabilidad de estar en el mismo estado.

Por tanto, al final la fórmula que usaremos, también repetida por comodidad, es:

$$P(A|B_0, ..., B_n) = \frac{P(A) \cdot \prod_{i=0}^{n} P(B_i|A)}{\prod_{i=0}^{n} P(B_i)} \tag{6.2}$$

En el caso de nuestro recomendador:

- $B_0, ..., B_n$ es la información conocida del *perfil del usuario*. Más concretamente, B_i es el estado de resolución de cada problema del sistema para el usuario al que estamos recomendando.
- A es el suceso cuya probabilidad queremos conocer, en particular la probabilidad de que el usuario con ese perfil sea capaz de resolver el problema A.

En relación a los B_i, como se contó en la sección 4.2, un problema puede estar, para un usuario, en los siguientes tres estados:

- Resueltos: el usuario ha realizado uno o varios envíos y al menos en uno de ellos ha obtenido *AC*.
- Intentados: el usuario ha realizado uno o varios envíos, pero en ninguno de ellos ha obtenido como veredicto un *AC*.
- No intentados: el usuario no ha realizado ningún envío, ya sea porque el problema le parece muy complicado o porque no lo ha leído todavía.

Estos estados se pueden reducir a "resuelto / no resuelto" juntando los estados "intentado / no intentado" en el estado "no resuelto".

Desde el punto de vista de un recomendador, nos hemos planteado si la información de saber que un usuario ha fracasado al resolver un problema (estado "intentado") es o no significativa, de modo que hemos hecho dos variantes distintas del recomendador, cada una con un modelo de estados distinto. En el primero usamos la versión simplificada fusionando los "intentados" con los "no intentados", de modo que los estados son "resuelto" y "no resuelto". Llamamos a ese recomendador "RN" por las siglas de ambos posibles estados. El segundo tendrá en cuenta los tres estados y lo llamaremos "RIN", también por las siglas de los posibles estados.

Por otro lado, como se contó en el capítulo 3.3.5, al aplicar la fórmula 6.2 nos encontramos con muchos $P(B_i|A)$ que valen 0, causando que el resultado final sea también 0. Para tratar este problema hemos optado por dos maneras distintas. Una utilizando la estimación de *Laplace* y la otra omitiendo los ceros.

Si mezclamos las dos formas de considerar los estados con las dos maneras de tratar los ceros obtenemos 4 posibles combinaciones que son las siguientes:

- RN: Será el recomendador que considera dos estados de un problema para un usuario y que aplica la estimación de *Laplace*. Las siglas vienen de los estados "Resuelto" y "No resuelto".

- RN-ceros: Este recomendador se diferenciará del anterior en la estimación. En vez de aplicar la estimación de *Laplace*, a la hora de multiplicar ignorará los ceros.

- RIN: Al contrario que los anteriores, este recomendador considerará los tres estados "Resuelto", "Intentado" y "No resuelto", de los que salen las siglas. También implementará la estimación de *Laplace*.

- RIN-ceros: Este último recomendador considerará los mismos estados que el RIN, pero ignorará los ceros en las multiplicaciones igual que el RN-ceros.

Para cada versión dado un usuario con un perfil $B_0, ..., B_n$, queremos calcular la probabilidad de que este resuelva cada uno de los ejercicios, es decir los $P(A|B_0, ..., B_n)$ para todos los problemas A, y ordenarlos de mayor a menor.

Por la ecuación 6.2 anterior, esto significa que tenemos que calcular la probabilidad simple de cada problema y la probabilidad condicionada de cada problema con cada problema, $P(A)$ y $P(B_i|A)$ respectivamente. Utilizaremos para ello la información del perfil de todos los usuarios. El denominador, $\prod_{i=0}^{n} P(B_i)$, es la probabilidad de que haya un usuario con ese perfil y se calcula como la multiplicación de las probabilidades simples de los problemas B_i.

El denominador de la ecuación no depende en ningún caso del problema del que queremos saber su probabilidad. Por este motivo, para un mismo usuario el denominador va a valer siempre lo mismo. Dado que el fin último es ordenar las probabilidades podemos omitirlo, pues no nos interesa el valor exacto sino el orden relativo. Si se quisiera obtener la probabilidad real habría que calcularlo.

Empezamos con la probabilidad simple. Utilizaremos la información a priori de la base de datos, que hará las veces de "entrenamiento" de nuestro recomendador. Se va a calcular como se muestra en la formula 6.3, dividiendo la cantidad de usuarios que han resuelto el problema A entre el número de usuarios totales. En la fórmula, U representa al conjunto de usuarios, y $E(A, u)$ representa el estado del problema A para el usuario u de los estados descritos antes. Según la versión del recomendador ("RIN" o "RN") tendremos más o menos opciones

$$P(A) = \frac{|\{u|E(A,u) == \text{“}\mathcal{R}''\}|}{|U|},\ \forall u \in U \tag{6.3}$$

Por otro lado, la probabilidad condicionada de A habiendo resuelto B_i se calcula dividiendo el número de usuarios que tienen los dos problemas resueltos, entre el número de usuarios que han resuelto B_i, independientemente de que hayan resuelto o no el A, tal como se muestra en la fórmula 6.4. También serían necesarias las tablas de probabilidad condicionada de A no habiendo resuelto B_i, y en el recomendador "RIN" también es necesario calcular la probabilidad condicionada de A habiendo intentado B_i. Estos cálculos se hacen aplicando análogamente la misma fórmula, sustituyendo donde pone "$\mathcal{R}''$ en $E(B, u) ==$ "$\mathcal{R}''$ por "$\mathcal{N}''$ y por "$\mathcal{I}''$ respectivamente.

$$P(B_i|A) = \frac{|\{u|E(A,u) == \text{“}\mathcal{R}'' \wedge E(B_i,u) == \text{“}\mathcal{R}''\}|}{|\{u|E(A,u) == \text{“}\mathcal{R}''\}|}, \forall u \in U \tag{6.4}$$

De nuevo, esto se calcula a partir de los datos de entrenamiento de la base de datos. En la práctica, muchas de las probabilidades $P(B_i|A)$ que hemos calculado dan como resultado 0, porque hay parejas de problemas que no han sido resueltos simultáneamente por el mismo usuario. Las versiones "RN-ceros" y "RIN-ceros" eliminan esos casos ignorándolos, o lo que es lo mismo, sustituyendo el 0 por 1. Las otras versiones utilizan la aproximación de *Laplace*, que consiste en asumir que hay al menos un usuario que tiene el problema B_i en el estado "$\mathcal{R}''$, otro usuario en el estado "$\mathcal{N}''$, y en el "RIN"

otro en el "$\mathcal{I}''$. Con esta mejora la fórmula anterior queda finalmente:

$$P(B_i|A) = \frac{|\{u|E(A,u) == \text{“}\mathcal{R}'' \wedge E(B_i,u) == \text{“}\mathcal{R}''\}| + 1}{|\{u|E(A,u) == \text{“}\mathcal{R}''\}| + x}, \forall u \in U$$
$$dondex = \begin{cases} 2, & \text{si } RN \\ 3, & \text{si } RIN \end{cases} \tag{6.5}$$

Con todo esto, podemos calcular ya la probabilidad que tiene un usuario de resolver cualquier problema, que nos sirven de base para la recomendación.

Para llevar a cabo los cálculos de las probabilidades para los diferentes usuarios será necesario generar tablas y operar sobre ellas. Por ello, se ha tomado la decisión de realizar la implementación en *Python* ya que dispone de unas librerías llamadas *Numpy* y *Pandas* McKinney (2011) con la que se optimizan los cálculos gracias a la paralelización de estos. Dicha optimización es necesaria, pues si se hiciera una implementación con las estructuras básicas del lenguaje, habría que realizar las operaciones de forma secuencial. Esto provocaría que con grandes cantidades de datos las operaciones tarden un tiempo excesivo. Por ejemplo, con los datos que se nos han proporcionado después de 6h de ejecución aún no se habrían obtenido resultados. Sin embargo, utilizando la librería de *Pandas* el tiempo total de los cálculos se reduce a una media de 7 minutos, pero para recomendaciones individuales el coste de tiempo es de 5 a 7 segundos.

6.2. Análisis de resultados

Tras la implementación de las cuatro variantes del recomendador bayesiano descritas, se procede a analizar los resultados obtenidos para cada una de las métricas. Se compararán siempre las diferentes implementaciones (*RN*, *RIN*, *RN-ceros* y *RIN-ceros*), a fin de estudiar cuál de ellas promete mejores resultados. La versión ganadora será la que se utilizará para competir con la mejor versión del recomendador por pesos de los K-Vecinos que se explicará en el próximo capítulo.

Para visualizar mejor los datos los mostraremos en gráficas. El eje horizontal representa el número de problemas recomendados. El eje vertical indica el valor de la métrica correspondiente.

6.2.1. *Precision*

La gráfica 6.1 permite comparar la precisión de las cuatro variantes del recomendador con distinto número de problemas recomendados al usuario.

Lo primero a destacar es la línea del SR RN-ceros, puesto que es la única que tiene una forma distinta y empieza en un valor de 0.001 como muestra

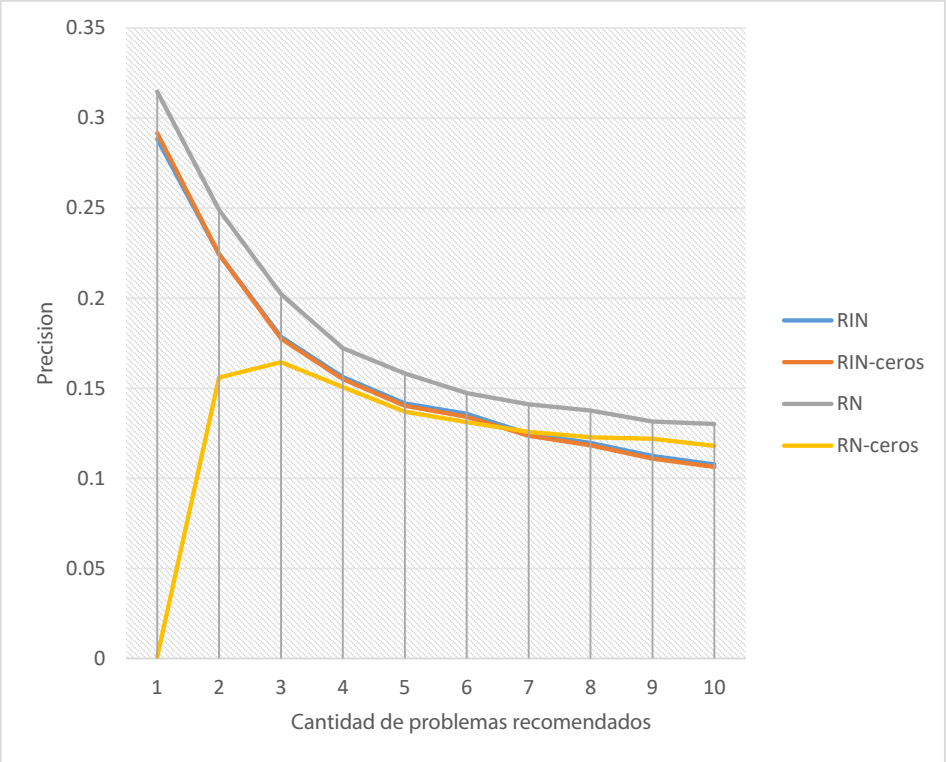

Figura 6.1: Gráfica comparación de precisiones.

la tabla 6.1 mientras que los otros tres recomendadores empiezan con valores superiores a 0.25. Antes de explicar el por qué este valor es tan bajo, hay que entender qué significa que sea tan bajo y luego suba rápidamente.

Según la definición de *precision*, una subida en la precisión significaría que se han recomendado problemas de bajo interés al usuario antes que los de alto interés. Y es esto lo que ha ocurrido con el RN-ceros, se ha recomendado a casi todos los usuarios en primer lugar el problema de *El profesor de música*, el cual en la fase de entrenamiento había sido resuelto por un único usuario, y en la de evaluación por 2. Resulta que este problema es bastante complicado, pero como llevaba poco tiempo en *¡Acepta el reto!* faltaba información.

Al calcular la probabilidad de que un usuario con perfil $B_0, ..., B_n$ resuelva lo resuelva, tenemos:

$$P(Musica|B_0, ..., B_n) = \prod_{i=0}^{n} P(B_i|Musica) \cdot P(Musica) = P(Musica)$$

Los posibles valores para $P(B_i|Musica)$ dado que solo un usuario lo ha resuelto son 0 si B_i no lo ha resuelto y 1 si lo ha resuelto. Este usuario en

	1	2	3	4	5	6	7	8	9	10
RN	0.314	0.249	0.202	0.172	0.158	0.147	0.141	0.137	0.131	0.130
RIN	0.288	0.224	0.178	0.156	0.141	0.135	0.124	0.119	0.112	0.107
RN-ceros	0.001	0.155	0.164	0.150	0.136	0.131	0.125	0.122	0.121	0.118
RIN-ceros	0.291	0.224	0.177	0.155	0.140	0.134	0.123	0.118	0.110	0.106

Tabla 6.1: Tabla *precision*

concreto tenía casi 300 problemas resueltos y solo había intentado 1 sin éxito. Por lo tanto, la formula anterior aplicada a este usuario quedaría:

$$P(Musica|B_0, ..., B_n) = 1 \cdot ... \cdot 1 \cdot P(Musica) = P(Musica)$$

En los recomendadores que aplican la estimación de *Laplace* (*RN* y *RIN*) al considerar que existe al menos un usuario para cada estado, ya no existe la posibilidad de que $P(B_i)$ sea ni 1 ni 0, sino que estos se transformarían 0.666 y 0.333 para el *RN*, y en 0.5 y 0.25 para el *RIN*. Esto hace que ya no esté en la primera posición.

En *RIN-ceros* no ocurre este problema porque al considerar los problemas intentados se favorece mucho la probabilidad de resolver problemas fáciles pues hay muchos difíciles intentados habiendo resuelto fáciles. Por ello aunque *El profesor de música* obtenga igualmente probabilidades altas, los fáciles obtienen mayores probabilidades y son recomendados primero.

En el recomendador *RN* se ha aplicado la aproximación de *Laplace* y por eso la estimación ha provocado que los ceros que conviertan en otros decimales, reduciendo así mucho la probabilidad de que este sea resuelto y por tanto pasando a ser recomendado bastante más tarde, y dejando en los primeros lugares los verdaderamente más probables. Es por eso que el recomendador *RN* obtiene unos valores de *precision* según lo esperado, con forma descendente. Que el valor inicial sea 0.31, aparentemente bajo, puede deberse al hecho de que en realidad no se ha recomendado nada al usuario sino que es una predicción de su comportamiento (sección 5.2).

Así mismo, se observa que las líneas de *precision* de los recomendadores *RIN* y *RIN-ceros*, son prácticamente coincidentes y que ambas siguen la forma esperada, al igual que el recomendador RN. Antes de estudiar esta casi coincidencia vamos a empezar explicando por qué en el recomendador *RIN-ceros* no ocurre igual que en el *RN-ceros*, donde el primer problema recomendado no es el más adecuado. Podríamos pensar que se debe a que la probabilidad de resolver este problema se reduce al considerar los intentados. Sin embargo no es así. La probabilidad de resolverlo es exactamente la misma, puesto que en la fase de entrenamiento el único usuario que resolvió dicho problema, también es el único que realizó un envío. Entonces, queda pensar que la probabilidad de resolver otros problemas ha aumentado. En efecto,

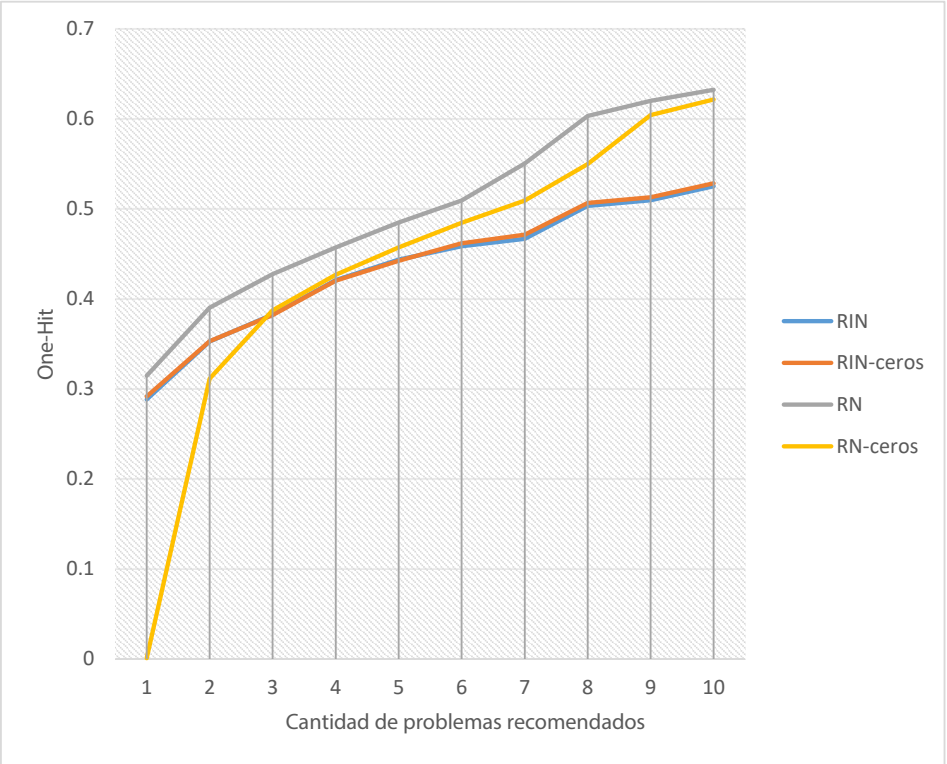

Figura 6.2: Gráfica comparación de one-hit.

al tener en cuenta los problemas intentados se obtienen unas probabilidades más altas para algunos problemas. Es también por esto que la aproximación de *Laplace* hace variar mínimamente las probabilidades finales, y en consecuencia los recomendadores *RIN* y *RIN-ceros* realizan recomendaciones prácticamente idénticas.

6.2.2. *One-hit*

Al contrario que en la métrica *precision*, el *one-hit* se espera que tenga una tendencia ascendente, dado que es obvio que cuantos más problemas sean

	1	2	3	4	5	6	7	8	9	10
RN	0.314	0.390	0.427	0.457	0.485	0.509	0.550	0.603	0.620	0.632
RIN	0.288	0.352	0.382	0.420	0.443	0.458	0.466	0.503	0.509	0.525
RN-ceros	0.001	0.311	0.387	0.426	0.457	0.484	0.509	0.550	0.604	0.621
RIN-ceros	0.291	0.352	0.382	0.420	0.442	0.461	0.471	0.506	0.512	0.528

Tabla 6.2: Tabla *one-hit*

	1	2	3	4	5	6	7	8	9	10
RN	0.039	0.061	0.075	0.085	0.098	0.109	0.122	0.136	0.146	0.161
RIN	0.035	0.055	0.066	0.077	0.087	0.101	0.108	0.118	0.125	0.133
RN-ceros	0.000	0.038	0.061	0.074	0.084	0.097	0.109	0.121	0.136	0.146
RIN-ceros	0.036	0.055	0.066	0.076	0.086	0.099	0.107	0.117	0.123	0.131

Tabla 6.3: Tabla *Recall*

recomendados más posibilidades hay de que al menos uno sea del agrado del usuario. En la figura 6.2 se comprueba que esto se cumple.

Primeramente se observa que se cumple que con un problema recomendado *one-hit* y *precision* son iguales. Se destaca el mismo valor próximo a cero del recomendador *RN-ceros*, cuya explicación es equivalente a la dada en el apartado anterior. Del mismo modo, la casi coincidencia del *one-hit* de *RIN* y *RIN-ceros*, se debe a que realizan prácticamente las mismas recomendaciones, como se explica al final del apartado anterior.

El recomendador vencedor vuelve a ser el *RN* que alcanza un *one-hit* del 50 % al recomendar 6 problemas y del 63 % al recomendar 10 problemas. Lo que hace pensar que el hecho de considerar las probabilidades de resolver un problema habiendo intentado otros, como es el caso de los recomendadores RIN, en lugar de favorecer a la recomendación como se esperaba que hiciera, la empeora ligeramente. Si con la *precisión* se espera que aumente al incluir el recomendador en *¡Acepta el reto!*, del mismo modo se espera que aumente también el *one-hit*, lo cual es esperanzador.

6.2.3. *Recall*

Es de esperar que desde el corte de la base de datos hasta el final los usuarios hayan resuelto muchos problemas, y por ello obtener valores de *recall* bajos no significa haber realizado malas recomendaciones, como se explica en la sección 5.1 con la definición de *recall*. En este caso concreto estamos evaluando recomendar hasta 10 problemas. Pues bien, a partir del corte hay 720 usuarios que han resuelto más de 10 problemas, y entre ellos suman un total de 18.512 problemas resueltos. Si asumiéramos que los diez primeros problemas resueltos de cada uno de ellos ha sido también recomendado, sumarían 11.512 problemas resueltos pero no recomendados.

Una vez más se obtienen valores similares para los recomendadores *RIN* y *RIN-ceros*, es de esperar como ya se ha mencionado anteriormente. Igualmente pasa con el recomendador *RN-ceros* que empieza en un valor muy próximo a cero. Y de nuevo el recomendador *RN* es el que obtiene mejores resultados.

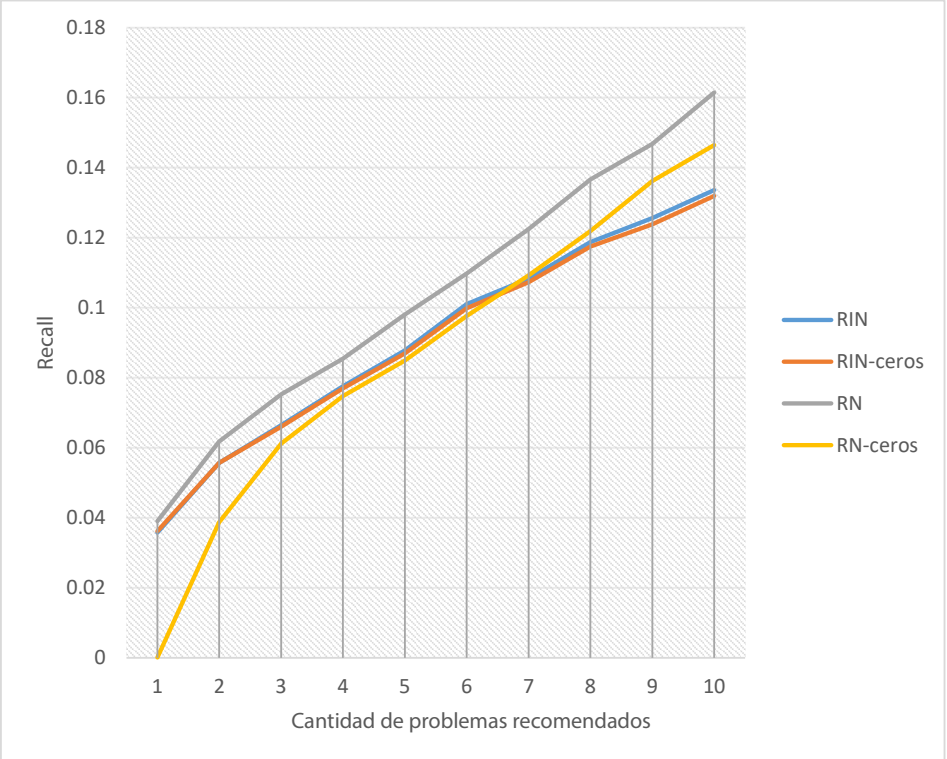

Figura 6.3: Gráfica comparación de recall.

6.2.4. *F-Score*

Por último, la cuarta métrica a analizar es f-score, que como se explica en la sección 5.1, atiende a una fórmula que utiliza las métricas de *precision* y *recall*. Por tanto, y dada la formula, es de esperar que las explicaciones dadas para los valores bajos de *recall*, sirvan para explicar también los que se observan en la gráfica de la figura 6.4.

Como viene siendo habitual los recomendadores *RIN* y *RIN-ceros* vuelven a tener valores casi coincidentes, el recomendador *RN-ceros* empieza en un valor próximo a cero, y el recomendador *RN* es claramente el que obtiene

	1	2	3	4	5	6	7	8	9	10
RN	0.069	0.099	0.109	0.114	0.121	0.125	0.131	0.137	0.138	0.144
RIN	0.063	0.089	0.096	0.103	0.108	0.115	0.115	0.119	0.118	0.119
RN-ceros	0.000	0.062	0.089	0.099	0.104	0.111	0.116	0.122	0.128	0.130
RIN-ceros	0.064	0.089	0.096	0.102	0.107	0.114	0.114	0.117	0.117	0.117

Tabla 6.4: Tabla *F-Score*

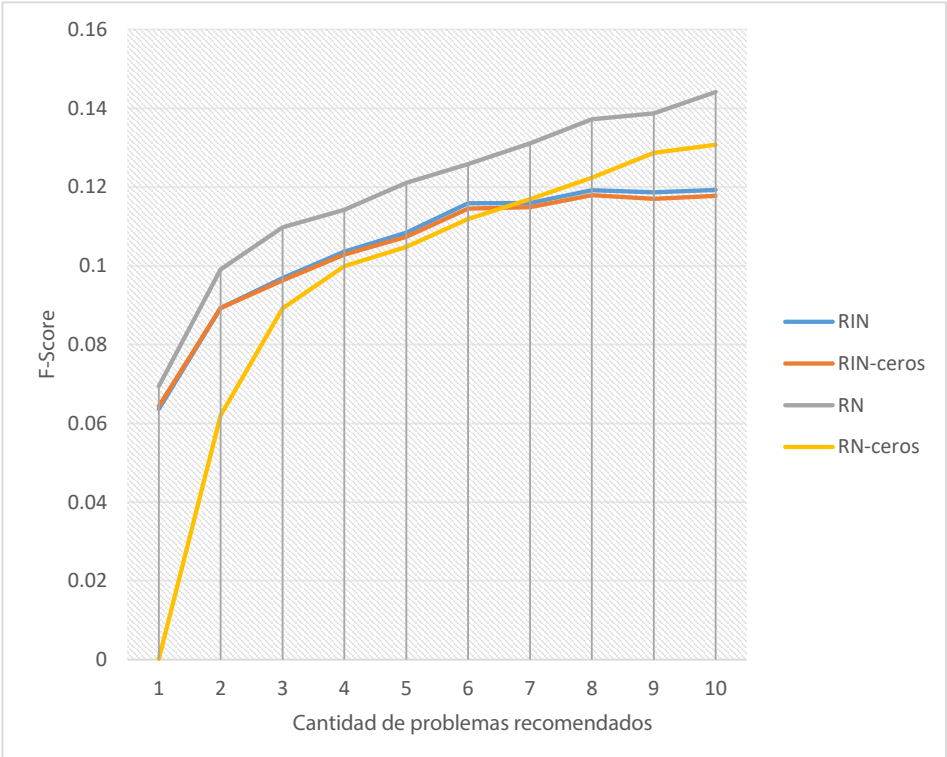

Figura 6.4: Gráfica comparación de f-score.

una eficacia superior a los demás, a pesar de que esta siga siendo baja por el efecto de *recall*.

6.2.5. Conclusiones

A la vista de los resultados y una vez realizado el análisis de estos, puede comprobarse que la versión del recomendador bayesiano que proporciona mejores recomendaciones es la *RN*. Con esto se concluye que la idea de considerar los problemas intentados pero no resueltos dentro del mismo grupo de problemas que los no resueltos, es bastante acertada, y es por esto que más adelante será la versión *RN* la que se comparará con una de las versiones del SR *k-vecinos* del capítulo siguiente.

Capítulo 7

Recomendador de pesos por los K-Vecinos más similares

Resumen: Este capítulo se centra en el recomendador de pesos por K-Vecinos más similares. Se explicará en detalle su funcionamiento aplicado a jueces en línea, y su evaluación y análisis de resultados, de manera individual.

El *recomendador de vecinos*, o *recomendador basado en correlaciones de usuarios*, es un recomendador cuyo algoritmo calcula la correlación entre usuarios dando un peso a cada problema que aún no ha hecho el usuario A, respecto al resto de usuarios Bi, teniendo en cuenta el grado de correlación de cada Bi.

La idea de este recomendador viene tomada del algoritmo K-Nearest Neighbor, explicado en el capítulo 4, donde se pretende buscar una serie de vecinos cercanos respecto al usuario a recomendar, y posteriormente se le genera una recomendación basada en los problemas realizados de esos vecinos más cercanos. Sin embargo, a diferencia del uso principal que se le da a K-Nearest Neighbor, que es el de clasificar, aquí se usan los vecinos calculando la correlación de los usuarios para después asignar pesos a los problemas que los vecinos obtenidos tienen resueltos y el usuario a recomendar no.

La finalidad y motivación para realizar este recomendador ha sido la necesidad de poder implementar un "algoritmo" más común en el mundo de los recomendadores, pero adaptado al caso de los jueces en línea y optimizado para estos, teniendo en cuenta, como se menciona en varios capítulos, que no disponemos de más información que usuarios y problemas anonimizados sin "atributos". Como ya se ha mencionado en el capítulo anterior, solo se consideran relaciones entre ellos de tal forma que se consideran, exclusivamente, los siguientes estados para un usuario respecto a un problema:

- Resuelto

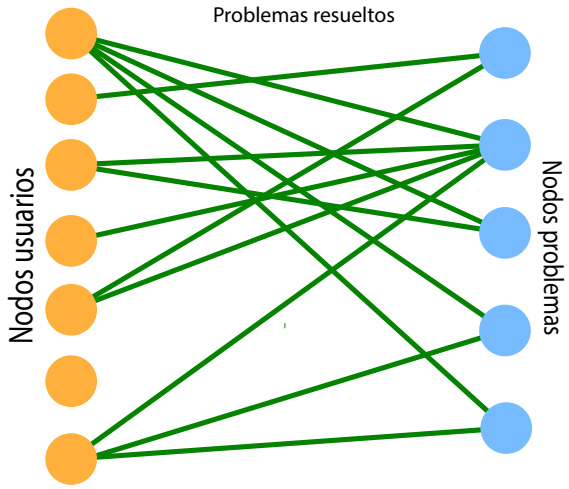

Figura 7.1: Ejemplo de relaciones entre nodos problemas y usuarios a través de los tipos de envíos.

- Intentado
- No intentado

Estos estados se aplican a las relaciones de los usuarios con los problemas. Para este recomendador se considerarán dos estados formados por dos subconjuntos de los mencionados. Estos son "resuelto" y "no resuelto" (que es la unión de los "intentados" y los "no intentados"). Es decir, este recomendador solo se va a fijar en si un usuario ha conseguido resolver un problema, y el resto de casos los considera como que el usuario aún no lo ha conseguido resolver, sin entrar a comprobar si está intentado o directamente no lo ha hecho. Por lo tanto, este recomendador va a tener equivalencia en cuanto los estados que se usan, al recomendador *RN* mencionado en el capítulo 6.

Este recomendador también sirve para contrastar los resultados con el primer recomendador que se ha realizado, y gracias a esto se podrá hacer un estudio para ver qué algoritmos funcionan mejor según qué ocasiones. Esto también permite la posibilidad de hacer un recomendador híbrido que decida qué algoritmo aplicar según que caso, entendiendo en qué momentos funciona mejor un algoritmo u otro.

Como dice el nombre completo de este recomendador, aunque el concepto de obtener los K-Vecinos más cercanos sea el mismo, a la hora de generar el listado de problemas de recomendación se le asignan pesos a los problemas que ha resuelto cada vecino, según las veces que se repiten.

La manera que se utiliza para encontrar los vecinos más cercanos, es la de calcular la correlación entre el usuario a recomendar y el resto de usuarios,

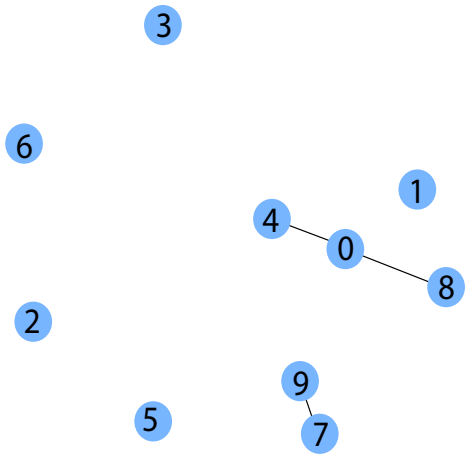

Figura 7.2: Relación de los 10 primeros usuarios de *¡Acepta el reto!* por sus problemas en común.

así una correlación fuerte implica tener un vecino más cercano que otro con correlación débil.

La forma de obtener la correlación entre usuarios, es fijarnos en los problemas que existen en común entre el usuario a recomendar respecto al otro usuario, y dividirlo por la cantidad de problemas que el usuario a recomendar ha resuelto en total. La fórmula se muestra con más detalle en la siguiente sección.

Tal como se observa en la figura 7.1, se puede entender *¡Acepta el reto!* como un grafo de nodos de problemas y usuarios donde las relaciones entre estos es el estado de un nodo usuario respecto si ha logrado o no resolver un nodo problema. De esta manera, para las líneas verdes, el usuario ha conseguido realizar un problema, y se genera una relación que este recomendador utiliza.

En la imagen de la figura 7.2, la *fuerza de la correlación* viene marcada por la distancia entre los nodos, y se calcula de la misma forma que la correlación entre usuarios usada para el algoritmo de este recomendador. Se muestra esta imagen para proporcionar una somera idea al lector de cómo trata este recomendador los datos de un juez en línea.

Como se puede observar en la figura 7.3, los nodos en azul, representan los 50 primeros usuarios de *¡Acepta el reto!* y las líneas junto con el grosor de estas, las relaciones y la correlación que hay entre los usuarios, según la cantidad de problemas resueltos que tienen en común. Esta figura es otra forma de representar la correlación entre usuarios de *¡Acepta el reto!* al igual que en la figura 7.2, pero vista de otra manera y con un conjunto más grande. No se van a mostrar ejemplos con más usuarios ya que empieza a resultar

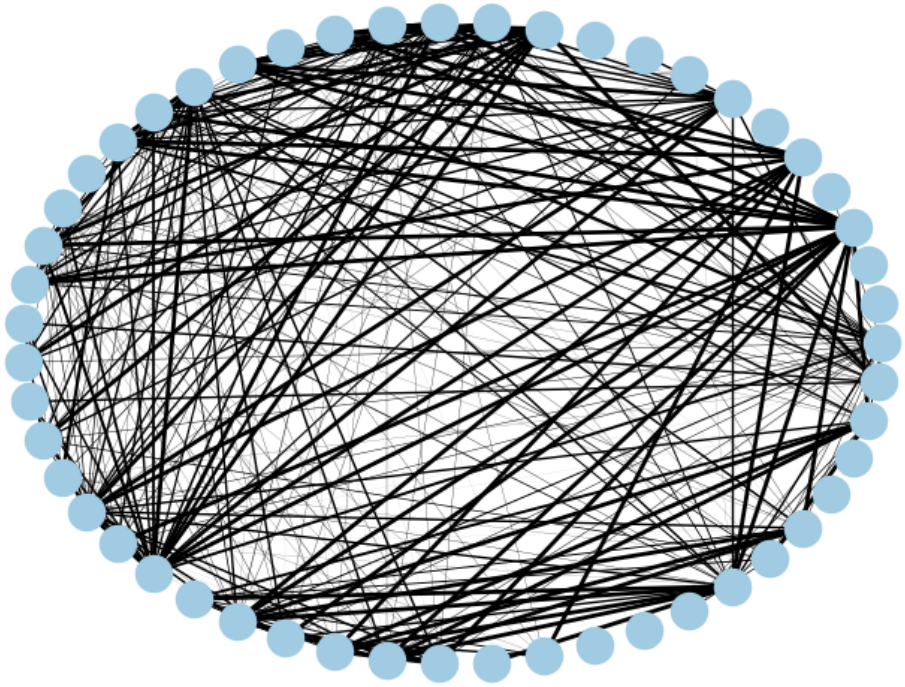

Figura 7.3: Relaciones entre usuarios (nodos), según la cantidad de problemas resueltos en común (grosor de la línea) en *¡Acepta el reto!*.

caótico y poco útil un ejemplo con cantidades mayores.

El recomendador por pesos de los K-Vecinos más similares, analiza un juez en línea como las dos últimas figuras mencionadas, calculando ese estilo de relaciones (entre usuarios) a partir de datos como los que la figura 7.1 representa.

Por lo general, este algoritmo recomendará problemas personalizados a un usuario según los problemas realizados por sus vecinos, de tal forma que por un lado el recomendador asegura que el usuario a ser recomendado tendrá buena tasa de éxito al resolver un problema de los recomendados, y por otro lado, se da una recomendación personalizada basada en el criterio de usuarios similares a este, que han resuelto otros problemas que aún no ha realizado el usuario a recomendar y es la similitud entre usuarios la que le da un peso importante a esos problemas no resueltos aún por el usuario al que se le va a realizar la recomendación.

Se puede considerar por lo tanto, que a parte de ese éxito alto, se está realizando una recomendación por gustos de usuarios, ya que si un usuario similar a mí ha querido hacer un problema y lo ha conseguido, seguramente yo también tenga ganas de intentar el mismo problema y también pueda

conseguirlo.

7.1. Funcionamiento del recomendador por K-Vecinos más similares

Como ya se ha comentado, este recomendador tiene similitudes con los Nearest Neighborhood Hall et al. (2008) mencionado en la sección 3.3.1. La diferencia de este recomendador es, que tras obtener los vecinos más cercanos, estos no se usan para clasificar en clases al usuario y generar una recomendación en base a la clase donde se ha clasificado. Se usa la correlación que determina la distancia de los vecinos, y se aplica esta correlación para asignar pesos a los problemas que tienen los vecinos y que el usuario a recomendar no ha intentado.

El recomendador internamente maneja una matriz donde sus filas representan los usuarios, sus columnas los problemas, y el valor fila/columna informa del estado.

En este caso el valor dispone únicamente de dos estados: conseguido, o el resto de casos que se consideran como no conseguido. Este resto de casos incluyen en el mismo conjunto, el caso de un usuario que ha intentado pero no resuelto un problema, y el caso en el que el usuario ni si quiera ha intentado realizarlo.

El recomendador realiza el siguiente algoritmo:

- Al usuario a recomendar se le llamará: $Owner = O$.

  Para un "O", obtenemos una lista de los K usuarios más similares con su correlación correspondiente. Esta lista se obtiene calculando la correlación para cada usuario β_i de la base de datos, con O. Posteriormente se ordena de mayor a menor.

  El cálculo de la correlación entre β_i y O se basa en, dados los elementos:

  - $\rho O =$*Conjunto de problemas de Owner*
  - $\rho\beta_i =$*Conjunto de problemas de usuario* β_i

  Ahora se define la correlación entre dos usuarios como:

$$Correlacion = \frac{|\rho\beta_i \bigcap \rho O|}{|\rho O|} \tag{7.1}$$

  Tras este cálculo para cada usuario de la base de datos, obtenemos aquellos N usuarios (β_N) con correlación más alta, que representarán los N vecinos más cercanos y creamos un *Diccionario(D)* cuya clave sea el id del problema y el valor indique el peso de la recomendación.

Este diccionario se usa para el cálculo de pesos de los problemas, para después ser ordenados por este peso al crear la lista de recomendación.

El peso de la recomendación se define finalmente de la siguiente manera:

- ρ_i =Problema "i" del conjunto de problemas existentes P.

$$\forall \rho_i \notin \rho O \rightarrow \sum_{j=0}^{N} \frac{correlacion(\beta_j, O)}{N} sii\ \rho_i \in \rho\beta_j \tag{7.2}$$

- Al dividir por N, se asegura que el resultado nunca sea mayor que 1. El sumatorio se realiza en cada posición del diccionario correspondiente a cada ρ_i. Finalmente se ordena el diccionario resultado por los valores, obteniendo una lista de recomendación ordenada con unos pesos asignados.

Como se puede observar, la fórmula que define la correlación es la que nos sirve para encontrar los vecinos más cercanos según esa norma, y quedarnos con los K mejores/más similares.

Por otro lado, para obtener la recomendación en sí, se buscan los problemas que cada *K-Vecino* obtenido ha resuelto pero el usuario a recomendar no, y se ordenan según una combinación entre la correlación de cada usuario y las veces que ese problema se repite en el resto de vecinos (esto es, el cálculo de los pesos de los problemas).

7.2. Optimización y modificaciones

Antes de entrar directamente en este punto, cabe destacar que el estudio de la aceleración se ha realizado siempre bajo las mismas condiciones de ejecución en este recomendador, pero no para ambos recomendadores realizados. Estas condiciones de ejecución son las siguientes:

- Procesador: Intel(R) Core(TM) i7 860.
  - Frecuencia: 2.80GHz
  - Número de núcleos: 4
- RAM: 8 GB
- GPU: NVidia Geforce 1050Ti GTX 4GB DRR
- SO: Windows 10 pro
- Almacenamiento de información: Disco duro convencional

Habiendo indicado con mayor claridad las condiciones bajo las que se ha estado ejecutando el recomendador, vamos a hablar en adelante de tiempos y la aceleración de las mejoras.

La primera versión del algoritmo trabaja con consultas directas sobre la BD de *¡Acepta el reto!*. Esta, para los casos de usos reales, es una opción lenta y no eficaz además de costosa y puede hacer que llegue a saturar una base de datos que no está pensada para informar directamente al recomendador.

Se han realizado nuevas versiones mejoradas para lograr abstraer esta información y tenerla en el recomendador de manera local.

Este paso, es necesario si se quiere tener el recomendador como servicio externo y que sirva también para portarlo a otros jueces en línea sin generar dependencias de la plataforma *¡Acepta el reto!*.

Por lo tanto, para esta mejora, se opta por trabajar con una *matriz de datos* en local que se cargue en memoria al arrancar el servicio realizando las actualizaciones pertinentes en memoria y haciendo sus volcados al sistema de archivos donde esté alojado el recomendador.

La matriz de datos representa los *usuarios y problemas* en base a sus *filas y columnas* de tal forma que cada posición fila de la matriz representa un id de usuario y cada posición columna representa un id de problema.

Al trabajar con *Python*, se aprovecha la eficacia y rendimiento que tiene *Numpy* sobre matrices. A diferencia del recomendador Bayesiano que como se ha comentado en el anterior capítulo, trabaja con *Pandas* para poder asignarle una clave a cada posición de fila y cada posición de columna, como *Numpy* no permite esto, se han creado dos arrays de conversión: Uno para las filas y otros para las columnas. Las posiciones de este array se corresponden a las de la matriz, y el valor de cada posición nos permite conocer el ID de usuario/problema asociado a este.

La elección de *Numpy* en este recomendador, respecto al uso de *Pandas* en el otro, es meramente la de poder contrastar diferencias en el uso de ambas tecnologías, y el interés formativo que requiere intentar desarrollar ambos recomendadores con recursos diferentes.

Cabe destacar que el control de obtención, actualización y mantenimiento de la información se abstrae en una clase que se ocupa de estos procesos para que el recomendador simplemente se encargue de hacer lecturas sobre esta matriz[1].

Esta implementación y los cambios mencionados para esta versión del recomendador, mejora el tiempo de cálculo por entrega en 5 respecto a la primera que se ha comentado[2]. (Se pasan de obtener tiempos de 5 minutos por recomendación a 1 minuto).

Aún así es necesario seguir optimizando el tiempo de recomendación, ya que sigue siendo muy alto. Para conseguir esta mejora se aprovecha todo lo

[1] La arquitectura y estructura del recomendador se hablará en la sección 7.3

[2] Estamos hablando de la aceleración

que ofrece *Numpy* en cuanto a paralelismo.

Para mejorar ciertas operaciones, sobre la matriz, que se realizan de manera secuencial se reprograman tareas para que funcionen en paralelo, observando que varias operaciones (principalmente de inserción, resta de conjuntos, intersección de conjuntos...) mejoran sustancialmente mediante los operadores y métodos que ofrece *Numpy*.

Realizando estas modificaciones se alcanza la tercera versión del recomendador, la cual obtiene una mejora de alrededor de 120, sin ni si quiera aprovechar el rendimiento máximo de procesamiento que el dispositivo de pruebas ofrece.

Tras la mejora, en esta última versión, el recomendador pasa a recomendar en menos de 1 (tiempos entre 0.1 y 0.7 segundos).

Estas mejoras se han medido con la base de datos parcial cortada a 2018. Por lo que con los registros a día de hoy, el tiempo de recomendación aumentaría un poco a medida que los usuarios y problemas crecen, pero de manera lineal y poco significativa.

Con estos nuevos resultados favorables, a falta de investigar con más profundidad qué nuevos cuellos de botella puedan surgir, la optimización es altamente satisfactoria y no se ha considerado necesario, por el momento, continuar con esta línea de investigación.

7.3. Arquitectura y estructura interna

Aunque el capítulo 9 tratará de la arquitectura de comunicación y actualización de ambos recomendadores, se menciona aquí someramente de la arquitectura *interna* de este recomendador. Es decir, esta sección se centra en tratar con más detalle, cómo funciona internamente la comunicación entre clases, de la parte interna del recomendador, dejando de lado la parte de la API y estructuras de nivel superior que permiten hacer del recomendador, un servicio externo y actualizable.

Este recomendador se ha seccionado en dos clases, mostradas en la figura 7.4, las cuales se encuentran separadas en dos ficheros diferentes.

Estas dos clases surgen de la necesidad de abstraer la parte de gestión de la BD interna con la parte del algoritmo de recomendación.

La lógica relacionada con el almacenamiento de datos, está agrupada en un fichero que contiene por un lado parámetros de configuración para hacer la conexión adaptativa de manera cómoda y rápida a diferentes jueces en línea, y por otro lado, contiene la clase *JuezDB*, mostrada en la figura 7.4. Esta clase está encargada de los siguientes aspectos:

- Cargar la matriz de recomendación en memoria de manera que si no existen los ficheros en local, los carga desde una BD MySQL y los guarda en local para cargarlos, a partir de entonces, desde ese almace-

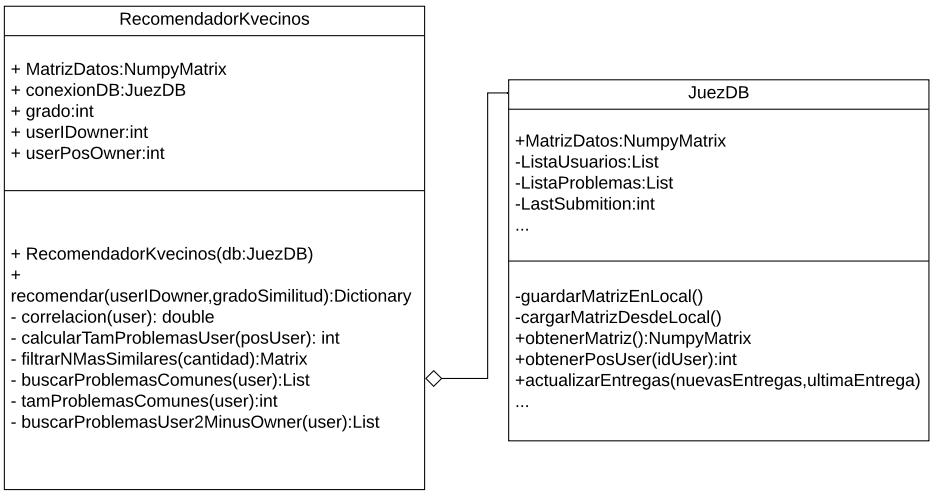

Figura 7.4: Diagrama de clases internas del recomendador K-Vecinos, simplificando atributos y métodos más interesantes.

namiento.

En caso de que existan los ficheros en local los carga directamente ya que supone menor coste y es este el contenedor real de la información del recomendador.

Esta carga también incluye información que *parsea* las posiciones de filas/columnas que representan usuarios y problemas, a sus respectivos IDs. Aunque para el caso de los problemas y en el juez en línea de *¡Acepta el reto!* se trabajan con los External ID, que corresponden directamente a la posición en la matriz más cien.

La carga desde la BD solo se debe hacer para cargar por primera vez la información al recomendador. A partir de ahí siempre trabajará desde local sin depender de una BD externa.

- Actualizar la BD en memoria y en local cada vez que se le llame al método "*actualizarEntregas*".

  Esta actualización se llama desde la clase *clientePeticiones* y se encarga de leer lo que *clientePeticiones* le pasa por parámetro para hacer los cambios en la matriz y los arrays correspondientes, añadiendo nuevas filas/columnas y elementos, así como modificar los valores de la matriz

de datos sobrescribiendo las posiciones en las que haya nuevos AC a uno.

- *Parsear* la información de la matriz de datos a quien se lo solicite, devolviendo el id de usuario asociado a una posición, y viceversa.

En cuanto a la clase *RecomendadorBasico* del fichero con el mismo nombre, se la inicializa con una instancia de la clase *JuezDB*, para obtener de ahí la matriz asociada.

Su principal método es "recomendar", al que se le asocia un id de usuario y el parámetro de K-Vecinos con el que se quiere realizar la recomendación.

Este método llama a otros métodos, que si bien pueden ser invocados externamente, tienen un fin de uso de métodos privados para la propia clase.

El método "recomendar" devuelve al solicitante un listado de recomendación tras ejecutar el algoritmo explicado en la sección 7.1.

El resto de métodos mostrados en la figura 7.4 para la clase recomendador básico, son propios del algoritmo explicado y no merecen profundizar en ellos en esta sección.

7.4. Entrenamiento y evaluación

En esta sección se habla sobre el entrenamiento realizado en el recomendador, haciendo un análisis de los resultados de este entrenamiento y la evaluación obtenida de las diferentes métricas descritas en el capítulo 5.

En particular, las métricas que se han obtenido de la evaluación son el "*precision*", el "*One-Hit*", el "*recall*" y finalmente el "*F-Score*".

Para cada tipo de métrica se han generado dos modelos de gráficas que representan diferentes configuraciones del recomendador y conjuntos de N mejores problemas obtenidos.

7.4.1. Resultados de Precision

Comenzando a analizar los datos del "*Precision*" se obtuvieron los resultados de evaluación de la tabla 7.1 tras el entrenamiento.

En la tabla 7.1 mencionada, las filas representan los parámetros del recomendador, ya que se realizó el entrenamiento para diferentes parámetros de K-Vecinos más similares. Por otro lado, las columnas representan la cantidad de problemas que se obtienen de la lista de recomendación, que se han tenido en cuenta a la hora de evaluar.

Los resultados del parámetro de configuración del recomendador para "*Precision*" se van a analizar mejor a través de la figura 7.5.

En la gráfica de la figura 7.5 mencionada, se muestra como cada $TOPN$[3],

[3]Top N representa los N mejores problemas que hemos cogido para evaluar.

	1	2	3	4	5	6	7	8	9	10
3	0.116	0.114	0.108	0.111	0.111	0.104	0.100	0.097	0.095	0.093
10	0.195	0.163	0.153	0.165	0.156	0.150	0.151	0.144	0.139	0.134
20	0.173	0.164	0.159	0.160	0.156	0.150	0.144	0.142	0.141	0.138
50	0.187	0.179	0.170	0.167	0.161	0.159	0.154	0.149	0.147	0.144
100	0.180	0.183	0.173	0.167	0.161	0.157	0.153	0.151	0.147	0.143
250	0.172	0.154	0.154	0.145	0.145	0.143	0.142	0.139	0.137	0.134
500	0.148	0.149	0.147	0.146	0.148	0.144	0.143	0.138	0.134	0.131
1000	0.142	0.137	0.139	0.143	0.144	0.139	0.137	0.132	0.127	0.125
2000	0.131	0.128	0.138	0.144	0.140	0.135	0.134	0.130	0.127	0.124
5000	0.133	0.129	0.140	0.145	0.140	0.136	0.134	0.130	0.127	0.123
all	0.133	0.129	0.140	0.145	0.141	0.136	0.135	0.130	0.127	0.123

Tabla 7.1: Tabla de resultados de *Precision* para el SR K-Vecinos.

marcados con una diferente línea, varía según los parámetros de recomendación que se le asignen al algoritmo *K-Vecinos más similares*.

Se observa cómo la recomendación para cualquier *N*, a medida que el parámetro de recomendación se aproxima a los 10 vecinos más similares la precisión crece, por lo que se entiende que el funcionamiento del recomendador tiende a mejorar con ligeras fluctuaciones.

Esa mejora dura hasta que se superan los 100 vecinos más similares, donde se observa cómo desciende ligeramente la precisión.

Por lo tanto, el intervalo óptimo para *K-Vecinos más similares* está en el rango entre 10 y 100. Esto sucede porque al incluir vecinos menos similares con el usuario a recomendar se distorsiona ligeramente la información de los problemas que realmente el usuario necesita. De manera que si se tiene en cuenta una similitud baja de un vecino más similar con el usuario, desencadenará en una recomendación peor.

La gráfica 7.6 representa en el eje de abscisas la cantidad de recomendaciones que pedimos al recomendador (Mejores N recomendaciones), y el eje de ordenadas representa el propio valor de la precisión al igual que en la gráfica analizada previamente. Las líneas representan cada parámetro de configuración K-Vecinos[4].

En esta gráfica (7.6) se puede observar que la precisión disminuye a medida que el eje de las "x" crece. Esto se debe a la naturaleza de la fórmula de precisión y la forma de como se han obtenido los *TP*, *FP* y *FN*, provoca una tendencia a disminuir, en nuestro caso, a partir de $x > 3$.

Sin embargo, para N = 2 y N = 3 no se observan en conjunto esos decrementos apenas. Los valores se mantienen y en algunos casos incluso se

[4]Aunque no se ha mencionado antes, se han seleccionado esos parámetros como más representativos como una manera incremental interesante donde duplicamos el parámetro en cada configuración respecto al anterior, permitiéndonos ver en diferentes grados cómo afecta dicho parámetro.

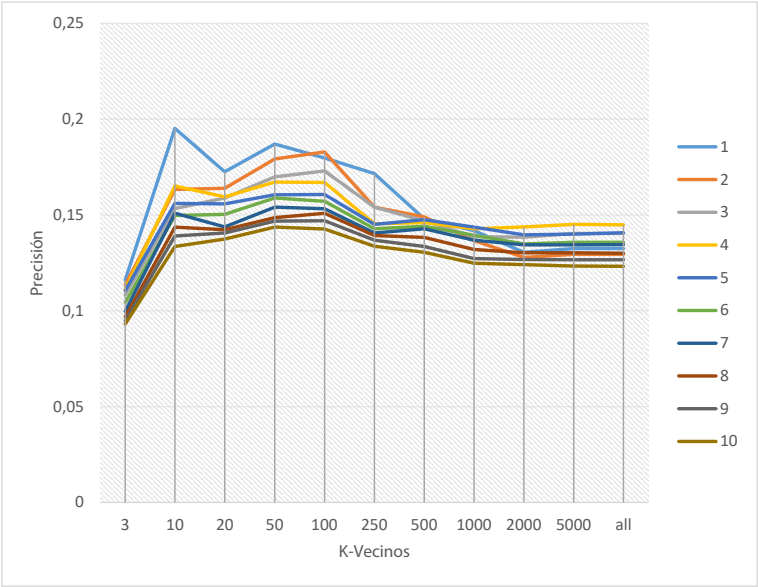

Figura 7.5: Gráfica de precisión 1.

obtienen mayores precisiones.

De esta gráfica se obtiene como información concluyente que un valor óptimo a tener en cuenta a la hora de recomendar sería utilizar los 3 mejores problemas, ya que, a medida que seguimos incluyendo problemas de esta lista ordenada de recomendación, esos problemas nuevos que se incorporan, cada vez tienen menos peso: son problemas que, aunque se incluyan, el recomendador considera que tiene menos éxito de que el usuario sea capaz de interesarse por ellos y de resolverlos.

Esto sucede, a pesar de que el conjunto de problemas a recomendar sea mayor y por lo tanto tengamos más probabilidades en recomendar problemas exitosos.

De todas formas, como estos dos conceptos se contraponen, apenas se nota ese decremento que implica obtener problemas con menos peso, ya que como el conjunto de problemas que se recomienda es mayor, el recomendador puede que acierte más al tener mayor rango de éxito.

Se puede concluir que la forma en la que se obtiene la precisión para cada Top N problemas, es la influyente en los resultados de este decrecimiento, al igual que pasará de manera similar en las próximas graficas del resto de métricas.

Según lo dicho, esta segunda gráfica para evaluar independientemente los problemas, no es tan interesante para nuestro recomendador, pero sí para compararla con otros recomendadores y confrontar los resultados.

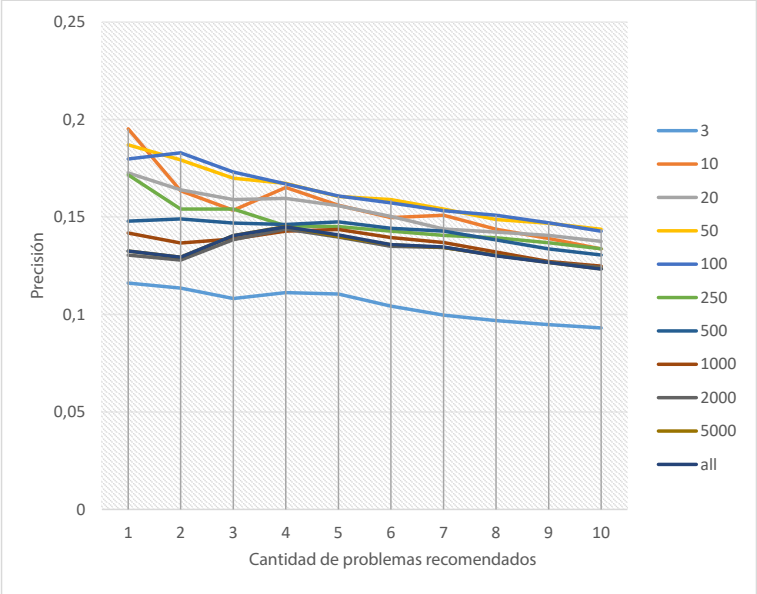

Figura 7.6: Gráfica de precisión 2.

Esto también se analizará mejor más adelante en un capítulo posterior, aunque reduciendo el conjunto de datos de este recomendador a los que han resultado más interesantes durante el análisis.

7.4.2. Resultados del One-Hit

La tabla de resultados se muestra en la figura 7.2.

De esta tabla (7.2), al igual que se ha hecho con la precisión, se han generado las dos gráficas de las figuras 7.7 y 7.8.

Como ya se ha comentado en otras secciones y capítulos, el "*One-Hit*" representa tener al menos un acierto en el conjunto de problemas recomendados.

Analizando la gráfica 7.7 para el "*One-Hit*", se observa que el conjunto de *TOPN* varía de forma similar para este concepto a medida que se aplica el algoritmo de recomendación incluyendo más vecinos similares.

El comportamiento que nos muestra la gráfica es el mismo con el que concluimos el de la precisión, y es que para unos valores de K-Vecinos entre 10 y 100 obtenemos los máximos "*One-Hit*" prácticamente.

Se observa también que para valores menores a 10 la función gráfica es creciente y para valores mayores a 100 la gráfica empieza a decrecer.

Lo que cambia aquí respecto a la precisión es que las líneas están a diferentes niveles (a parte de la obviedad de que los valores obtenidos son más

	1	2	3	4	5	6	7	8	9	10
3	0.116	0.178	0.213	0.266	0.304	0.319	0.330	0.351	0.364	0.376
10	0.195	0.255	0.295	0.376	0.402	0.428	0.445	0.461	0.475	0.490
20	0.173	0.248	0.299	0.345	0.380	0.403	0.414	0.448	0.481	0.497
50	0.187	0.264	0.316	0.355	0.377	0.416	0.438	0.458	0.486	0.497
100	0.180	0.275	0.327	0.377	0.396	0.432	0.450	0.468	0.485	0.498
250	0.172	0.233	0.303	0.331	0.367	0.390	0.409	0.432	0.453	0.465
500	0.148	0.236	0.294	0.348	0.382	0.406	0.427	0.440	0.446	0.457
1000	0.142	0.223	0.281	0.339	0.376	0.396	0.416	0.424	0.431	0.448
2000	0.131	0.212	0.276	0.339	0.371	0.388	0.408	0.420	0.431	0.448
5000	0.133	0.214	0.282	0.342	0.373	0.391	0.408	0.420	0.431	0.445
all	0.133	0.214	0.282	0.342	0.372	0.391	0.408	0.419	0.431	0.445

Tabla 7.2: Tabla de resultados de *one-hit* para el SR K-Vecinos.

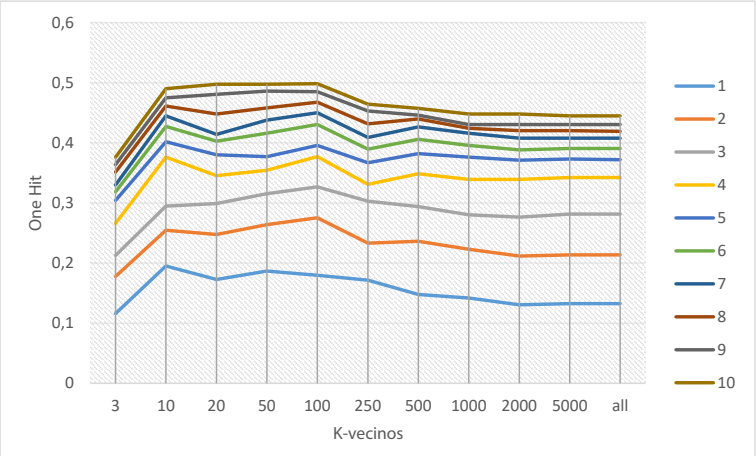

Figura 7.7: Gráfica One-Hit 1.

altos por lo que significa el One-Hit), pero eso se entiende mejor analizando la segunda gráfica de esta subsección.

En la gráfica 7.8 se puede observar como el *One-Hit* crece a medida que crece N, como es de esperar, el *One-Hit* debe crecer, pues si aumentas el conjunto de problemas a recomendar, aumenta la probabilidad de que *al menos* uno tenga éxito y sobre todo teniendo en cuenta que en ese conjunto están los que el recomendador ha considerado como mejores.

7.4.3. Resultados del recall

En cuanto a los resultados obtenidos por parte del "*recall*", la tabla resultante se puede observar en la figura 7.3.

De esta tabla (7.3) se han sacado las dos gráficas correspondientes, siguiendo el mismo esquema de las subsecciones anteriores. Estas gráficas se

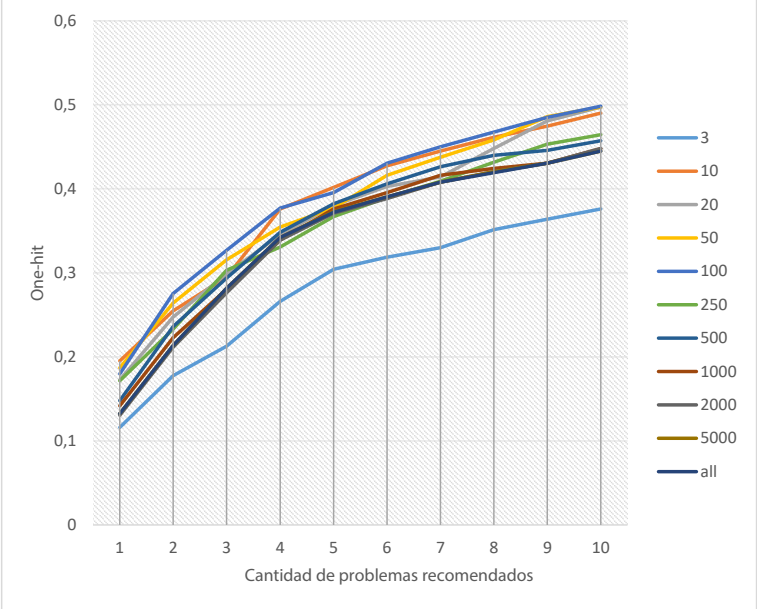

Figura 7.8: Gráfica One-Hit 2.

pueden observar en las figuras 7.9 y 7.10.

En la figura 7.9 se observa como el "*recall*", que nos indicaba la proporción *(en nuestro caso sobre 1)* de aciertos reales que se calcularon correctamente, mejora para todos los *TOPN*, con los parámetros de configuración dentro del intervalo $[10 - 100]$.

Esta mejora es ligera, pero suficientemente notable ya que sufre un efecto similar al analizado ya con el "*One-Hit*".

Se concluye de la misma manera para el "*recall*" que como se concluía con el "*One-Hit*", y es que el motivo de que la gráfica nos muestre un decremento a partir de cierto punto y un incremento para valores de K-Vecinos muy bajos, es principalmente por el propio parámetro de recomendación, que para valores muy bajos nos viene a indicar la falta de datos (insuficientes valores), y el exceso de datos que ensucian la calidad del resultado al coger tantos K-Vecinos que dejan de ser tan similares muchos de ellos.

En la gráfica 7.10, se puede observar un incremento lineal destacable para todos los parámetros de configuración, a medida que aumentamos el subconjunto de los *TOPN* mejores.

Es de esperar que el *recall* mejore linealmente a medida que este subconjunto crece ya que la proporción de aciertos reales que se obtiene correctamente debe mejorar si el subconjunto del total de problemas se acerca a tener el mismo cardinal del conjunto del que proviene.

Es destacable que para el parámetro de configuración 3, el "*recall*" man-

	1	2	3	4	5	6	7	8	9	10
3	0.009	0.018	0.025	0.034	0.043	0.048	0.054	0.060	0.066	0.072
10	0.015	0.025	0.036	0.051	0.060	0.069	0.082	0.089	0.097	0.103
20	0.013	0.025	0.037	0.049	0.060	0.070	0.078	0.088	0.098	0.106
50	0.014	0.028	0.039	0.052	0.062	0.074	0.083	0.092	0.102	0.111
100	0.014	0.028	0.040	0.052	0.062	0.073	0.083	0.093	0.102	0.110
250	0.013	0.024	0.036	0.045	0.056	0.066	0.076	0.086	0.095	0.103
500	0.011	0.023	0.034	0.045	0.057	0.067	0.077	0.085	0.093	0.101
1000	0.011	0.021	0.032	0.044	0.055	0.065	0.074	0.082	0.088	0.096
2000	0.010	0.020	0.032	0.044	0.054	0.062	0.073	0.080	0.088	0.096
5000	0.010	0.020	0.033	0.045	0.054	0.063	0.073	0.080	0.088	0.095
all	0.010	0.020	0.033	0.045	0.054	0.063	0.073	0.080	0.088	0.095

Tabla 7.3: Tabla de resultados de *recall* para el SR K-Vecinos.

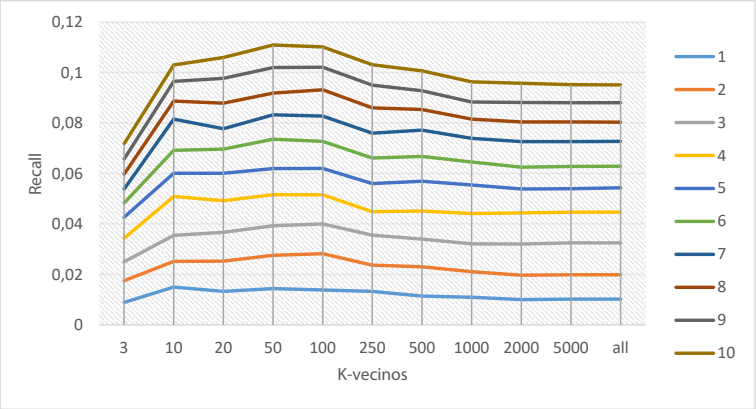

Figura 7.9: Gráfica 1 del recall para el SR K-Vecinos.

tiene proporciones significativamente más bajas respecto al resto.

7.4.4. Resultados del F-Score

Por último, vamos a ver como se comporta el *F-Score*.

Haciendo un breve resumen de lo que representaba esta métrica, venía a ser la precisión del entrenamiento/evaluación del recomendador. Para ello, los valores obtenidos se pueden ver en la tabla de la figura 7.4.

Partiendo de la tabla 7.4 se han representado las dos gráficas que se han usado hasta ahora, con estos nuevos valores. Estas gráficas se muestran en las figuras 7.11 y 7.12 correspondientes.

Para la gráfica 7.11 destacar esa mejora significativa en los parámetros de configuración entre $[10 - 100]$ como ya pasaba con las anteriores métricas, por lo que el *F-Score* también funciona mejor con esos parámetros de

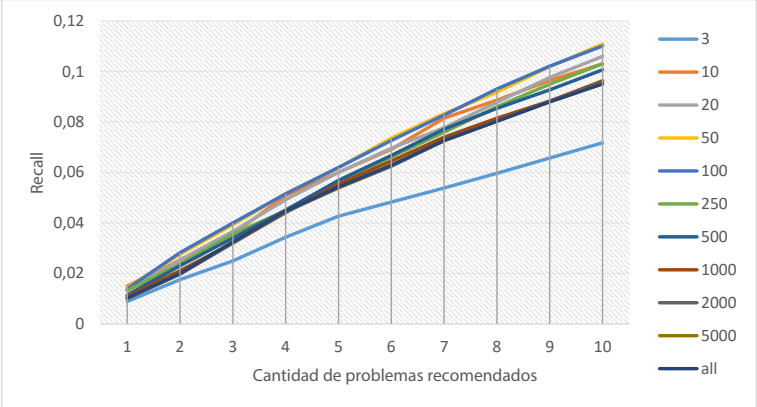

Figura 7.10: Gráfica 2 del Recall para el SR K-Vecinos.

	1	2	3	4	5	6	7	8	9	10
3	0.017	0.030	0.041	0.052	0.062	0.066	0.070	0.074	0.078	0.081
10	0.028	0.044	0.058	0.078	0.087	0.095	0.106	0.110	0.114	0.116
20	0.025	0.044	0.060	0.075	0.087	0.095	0.101	0.109	0.115	0.120
50	0.027	0.048	0.064	0.079	0.089	0.101	0.108	0.114	0.120	0.125
100	0.026	0.049	0.065	0.079	0.089	0.100	0.108	0.115	0.121	0.124
250	0.025	0.041	0.058	0.069	0.081	0.090	0.099	0.106	0.112	0.116
500	0.021	0.040	0.055	0.069	0.082	0.091	0.100	0.105	0.110	0.114
1000	0.020	0.037	0.052	0.067	0.079	0.088	0.096	0.101	0.104	0.109
2000	0.019	0.034	0.052	0.068	0.078	0.085	0.094	0.100	0.104	0.108
5000	0.019	0.035	0.053	0.068	0.079	0.086	0.094	0.100	0.104	0.108
all	0.019	0.035	0.052	0.068	0.078	0.085	0.094	0.099	0.104	0.107

Tabla 7.4: Tabla de resultados de *F-Score* para el SR K-Vecinos.

configuración como ya venía pasando con el resto.

Al parecer el *F-Score*, al ser métrica que viene generada por el recall, hereda los comportamientos para esta gráfica.

Se ve como a partir de 1000 los valores, que han decrementado desde 100, empiezan ahora a mantenerse prácticamente similares.

El comportamiento de la gráfica 7.12 sigue siendo también similar al del "*recall*", y sus valores rondan entre el 2 y el 13 por ciento.

De esta gráfica se puede destacar también un crecimiento, pero esta vez con cierta curva cóncava que nos informa de que a medida que el subconjunto *TOPN* crece, tiende a algún tipo de límite o le cuesta más mejorar/incrementar el valor entre el TOPN-1 y TOPN.

Destacar también que para el parámetro K-Vecinos igual a 3, el valor de *TOPN* apenas se incrementa respecto al resto, pero esto es por esa falta de

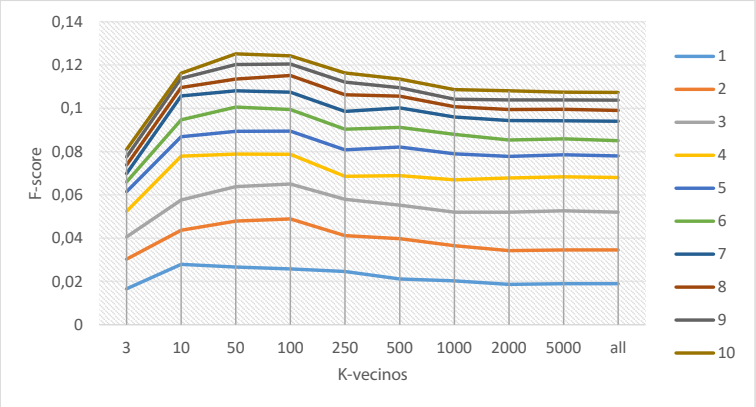

Figura 7.11: Gráfica 1 del *F-Score* para el SR K-Vecinos.

información de tener tan pocos vecinos, que provoca peores resultados.

7.4.5. Conclusiones del análisis

Estas gráficas y el análisis pertinente realizado, ha permitido optimizar los parámetros de configuración encontrando los máximos y los mínimos óptimos y permitiendo concluir que los valores más adecuados para el recomendador, que además se utilizarán para contrastar resultados con el *recomendador Bayesiano*, son los parámetros de K-Vecinos 10-50-100 y los TOPN 2-3, principalmente.

Por lo general se ha intentando analizar principalmente el comportamiento, ya que el valor en sí es menos llamativo que las propias subidas y bajadas que nos muestra la gráfica.

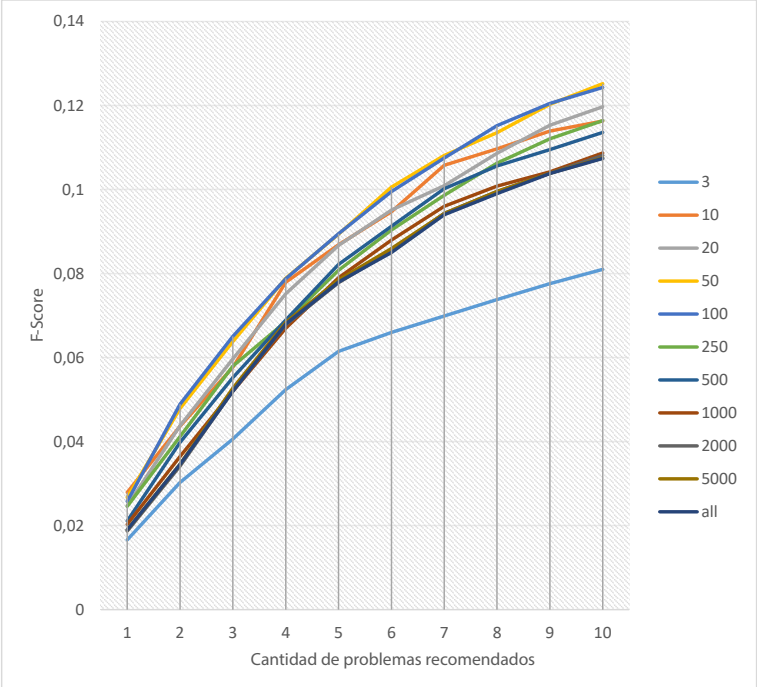

Figura 7.12: Gráfica 2 del *F-Score* para el SR K-Vecinos.

Capítulo 8

Comparación entre recomendadores

Resumen: Una vez seleccionadas las mejores versiones de los recomendadores anteriormente descritos nos disponemos en este capítulo a compararlas con el fin de comprobar cuál de las dos se ajusta más al juez online *¡Acepta el reto!*.

Tras haber implementado dos tipos de recomendadores, *bayesiano* y *K-Vecinos*, y haber realizado una evaluación de las diferentes versiones, se ha seleccionado la versión más prometedora de cada uno. El objetivo es comparar ambos resultados en base a las métricas descritas con el objetivo de concluir cuál de los dos tipos de recomendación sería más fiable dentro del sistema de *¡Acepta el reto!*. El que mejores resultados obtenga será el primer candidato a ser llevado a evaluación en un caso real con usuarios.

Las versiones seleccionadas han sido:

- *Bayesiano*: se ha seleccionado la versión *RN*, que considera dos estados y aplica la estimación de *Laplace*.
- *K-Vecinos*: la versión ganadora de este SR has sido la que considera los *100-vecinos* más similares.

Al igual que en los capítulos anteriores los resultados se mostrarán en gráficas. El eje horizontal se corresponderá con el número de problemas recomendados, mientras que el eje vertical representará el valor de la métrica correspondiente.

8.1. *Precision*

En la figura 8.1 se muestra la gráfica de *precision*. Es muy destacable la diferencia cuando el número de problemas recomendados es 1. El reco-

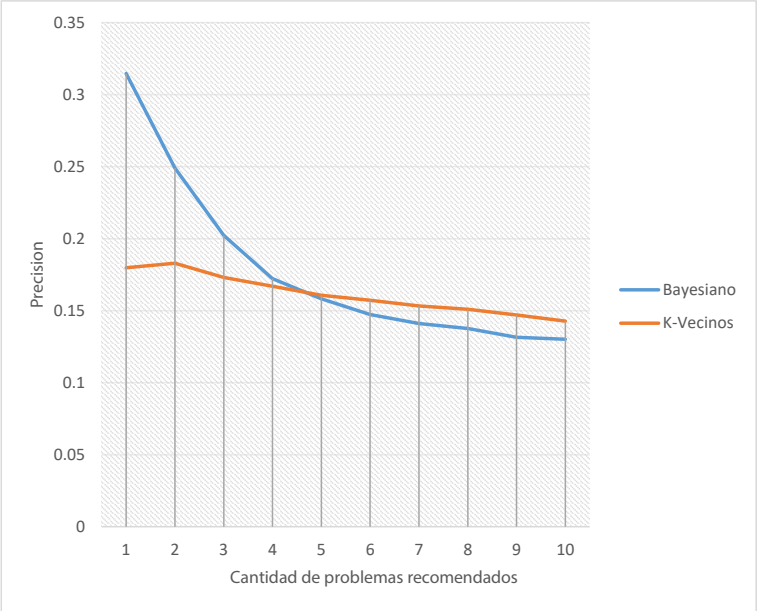

Figura 8.1: Gráfica de comparación de *precision*

mendador *bayesiano* obtiene un resultado casi un 60 % superior al resultado del recomendador *K-Vecinos*. Lo que significa que el primer problema que propone el recomendador *bayesiano* es bastante mejor que la del otro.

Sin embargo, al ir recomendando más problemas la precisión del recomendador *bayesiano* cae rápidamente, mientras que el recomendador *K-Vecinos* es bastante más estable y cae más lentamente, incluso presenta alguna subida. Es tal la diferencia en la velocidad de caída que cuando se recomiendan 5 problemas el recomendador que tiene mayor precisión aunque por poca diferencia es el *K-Vecinos*.

Esto parece ser debido a que el recomendador *bayesiano* ha recomendado mayoritariamente como quinta opción el mismo problema a todos los usuarios, mientras que el recomendador *K-Vecinos* ha recomendado problemas diferentes a muchos de ellos. Que sea un problema sencillo y muy probable de ser resuelto por todos los usuarios no quiere decir que el problema sea interesante para el usuario. Por ello parece ser más acertada una recomendación de problemas más distribuida y más personalizada a cada usuario.

8.2. *One-hit*

Al contrario que en la métrica *precision*, el claro y constante vencedor en *one-hit* es el recomendador *bayesiano*. Este recomienda problemas fáciles de resolver para el usuario y por ello es más probable que sea resuelto. El

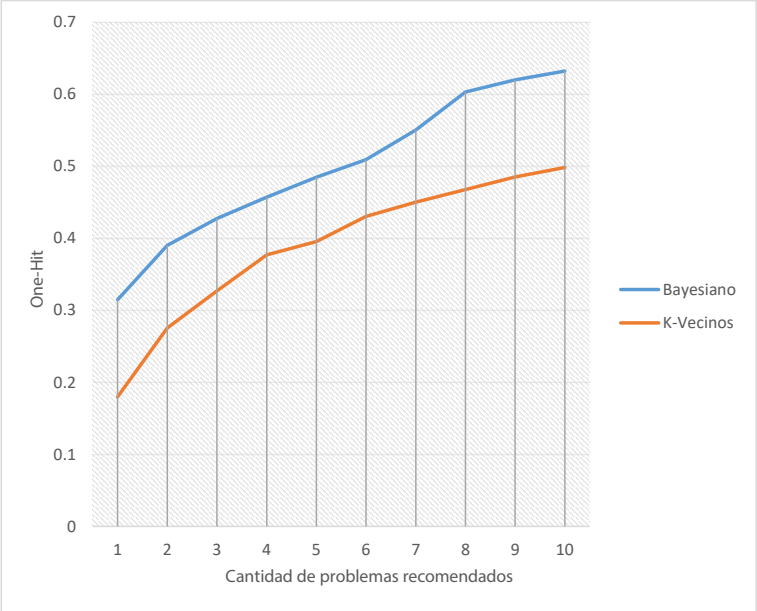

Figura 8.2: Gráfica de comparación de *one-hit*

recomendador *K-Vecinos*, al personalizar más la recomendación no llega a recomendar los más fáciles, sino alguno más interesante para el usuario, por ello le cuesta más tener al menos un acierto.

Aun así, como se muestra en la figura 8.2, la diferencia en el valor de *one-hit* no es muy grande, por lo que ambos recomendadores tienen un gran porcentaje de acierto en al menos una recomendación.

8.3. *Recall*

Los resultados obtenidos de *recall* se muestran en la figura 8.3. Al igual que en los resultados de *one-hit*, se observa que el recomendador *bayesiano* vuelve a estar por encima en todo momento del recomendador *K-Vecinos*. Los valores de *recall* son aparentemente bajos como ya se explicó anteriormente por el hecho de que hay muchos más problemas resueltos que recomendados.

Aunque en la gráfica parezca que las líneas están muy distantes, si observamos los valores en las tablas correspondientes (39 y 58) la diferencia oscila entre 2 y 5 centésimas por lo que en realidad están bastante a la par aunque si es verdad que el recomendador *bayesiano* toma ventaja.

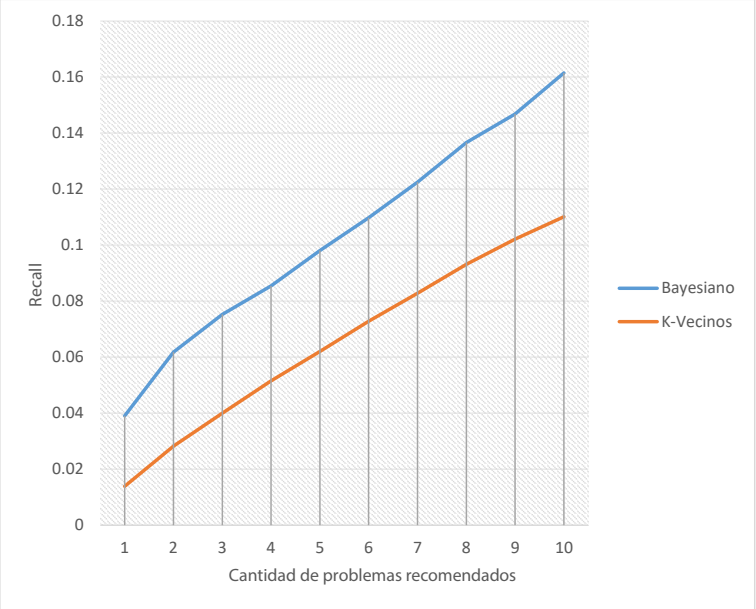

Figura 8.3: Gráfica de comparación de *recall*

8.4. *F-Score*

No es de extrañar que con la métrica *f-score* se obtenga una gráfica en la que también el recomendador vencedor sea el *bayesiano*. En efecto así es. En la gráfica de la figura 8.4 se ve claramente que el recomendador *K-Vecinos* está siempre por debajo.

Se destaca la diferencia entre la distancia inicial entre las líneas y la distancia final. Esta distancia se va reduciendo a consecuencia de la métrica *precision*, pues descendía muy rápidamente, y como ya sabemos *f-score* se calcula con una fórmula que utiliza las métricas *precision* y *recall*.

8.5. Conclusiones de la evaluación

El entrenamiento aplicado a ambos recomendadores ha sido de utilidad principalmente a la hora de estudiar el comportamiento de los diferentes parámetros de recomendación con un subconjunto de los mejores problemas de la lista de recomendación que los recomendadores proponen. De esta comparación se pueden observar por lo general mejores resultados en las diferentes métricas por parte del recomendador *bayesiano*.

Es curioso el caso de la precisión donde el comportamiento que tiene el recomendador *bayesiano* se asemeja a una función exponencial decreciente de la forma $f(x) = a^x$, $(0 < a < 1)$, así como el recomendador *K-Vecinos*

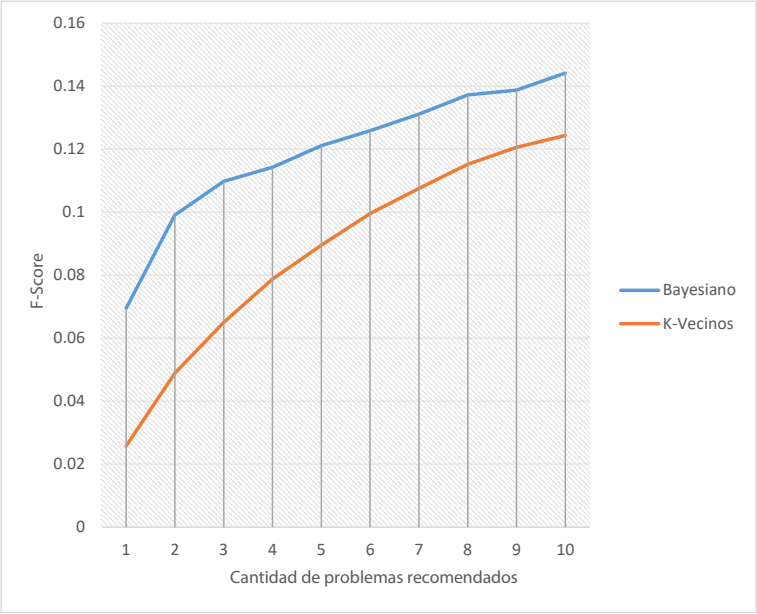

Figura 8.4: Gráfica de comparación de *f-score*

a una función lineal decreciente con poca caída, de la forma $f(x) = b - mx$, permitiendo en algunos casos superar el resultado de la precisión del *bayesiano*.

Se puede entender que el comportamiento del recomendador *bayesiano* haya dado mejores resultados de recomendación, ya que el concepto del teorema de bayes aplicado en recomendaciones resulta más preciso ya que busca calcular la probabilidad de que un usuario pueda resolver un problema. Por otro lado, el recomendador por K-Vecinos no ha considerado detalles que podrían refinar información como son los casos en los que un usuario ha intentado resolver un problema pero no ha tenido éxito.

El hecho de que el recomendador *bayesiano* haya resultado victorioso en estas métricas, no descarta idea de que sus recomendaciones tengan que ser siempre las más personalizadas para un usuario, ya que una recomendación personalizada no siempre tiene por qué ser la opción más fácil para este.

Por otro lado, también cabe contrastar el tiempo de recomendación generado por cada SR. Como se ha comentado en los correspondientes capítulos, aunque para ambos recomendadores el tiempo de recomendación por usuario es bastante bajo, se aprecia que *K-Vecinos* está mejor optimizado y su coste algorítmico es menor. Esta diferencia es también interesante de resaltar ya que puede merecer la pena empeorar la calidad de recomendación un poco a costa de mejorar la velocidad y eficiencia de esta, sobre todo si se están calculando muchas recomendaciones a la vez.

Capítulo 9

Implementación como servicio externo

Resumen: Este capítulo engloba la arquitectura e ingeniería de implementación de los dos *recomendadores* realizados, para poder ser implementados como servicio externo, siguiendo los principios de adaptabilidad y portabilidad. La estructura interna de cada recomendador se expone en los capítulos referentes a cada uno de ellos.

9.1. API de comunicación

Para llevar a cabo un funcionamiento completo del recomendador, es necesario implementar los algoritmos sobre un diseño eficaz que permita una retroalimentación, y que a la vez sea capaz de dar servicio múltiple a quien le solicite recomendaciones.

Para ello se ha implementado una arquitectura adecuada que soluciona de manera eficaz estos problemas: se ha dividido en tres partes esta arquitectura que separan de manera lógica las diferentes funcionalidades que el *recomendador* debe poder realizar.

Los *recomendadores* realizados hasta ahora han trabajado en un entorno de pruebas local. Para hacerlos totalmente funcionales en casos de usos reales, son necesarias nuevas estructuras que permitan al *recomendador* poner al día la información de este, sobre el *juez en línea* al que está dando servicio.

También son necesarias estas nuevas estructuras para tener una vía sobre la que el *juez en línea* pueda hacer peticiones al *recomendador*, y obtener los datos que sean necesarios y útiles.

En los *recomendadores* debe incluirse un nuevo módulo lógico para que se encargue de resolver estos problemas. Este nuevo bloque lógico debe funcionar como una API de comunicación entre el *recomendador* y el *juez en*

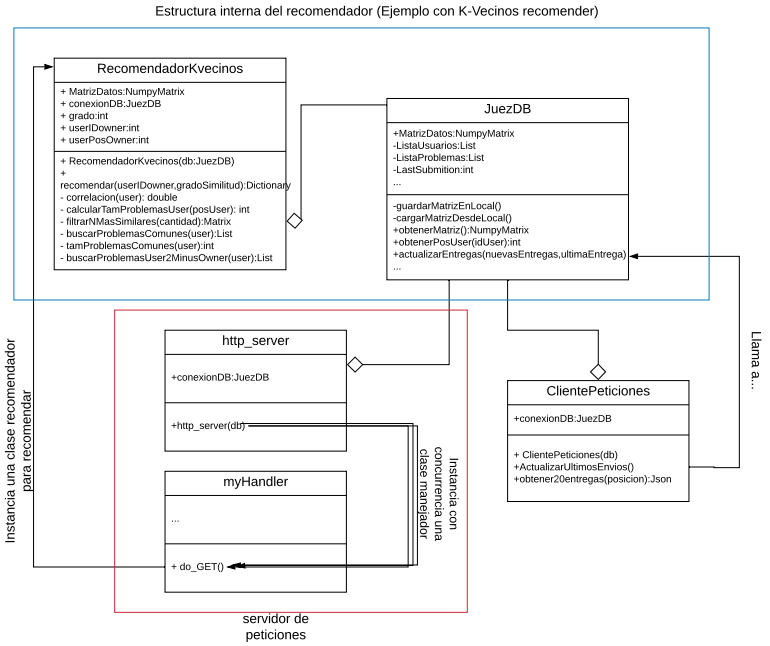

Figura 9.1: Diagrama de clases extendido para el recomendador K-Vecinos.

línea.

Cada uno de los *recomendadores* se ha programado con diferentes tecnologías. Las partes propias de cada uno de ellos se explican con mayor detalle en los capítulos 6 y 7 respectivamente.

Aquí se va a tratar la lógica relacionada con la API común a ambos *recomendadores* que representa un nivel superior en las capas de la arquitectura de esta, ya que esta lógica instanciará objetos de recomendación, para hacer uso de la recomendación y la actualización de la BD.

La lógica relacionada con la API, es decir, la parte que gestiona la comunicación con el *juez en línea*, se encuentra totalmente separada de la lógica del recomendador.

Este fichero contiene tres clases: dos que funcionan como servidor http y manejador de peticiones, y una que funciona como cliente que hace peticiones *HTTP* al *juez en línea* sobre el que opera.

De manera detallada se observa cómo funciona y se relaciona este servicio externo, aplicado al caso del *recomendador K-Vecinos* y el *juez en línea ¡Acepta el reto!*, como la figura 9.2 indica.

Mediante este mecanismo, está completamente separado el *recomendador* del juez en línea permitiendo su portabilidad.

Haciendo ligeras modificaciones en el conjunto de la API y esta estructura

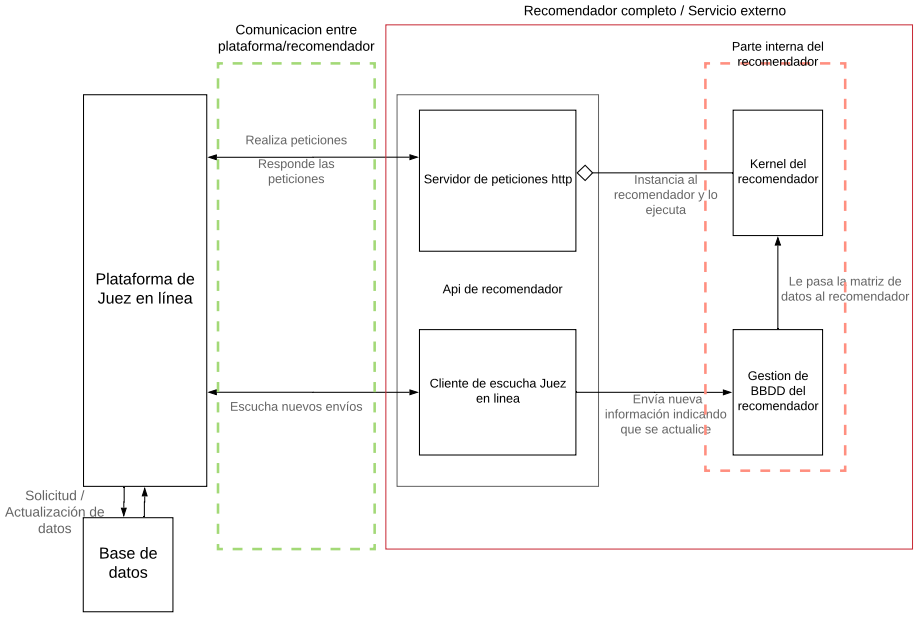

Figura 9.2: Funcionamiento del recomendador K-Vecinos con API implementada para su uso como servicio externo.

superior, se pueden adaptar otros *recomendadores* a la API, o aplicar varios *recomendadores* sobre una misma API y estructura superior.

De esta manera se puede conseguir que el *juez en línea* pida al servicio de recomendación que *recomendador* utilizar, o sea la propia estructura superior será la que decida qué *recomendador* se debe utilizar para un usuario concreto, en función de los resultados obtenidos para el mismo.

La figura 9.1 muestra el diagrama de clases de lo que nos interesa en cuanto a la API del recomendador. En este se tiene un ejemplo aplicado para el recomendador *K-Vecinos*.

Las clases internas de este *recomendador* en concreto, no van a ser analizadas en las siguientes subsecciones ya que son propias de este y no comunes de ambos recomendadores.

Se ha decidido mostrar el diagrama de clases extendido de la API con este ejemplo, ya que se ha logrado implementar y probar con correcto funcionamiento este módulo del que estamos hablando, con este recomendador mencionado en el capítulo anterior.

De todas formas, hay que dejar claro que la API se comunica de manera similar con ambos *recomendadores*, es decir, siempre va a tener que instanciar una clase recomendador, como se verá en las siguientes secciones del capítulo,

para llamar a su método recomendar.

La API también va a tener que poder llamar a un método que permita actualizar los nuevos envíos a la BD del *recomendador* con el que se está trabajando.

9.2. Cliente de actualización

El cliente de actualización está gestionado por la clase *ClientePeticiones* del fichero "*server.py*".

Esta clase se encarga de hacer que el *recomendador* sea capaz de actuar como un cliente más para la web de *¡Acepta el reto!*, y de esta manera observe los cambios en la sección de "últimos envíos" que hacen los usuarios en *¡Acepta el reto!*.

Tras observar estos cambios nuevos que el recomendador no tiene actualizados, *ClientePeticiones* se encarga de pasárselos a la clase pertinente que se dedique a actualizar la BD.

ClientePeticiones se encarga por lo tanto de renovar los nuevos envíos del juez en línea. Tanto al arrancarse el servicio de recomendación, como cada cierto periodo de tiempo, buscando en qué última entrega se quedó actualizado este servicio, de tal forma que *¡Acepta el reto!* o el *juez en línea* pertinente lo trata como un cliente más que quiere ver esa información de la sección donde muestre los últimos envíos.

Esta pequeña clase se conecta para el caso específico de *¡Acepta el reto!* a la zona donde se actualizan los últimos envíos con la información detallada y necesaria para el recomendador, y obtiene un JSON mediante peticiones GET a este servicio del *juez en línea*.

En el caso concreto de *¡Acepta el reto!*, se tiene planteado crear un módulo dentro del *juez en línea* que facilite al cliente esta información.

Hasta que llegue este caso, el cliente simplemente "espía" la información de *¡Acepta el reto!*[1], el cual muestra entregas en grupos de 20 filas, y va desplazándose de 20 en 20 entregas por esta página hasta llegar al último submit que el *recomendador* tenía actualizado.

En el ejemplo de código 9.1 se observa un *XML* de cómo *¡Acepta el reto!* presenta las entregas para extraer la información necesaria de ellas.

```
<submissions>
  <nextLink>
    https://www.aceptaelreto.com/ws/submission/?start=21&size=20
  </nextLink>
  <submission>
    <num>334785</num>
    <user>
      <id>12426</id>
```

[1] `https://www.aceptaelreto.com/ws/submission/`

```
      <nick>ProConrad</nick>
      <name>ProConrad</name>
      <avatar>https://www.aceptaelreto.com/pub/user/noavatar.jpg</avatar>
    </user>
    <problem>
      <num>168</num>
      <title>La pieza perdida</title>
    </problem>
    <result>AC</result>
    <language>JAVA</language>
    <executionTime>1.953</executionTime>
    <memoryUsed>1746</memoryUsed>
    <ranking>702</ranking>
    <submissionDate>2019-05-16T19:14:32+02:00</submissionDate>
  </submission>
  .
  .
  .
</submissions>
```

Listing 9.1: Ejemplo en XML de las entregas observadas de *¡Acepta el reto!*

Posteriormente, *ClientePeticiones* pasa esta información al método correspondiente que se encarga de actualizar la BD.

La información se proporciona de manera ordenada por id de entrega, desde la entrega con id más bajo a la entrega con id más alto.

Cada elemento de esta lista ordenada es un diccionario de 3 elementos: -id de usuario, problema que ha intentado y el estado, (Resuelto, No Resuelto[2]).

Esta técnica se realiza únicamente en el arranque del recomendador aunque está preparada para poder ejecutarse concurrentemente con el pequeño servidor *HTTP* que se crea posteriormente del paso actualización, tras el arranque del *recomendador* como servicio.

De esta forma podría, actualizar la BD del *recomendador* cada cierto tiempo, a la vez que da servicio el servidor.

Se puede observar gráficamente esta explicación dada a través de las figuras 9.1 y 9.2.

Cabe destacar que el cliente de peticiones hace uso de la librería "*requests*", la cual facilita enormemente el tratamiento del envío de peticiones *HTTP* y su posterior respuesta.

9.3. Servidor de peticiones

El servidor de peticiones es un servidor *HTTP* preparado para atender peticiones Get específicas. Hace uso de la librería de python "*http.server*".

[2]No resuelto viene a representar en este caso un WA y derivados de intentos que no han conseguido el AC, es decir, intentando pero sin éxito.

De esta librería específicamente se utilizan las clases "*BaseHTTPRequestHandler*" y "*HTTPServer*" que son adaptadas para realizar las tareas de recomendación.

La adaptación de estas dos clases para su uso se explican con mayor detalle a continuación:

- Clase *http_server*: Esta clase es la adaptación de "*HTTPServer*"de la librería de Python "*http.server*" mencionada, y acoge el servidor web para atender peticiones de clientes que serán solicitudes de recomendaciones con diferentes parámetros de entrada.

  Esta clase permite concurrencia, es decir, hace uso de la siguiente clase que se va a explicar, de manera que puede estar atendiendo y gestionando varias peticiones a la vez.

  Esta clase contendrá como atributo la clase "*JuezDB*" para tener a mano la BD de recomendación y hacer uso de estas en las peticiones que reciba.

- Clase *myHandler*: Esta clase es la adaptación de "*BaseHTTPRequestHandler*" de la librería de python "*http.server*" mencionada, y hace de manejador de las peticiones que recibe el servidor, se encarga de atender los *GET* del protocolo *HTTP*.

  Realiza directamente la recomendación instanciando el objeto recomendador y asignándole la BD correspondiente.

  Esta recomendación la *parsea* a formato *JSON* y la devuelve al cliente al que le está gestionando la petición. Del formato *JSON* que devuelve a los clientes, se muestra la estructura unas líneas más abajo.

  Por otro lado, los parámetros de las peticiones get que maneja la clase "*myHandler*" son los siguientes:

  - *idUsuario*: Id de usuario a quien se le va a recomendar.
  - *topk*: Devuelve solo los mejores K elementos. Si no existe el parámetro, devuelve todos ordenados. Si *topk* es negativo, devuelve los "*K peores*".
  - *k-vecinos*: Parámetro específico para el recomendador *K-Vecinos*. Realiza la recomendación con los *K-Vecinos* dados. Si no existe el parámetro y se utiliza este recomendador, se usa la configuración elegida por defecto, cuyos parámetros se han seleccionado a través de la subsección de conclusiones del capítulo 7.

  El fragmento de código 9.2 escrito en *JSON* es un ejemplo de respuesta del servidor para una petición de recomendación con los parámetros "$topk = 5$" y "$kvecinos = 100$".

```
{
  "error": 0,
  "recomendacion": [
    {
      "problemaID":155,
      "valor": 0.23219512195121966
    },
    {
      "problemaID":119,
      "valor": 0.22926826268269689
    },
    {
      "problemaID":180,
      "valor": 0.21861788617886185
    },
    {
      "problemaID":207,
      "valor": 0.21265856455636545
    },
    {
      "problemaID":168,
      "valor": 0.20959325865845632
    },
  ]
}
```

Listing 9.2: Ejemplo en JSON de la respuesta del recomendador como servicio para *¡Acepta el reto!*

Se puede observar con detalle el funcionamiento explicado en esta sección complementado con el diagrama de clases y el esquema, de las figuras 9.1 y 9.2.

Capítulo 10

Conclusiones y trabajo futuro

Este trabajo ha estado enfocado en aplicar varios sistemas de recomendación interesantes en el ámbito de las plataformas de jueces en línea.

De esta manera, se ha podido hacer una investigación detallada de modelos de recomendación destacables y se han seleccionado dos de ellos para llevar a cabo una implementación junto con su evaluación de resultados, en una plataforma de juez en línea con buena popularidad.

Estos dos recomendadores realizados han sido el recomendador *bayesiano* y el recomendador de pesos por los *K-Vecinos* más cercanos que, gracias a una BD anonimizada de la plataforma *¡Acepta el reto!*, se ha podido hacer un estudio sobre la eficacia de estos sistemas de recomendación para esta adaptación a jueces en línea.

Además se ha desarrollado una propuesta de comunicación (la API) para estos recomendadores, que permitiría que fuesen usados por un juez en línea como un servicio externo y se ha podido probar para el caso de *K-Vecinos* en la plataforma *¡Acepta el reto!*sin llegar a implementarse de manera directa.

Gracias a esto y como se ha podido observar al final del capítulo 8, se ha llegado a la conclusión de que el recomendador *bayesiano* funciona mejor para estas plataformas que son los jueces en línea, ya que sus resultados mostrados durante el análisis del capítulo mencionado, han dejado claramente que un modelo de recomendación bayesiano resulta más preciso en la selección de los problemas que un usuario intentaría realizar.

Sin embargo, también queda como otra idea de las conclusiones, que el estilo de recomendación del bayesiano es bueno para seleccionar una lista de problemas que un usuario tendría más probabilidades de resolver, por el propio concepto del teorema de bayes, pero no siempre se puede considerar que sea la mejor recomendación ya que el interés de un usuario no se limita a resolver el problema que más probabilidades tenga de ser resuelto, sino que muchas veces los usuarios de estas plataformas buscan retos y la necesidad de algo más difícil y nuevo para ellos.

Aquí entra en juego el estilo de recomendación del otro recomendador, el

recomendador *K-Vecinos*, que intenta buscar usuarios similares al que se está recomendado y fijarse qué otros problemas han seguido haciendo, para así seleccionarlos para el usuario a recomendar. Esta forma de recomendación, es más personalizada, aunque sus resultados no son tan buenos como el bayesiano para la evaluación que se ha aplicado, pero intenta seleccionar esa personalización que el bayesiano no tiene en cuenta, permitiendo ser un recomendador interesante como propuesta de implementación en un juez en línea.

Por otro lado, el trabajo en un principio se planteó como algo que abarcaba más detalles que no se han podido llegar a realizar, por lo que en adelante, se comenta el trabajo futuro y las líneas que puede seguir este proyecto que se ha llevado a cabo. Las siguientes secciones, introducen diferentes líneas a seguir de cara al futuro para cuatro aspectos fundamentales que tiene este proyecto.

10.1. Recomendador Bayesiano

La implementación del *recomendador bayesiano* contada en el capítulo 6 ha sido optimizada para reducir el tiempo de cálculo. Aún así, si se tuviera que recalcular todo cada vez que se hace un nuevo envío, podría ocurrir que en un corto espacio de tiempo se realizaran varios envíos y entre medias de cada uno aún no haya finalizado el recálculo anterior.

Por este motivo se estudiaría a fondo en qué modo afecta un nuevo envío, con el fin de realizar el menor número de cálculos posibles tomando como base los resultados del cálculo anterior.

Además se plantea la posibilidad de acumular los nuevos envíos que se produzcan y realizar el recálculo en determinados momentos del día. Esto provocaría que las recomendaciones no estuvieran 100 % actualizadas, por lo que, habiendo estudiado el efecto de un nuevo envío, habría que analizar si compensa. Eso sí, teniendo cuidado siempre de no recomendar a un usuario un problema que ya haya resuelto, incluso en el tiempo que los cálculos no estén actualizados.

Otra propuesta a realizar es calcular el porcentaje exacto de probabilidad ya que la versión implementada se ahorra calcular el denominador puesto que solo interesaba el orden de probabilidad en un principio. La probabilidad básica nos dice que la suma de la probabilidad de todos los posibles sucesos tiene que ser 1. Por tanto, podemos aprovechar los cálculos realizados ($P(A|B)$ y $P(\neg A|B)$) a los que les falta el denominador y despejar la siguiente fórmula para obtenerlo (de la que resulta que el denominador es tan solo sumar ambos valores):

$$\frac{P(A|B)}{denominador} + \frac{P(\neg A|B)}{denominador} = 1$$

10.2. Recomendador de pesos por los K-Vecinos más similares

El recomendador de pesos por los *K-Vecinos* más similares ha sido implementado tal como se explica en el capítulo 7. Cabe la posibilidad de modificar el algoritmo con la idea de mejorar y pulir aspectos de este, tanto a nivel de optimización, como en su funcionamiento interno.

De este segundo caso (la mejora de los resultados de las métricas), no se puede saber a ciencia cierta si mejoraría el algoritmo sin su evaluación posterior para contrastar resultados.

Una de las modificaciones del algoritmo propuestas en cuanto a maneras de recomendar, es la idea de considerar un nuevo caso en los envíos, que actualmente el algoritmo considera como no realizado. Este caso nuevo se añade haciendo que el recomendador también tenga en cuenta los envíos sin éxito de los usuarios. Esta modificación sería equivalente a la versión RIN del recomendador *bayesiano*.

De esta forma, a la hora de calcular la correlación de un usuario con otros, se podría también añadir a la fórmula los intentos sin éxito que este ha tenido, para fijarse en esas similitudes respecto a otros usuarios, junto con los problemas que han logrado resolver en común.

Gracias a esta nueva manera de calcular la correlación, se podrían encontrar nuevas similitudes o acercar vecinos que de la manera actual los mantiene más alejados, y de esta forma, cambiaría en cierto modo su recomendación.

Otro momento donde poder considerar este nuevo estado en el que se fijaría el recomendador, sería a la hora de calcular los pesos por problema. Así, si uno de los vecinos tiene un problema "intentado y no resuelto" que el usuario a recomendar no haya ni intentado, se le puede bajar el peso de recomendación a ese problema en concreto para dificultar que este escale en el ranking de recomendación.

El factor de intentado y no resuelto que se puede añadir a considerar en el algoritmo de recomendación, es un dato más que enriquece la información con la que este recomendador puede trabajar y genera la curiosidad de ver si la recomendación resultante de este algoritmo mejoraría notablemente, o incluso si pudiese empeorar.

Por otro lado, en el aspecto de la optimización, el recomendador aún puede ser mejorado encontrando nuevos cuellos de botella, ya que el punto de optimización se dejó de trabajar en cuanto los tiempos de recomendación se redujeron considerablemente.

Se podría también manejar un diccionario al parsear el ID de usuario por su posición en la matriz, en lugar de usar un array cuya posición represente la posición de la matriz y su valor el ID, ya que Python maneja de manera óptima este tipo de estructuras y facilitaría la búsqueda de una posición en la matriz en base a su ID de usuario. Actualmente esa búsqueda aplica bús-

queda binaria ya que el array está ordenado por ID de menor a mayor. Esto también facilitaría las inserciones a la hora de actualizar el recomendador, cada vez que existen nuevos usuarios, ya que actualmente se debe desplazar el array para insertar en una posición concreta y mantener el orden por ID. De todas formas este desplazamiento se realiza mediante los métodos que ofrece *Numpy*, así que su orden de complejidad está bastante optimizado.

En cuanto a otras formas de optimización que no implicasen buscar cuellos de botella, se ha planteado la idea de mejorar los resultados de recomendación pre-cacheando la información para tenerla disponible en el momento de la consulta de recomendación, aunque el pre-cacheado generaría la necesidad de nuevas clases que gestionen todo este tema, ya que las consultas de recomendación obtendrían directamente un resultado precalculado para tenerlo más accesible y sin necesidad de espera. Esta idea se descartó gracias a que mejoraron significativamente los tiempos de cálculo por recomendación aplicando el mayor paralelismo posible.

10.3. API de comunicación

La API de comunicación que se ha realizado, no maneja concurrencia entre el servidor de recomendación y las consultas que se hacen a *¡Acepta el reto!* para actualizar la BD del recomendador.

Actualmente su comportamiento es: al ejecutar el servicio de recomendación, la API actualiza la BD del recomendador con los últimos envíos de *¡Acepta el reto!* y posteriormente lanza el servidor. Una vez se lanza el servidor el recomendador no mantiene una actualización periódica.

La idea en este punto de la API es poder añadir concurrencia entre el servidor de recomendaciones y el cliente que hace las actualizaciones, para poder mantener esa actualización de la BD del recomendador sin tener que reiniciar el servicio.

Otra intención original era la de llegar a conseguir una API que se conectase a un servicio que ofreciese *¡Acepta el reto!* para los recomendadores, de tal manera que no actuase de "espía" como hace actualmente, si no que *¡Acepta el reto!* tuviese una parte que le facilitase esta información de actualización ya filtrada, y con la que se comunicase de una manera más cómoda.

10.4. Implementación en *¡Acepta el reto!*

La idea de poder implementar los recomendadores en *¡Acepta el reto!* era sobre todo para poder hacer una evaluación de ambos recomendadores en un entorno de ejecución real, permitiendo analizar con mayor precisión los resultados de recomendación, ya que bajo un entorno de implementación real, el usuario puede ver la recomendación y decidir si hacer el problema

recomendado o no, y en caso de intentarlo, ver si ha tenido éxito o ha fracasado.

Esta implementación habría sido un gran juego en la evaluación de resultados de los recomendadores, debido a su fidelidad con un caso de uso real, ya que se podría entender mejor si de lo que se recomienda a un usuario, es ignorado o intentado, y en este último caso, si ha habido éxito o fracaso en la resolución del ejercicio.

Capítulo 11

Conclusions and future work

This work has been focused on applying several interesting recommendation systems in the field of online judge platforms.

So, it has been possible to make a detailed investigation of importants recommendations models and two of them have been selected to carry out an implementation together with its evaluation of results, in a popular online judge platform.

These two recommenders have been the Bayesian recommender and the recommender of weights for the nearest K-Neighbors that, thanks to an anonymous database of the platform *¡Acepta el reto!* it has been possible to make a study about the effectiveness of these recommendation systems for this adaptation to online judges.

In addition, a communication proposal (the API) has been developed to these recommenders, which would allow them to be used by an online judge as an external service and it has been possible to prove for the case of K-Neighbors in the platform acepta el reto! without being implemented directly on it.

Thanks to this and as it has been observed at the end of chapter 8, has come to the conclusion that the Bayesian recommender works better for these platforms that are online judges, since their results shown during the analysis of the chapter mentioned, have proved that a Bayesian recommendation model is more accurate in the selection of the problems that a user would try to perform.

However, it also remains as another idea of the conclusions, that the Bayesian recommendation style is good for selecting a list of problems that a user would be more likely to solve, by the own concept of the bayes theorem, but can not always be considered that is the best recommendation since the interest of a user is not limited to solve the problem that is most likely to be resolved, but rather many times the users of these platforms look for challenges and the need of something more difficult and new for them.

Here comes into play the recommendation style of the other recommen-

der, the K-Neighbors recommender, who tries to find users similar to the one who is being recommended and note what other problems have been choosen to be selected them for the user to recommend. This form of recommendation, is more personalized, although its results are not as good as Bayesian for the evaluation that has been applied, but try to select that customization that the Bayesian does not take into account, allowing to be a interesting recommendation as a proposal for implementation in a judge in line.

On the other hand, the work at first was proposed as something that encompassed more details that could not be made, so on, the future work is discussed and the lines that can be followed in order to continue this project that has been carried out. The following sections, introduce different lines to follow in the future for four fundamental aspects that this project has.

11.1. Bayesian Recommender

The implementation of the *Bayesian recommender* ,told in chapter 6, has been optimized to reduce calculation time. Still, if you had to recalculate everything every time a new shipment is made, it could happen that In a short space of time several shipments will be made and in between each one has not finished the recalculation yet.

For this reason we will study in depth how a new shipment affects, in order to perform the least number of possible calculations taking as base the results of the previous calculation.

It also raises the possibility of accumulating new shipments that are produce and perform the recalculation at certain times of the day. This would cause the recommendations not to be 100 % up-to-date, so that, having studied the effect of a new shipment, it would be necessary to analyze whether compensates or not. Of course, always taking care not to recommend an user a problem that has already been solved, even in the time that the calculations do not are updated.

Another proposal to make is to calculate the percentage of the probability and the version implemented. The basic probability tells us that the sum of the probability of all possible events must be 1. Therefore, we can take advantage of the calculations made ($P(A|B)$ and $P(\neg A|B)$) a The following formula to obtain it is the following:

$$\frac{P(A|B)}{denominator} + \frac{P(\neg A|B)}{denominator} = 1$$

11.2. Weight recommendation by most Similar K-Neighbors

The recommender of weights for the most similar K-Neighbors has been implemented as explained in chapter 7. It is possible to modify the algorithm with the idea of improving and polishing aspects of this, both at the optimization, as in its internal functioning.

In this second case (the improvement of the results of the metrics), we can't know for sure if it would improve the algorithm without its subsequent evaluation to compare results.

One of the algorithm modifications proposed in terms of ways to recommend, is the idea of considering a new case in the shipments, which currently the algorithm considers as not performed. This new case is adds by having the recommender also consider shipments without success of the users. This modification would be equivalent to the RIN version of the Bayesian recommender.

In this way, when calculating the correlation of a user with others, you could also add to the formula unsuccessful attempts that this has had, to look at those similarities with other users, together with the common problems they have managed to solve.

Thanks to this new way of calculating the correlation, you could find new similarities or bring neighbors that in the current way keeps them further away, and in this way, it would change its recommendation somewhat.

Another moment where we can consider this new state in which would fix the recommender, it would be at the moment of calculating the weights per problem. So, if one of the neighbors has a problem "tried and not resolved"that the user to recommend has not tried, you can lower the weight of recommendation to that problem in particular to make it difficult for it to rise in the recommendation raking.

The factor of attempted and unresolved that can be added to consider in the recommendation algorithm, is a data that enriches the information with which this recommender can work and generates the curiosity to see if the recommendation resulting from this algorithm would improve significantly or, even, if it could get worse.

On the other hand, in the aspect of optimization, the recommender still can be improved by finding new bottlenecks, since the point of optimization was stopped working as soon as the recommendation times they were considerably reduced.

A dictionary could also be handle when parsing the user ID by its position in the matrix, instead of using an array whose position represents the position of the matrix and its value the ID, since python handles in a optimal this type of structures and would facilitate the search for a position in the matrix based on your user ID. Currently that search applies search binary since the

array is sorted by ID from least to greatest. This it would also facilitate the insertions when updating the recommender, every time there are new users, since currently it is necessary to move the array to insert in a specific position and maintain the order by ID. In any case, this displacement is carried out through the methods that *Numpy* offers, so its order of complexity is quite optimized.

As to the other forms of optimization that did not involve looking for collars bottleneck, the idea of improving the recommendation results has been pre-caching the information to have it available at the time of the recommendation consultation, although pre-caching would generate the need of new classes that manage this whole topic, since the queries of recommendation would directly obtain a precalculated result for have it more accessible and without waiting. This idea was ruled out thanks which significantly improved calculation times by recommendation applying the greatest possible parallelism.

11.3. Communication API

The communication API that has been made, does not handle concurrency between the recommendation server and the queries that are made to *¡Acepta el reto!* to update the recommender's BD.

Currently his behavior is: when executing the recommendation service, the API updates the recommender's BD with the latest shipments of Accept the challenge! and then launch the server. Once the server the recommender does not maintain a periodic update.

The idea at this point of the API is to be able to add concurrency between the server of recommendations and the client that makes the updates, for able to maintain that update of the recommender's BD without having to restart the service.

Another original intention was to get an API that connects to a service that offers *¡Acepta el reto!* for the recommenders, in such a way that he did not act as a "spy" as he does now, but that *¡Acepta el reto!* had a part that would provide this update information already filtered, and with which it communicated in a more comfortable way.

11.4. Implementation in *¡Acepta el reto!*

The idea of being able to implement the recommenders in *¡Acepta el reto!* it was mostly to be able to make an evaluation of both recommenders in a real execution environment, allowing to analyze with greater precision the recommendation results, since under an implementation environment real, the user can see the recommendation and decide whether to do the problem

recommended or not, and if you try, see if it has been successful or has failed.

This implementation would have been a great game in the evaluation of results of the recommenders, due to their fidelity with a case of real use, since it could be better understood if what is recommended to a user, is ignored or tried, and in this last case, if there has been success or failure in the resolution of the exercise.

Contribuciones al proyecto

En este capítulo extra se detallan las tareas realizadas por cada integrante del grupo en las distintas fases que ha tenido este proyecto. Estas tareas se van a separar en tres fases importantes: documentación, desarrollo y memoria.

Contribuciones de Pedro Pablo Doménech Arellano

Fase de documentación

- Documentación sobre sistemas de recomendación y varias técnicas de recomendación.
- Documentación sobre métodos de evaluación y entrenamiento.
- Documentación sobre jueces en línea. Recabación de esos.
- Análisis de juez en línea *¡Acepta el reto!*.
- Análisis de juez en línea *Kattis* así como del SR que incorpora.
- Documentación sobre implementación de recomendadores en jueces enlínea.
- Documentación sobre uso de numpy y pandas para análisis de datos.
- Documentación de arquitecturas de servicio externo y API's de comunicación.

Fase de desarrollo

- Elaboración del recomendador bayesiano.
- Cálculo de las métricas de evaluación con los resultados obtenidos del recomendador bayesiano.
- Cálculo de las métricas de evaluación con los resultados obtenidos del recomendador por pesos de los k-vecinos más similares.

- Realización de análisis de resultados del recomendador bayesiano.

Elaboración de la memoria

- Elaboración de introducción en inglés.
- Elaboración de la sección 3.3.5: recomendador bayesiano.
- Elaboración de capítulo 4: jueces en línea.
- Elaboración de capítulo 5: contribuciones en las métricas.
- Elaboración de capítulo 6: recomendador bayesiano.
- Elaboración de capítulo 8: comparación entre recomendadores excluyendo las conclusiones.
- Elaboración de las tablas.
- Elaboración de trabajo futuro para el recomendador bayesiano.
- Lecturas de corrección. Corrección de fallos mayores y menores.
- Corrección sobre feedback de los tutores.

Contribuciones de Alfonso Soria Muñoz

Fase de documentación

- Documentación sobre sistemas de recomendación y varias técnicas de recomendación.
- Documentación sobre métodos de evaluación y entrenamiento.
- Documentación sobre jueces en línea. Recabación de estos (Kattis, UVa..).
- Análisis de juez en línea *¡Acepta el reto!*.
- Análisis de juez en línea *Kattis* así como del SR que incorpora.
- Documentación sobre implementación de recomendadores en jueces en-línea.
- Documentación sobre uso de numpy y pandas para análisis de datos.
- Documentación de arquitecturas de servicio externo y API's de comunicación.

Fase de desarrollo

- Elaboración del recomendador por pesos de los k-vecinos más similares.
- Entrenamiento del recomendador por pesos de los k-vecinos más similares para dejar preprocesados los datos y poder obtener sus métricas.
- Realización de análisis de resultados del recomendador por pesos de los k-vecinos más similares.
- Realización de API de comunicación genérica para ambos modelos de recomendación. Se ha analizando que información proporciona *¡Acepta el reto!* para que el recomendador pueda nutrirse de esta y mantenerse actualizado, y se ha creado un arquitectura de comunicación para que el recomendador pueda funcionar como servicio web.

Elaboración de la memoria

- Elaboración de introducción.
- Elaboración de notas de los autores.
- Elaboración de conclusiones, gran parte de trabajo futuro (todas las secciones menos bayes)
- Elaboración de conclusiones en su versión en inglés.
- Elaboración de resumen.
- Elaboración de abstract (versión en inglés del resumen).
- Elaboración de capítulo 4: sistemas de recomendación, excluyendo la sección del recomendador bayesiano.
- Elaboración de capítulo 5: evaluación de recomendadores.
- Elaboración de capítulo 7: recomendador de pesos por los k-vecinos más similares.
- Elaboración de conclusiones del capítulo 8: comparación entre recomendadores.
- Elaboración de capítulo 9: implementación como servicio externo.
- Elaboración de las figuras vectoriales (grafos, diagramas uml, etc).
- Elaboración de las gráficas.
- Lecturas de corrección. Corrección de fallos mayores y menores.
- Corrección sobre feedback de los tutores.

Bibliografía

No es la conciencia del hombre la que determina su ser, sino, por el contrario, el ser social es lo que determina su conciencia

Karl Marx, filósofo.

Aggarwal, C. C. *Recommender Systems*. Springer, 2016. Disponible en `https://www.springer.com/gp/book/9783319296579`.

Bellogín, A. y de Vries, A. P. Understanding similarity metrics in neighbour-based recommender systems. En *Proceedings of the 2013 Conference on the Theory of Information Retrieval*, ICTIR '13, páginas 13:48–13:55. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2107-5.

Clark, P. y Boswell, R. Rule induction with cn2: Some recent improvements. En *Machine Learning — EWSL-91* (editado por Y. Kodratoff), páginas 151–163. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. ISBN 978-3-540-46308-5.

Furht, B. *Handbook of Social Network Technologies and Applications*. Springer US, 2010. ISBN 978.

Hall, P., Park, B. U. y Samworth, R. J. Choice of neighbor order in nearest-neighbor classification. *Ann. Statist.*, vol. 36(5), páginas 2135–2152, 2008.

Herlocker, J. L., Konstan, J. A., Terveen, L. G. y Riedl, J. T. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, vol. 22(1), páginas 5–53, 2004. ISSN 1046-8188.

Jimenez-Diaz, G., Gómez-Martín, P., Gómez-Martín, M. y Sánchez-Ruiz, A. Similarity metrics from social network analysis for content recommender systems. *AI Communications*, vol. 30, páginas 1–12, 2017.

McKinney, W. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2011. ISBN 978-1-449-31979-3.

REVILLA, M. A., MANZOOR, S. y LIU, R. Competitive learning in informatics: The uva online judge experience. *Olympiads in Informatics*, vol. 2(10), páginas 131–148, 2008.

SANZ MOLINA, A. y MARTIN DEL BRIO, B. *Redes Neuronales y Sistemas Borrosos.*. Ra-Ma editorial, 2006. Disponible en `http://www.ra-ma.es/libros/REDES-NEURONALES-Y-SISTEMAS-BORROSOS-3-EDICION/241/978-84-7897-743-7`.

SÁNCHEZ-RUIZ, A., JIMENEZ-DIAZ, G., GÓMEZ-MARTÍN, P. y GÓMEZ-MARTÍN, M. Case-based recommendation for online judges using learning itineraries. páginas 315–329. 2017. ISBN 978-3-319-61029-0.

WIKIPEDIA (Deep learning). Disponible en `https://es.wikipedia.org/wiki/Aprendizaje_profundo`.

WIKIPEDIA (Elo rating system). Entrada: "Elo rating system". Disponible en `https://en.wikipedia.org/wiki/Elo_rating_system` (último acceso, Marzo, 2019).

WIKIPEDIA (k-nearest neighbors). Disponible en `https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm`.

ZHENG, Y., MOBASHER, B. y BURKE, R. Context recommendation using multi-label classification. 2014.

Lista de acrónimos

AC ..... *Accepted*

API ..... *Application Programming Interface*

BD ..... *Base de datos*

FN ..... *False Negative*

FP ..... *False Positive*

RRNN ..... *Redes neuronales*

SR ..... *Sistema de Recomendación*

TN ..... *True Negative*

TP ..... *True Positive*