Trabajo de Fin de Grado

Grado en Ingeniería Informática

Análisis e implementación de un sistema recomendador en el ámbito de GitHub

Analysis and implementation of a recommender system in the scope of GitHub

Adrián Gutiérrez Álvarez

La Laguna, 30 de junio de 2017

D. **Dagoberto Castellanos Nieves**, con N.I.F. 79.234.766-L profesor Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Análisis e implementación de un sistema recomendador en el ámbito de GitHub."

ha sido realizada bajo su dirección por D. **Adrián Gutiérrez Álvarez**, con N.I.F. 43.380.435-C.

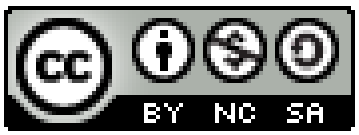
Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 30 de junio de 2017.

Agradecimientos

En primer lugar, quería agradecer a **D. Dagoberto Castellanos Nieves**, tutor del trabajo de fin de grado, por ayudarme en todo lo posible a lo largo de la realización de este proyecto.

En segundo lugar, me gustaría dar las gracias a **Luz Marina Moreno de Antonio**, coordinadora del TFG, por la gran planificación elaborada junto a la creación de una plantilla simple y de calidad para redactar esta memoria.

Licencia



Resumen

El impacto en la investigación de redes sociales es cada vez más grande debido al crecimiento exponencial de datos existentes año tras año. El pensamiento de un grupo de personas con características similares, cómo actúan las personas frente a un contenido viral y la distribución de la información por todo el planeta pueden ser algunos de los puntos importantes para acreditar a cualquier investigador a obtener patrones o resultados sobre el raciocinio humano.

El objetivo de este trabajo consistirá en el análisis y elaboración de un sistema recomendador. Para ello, se explorará y desarrollarán métodos utilizando la API de GitHub. Estos métodos darán como resultado un prototipo software capaz de recomendar usuarios con características similares aparte de asesorar a programadores capaces de resolver errores los cuales disponen ciertos proyectos.

Palabras clave: Sistema de recomendación, GitHub, API, redes sociales, GUI.

Abstract

The impact of investigations in social networks is increasing year after year due to the exponential data growth. What a group of people think, how people act in relation to viral content on the Internet and how information is distributed around the world are some of the most important points in order let any researcher obtain patterns or results about human reasoning.

The purpose of this project will be the analysis and elaboration of a recommendation system. To do this, we will explore and develop methods using GitHub API. These approaches are going to result in a software prototype that will be able to recommend users with similar characteristics, apart from advising programmers who can resolve mistakes that may appear in some projects.

Keywords: *Recommendation system, GitHub, API, Social Networks, GUI.*

Índice General

Índice de figuras

Índice de tablas

Capítulo 1. Introductorio

1.1 Introducción a las Redes Sociales

Dado el alto nivel tecnológico que se acapara hoy en día, la gran mayoría de los usuarios invierten varias horas de su tiempo diario a leer, emplear, crear y compartir información en las redes sociales. Es muy importante destacar la gran cantidad de datos que se originan en ellas, siendo tan elevados que muchas personas las emplean para investigar y comprender el comportamiento del ser humano.

La Web 2.0 ha focalizado la importancia de la información, no en unos pocos expertos en un tema, sino en una multitud de opiniones vertidas por usuarios a través de diversos medios en las redes sociales. Debido a ello, han cobrado un mayor interés los sistemas que son capaces de determinar qué es lo que piensan los usuarios sobre un determinado concepto, agregando diferentes fuentes de datos y aplicando cálculos de polaridad de las opiniones, que permiten determinar y comparar esos conceptos con otros similares [1].

Como podemos observar, el impacto de investigación en las redes sociales es cada vez más grande debido al crecimiento exponencial de datos existentes año tras año. El pensamiento de un grupo de personas con características similares, cómo actúan las personas frente a un contenido viral y la distribución de la información por todo el planeta son algunos de los puntos importantes para acreditar a cualquier investigador a obtener patrones o resultados sobre el raciocinio humano.

Dentro de la Web 2.0 existen empresas que buscan ser la referencia mundial debido a la popularidad y al número de ingresos tan elevados que pueden llegar a conseguir en un período relativamente corto. Es por ello que actualmente dichas compañías pretenden innovar con nuevo contenido, nuevas funciones o

nuevas formas de interacción con los demás usuarios para convertir su red social en la principal atracción para los consumidores. Por ello, es muy importante recaudar toda la información posible para así mejorar la interfaz en función al usuario, ya que es la única manera efectiva de comprender las carencias y procedimientos a ejecutar para llevar a cabo el objetivo deseado.

1.2 Sistemas de recomendación

¿Cuántas veces hemos tomado una mala decisión a causa de tanta información disponible en la red? Los Sistemas de Recomendación (SR) son útiles para tratar la sobrecarga de información de la web. Cada técnica de SR tiene un propósito diferente y es importante elegir la adecuada con el fin de obtener resultados óptimos [2].

Cada día el número de usuarios de internet aumenta exponencialmente junto a la cantidad de información circulando por la red, siendo prácticamente imposible tomar decisiones acertadas acerca de los intereses de dichos usuarios. Una de las soluciones prácticas para resolver este problema es el uso de los sistemas recomendadores ya que, con técnicas predictivas, características de las personas o clasificaciones para generar un "ranking" podemos dar soluciones concretas y precisas a los problemas presentados por los usuarios.

Los sistemas recomendadores aparecen en varias áreas de la informática como por ejemplo el comercio (eBay, Amazon, AliExpress, etc.).

Los sistemas de comercio electrónico actuales requieren de manera permanente proveer personalización en la presentación de sus contenidos. En este sentido, los sistemas de recomendación realizan sugerencias y facilitan información acerca de los ítems disponibles en el sistema. Actualmente se dispone de una gran cantidad de métodos, incluyendo técnicas de minería de datos, que pueden utilizarse para la personalización en dichos sistemas [3].

Ilustración 1.1 Ejemplo de aplicaciones web que utilizan SR [4]

1.3 Objetivo principal

La función de este proyecto es, principalmente, destacar la relación de información que digiere el usuario en su día a día con contenido recomendado, conociendo nuevas fuentes de información gracias a la "Interfaz de Programación de Aplicaciones" (API) y así mostrar un mayor interés en la red social. Este procedimiento puede garantizar un mayor número de horas dedicadas al uso de la aplicación, una mayor interacción con otras partes del mundo y facilidad de uso, algo muy importante a día de hoy dado el alto uso de la tecnología en personas con distintos niveles de habilidad.

Además, este contenido recomendado será capaz de ayudar al usuario con sus problemas, aportando una serie de sugerencias de personas capaces de resolver conflictos relacionados.

En resumen, los objetivos que pretende conseguir este proyecto son los siguientes:

- Estudio del estado del arte relacionado con las redes sociales, así como recomendaciones a diferentes fuentes de información.
- Búsqueda, definición y creación de métodos de recomendación usando fuentes de información acorde a los medios de comunicación del usuario en GitHub.
- Validación de los métodos de recomendación con pruebas y datos del usuario.
- Obtención de un prototipo software en lenguaje R.
- Creación de la documentación técnica, entregables y la memoria del proyecto.

1.4 GitHub

Básicamente, GitHub es una red social de desarrolladores con muchas herramientas que ayudan a mantener el trabajo colaborativo entre desarrolladores [5].

GitHub ofrece toda la funcionalidad de *Git* (software de control de versiones) e integra diversas herramientas de control de acceso, colaboración, trazabilidad, gestión de tareas y control de proyectos. Recientemente, educadores dentro y fuera del mundo académico relacionado con la Informática han comenzado a usar GitHub en sus cursos [6].

Además de ello, en la API disponemos de otras características como seguir a usuarios o repartir "estrellas" en repositorios que consideremos importantes o de gran calidad. Esto último es muy importante para nuestro sistema de recomendación, ya que podemos utilizar todos esos datos para clasificar a los usuarios y analizar su comportamiento y habilidades para resolver ciertos problemas.

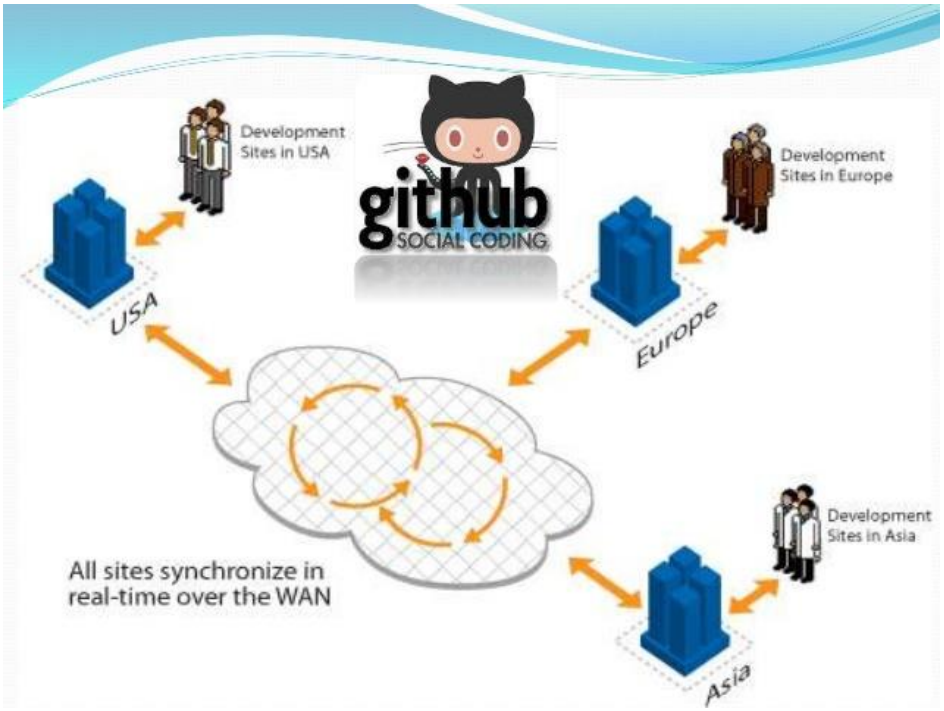


Ilustración 1.2 Diagrama de ejemplo de uso en GitHub [7]

Uno de los beneficios de GitHub es que tienen disponible una segunda interfaz de programación para favorecer el acceso a los datos llamada "REST API v3". Con ella, podemos hacer hasta 5000 consultas por hora si estamos registrados como desarrolladores. Es muy intuitiva y fácil de usar, ya que todos los datos públicos de usuarios están en formato "JSON" además de ciertos parámetros para filtrar los resultados.

1.5 Lenguaje R

Para crear el software que resolverá los objetivos planteados, se ha decidido utilizar el lenguaje R junto a su interfaz, R Studio.

R es un entorno de programación, análisis estadístico y generación de gráficos distribuido bajo licencia GNU. Es un poderoso aliado para la investigación y una excepcional herramienta de trabajo para la docencia. Está constituido por más de 1.400 paquetes integrados con los que es posible ejecutar simples análisis descriptivos o aplicar los más complejos y novedosos modelos formales [8].

Este lenguaje cumple las expectativas para analizar los datos de la API de GitHub, utilizar algoritmos para clasificar, crear un "ranking", detallar las características de los usuarios y, entre otras funciones, crear nuestro sistema de recomendación personalizado.

1.6 Aprendizaje automático

El aprendizaje automático (Machine Learning) es una disciplina científica que se dedica a buscar patrones en un conjunto muy grande de datos. Se utiliza mucho en inteligencia artificial, bases de datos, aprendizaje automático, etc.

La existencia de voluminosas bases de datos conteniendo grandes cantidades de datos, que exceden en mucho las capacidades humanas de reducción y análisis a fin de obtener información útil, actualmente son una realidad en muchas organizaciones [9].

Esto se puede resolver mediante KDD (Knowledge Discovery in Databases), un área de la computación capaz de obtener conocimiento de grandes cantidades de datos imposibles de procesar por los humanos. En el caso de GitHub, podemos analizar el comportamiento de los usuarios con el fin de poder clasificarlos acorde a una serie de características comunes con otros programadores y obtener similitudes a la hora de crear nuestro SR.

1.7 Inteligencia colectiva

En informática, la inteligencia colectiva es una especie de inteligencia procedente de las aportaciones de usuarios en la red. En la web 2.0 esta inteligencia es muy importante dado que los creadores de contenido tienen un papel fundamental para que aparezcan nuevas formas de relación entre usuarios. Tanto consumidores como creadores forman parte, en su mayoría, de internet, con gran facilidad para aportar conocimientos y contenidos de manera masiva posibilitando, entre otras cosas, el análisis de datos.

El principio fundamental que se esconde detrás del éxito de los gigantes nacidos en la era de la Web 1.0 que han sobrevivido para liderar la era de la Web 2.0 parece ser éste, que han abrazado el poder de la web para explotar inteligencia colectiva [10]:

- Los hipervínculos constituyen los cimientos de la Web 2.0. A medida que se agregan nuevos contenidos a la red, éstos se enlazan a páginas ya existentes. De forma similar a la sinapsis cerebral, la red de conexiones crece orgánicamente como resultado de la actividad de todos los usuarios de la Internet [11].

- Otro ejemplo es Yahoo! Answers (http://answers.yahoo.com/), pues sirve como un espacio donde la gente hace preguntas no a un gurú que teóricamente lo debe saber todo, sino que la gente se pregunta entre sí y las respuestas que obtiene vienen no de un equipo de expertos, sino de otras personas [12].
- El PageRank es el método inicial de cálculo que usaron los fundadores de Google para clasificar las páginas web según su importancia. Hasta ahora, su algoritmo viene siendo objeto de continuas mejoras y adaptaciones a diversos problemas [13].
- El servicio de subastas virtuales eBay, por ejemplo, se cimienta en la opinión manifestada por los usuarios al completar una transacción. Aunque cada usuario valora de forma individual cómo se ha desarrollado la operación en la que ha participado personalmente, es el porcentaje final, determinado por la cantidad de transacciones valoradas positivamente, el que configura el índice de confianza asignado a un determinado usuario [14].

Como podemos ver, la inteligencia colectiva en la red se forma con todas las aportaciones de los usuarios. Gracias a ella, se pueden clasificar, identificar, almacenar datos para estudiarlos, aplicar técnicas y diferentes algoritmos para dar respuesta a miles de preguntas.

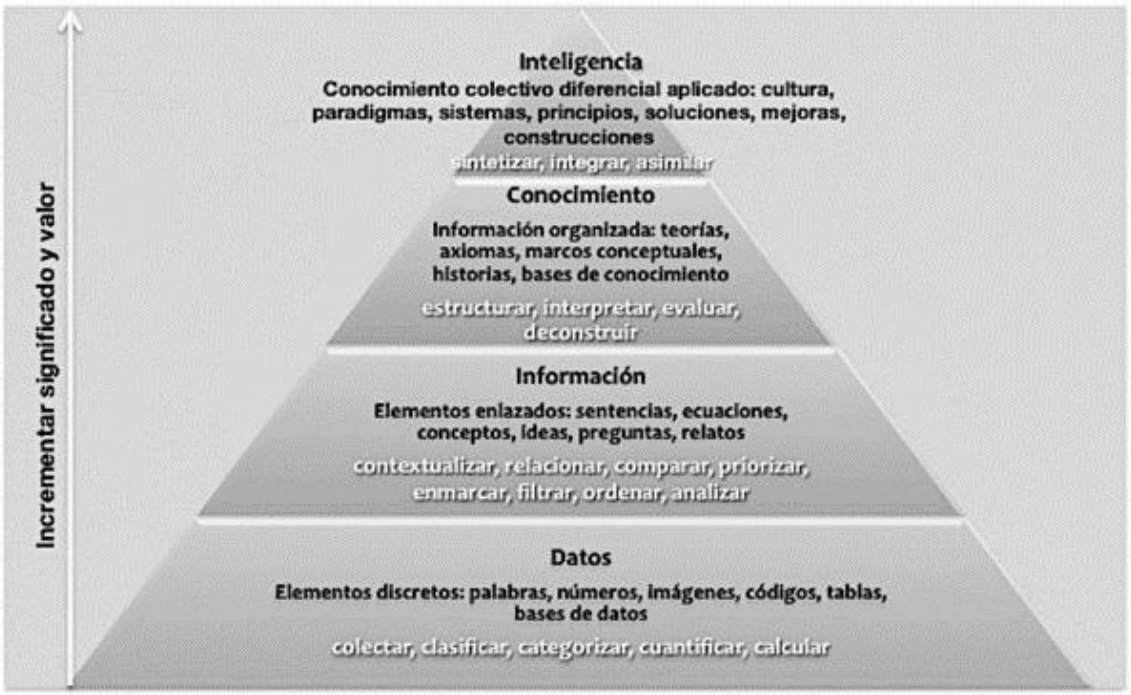


Ilustración 1.3 Transformación de los datos a la inteligencia colectiva [15]

En la figura 1.3 se aprecia una pirámide que muestra cómo se puede transformar la inteligencia colectiva desde los datos. Se necesita obtener información como relaciones, comparaciones, filtros... para obtener el conocimiento necesario de estructurar esa información y transformarla en conocimiento.

Capítulo 2.
Fuentes de datos

En el capítulo anterior se ha hecho una pequeña introducción al aprendizaje automático, el lenguaje R y GitHub, entre otros.

Para cumplimentar los objetivos de este proyecto necesitamos extraer los datos públicos de usuarios de GitHub. Lo primero que debemos hacer es obtener una forma de representar los diferentes usuarios del API y una forma muy sencilla es mediante un diccionario anidado. Para la clasificación del diccionario, necesitamos definir una métrica acorde a los objetivos propuestos que exprese las diferentes características de los usuarios.

2.1 Métrica propuesta

Antes de realizar el software para la obtención de datos, debemos tener una métrica acorde a las ideas planteadas en este proyecto. La métrica propuesta consiste en definir y clasificar los usuarios en estos campos para crear el diccionario:

- Número de seguidores.
- Número de seguidos.
- Número de repositorios públicos.
- Lenguajes de programación usados en los repositorios.
- Número de "Gists".
- Lenguajes de programación usados en "Gists".
- Número de estrellas asignadas a proyectos.
- Compañía.
- Fecha de la última aportación a la red.

El **número de seguidores** de un usuario en GitHub es un elemento a tener en cuenta dado que podemos determinar si la persona es famosa o influenciable por sus repositorios, por algún proyecto en particular, por sus contribuciones... o, por el contrario, es una persona no influenciable, no se tiene en cuenta como modelo a seguir, no realiza contribuciones, etc. Esta información no es la más importante a la hora de realizar las recomendaciones (que veremos en los siguientes capítulos), pero es un elemento que puede ayudar a aportar mayor precisión a la hora de realizar la clasificación.

El **número de seguidos** es un componente importante para el sistema porque podemos recoger información acerca de los usuarios que sigue una persona en GitHub. Esta información es útil para analizar características similares entre los usuarios seguidos y determinar un patrón o modelo a tener en cuenta.

El **número de repositorios públicos** junto al **lenguaje de programación usado en los repositorios** son quizás los elementos más importantes para la clasificación de los usuarios. El primer componente nos da información acerca de los proyectos que ha subido a la web, así como determinar si el usuario aporta gran cantidad de proyectos o de lo contrario, su contribución es prácticamente nula. El segundo componente es una ampliación del primero puesto que permite conocer los lenguajes de programación usados en cada uno de los repositorios. Con ello, podemos determinar si el usuario es un programador multilenguaje o de lo contrario, tiene un alto nivel de programación en alguno en particular, entre otras características.

El **número de "Gists"** y el **lenguaje de programación usado** son elementos secundarios sobre los repositorios que el usuario sube a la web, pero es una información útil para analizar y complementar las características del lenguaje de programación y aportaciones del programador.

El **número de estrellas asignadas a proyectos** es una variable usada en GitHub para que el usuario premie, según su criterio, los proyectos de otros usuarios. En nuestro sistema, este elemento es una aportación de información acerca de los gustos y criterio del programador a analizar.

La **compañía** es el elemento menos importante en nuestro sistema recomendador, simplemente nos dará información sobre si el usuario trabaja para alguna empresa o es independiente.

Por último, la **fecha de la última aportación a la red** nos proporcionará la actividad más reciente del usuario en GitHub, puesto que un usuario inactivo tiene un grado de importancia más bajo frente a un usuario en constante actividad.

2.2 Conexión a GitHub

Una vez hemos definido la métrica a usar, el siguiente paso es conectar la API de GitHub con R Studio (su interfaz de programación) para la obtención de datos. Lo primero que debemos hacer es acceder a GitHub e iniciar sesión con nuestra cuenta de usuario. Una vez hecho esto, nos dirigimos al apartado *settings* o a la dirección "https://github.com/settings/developers" y registramos una aplicación nueva. Cuando tengamos rellenados los datos de la aplicación, se generará un "token" de acceso a los datos de la API y podremos acceder a los datos desde R Studio.

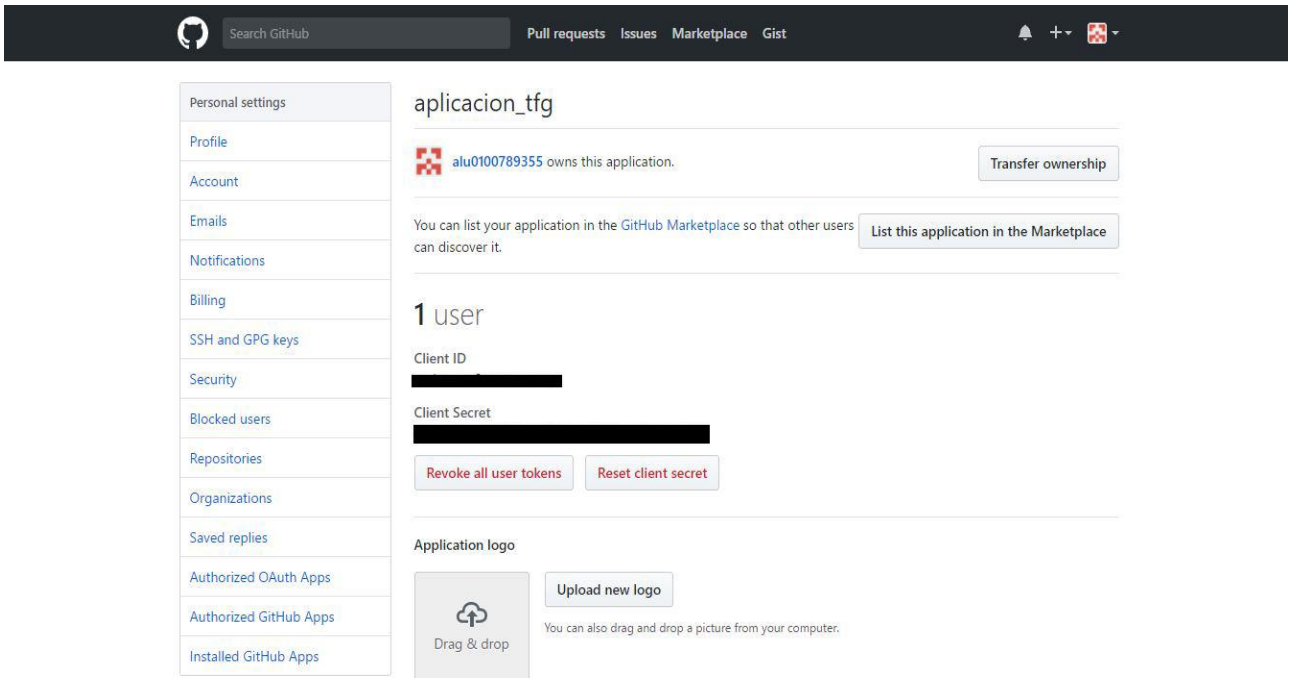


Ilustración 2.1 Aplicación de autorización creada para la conexión a GitHub

En la figura 2.1 se muestra una captura de una aplicación de autorización (OAuth) creada con éxito con el ID y clave para utilizar en cualquier lenguaje de programación que acepte peticiones por HTTP.

GitHub trabaja al completo en su REST API con archivos de tipo *JSON* haciéndola muy intuitiva de usar. Todo el acceso a dicha API se hace mediante

el protocolo *HTTPS* y se accede en https://api.github.com. Podemos añadir parámetros a las consultas como por ejemplo "per_page", "repos", "issues" o "users" para especificar más los datos a obtener de los usuarios.

2.3 Librerías y conexión en R

Para poder usar correctamente la API de GitHub en R Studio necesitamos utilizar ciertos paquetes de R para la conexión, tratamiento de datos y generación de archivos.

El primer paquete o librería que necesitamos para la conexión https con la API se llama "**httpuv**": Httpuv proporciona soporte de bajo nivel de socket y protocolo para el manejo HTTP y WebSocket directamente desde dentro de R. Es como un bloque de construcción principalmente para otros paquetes como, por ejemplo, el paquete "httr".

La segunda librería a utilizar se llama "**httr**", una herramienta muy útil para consultas http como "*get()*", "*post()*", "*authenticate()*", etc. En nuestro caso, la usaremos para las consultas a la API mediante la función *get()* y para autenticarnos en la *REST API V3*.

Para el tratamiento de los datos usaremos el paquete "**jsonlite**", un analizador y generador de archivos *JSON*. El paquete ofrece flexibilidad, robustez, herramientas de rendimiento para trabajar con JSON en R y es muy potente para construir tuberías e interactuar con una API web.

Otro de los paquetes a utilizar se llama "**pipeR**". Esta librería es capaz de unir diferentes variables, cadenas de caracteres y otras fusiones con la finalidad de hacer más interactivo el uso de R. Es un paquete muy importante para conseguir la automatización de nuestro programa, ya que podemos unir variables con cadenas de caracteres y aplicar diferentes bucles para obtener datos.

Por último, un paquete útil a la hora de hacer consultas al api es "**rlist**", una librería que facilita el uso de las listas, así como el uso de filtros, búsquedas, grupos de datos, "mapping", etc. En nuestro caso, usaremos este paquete para filtrar los datos obtenidos de los *JSON* y facilitar el manejo de datos.

Una vez se hayan instalados estos paquetes, se puede empezar a crear el programa para que sea capaz de conectarse al api de GitHub y obtener los datos deseados. En la siguiente figura podemos ver un ejemplo sencillo de una conexión:

```
# Can be github, linkedin etc depending on application
oauth_endpoints("github")

# Change based on what you
myapp <- oauth_app(appname = "aplicacion_tfg",
                   key = "AQUI VA EL ID DE LA APP",
                   secret = "AQUI VA EL KEY DE LA APP")

# Get OAuth credentials
github_token <- oauth2.0_token(oauth_endpoints("github"), myapp)

# Use API
gtoken <- config(token = github_token)
```

Ilustración 2.2 Ejemplo de conexión al API de GitHub en R

Como podemos observar, con insertar los datos de la aplicación creada en GitHub a una variable local y crear el token, tendremos la conexión creada y capacidad para utilizar todas las funciones de la api como, por ejemplo "*get()*".

2.4 Script para la obtención de datos

Una vez se ha establecido la conexión entre el api de GitHub y nuestro código en R, el siguiente paso es obtener los usuarios del api. Si vamos a la documentación de REST API V3, podemos ver que en https://api.github.com/users tenemos los 30 primeros usuarios de GitHub ordenados por su id. A esta consulta se le pueden hacer variaciones para obtener más resultados, como por ejemplo añadir los parámetros "*per_page*" y "*since*". El primer parámetro se utiliza para indicar al api el número de usuarios que deseamos obtener por página (el límite es de 100 usuarios), y el segundo parámetro se utiliza para continuar la consulta a partir del id de usuario que deseemos. Un ejemplo de una consulta aplicando estos dos parámetros sería así:

https://api.github.com/users?since=237&per_page=100

En este ejemplo, le indicamos al api que buscamos los siguientes 100 usuarios a partir del id de usuario número 237 y se genera lo siguiente:

```
[
  {
    "login": "superphly",
    "id": 238,
    "avatar_url": "https://avatars1.githubusercontent.com/u/238?v=3",
    "gravatar_id": "",
    "url": "https://api.github.com/users/superphly",
    "html_url": "https://github.com/superphly",
    "followers_url": "https://api.github.com/users/superphly/followers",
    "following_url": "https://api.github.com/users/superphly/following{/other_user}",
    "gists_url": "https://api.github.com/users/superphly/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/superphly/starred{/owner}{/repo}",
    "subscriptions_url": "https://api.github.com/users/superphly/subscriptions",
    "organizations_url": "https://api.github.com/users/superphly/orgs",
    "repos_url": "https://api.github.com/users/superphly/repos",
    "events_url": "https://api.github.com/users/superphly/events{/privacy}",
    "received_events_url": "https://api.github.com/users/superphly/received_events",
    "type": "User",
    "site_admin": false
  },
  {
    "login": "swhitt",
    "id": 239,
    "avatar_url": "https://avatars1.githubusercontent.com/u/239?v=3",
    "gravatar_id": "",
    "url": "https://api.github.com/users/swhitt",
    "html_url": "https://github.com/swhitt",
    "followers_url": "https://api.github.com/users/swhitt/followers",
    "following_url": "https://api.github.com/users/swhitt/following{/other_user}",
    "gists_url": "https://api.github.com/users/swhitt/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/swhitt/starred{/owner}{/repo}",
    "subscriptions_url": "https://api.github.com/users/swhitt/subscriptions",
    "organizations_url": "https://api.github.com/users/swhitt/orgs",
    "repos_url": "https://api.github.com/users/swhitt/repos",
    "events_url": "https://api.github.com/users/swhitt/events{/privacy}",
    "received_events_url": "https://api.github.com/users/swhitt/received_events",
    "type": "User",
    "site_admin": false
  },
  {
    "login": "bryanl",
    "id": 240,
    "avatar_url": "https://avatars0.githubusercontent.com/u/240?v=3",
```


Ilustración 2.3 Fragmento de datos en formato JSON en la api de GitHub

La figura 2.3 representa una porción del resultado de la petición creada en el ejemplo en formato JSON. Las variables declaradas en el archivo tienen como resultado, en su mayoría, un enlace a otro fichero JSON con datos específicos sobre ese campo. El siguiente paso consiste en usar el paquete "jsonlite" para leer y transformar los datos obtenidos por el api, simplemente usando la función "*fromJSON()*" y pasándole por parámetro la variable que contenga el contenido mencionado anteriormente. El resultado de aplicar esta función es un "*data.frame*", un objeto de tipo matriz que permite almacenar variables de distinto tipo (caracteres, enteros, null, true/false…) muy típico en el lenguaje R.

Una vez visto como obtener los datos de una petición mediante la función "*get()*", implementaremos un bucle que recorrerá todos los elementos que definimos en la métrica propuesta además de crear el *data.frame* correspondiente. Es muy importante recorrer todas las URL que contengan los campos necesarios en la métrica como "*url*", "*repos_url*" o "*starred_url*", dado que esos enlaces contienen campos correspondientes a extraer. Cuando se hayan creado los bucles correspondientes y se haya generado el *data.frame*, obtendremos nuestro "dataset" de usuarios para empezar la clasificación.

	usuario	lugar	nombre	compania	repositorios	lenguajes	estrellas	lenguaje_estrellas	gists	seguidores	siguiendo	ult_actualizacion
1	sevenwire	WI & CA	Sevenwire	null	8	Ruby	0		1	0	0	2015-07-29 08:04:20
2	brandonarbini	Costa Mesa, CA	Brandon Arbini	null	12	Ruby	13	Ruby	2	18	1	2017-05-30 03:18:01
3	al3x	Portland, Oregon, USA	Alex Payne	null	48	Ruby, Scala	237	Clojure, JavaScript, Ruby, Scala, Go, Java, Python, Sh...	44	1024	54	2017-04-26 19:26:14
4	toolmantim	Melbourne, Australia	Tim Lucas	null	56	Ruby, JavaScript, Shell, CSS	101	Ruby, JavaScript, Python	90	294	64	2017-06-03 22:09:37
5	nicksieger	Minneapolis, MN, USA	Nick Sieger	@sportngin	70	Ruby	220	Ruby, Java, Clojure, Emacs Lisp, JavaScript, Python	218	211	14	2017-01-13 17:49:56
6	jicksta	San Francisco, California	Jay Phillips	@neighborly	49	Ruby, JavaScript	300	JavaScript, Go, Python, Scala, Ruby, Java, C++, Shell, ...	39	78	16	2017-05-12 10:48:06
7	joshsusser	San Francisco, CA	Josh Susser	null	16	Ruby	66	Ruby, JavaScript	38	276	10	2017-05-23 09:55:04
8	jcrosby	SF Bay Area, CA	Jon Crosby	Medium	27	Ruby, JavaScript	96	Ruby, JavaScript, Objective-C	13	144	17	2017-05-18 10:08:31
9	thewoolleyman	Tucson, AZ	Chad Woolley	Pivotal Labs - http://pivotallabs.com	137	Ruby, Shell	300	Ruby, JavaScript, Shell, Elm, Go, Haskell, Java, Objecti...	50	72	36	2017-06-01 02:59:07
10	technicalpickles	Savannah, GA	Josh Nichols	@github	217	Ruby, JavaScript, CoffeeScript, Python, Shell	300	Ruby, JavaScript, Python, CoffeeScript, Vim script, Go...	294	610	66	2017-06-02 02:23:15
11	ryanb	Southern Oregon	Ryan Bates	RailsCasts	52	Ruby	300	Ruby, JavaScript, Shell	60	6922	94	2017-05-30 06:02:01
12	cnix	Virginia Beach, VA	Claude Nix	JibeSet	21	Ruby	36	Ruby, JavaScript	22	20	15	2016-02-26 22:34:28
13	tpitale	Saint Louis, MO	Tony Pitale	null	45	Ruby	300	Ruby, Rust, JavaScript, Elixir, CSS, C, Objective-C, Pyt...	99	79	68	2017-05-31 09:47:57
14	atduskgreg	Los Angeles, CA	Greg Borenstein	Riot Games	223	Ruby, Java, C++, JavaScript, Processing, Objective-C, ...	300	Ruby, C++, Java, JavaScript, C, Python, Objective-C	526	506	39	2017-05-22 01:37:12

Ilustración 2.4 Fragmento de un "Data.frame" en R creado con la métrica propuesta

En la figura 2.4 se observa una pequeña parte de un diccionario de usuarios en GitHub, el cual cumple con la métrica propuesta en el punto 2.1. Se puede obtener el número de usuarios deseado con tan sólo indicar las iteraciones a realizar del bucle que, en mi caso, fue de 1000 programadores. Una vez obtenido nuestro diccionario, el siguiente paso consistirá en preprocesar los datos obtenidos y utilizar un modelo coherente para nuestro ámbito (programación).

Capítulo 3.
Modelo y procesamiento de datos

En los siguientes apartados se preprocesarán los datos obtenidos del diccionario de usuarios en GitHub (capítulo 2) y se elegirá un modelo para realizar recomendaciones acordes a los objetivos planteados en el proyecto.

3.1 Elección del modelo

Antes de realizar el preprocesamiento de datos, necesitamos conocer el modelo con el que vamos utilizar nuestro diccionario para obtener recomendaciones. El modelo que vamos a utilizar es el **filtrado colaborativo** porque es una técnica que consiste en hacer predicciones automáticas mediante los gustos o intereses de los usuarios. Es muy importante utilizarlo con el diccionario de usuarios en GitHub puesto que el objetivo a cumplir consiste en realizar recomendaciones automáticas mediante la similitud entre las características de los programadores.

Existe un paquete en R que contiene, entre otras características, dicho modelo y se llama "Recommenderlab".

RecommenderLab, es un *framework* que provee algoritmos básicos de filtrado colaborativo y permite al usuario desarrollar sus propios métodos. Este paquete forma parte del entorno de programación para análisis estadístico y gráfico de R. Su fuerte es el manejo consistente y eficiente de datos, facilitando la incorporación de algoritmos, experimentación y evaluación de resultados [16].

Este paquete utiliza funciones y algoritmos de filtrado colaborativo a destacar como "*dissimilarity*", "*Recommender*", "*evaluate*" y "*normalize*". En nuestro caso, usaremos la función "*Recommender*" dado que crea un modelo de

recomendación con tan sólo añadir un "dataset" (el diccionario de usuarios) y la elección de uno de sus métodos. Entre los más importantes se encuentran:

- **UBCF**: Este método utiliza el algoritmo "KNN" (vecinos más cercanos) y consiste en encontrar los usuarios con características más similares en cuanto a distancia.
- **IBCF**: Este método parte al igual que UBCF y sólo se diferencia en que este procedimiento encuentra los *ítems* u objetos más cercanos.
- **FunkSVD**: Un método que utiliza el algoritmo SVD, el cual se basa en recomendar a un usuario características u objetos que han gustado o tienen en común con otras personas.
- **Association rule-based algorithms**: Este procedimiento consiste en crear reglas de asociación entre un conjunto de datos con hechos en común.
- **Popular**: Como se expresa el nombre del método, consiste en recomendar los usuarios más populares en cuanto a unas características determinadas en común.
- **Random**: Este último procedimiento utiliza n usuarios seleccionados aleatoriamente para obtener las diferentes características entre ellos y recomendar otros usuarios en común.

El método más favorable para realizar recomendaciones en nuestro sistema es IBCF dado que en nuestro diccionario necesitamos clasificar los usuarios acordes a la cercanía con respecto a los lenguajes principalmente. De esta manera podremos analizar al programador en específico con la misma metodología de trabajo y recomendar los usuarios correspondientes a las características encontradas en común.

3.2 Preprocesamiento acorde al modelo

Una vez elegido el modelo con el que vamos a realizar las recomendaciones, el siguiente paso a realizar para construir nuestro SR es preprocesar los datos.

El propósito del preprocesamiento de datos es principalmente corregir las inconsistencias de los datos que serán la base de análisis en procesos de minería de datos. En el caso de las fuentes de datos estructuradas, el propósito no es

distinto y pueden ser aplicadas diversas técnicas estadísticas y de aprendizaje computacional [17].

En nuestro caso, el preprocesamiento de los datos aportará la comprensión del modelo (Recommenderlab) junto al diccionario de usuarios. Para ello, necesitamos analizar la métrica usada por este paquete, la cual se basa en 3 campos multivaluados:

- **User**: Este primer campo indica la identificación del usuario (puede ser numérico o una cadena).
- **Item**: El campo "item" se utiliza para relacionar el campo "user" con el ítem o característica a recomendar (puede ser numérico o una cadena).
- **Rating**: Por último, "rating" describe la valoración que tiene el usuario con respecto al ítem relacionado (sólo puede ser de tipo numérico).

	user	item	rating
1	1	Toy Story (1995)	5
263	1	GoldenEye (1995)	3
386	1	Four Rooms (1995)	4
460	1	Get Shorty (1995)	3
640	1	Copycat (1995)	3
709	1	Shanghai Triad (Yao a yao yao dao waipo qiao) (1995)	5
726	1	Twelve Monkeys (1995)	4
962	1	Babe (1995)	1
1133	1	Dead Man Walking (1995)	5
1306	1	Richard III (1995)	3
1370	1	Seven (Se7en) (1995)	2
1565	1	Usual Suspects, The (1995)	5
1760	1	Mighty Aphrodite (1995)	5
1884	1	Postino, Il (1994)	5
1988	1	Mr. Holland's Opus (1995)	5
2156	1	French Twist (Gazon maudit) (1995)	5
2187	1	From Dusk Till Dawn (1996)	3

Ilustración 3.1 Métrica usada en Recommenderlab con datos de películas

La figura 3.1 muestra un ejemplo creado en el que se usan los datos "Movielense" del propio paquete. Se puede observar como la métrica usada por Recommenderlab es simple y práctica, pero los datos obtenidos en el diccionario de usuarios tienen 15 campos diferentes y es un problema.

Para resolver este inconveniente, el primer paso a seguir consistirá en trabajar todos los campos en un solo tipo de dato, a excepción del nombre de usuario y el lenguaje (lo necesitamos para el campo "user" e "item"). Una técnica sencilla y simple para convertir una cadena de caracteres a otro tipo de dato (en nuestro caso un entero), es asignando un valor a cada característica. Por ejemplo, la columna "compañía" del diccionario es un dato que podemos transformar a un entero con tan solo asignar un "0" a los usuarios que no disponen de empresa y un "1" a los usuarios que sí la tienen asignada. Otro campo fácil de convertir del diccionario de usuarios es la fecha de última actualización, ya que podemos dividir este tipo de dato en dos columnas diferentes: una columna para el año y otra columna para el mes (el día y la hora no es tan importante como para tenerlo en cuenta). El nombre de usuario completo y la localización no son datos relevantes para tenerlos en cuenta así que podemos omitir esas dos columnas. El problema más grande se encuentra en los lenguajes de cada usuario, dado que más de una persona tiene asignado varios lenguajes y esto no es comprensible por Recommenderlab. Una manera fácil de solucionar este inconveniente y no perder esta característica tan importante, es crear un campo llamado "multi" que contendrá un "0" si la persona no programa en varios lenguajes o un "1" si resulta ser lo contrario. Además, para que el sistema recomendador encuentre similitudes entre usuarios, se pueden duplicar las filas tantas veces como número de lenguajes posea. Esto hace que se pueda añadir un último campo a nuestro diccionario preprocesado que caracterizará el número de lenguajes que cada usuario ha programado.

	usuario	repositorios	estrellas	gists	seguidores	siguiendo	multi	lenguajes	lenguaje_codif	lenguaje_gist	anio	mes	company	rate
1	sevenwire	8	0	1	0	0	0	1	100	5000	2015	7	0	1
2	brandonarbini	12	13	2	18	1	0	1	100	100	2017	5	0	2
3	al3x	48	237	44	1024	54	1	2	100	2000	2017	4	0	5
4	toolmantim	56	101	90	294	64	1	4	100	100	2017	6	0	4
5	nicksieger	70	220	218	211	14	0	1	100	100	2017	1	1	5
6	jicksta	49	300	39	78	16	1	2	100	300	2017	5	1	6
7	joshsusser	16	66	38	276	10	0	1	100	100	2017	5	0	2
8	jcrosby	27	96	13	144	17	1	2	100	100	2017	5	1	5
9	thewoolleyman	137	300	50	72	36	1	2	100	100	2017	6	1	6
10	technicalpickles	217	300	294	610	66	1	5	100	100	2017	6	1	8
11	ryanb	52	300	60	6922	94	0	1	100	100	2017	5	1	6
12	cnix	21	36	22	20	15	0	1	100	100	2016	2	1	1
13	tpitale	45	300	99	79	68	0	1	100	100	2017	5	0	4
14	atduskgreg	223	300	526	506	39	1	9	100	100	2017	5	1	8
15	heff	65	123	7	187	10	0	1	300	300	2017	6	1	4
16	entryway	7	123	14	0	0	0	0	5000	5000	2017	3	0	1

Ilustración 3.2 Fragmento de diccionario

Aquí podemos ver el resultado de aplicar estas técnicas simples de preprocesamiento de datos para poder utilizar el método IBCF de Recommenderlab.

Es muy importante recordar que la métrica que utiliza este paquete sólo permite la inserción de 3 campos en su tipo de dato "*RealRatingMatrix*", así que aún no hemos terminado el preprocesado debido al número de campos usados para caracterizar a los usuarios.

Para poder utilizar correctamente Recommenderlab, debemos dar una puntuación general a cada usuario y colocarla en el campo "rate". Una técnica muy buena para valorar cada persona es la suma de cualidades, por ejemplo, un usuario multilenguaje, con muchos seguidores y gran cantidad de repositorios obtendrá una calificación más alta frente a un usuario con pocos seguidores, bajo número de repositorios y programador de un solo lenguaje. Este ejemplo sólo muestra unas pocas características recogidas en nuestro diccionario, así que para una valoración precisa se sugiere lo siguiente:

- **Repositorios**: Un número clave para diferenciar los usuarios con gran cantidad de proyectos frente a los usuarios con poca cantidad es 25, dado que a partir de esa cifra el programador demuestra claramente su capacidad de creación.
- **Estrellas**: Las estrellas que una persona asigna a un proyecto es un dato poco relevante, pero da información acerca de la actividad y

criterio de calificación en GitHub. Un número preciso para diferenciar esta característica es 150.

- **Gist:** El número de "gists" que tiene un usuario es un complemento a la característica de la programación. 100 es un buen número ya que un usuario multilenguaje y con gran número de repositorios coincide en una mayor cantidad (más de 150 gists) con respecto a otro tipo de programadores.
- **Seguidores:** Este campo es importante ya que determina la "fama" o potencial del usuario. Un programador con buenos proyectos y en continuo avance obtendrá un mayor número de seguidores frente a los demás. Un número preciso para separar los usuarios más relevantes del resto es 1000, dado que esta cifra la cumple perfectamente este tipo de programador en comparación al resto.
- **Siguiendo:** Un campo no tan importante como el número de seguidores, pero relevante en cuanto a sociabilidad del usuario. Se puede considerar una persona sociable si supera en 100 el número de seguidos en GitHub.
- **Multilenguaje:** Esta característica es muy fácil de valorar. Si la persona tiene un "1", significa que es multilenguaje y se debe tener en cuenta más que si tiene un valor de "0" (1 o ningún lenguaje).
- **N.º lenguajes:** Un campo bastante relevante porque permite diferenciar, dentro de los usuarios multilenguaje, el número de lenguajes que es capaz de programar la persona en cuestión. Una persona que sabe programar en más de 4 lenguajes es un programador con una valoración mayor frente a un usuario con menor número.
- **Año y mes:** Dos campos referentes a la última actualización del usuario en la API. El primero se considera a valorar si está en el año vigente (2017) y el segundo si el mes de última aportación fue a partir de abril (4).
- **Compañía:** Por último, la compañía del usuario es una característica fácil de valorar, puesto que un "1" significa que el usuario trabaja para alguna empresa y un "0" determina que no.

Esta técnica se puede implementar fácilmente mediante preguntas de tipo "*if*" y "*else*" y almacenar el valor resultante en un campo nuevo. Cuando se procese de nuevo nuestro diccionario de usuarios obtendremos algo así:

	usuario	lenguaje	valoracion
87	bernerdschaefer	Ruby	5
88	bernerdschaefer	JavaScript	5
89	zapnap	Ruby	6
90	zapnap	JavaScript	6
91	NZKoz	Ruby	5
92	NZKoz	Objective-C	5
93	croaky	Ruby	6
94	jeremy	Ruby	6
95	ELLIOTTCABLE	JavaScript	8
96	ELLIOTTCABLE	Ruby	8
97	ELLIOTTCABLE	C	8
98	ELLIOTTCABLE	Shell	8
99	ELLIOTTCABLE	CoffeeScript	8
100	ELLIOTTCABLE	CSS	8
101	monde	Ruby	4
102	ryanbriones	Ruby	5
103	ryanbriones	JavaScript	5

Ilustración 3.3 Fragmento del diccionario de usuarios procesado

Como se puede apreciar, hemos transformado los datos del diccionario principal a una métrica equivalente para la comprensión en Recommenderlab. En la columna "usuario" se encuentra el nombre de cada persona repetido tantas veces como número de lenguajes programa. El lenguaje es una columna que muestra los idiomas de programación relacionados con cada usuario y la valoración es la calificación final obtenida de la suma de cada característica personal.

Capítulo 4. Propuesta sistema recomendador

En este capítulo se explicará más detalladamente el método IBCF (Item Based Collaborative Filtering) visto en el episodio anterior y su aplicación junto a recomendaciones elaboradas con el paquete "Recommenderlab". Además, se necesitará automatizar el proceso para poder utilizar cualquier usuario de GitHub.

4.1 Filtrado colaborativo basado en objetos

Como ya se nombró en el capítulo anterior, IBCF o Filtrado Colaborativo Basado en Objetos, es un método basado en el algoritmo "KNN" (vecinos más cercanos) y trata de agrupar por similitud los usuarios con sus características (valoración de una película, por ejemplo).

Los usuarios de un sistema colaborativo comparten sus valoraciones y opiniones con respecto a los ítems que conocen de forma que otros usuarios puedan decidir qué elección realizar. A cambio de compartir esta información, el sistema proporciona recomendaciones personalizadas para aquellos elementos que pueden resultar interesantes al usuario [18].

Este sistema es muy útil para recomendar usuarios dado elementos como son los lenguajes junto a una valoración. Nuestro diccionario preprocesado anteriormente contiene todas las características necesarias para la aplicación de este método con el fin de conseguir una clasificación correcta y recomendaciones precisas entre elementos.

4.2 Medidas de similitud

Existen diferentes medidas de similitud para encontrar elementos semejantes entre sí como comparar personas y calcular un resultado de semejanza. Entre las medidas más importantes destacan la distancia euclídea, el coeficiente de correlación de Pearson y el coseno. La primera medida consiste en el cálculo, mediante el teorema de Pitágoras para la distancia entre dos puntos. El coeficiente de correlación de Pearson es otra medida que mide el encaje de dos puntos en una línea recta, proporcionando mejores resultados cuando el uso de datos no está normalizado. Por último, la similitud coseno es una ponderación entre dos vectores los cuales, mediante el cálculo del coseno, se obtiene un valor que determinará la similitud entre estos.

Cabe destacar que la similitud coseno es la medida usada por defecto en "Recommenderlab" y es la que mejores resultados proporciona con respecto a los datos de usuarios en GitHub.

4.3 Recomendaciones

La figura 3.3 del capítulo anterior muestra un fragmento del diccionario de usuarios preprocesado para poder utilizar correctamente "Recommenderlab". Para usar este paquete, debemos transformar nuestro *dataset* en un tipo de estructura llamada "RealRatingMatrix". Esta configuración es un tipo de matriz que tiene por regla general el no existir ningún campo vacío además de utilizar el mismo tipo de dato para cada columna. En nuestro caso, el preprocesamiento creado anteriormente hace que se cumplan estas reglas a la perfección, así que podemos transformar nuestro diccionario.

```
# Parte recomendación ----------------------------------------------------

library("recommenderlab")
tr<-read.csv("diccionario_usuarios.csv",header=TRUE, stringsAsFactors=FALSE)

tr.matrix<- as(tr,"realRatingMatrix")
```

Ilustración 4.1 Transformación a "RealRatingMatrix"

En la figura 4.1 se muestra como modificar nuestro diccionario de usuarios a una matriz de tipo *RealRatingMatrix.* Si cumplimos los requisitos previos no debería de haber ningún error en la conversión.

El siguiente paso a realizar es crear el modelo de recomendación para el cual se van a introducir todos los datos. En nuestro caso, nuestro modelo es IBFC y la medición de similitud se hará mediante el coseno. Esto lo podemos indicar de la siguiente manera:

```
Rec.model=Recommender(tr.matrix[1:nrow(tr.matrix)],method="UBCF",
                      param=list(method="Cosine"))
```

Ilustración 4.2 Creación del modelo de recomendación

Como podemos observar, es muy intuitivo y sencillo de utilizar, simplemente se le indica el número de elementos a procesar (en nuestro caso interesa que sean todos), el modelo a utilizar y como parámetros la medida de similitud.

Ya con nuestro modelo creado y elaborado, sólo queda predecir el número de personas que queramos recomendar al usuario en cuestión. Para ello, existe una función creada por el paquete que analiza y encuentra los usuarios más cercanos al programador introducido llamada "predict". Esta función tiene como parámetros el modelo, el usuario dentro de nuestro diccionario y el número de recomendaciones. Un ejemplo práctico sería mostrar 5 usuarios recomendados para la persona número 1 de nuestro diccionario:

```
recommended.user <- predict(rec, tr.matrix[1,], n=5)
recomendados.top5 = as(recommended.user, "list")
```

Ilustración 4.3 Predicción y creación de lista con 5 usuarios recomendados

Con tan sólo usar la función "predict", ya hemos obtenido los 5 usuarios a recomendar para el usuario número 1. Con esto finaliza la creación del modelo elegido en "Recommenderlab" y las recomendaciones. El siguiente paso consistirá en automatizar el proceso dado que en este caso se está recomendando usuarios dentro del diccionario. Si queremos recomendar cualquier usuario de GitHub es necesario utilizar los métodos y técnicas vistas hasta este capítulo para poder añadir a nuestro set de datos en particular, la persona a recomendar.

4.4 Automatización

Para la creación de nuestro sistema de recomendación necesitamos que los diferentes métodos y procedimientos llevados a cabo hasta ahora se procesen de forma automática. Es muy importante automatizar todo para poder añadir, entre otros, una pequeña interfaz gráfica y que cualquier usuario pueda utilizar nuestra aplicación.

El primer paso a realizar para mecanizar nuestro SR es reutilizar los métodos vistos en el capítulo 2 para la obtención de los datos. Lo primero que debemos hacer es guardar en alguna variable el nombre del usuario para poder conectarnos a la API y guardar sus datos en varios campos. Una vez obtenidos sus datos, debemos aplicar las técnicas de preprocesamiento vistas hasta ahora y almacenar el resultado en un set de datos.

usuario	lugar	nombre	repositorios	estrellas	gists	seguidores	siguiendo	multi	lenguajes	lenguaje_codif	lenguaje_gist	anio	mes	company
alex	Washington D.C.	Alex Gaynor	348	150	66	3476	41	1	9	Python	Python	2017	06	1

Ilustración 4.4 Ejemplo de usuario preprocesado

La tabla 4.4 muestra un ejemplo de preprocesamiento de datos una vez obtenidos los campos correspondientes. En este caso, se ha utilizado la información del usuario "*alex*" y se ha organizado de manera idéntica a la métrica usada en el diccionario. Una vez completado el preprocesamiento, tan sólo debemos cargar el fichero correspondiente a nuestro diccionario preprocesado y añadir una fila nueva con los datos de este nuevo individuo. Para no perder el control del usuario insertado a la hora de crear el modelo de datos y obtener recomendaciones, podemos insertar siempre en la primera posición del set de datos para elegir esa posición cuando usemos la función "predict" de "Recommenderlab".

```
data0 <-  read.csv("MyData3.csv", header = TRUE, stringsAsFactors=F)
data1 <- merge.data.frame(usuario_nuevo,data0, all = TRUE)
```

Ilustración 4.5 Unión de dos *data.frame*

En este ejemplo podemos apreciar que la función "*merge.data.frame()*" se encarga de crear un objeto nuevo con la unión de los datos, tanto del usuario reciente, como del diccionario. Con este paso realizado, tan sólo queda aplicar el método visto en el apartado anterior y usar la función para guardar los n usuarios a recomendar. Una buena práctica para automatizar aún más el proceso y facilitar la elaboración de nuevos códigos consiste en crear una función que sea capaz de realizar todo el proceso y devolver el nombre de cada usuario, si existe. En este caso, es bastante sencillo, tan sólo debemos de recoger por parámetro el nombre del usuario a recomendar y devolver en un vector el resultado del proceso.

4.5 Interfaz gráfica

El último paso a realizar para tener un sistema recomendador automatizado y con una visión agradable al usuario es añadir una pequeña interfaz gráfica. Existe un paquete en el lenguaje R capaz de crear y modificar a nuestro gusto todo tipo de ventanas con botones, etiquetas, paneles, campos de texto... llamado **RGTK2.**

RGTK2 es una interfaz para "GTK" que consiste en la creación de "GUIs" bajo el entorno de programación R. Este paquete facilita mucho el aprendizaje y programación de interfaces gráficas mediante funciones y llamadas a la librería interna "GTK".

En el apartado anterior se definió y creó una función en el lenguaje R que, al pasarle por parámetro el nombre de usuario a recomendar, ésta devuelve un vector de n posiciones con el resultado obtenido. Vamos a utilizar esa función dentro de las propiedades de RGTK2 para poder insertar el nombre de la persona a recomendar y mostrar los resultados en la ventana.

Lo primero que debemos hacer es diseñar la "GUI" acorde al trabajo propuesto. Una idea simple y eficaz es utilizar un cuadro de texto para que el usuario introduzca su nombre de usuario, un botón que llame a la función creada y un espacio que muestre los resultados obtenidos (podría ser otro cuadro de texto o una tabla). Para implementar este diseño, debemos crear una ventana mediante la función "*gtkWindow()*" y añadir un título para que aparezca como nombre de aplicación con "*gtkFrameNew()*". Una vez realizado este paso, necesitamos indicarle al paquete que buscamos añadir elementos como un cuadro de texto o botones. Para ello, existe diferentes "boxs" con varias características en particular. Por ejemplo, "*gtkLabelNewWithMnemonic*" crea un "label" o etiqueta en la posición que deseemos dentro de la ventana. Otro ejemplo es "*gtkEntryNew()*", el cual crea un campo de tipo texto tanto para entrada de datos como para mostrar resultados. Esta última función es muy útil para crear un campo en el que un usuario introduzca su nombre de GitHub y lo recoja nuestra función. Este es el resultado una vez se ha diseñado e implementado la ventana:

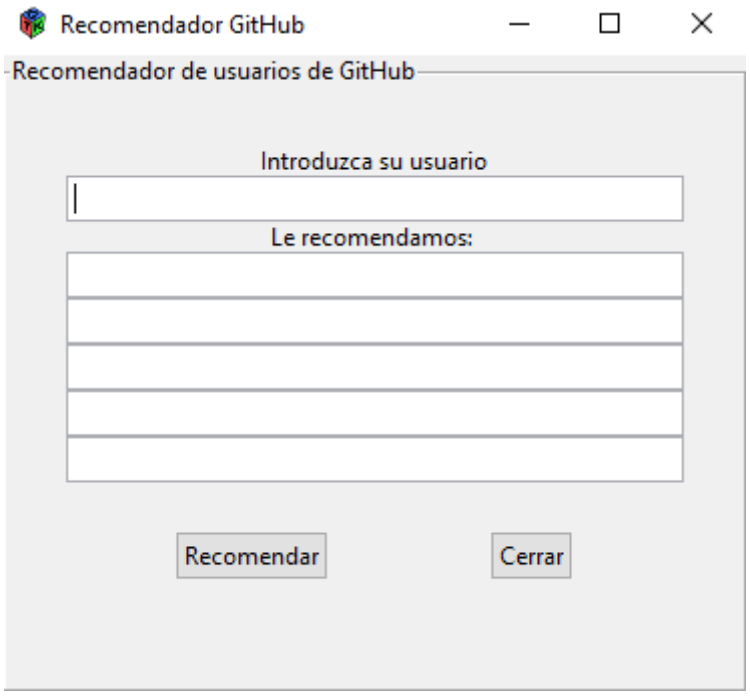


Ilustración 4.6 Ventana creada y configurada con RGTK2

La figura 4.6 muestra una ventana creada con este paquete de una forma sencilla: utiliza una etiqueta principal con el título "Recomendador de usuarios de GitHub" junto a dos etiquetas más, un cuadro de texto para insertar el nombre de usuario y 5 cuadros más para mostrar las recomendaciones, un botón

para llamar a la función vista en el apartado anterior y otro botón para salir de la aplicación.

El siguiente paso consistirá en conectar cada funcionalidad de la ventana con nuestro código, tanto para la entrada de datos, como la salida del resultado. Para ello, debemos conectar el botón "*recomendar*" junto a una función que recogerá la información escrita por el usuario y llamará a la función de recomendación.

```
gSignalConnect(Recomendar, "clicked", DoCalculation) #señal conectada a funcion

buttonCancel2 = gtkButton("Cerrar")
gSignalConnect(buttonCancel2, "clicked", window$destroy) # señal para cerrar la ventana
box2$packStart(buttonCancel2,fill=F)
```

Ilustración 4.7 Conexión de botones y funciones

Se puede apreciar claramente mediante la función "*gSignalConnect()*" la conexión entre el botón *"recomendar"* (cuando sea pulsado) junto a la función "*DoCalculation*". También está conectado el botón "cerrar" a una señal llamada "*window$destroy*" para eliminar la ventana y cerrar la aplicación. Cuando hayamos realizado este paso, debemos rellenar la función conectada al botón *recomendar* con las siguientes características:

- Debe comprobar que el usuario ha escrito algo en el campo correspondiente para la lectura de datos. Si queremos verificar que la persona ha escrito, podemos comprobar que la entrada de texto esté vacía. Si es así, entonces no ejecutaremos nada.
- El nombre de usuario introducido debe pasarse por parámetro a la función creada en el anterior apartado. Una manera sencilla es guardando la entrada de texto en una variable y llamando a la función en cuestión con ese parámetro.
- Se debe mostrar el resultado en la misma ventana que dispone el usuario. Para ello, guardaremos el resultado de la función en un objeto local y así ir mostrando cada elemento del vector en un *textbox* nuevo.

```
DoCalculation<-function(button)
{
  if ((TextoUsuario$getText())=="") return(invisible(NULL)) #si no hay texto no hacer nada

  tryCatch(
    if (TextoUsuario$getText()!=""){
      x <- TextoUsuario$getText()
      m<-recomendaciones(x)

      m<- paste0("Usuario: ",m," | link: ","https://github.com/", m)

      result$setText(m[1])
      result2$setText(m[2])
      result3$setText(m[3])
      result4$setText(m[4])
      result5$setText(m[5])
```

Ilustración 4.8 Implementación del método usado en GUI

La figura 4.8 muestra la implementación necesaria para cumplir las características expuestas anteriormente (el *trycatch* está incluido para abrir una nueva ventana de error si el código no se ejecutara correctamente).

Una vez creada la ventana, añadido los botones y campos de texto, conectado los objetos e implementadas las características de resultados tenemos disponible nuestra pequeña "GUI" o interfaz gráfica personalizada y lista para utilizar.

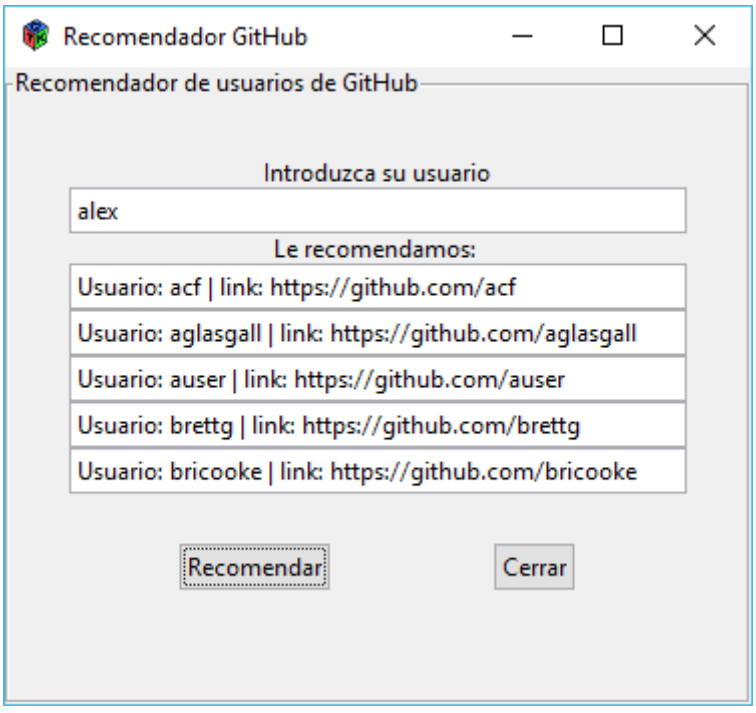


Ilustración 4.9 Ejemplo de uso de la interfaz gráfica

Para comprobar que la interfaz funciona correctamente, se ha hecho un ejemplo con el usuario "alex", el cual es un programador multilenguaje con gran cantidad de seguidores y repositorios. Los usuarios recomendados son bastante

similares a él y cada enlace correspondiente funciona a la perfección, además de mostrarse correctamente en cada celda de texto. Con esta demostración práctica se cierra el capítulo 4 obteniendo un sistema de recomendación completo con su correspondiente interfaz gráfica.

Capítulo 5. Mejoras sistema recomendador

Nuestro sistema recomendador actual analiza a los usuarios, le asigna una valoración según una métrica y unas características y recomienda usuarios similares. Esta funcionalidad no se encuentra implementada actualmente en la API de GitHub y es una buena funcionalidad de cara a conocer más gente y ampliar el rango de conocimiento, pero ¿qué pasaría si además nuestro SR tuviera la habilidad de recomendar usuarios capaces de solucionar errores de otros programadores? GitHub ofrece para cada repositorio un campo llamado "issue", el cual tiene como uso marcar errores en el repositorio correspondiente con el fin de hacérselo saber al titular del proyecto para correcciones. También dispone de un campo llamado "events" dentro de la API que recopila información acerca de las contribuciones que cada usuario aporta a la web. Con esta información, podemos obtener y clasificar usuarios que realicen varias contribuciones en uno o varios lenguajes junto a los últimos "issues" o errores que aparezcan abiertos en diferentes programadores.

5.1 Obtención de datos y filtrado

El primer paso a realizar para conseguir aplicar esta funcionalidad es obtener un pequeño diccionario de usuarios con una métrica simple:

- **Nombre de usuario:** Este campo va a definir el usuario por su nombre en GitHub y será el campo a mostrar hacia el usuario.
- **Lenguaje usado:** Esta variable contendrá el lenguaje más utilizado en las contribuciones realizadas por cada usuario. Es muy importante determinar el lenguaje de programación que cada persona usa, ya que si recomendásemos programadores omitiendo este campo llegaríamos, en su mayoría, a recomendar usuarios incapaces de resolver cada problema en particular.

- **Número de contribuciones:** El número de contribuciones es un campo importante para clasificar los usuarios acordes al lenguaje y al número de aportaciones realizadas. No tendrá la misma importancia un usuario con pocas aportaciones frente a un usuario en constante actividad.

Para extraer los datos de GitHub, el enlace https://api.github.com/users/nombre-de-usuario/events proporciona las últimas aportaciones de cada usuario. Podemos utilizar exactamente la misma técnica nombrada en el primer capítulo para la obtención de datos y almacenar en un *data.frame* el resultado. Como necesitamos tener un nombre de usuario vinculado por cada enlace, podemos reutilizar las personas almacenadas en nuestro diccionario y recoger toda la información pública disponible.

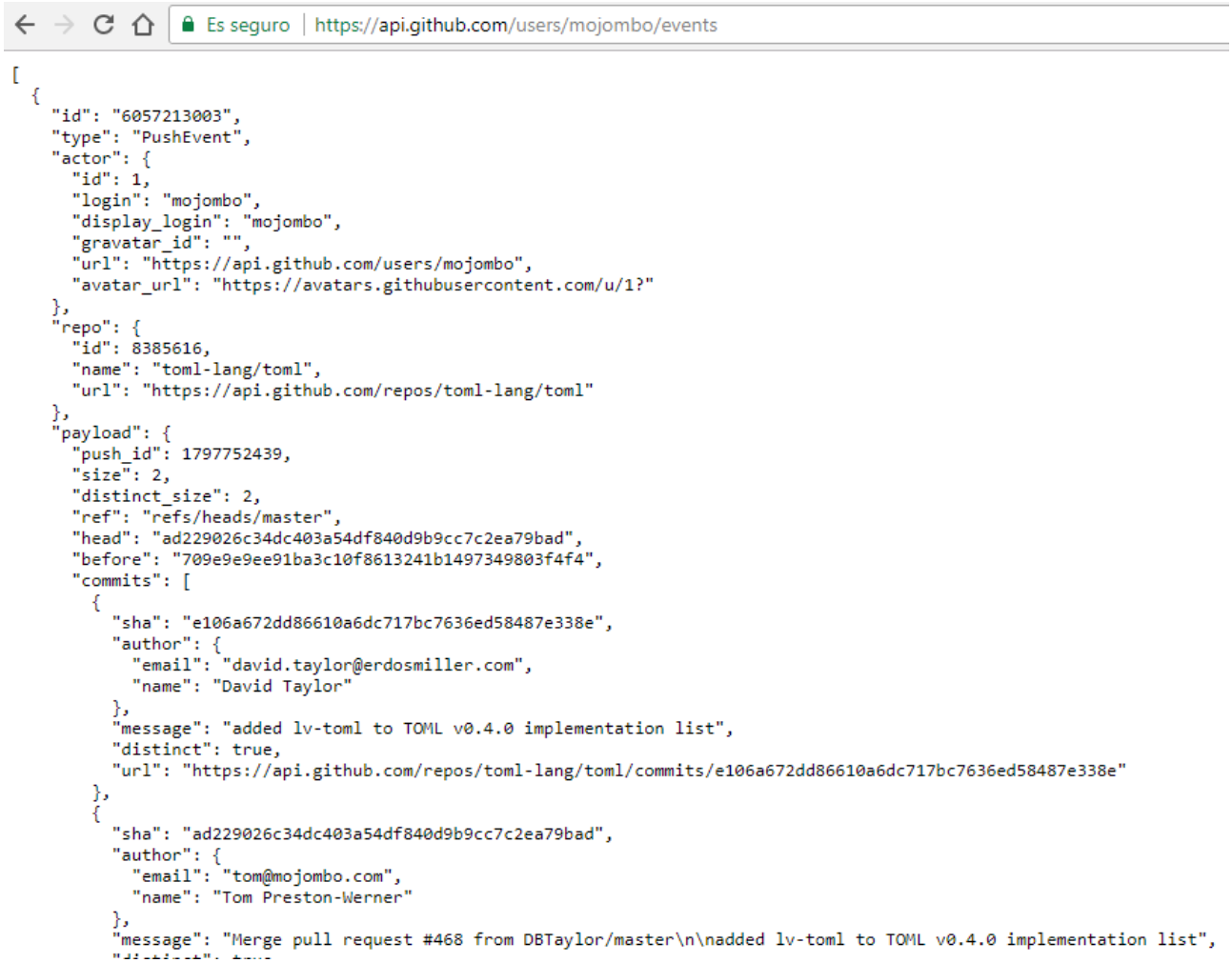


Ilustración 5.1 Fragmento de datos en formato JSON devuelto por la API V3 de GitHub para el usuario "mojombo"

La figura 5.1 muestra una pequeña parte de una consulta enviada a las contribuciones del usuario “mojombo”. Se puede observar que la API tiene en cuenta para asignar contribuciones los “issues”, “pull request”, “commit” realizados en repositorios ajenos y los “fork” a su cuenta de usuario.

Una vez obtenido todos los datos, el último paso a realizar consiste en filtrar la información y recoger sólo los campos correspondientes a la métrica definida. Para ello, debemos recorrer cada enlace en busca del lenguaje de cada repositorio en el cual se ha hecho una contribución y la suma de aportaciones en lo que va de año, todo ello vinculado al nombre de usuario.

	usuario	lenguajes	contribuciones
1	sevenwire	Ruby	30
2	brandonarbini	JavaScript	30
3	al3x	JavaScript	15
4	toolmantim	Ruby	30
5	nicksieger	TypeScript	12
6	jicksta	JavaScript	30
7	joshsusser	Ruby	2
8	thewoolleyman	JavaScript	30
9	technicalpickles	Elixir	30
10	ryanb	Java	2
11	tpitale	HTML	30
12	atduskgreg	JavaScript	30
13	heff	Python	30
14	aflatter	Ruby	7

Ilustración 5.2 *Data.frame* creado siguiendo la métrica y filtro propuesto

En este fragmento de datos se puede ver con claridad el uso de la métrica definida junto al filtrado de datos. En el primer campo se encuentra el nombre de usuario, el segundo campo muestra el lenguaje más utilizado en las contribuciones y, por último, el número de aportaciones durante el año realizadas lo apunta la última columna. Cabe destacar que para acelerar el proceso de descarga de datos hubo que limitar el número de contribuciones a 30 debido al tiempo tan excesivo que tardaba la aplicación en ejecutar. Aun así, 30 es un número más que razonable para indicar que el usuario ha realizado un buen número de contribuciones.

5.2 Clasificación

Para recomendar un número de usuarios capaces de solucionar un problema en concreto, debemos crear un "ranking" ordenado por número de contribuciones y el lenguaje asociado. Una manera limpia y ordenada de llevar a cabo esta tarea es importando el "diccionario de contribuciones" y usar las funciones básicas implementadas por R Studio para los *data.frame*. Una función que ordena según nuestro criterio es "*order()*", la cual nos permite indicar la columna o fila a ordenar, orden ascendiente o descendente y demás parámetros entre otros. Lo primero que debemos ordenar es la columna del lenguaje de modo que estén todos agrupados por similitud sin importar el orden. Una vez hecho esto, se agrupará el tercer campo por orden descendente para tener los usuarios más relevantes en las primeras posiciones. El resultado debe quedar así:

	usuario	lenguajes	contribuciones
1	mysmallidea	Vue	30
2	spicycode	Vim script	30
3	tpope	Vim script	30
4	cypher	Vim script	30
5	billturner	Vim script	27
6	nickel	Vim script	23
7	mariusz	Vim script	15
8	benwad	Vim script	11
9	dmondark	Vim script	2
10	jicksta	TypeScript	30
11	usergenic	TypeScript	30
12	mbleigh	TypeScript	30
13	willnorris	TypeScript	30
14	Cordobo	TypeScript	30
15	aggieben	TypeScript	30

Ilustración 5.3 Fragmento del diccionario de contribuciones ordenado

La figura 5.3 muestra el orden correcto para llevar a cabo el "ranking" de usuarios con más contribuciones por lenguaje utilizado. Es muy importante no modificar este orden puesto que, posteriormente, se va a consultar esta tabla en busca del lenguaje en común con el usuario analizado y así poder recomendar en base al "ranking" establecido.

5.3 Automatización

Con nuestro "ranking" ya elaborado, sólo tenemos que buscar un usuario de GitHub, obtener sus datos, procesarlos y buscar en nuestra clasificación los programadores que más contribuyen en ese ámbito. Estos pasos son similares al sistema de recomendación realizado hasta ahora, así que podemos reutilizar lo aprendido para automatizar el proceso y poder, posteriormente, añadirlo a nuestra interfaz gráfica.

En primer lugar, usaremos la técnica vista en el primer apartado de este episodio para obtener los datos del usuario en cuestión. En este caso, debemos de filtrar los datos por los "issues" que tiene abierto el usuario a analizar. Esta consulta se puede generar ya filtrada a través del siguiente enlace: https://api.github.com/search/issues?q=type:pr+state:open+author:NOMBRE &per_page=100&page=1

En ella se está indicando la búsqueda de "issues" abiertos por el usuario con x nombre y expandiendo el resultado a 100 elementos, si existen. El resultado a esta consulta es el siguiente:

```
{
  "total_count": 31,
  "incomplete_results": false,
  "items": [
    {
      "url": "https://api.github.com/repos/taskcluster/rust-hawk/issues/2",
      "repository_url": "https://api.github.com/repos/taskcluster/rust-hawk",
      "labels_url": "https://api.github.com/repos/taskcluster/rust-hawk/issues/2/labels{/name}",
      "comments_url": "https://api.github.com/repos/taskcluster/rust-hawk/issues/2/comments",
      "events_url": "https://api.github.com/repos/taskcluster/rust-hawk/issues/2/events",
      "html_url": "https://github.com/taskcluster/rust-hawk/pull/2",
      "id": 238963332,
      "number": 2,
      "title": "Fixed linkification in readme",
      "user": {
        "login": "alex",
        "id": 772,
        "avatar_url": "https://avatars1.githubusercontent.com/u/772?v=3",
        "gravatar_id": "",
        "url": "https://api.github.com/users/alex",
        "html_url": "https://github.com/alex",
        "followers_url": "https://api.github.com/users/alex/followers",
        "following_url": "https://api.github.com/users/alex/following{/other_user}",
        "gists_url": "https://api.github.com/users/alex/gists{/gist_id}",
        "starred_url": "https://api.github.com/users/alex/starred{/owner}{/repo}",
        "subscriptions_url": "https://api.github.com/users/alex/subscriptions",
        "organizations_url": "https://api.github.com/users/alex/orgs",
        "repos_url": "https://api.github.com/users/alex/repos",
        "events_url": "https://api.github.com/users/alex/events{/privacy}",
        "received_events_url": "https://api.github.com/users/alex/received_events",
        "type": "User",
        "site_admin": false
      },
      "labels": [

      ],
      "state": "open",
      "locked": false,
      "assignee": null,
      "assignees": [
```


Ilustración 5.4 Fragmento de un ejemplo de consulta sobre "issues" a la API de GitHub

Esta respuesta generada por la API nos devuelve, ordenado por fecha, un archivo en formato JSON con las características de cada repositorio que contenga algún “issue” abierto.

Con esta información, podemos acceder al repositorio y extraer datos más específicos como, por ejemplo, determinar el lenguaje del proyecto en cuestión.

Con este paso realizado ya podemos hacer una búsqueda del lenguaje de programación equivalente a nuestro diccionario de contribuciones y poder mostrar una lista de elementos que pueden ayudar a la persona en cuestión a resolver el/los problema/s.

5.4 Adición a la GUI creada

La interfaz gráfica creada hasta el momento permite que un usuario introduzca su nombre y al pulsar un botón aparezcan personas con características similares como recomendación a seguir. Podríamos añadir más "widgets" a la ventana para configurar las recomendaciones de errores abiertos en repositorios públicos, pero ¿y si un usuario no tiene ningún "issue" abierto? La solución a este inconveniente es sencilla: podemos analizar al usuario y determinar si dispone de errores en proyectos o, por el contrario, tiene todo en perfectas condiciones. Una vez sepamos la respuesta, implementaremos una ventana nueva que muestre automáticamente las personas que pueden ayudarle a resolver los problemas siempre que exista algún "issue" abierto. Para ello, vamos a crear una función llamada "contribuciones" con los métodos implementados hasta ahora además de devolver un vector de n usuarios a recomendar. Un mecanismo para determinar si un usuario tiene errores recientes en alguno de sus proyectos es comprobando si el vector devuelto por la función está vacío. Este método tiene como resultado dos soluciones: o bien el usuario no dispone de "issues" abiertos, o bien no existe ninguna persona en nuestro diccionario capaz de poder ayudarle. En cualquiera de los dos casos, no se mostrará la ventana auxiliar y así resolvemos el problema.

Para la implementación, esta vez no será necesario que el usuario introduzca su nombre de nuevo ni pulse ningún botón, puesto que nuestro sistema detectará cuando es necesario mostrar la nueva ventana. Un simple botón auxiliar para cerrar la ventana y n cuadros de texto será suficiente para mostrar el resultado al usuario de forma sencilla y visible.

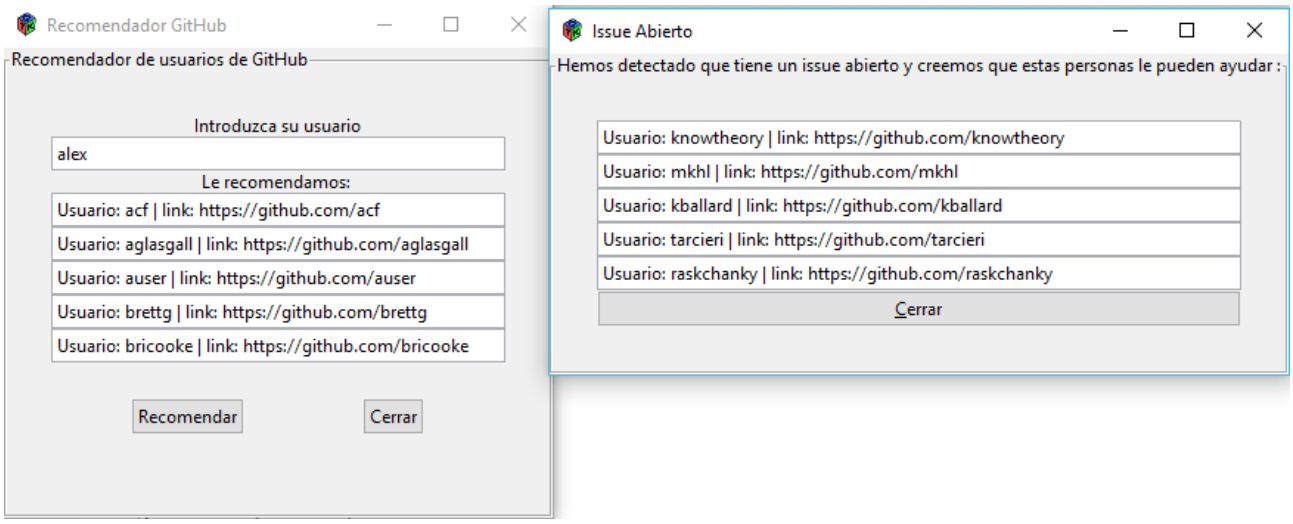


Ilustración 5.5 Ejemplo de interfaz gráfica con el usuario “alex”

En la figura 5.5 se muestra un ejemplo del resultado final obtenido, el cual el usuario “alex” tiene un “issue” recientemente abierto y, por consecuente, aparece una segunda ventana con 5 usuarios que pueden ayudar a resolver su problema. En todo el SR se muestra el enlace de cada usuario para que sea más simple buscar el perfil por GitHub y conocer, en mejor medida, el correspondiente programador.

Con este último ejemplo, queda por concluido nuestro sistema de recomendación con la funcionalidad extra de recomendar programadores capaces de resolver errores abiertos en repositorios públicos.

Capítulo 6. Conclusiones y líneas futuras

Los objetivos de este proyecto han sido varios, dando como resultado un prototipo software de un sistema de recomendación junto a la sugerencia de contribuyentes capaces de resolver problemas relacionados con un repositorio en GitHub.

Hemos visto a lo largo de esta memoria cómo funcionan los sistemas recomendadores en la actualidad y la importancia del procesamiento de datos junto a los algoritmos para llevarlos a cabo. Es fundamental aprovechar la inteligencia colectiva para realizar estudios, buscar patrones en común, entender mejor el raciocinio humano y, entre otras características, realizar sugerencias a personas con perfiles similares.

Gracias a este trabajo de fin de grado se han aportado ideas e implementaciones que pueden ayudar en futuro al desarrollo de sistemas más sofisticados y precisos para la API.

Por otro lado, se han creado métodos que definen las propiedades más importantes de cada usuario en GitHub, demostrando una buena precisión y unos resultados acordes al proyecto.

6.1 Líneas futuras

Como líneas futuras, este proyecto se podría extender aplicando mejoras en las técnicas de procesamiento y aplicación de algoritmos. Además, con el aprovechamiento de los datos, cabe la posibilidad de añadir funcionalidades adicionales al sistema recomendador con el fin de ayudar al programador en sus proyectos y en sus habilidades de interacción.

Capítulo 7.
Summary and Conclusions

There have been a variety of aims in this project, and the final result is a software prototype of a recommendation system, along with suggestions of contributors who are able to resolve problems related to a GitHub repository.

It has been seen throughout this report how current recommendation systems work, the importance of data processing and the algorithms to accomplish them. It is essential to take advantage of collective intelligence to carry out studies, to find common patterns, to have a better understanding of human reasoning and, among other options, to make suggestions to people with similar characteristics.

Apart from that, different methods have been created, which define the most important attributes of each GitHub user, showing good precision and valuable results.

Thanks to this final degree project there have been proved multiple ideas and implementations, which can help to the future development of more sophisticated and accurate systems for the API.

7.1 Future work lines

Focusing on possible future developments, this project could be extended by the applying of improvements in the processing and implementation of the algorithms. Furthermore, the data use allows the possibility of adding additional functionalities to the recommendation system, in order to help the programmers in their projects and in their social interaction abilities.

Capítulo 8. Presupuesto

El presupuesto estimado para la realización de este proyecto se basa en la duración en horas de éste junto a la implementación. Cada hora se estima un total de 10 euros.

Objetivo	Descripción	Horas totales	Costo
Estudio del estado del arte	Estudio del estado del arte relacionado con las redes sociales, así como recomendaciones a diferentes fuentes de información	120	1200
Búsqueda, definición y creación de métodos de recomendación	Búsqueda, definición y creación de métodos de recomendación usando fuentes de información acorde a los medios de comunicación del usuario en GitHub	240	2400
Validación de los métodos de recomendación	Validación de los métodos de recomendación con pruebas y datos del usuario.	80	800
Prototipo software	Obtención de un prototipo software en lenguaje R	280	2800
Documentación	Creación de la documentación técnica, entregables y la memoria del proyecto.	136	1360
Total		**856**	**8560**

Tabla 8.1. Tabla resumen de los Tipos.

Referencias

[1] J. Fernández Martínez, F. Llopis, P. Martínez-Barco, Y. Gutiérrez, Á. Díez and others, "Analizando opiniones en las redes sociales," -, 2017.

[2] A. A. Pérez and J. G. Close, "Sistemas de recomendación," *Robots Autónomos: Navegación,* 2015.

[3] J. Pinho Lucas, "Métodos de clasificación basados en asociación aplicados a sistemas de recomendación," -, 2010.

[4] L. E. M. Mislej, "Slideshare," 27 Junio 2013. [Online]. Available: https://es.slideshare.net/ErnestoMislej/charla-recsys.

[5] J. C. Caiza, D. S. Guamán and G. R. López, "Herramientas de desarrollo con soporte colaborativo en Ingeniería de Software," *Enfoque UTE,* vol. 6, pp. pp--102, 2015.

[6] F. J. Lopez-Pellicer, R. Béjar, M. A. Latre, J. Nogueras-Iso and F. J. Zarazaga-Soria, "GitHub como herramienta docente," in *Actas de las XXI Jornadas de la Enseñanza Universitaria de la Informática*, 2015.

[7] G. Joseph, "Slideshare," 28 Septiembre 2014. [Online]. Available: https://www.slideshare.net/SeemzGrace/source-code-management-system-github.

[8] P. E. Oliden, "¿Existe vida más allá del SPSS? Descubre R," *Psicothema,* vol. 21, pp. 652-655, 2009.

[9] E. Durán and R. Costaguta, "Minería de datos para descubrir estilos de aprendizaje," *Revista Iberoamericana de Educación,* vol. 42, pp. 1-10, 2007.

[10] M. C. Herreros, Desarrollos del periodismo en Internet, vol. 37, Comunicacion Social, 2010.

[11] A. Bartolomeo, "Análisis de la última tenden," *Marketing,* vol. 2, p. 11, 2011.

[12] P. Hernández, "Tendencias de Web 2.0 aplicadas a la educación en línea," *No solo usabilidad,* 2007.

[13] M. Herrera, J. Gutiérrez-Pérez, J. Izquierdo and R. Pérez-García, "Ajustes en el modelo PageRank de Google para el estudio de la importancia relativa de los nodos de la red de abastecimiento," *X Seminario Iberoamericano de planificación, proyecto y operación de sistemas de abastecimiento de agua,* 2011.

Brin, S., Page L. (1997) "The Anatomy of a Large-Scale Hypertextual Web Search Engine" http://infolab.stanford.edu/backrub/google.html.

Chung, F., Zhao, W. (2008) "Pagerank and random walks on graphs" Proceedings Fete of Combinatorics and Computer Science Conference en honor de Laci Lovász (Keszthely, Hungría).

[14] X. Ribes, "La Web 2.0. El valor de los metadatos y de la inteligencia colectiva," *Telos,* vol. 73, pp. 36-43, 2007.

[15] V. Reyes, Diciembre 2015. [Online]. Available: http://www.scielo.cl/fbpe/img/jotmi/v8s1/art36.fig1.jpg.

[16] N. I. C. O. L. Á. S. I. G. N. A. C. I. O. T. O. R. R. E. S. RUDLOFF, "SISTEMAS DE RECOMENDACIÓN BASADOS EN MÉTODOS DE FILTRADO COLABORATIVO," -, 2015.

[17] C. Hernández and J. E. R. Rodríguez, "Preprocesamiento de datos estructurados," *Revista Vinculos,* vol. 4, pp. 27-48, 2013.

[18] E. J. Castellano, L. Martínez, M. Barranco and L. G. PÉREZ, "Recomendación de perfiles académicos mediante algoritmos colaborativos basados en el expediente," in *Conferencia IADIS Ibero-Americana WWW/Internet 2007*, 2007.

Apéndice A.
Descarga de datos

A.1. Obtención de usuarios

```
#*********************************************************************************
#
# obtención_datos.R
#
#*********************************************************************************
#
#
# AUTOR: Adrian Gutierrez Alvarez
#
#
# FECHA 28/06/2017
#
#
# DESCRIPCION método para la extracción de datos y generación de diccionario de forma
# automática
#
#
#*********************************************************************************/
library(jsonlite)
library(httpuv)
library(httr)
library(pipeR)
library(rlist)

oauth_endpoints("github")

myapp <- oauth_app(appname = "aplicacion_tfg",
                   key = "Introducir key",
                   secret = "introducir clave")

# Obtener credenciales OAuth
github_token <- oauth2.0_token(oauth_endpoints("github"), myapp)
```

```
  # Usar API
  gtoken <- config(token = github_token)
  contador = 0
  id = 147
  datos = 1
  vector.lenguaje.frecuencia = vector()
  informacion.usuario = ""
  vector.lenguaje.frecuencia2 = vector()


  while(contador < 50){
    print(contador)
    req <- GET("https://api.github.com/users?per_page=100&since=%d"%>>%sprintf(id),
  gtoken)


    # accion de http error
    stop_for_status(req)


    # extraer contenido de la consulta
    json1 = content(req)


    # Convertir a data.frame
    gitDF = jsonlite::fromJSON(jsonlite::toJSON(json1))



      req2 <- GET(gitDF$url[[datos]], gtoken)
      stop_for_status(req2)
      json2 = content(req2)
      gitDF2 = jsonlite::fromJSON(jsonlite::toJSON(json2))


  # Hasta aquí se ha basado el código en: https://medium.com/towards-data-
science/accessing-data-from-github-api-using-r-3633fb62cb08


    if(length(gitDF2$company)== 0){
      gitDF2$company= "null"
    }
    if(length(gitDF2$name)== 0){
        gitDF2$name= "null"
    }


        if (length(gitDF2$location)!= 0){
```

```
            repos <- paste0(gitDF2$repos_url,"?per_page=100&page=%d"%>>%
            sprintf(1:3),"&client_id=introducir-key&client_secret=introducir-clave")
            %>>%
            list.load("json") %>>%
            list.ungroup
          lenguaje.frecuencias = data.frame(repos %>>%
            list.filter(!is.null(language)) %>>%
            list.table(language) %>>%
            list.sort(-.))


          repos2 <- paste0("https://api.github.com/users/",
gitDF2$login,"/starred?per_page=100&page=%d" %>>% sprintf(1:3),"&client_id=introducir-
key&client_secret=introducir-clave") %>>%
            list.load("json") %>>%
            list.ungroup
          lenguaje.frecuencias2 = data.frame(repos2 %>>%
                                list.filter(!is.null(language)) %>>%
                                list.table(language) %>>%
                                list.sort(-.))


          j = 1
          if(!is.null(lenguaje.frecuencias[j,2])){
          while(as.numeric(lenguaje.frecuencias[j,2]) >= 5 &&
paste0(lenguaje.frecuencias[j,2])!= "NA"){
            vector.lenguaje.frecuencia =
c(vector.lenguaje.frecuencia,paste0(lenguaje.frecuencias[[j,1]]))
            j = j+1
          }
        }



        k = 1
        if(!is.null(lenguaje.frecuencias2[k,2])){
          while(as.numeric(lenguaje.frecuencias2[k,2]) >= 5 &&
paste0(lenguaje.frecuencias2[k,2])!= "NA"){
            vector.lenguaje.frecuencia2 =
c(vector.lenguaje.frecuencia2,paste0(lenguaje.frecuencias2[[k,1]]))
            elementos.frecuencia = repos2 %>>% list.table(id) %>>%list.count()
            k = k+1
          }
        }
```

```
          vector.lenguaje.frecuencia = toString(vector.lenguaje.frecuencia)
          vector.lenguaje.frecuencia2 = toString(vector.lenguaje.frecuencia2)
            usuario =
matrix(c(gitDF2$login,gitDF2$location,gitDF2$name,gitDF2$company,gitDF2$public_repos,pa
ste0(vector.lenguaje.frecuencia[1:length(vector.lenguaje.frecuencia)]),elementos.frecue
ncia,paste0(vector.lenguaje.frecuencia2[1:length(vector.lenguaje.frecuencia2)]),gitDF2$
public_gists,gitDF2$followers,gitDF2$following,gitDF2$updated_at),nrow=1,ncol =
12,byrow=T)
            #dimnames(datos)<-
list(c("usuario","lugar","nombre","compania","repositorios","lenguajes","gists","seguid
ores","siguiendo","ult_actualizacion"))
            informacion.usuario = rbind(informacion.usuario,usuario)


                contador = contador + 1
        }


    datos = datos + 1
    vector.lenguaje.frecuencia = vector()
    vector.lenguaje.frecuencia2 = vector()



      if(datos == 100){
        id = gitDF$id[[100]]
        datos = 1
      }
    }

colnames(informacion.usuario)=list("usuario","lugar","nombre","compania","repositorios"
,"lenguajes","estrellas","lenguaje_estrellas","gists","seguidores","siguiendo","ult_act
ualizacion")
    informacion.usuario <- informacion.usuario[-c(1),]
    informacion.usuario = data.frame(informacion.usuario)
    format.git.date<- function(datestring) {
      date <- as.POSIXct(datestring, format = "%Y-%m-%dT%H:%M:%SZ",
                         tz = "GMT")
  }
  informacion.usuario$ult_actualizacion <-
format.git.date(informacion.usuario$ult_actualizacion)
  write.csv(informacion.usuario, file = "diccionario_usuarios.csv", row.names=FALSE)
```

A.2. Procesamiento del diccionario de usuarios y recomendaciones

```
#*****************************************************************************
#
# procesamiento_recomendacion.R
#
#*****************************************************************************
#
#
# AUTOR: Adrian Gutierrez Alvarez
#
#
# FECHA 28/06/2017
#
#
# DESCRIPCION método para el procesamiento del diccionario de forma automática además
# de utilizar Recommenderlab para recomendar usuarios.
#
#
#*****************************************************************************/
# Procesamiento de los datos ---------------------------------------------


data1 <-  read.csv("diccionario_usuarios.csv", header = TRUE)
#declaramos las variables
contador = 1
informacion.usuario = data.frame()
primer_leng = vector()
primer_gist = vector()
#vector multilenguaje si o no
lenguages <- matrix(ncol=2)
multi=vector()
anio= vector()
mes = vector()
lenguaje_codif=vector()
lenguaje_gist = vector()
codigo_lenguaje= data.frame()
lenguajes <- matrix(ncol=1)
company = vector()
id= vector()
```

```
while(nrow(data1) >= contador){

  separar = unlist(strsplit(paste0(data1[contador,6]),", "))
  separar2 = unlist(strsplit(paste0(data1[contador,8]),", "))
  fecha = unlist(strsplit(paste0(data1[contador,12]),"-"))

  anio <- c(anio,fecha[1])
  mes <- c(mes,fecha[2])

  if(data1[contador,4] == "null"){
    company <- c(company,0)
  }
  else{
    company <- c(company,1)
  }

  if(is.na(separar[1]) && length(primer_leng)!=0){
    primer_leng <- c(primer_leng,"null")
  }
  else{
  primer_leng <- c(primer_leng,separar[1])
  }


  if(is.na(separar2[1]) && length(primer_gist)!=0){
    primer_gist <- c(primer_gist,"null")
  }
  else{
    primer_gist <- c(primer_gist,separar2[1])
  }

  if(length(separar) > 1){
    multi<- c(multi,1)

  }else{
    multi<- c(multi,0)
  }

  for(x in 1:length(separar)){
```

```
    datos <- c(paste0(data1[contador,1]),paste0(as.character(separar[x])))
    lenguages <- rbind(lenguages,datos)
    id <- c(id,contador)
  }


  if(length(separar) !=0){

    for(i in 1:length(separar)){
      insertado = 0
      if(length(codigo_lenguaje)==0){
        codigo_lenguaje <- rbind(separar[i])
        insertado = 1
      }
      else{
      for(j in 1:length(codigo_lenguaje)){
        if(!is.na(separar[i])){
          if(separar[i]==codigo_lenguaje[j]){
            insertado= 1
          }
        }
      }
    }
      if(insertado !=1){
        codigo_lenguaje <- rbind(codigo_lenguaje,separar[i])
      }
    }

  }

  datos <- c(length(separar))
  lenguajes <- rbind(lenguajes,datos)

  contador = contador + 1

}

codigo_lenguaje <- rbind(codigo_lenguaje, "null")
k = 1
limite = 100
```

```
while(k <= length(codigo_lenguaje)){
  informacion.usuario <- rbind(informacion.usuario,limite)
 k = k+1
 limite = limite + 100
}
codigo_lenguaje <- cbind(codigo_lenguaje,informacion.usuario)
colnames(codigo_lenguaje) = list("lenguaje","numero")


for(l in 1:length(primer_leng)){
  for(m in 1:nrow(codigo_lenguaje)){
    if(primer_leng[l]==codigo_lenguaje[m,1]){
      lenguaje_codif <- c(lenguaje_codif,codigo_lenguaje[m,2])
    }
    if(is.na(primer_gist[l])){
      primer_gist[l] = "null"
    }
    if(primer_gist[l]==codigo_lenguaje[m,1]){
      lenguaje_gist <- c(lenguaje_gist,codigo_lenguaje[m,2])
    }
  }

}

datos_recommender <- data.frame(lenguages)
colnames(datos_recommender) = list("usuario","lenguaje")


lenguajes <- lenguajes[-c(1),]

#formamos el diccionario con los datos

parte1 <- cbind(multi,lenguajes)
parte2 <- cbind(parte1,lenguaje_codif)
parte3 <- cbind(parte2,lenguaje_gist)
parte4 <- cbind(parte3,anio)
parte5 <- cbind(parte4,mes)

diccionario <- cbind(parte5,company)
```

```
final <- cbind(informacion.usuario,diccionario)

final <- final[,-c(4,6,8,12)]
final[is.na(final)] <- 0

datos_recommender <- data.frame(lenguages)
colnames(datos_recommender) = list("usuario","lenguaje")
datos_recommender <- datos_recommender[-c(1),]
rownames(datos_recommender) <- NULL


# valoracion de cada usuario ----------------------------------------------


rate= vector()
numero.usuarios = 1
puntuacion = 0
while(nrow(final) >= numero.usuarios){


  if(as.integer(paste0(final[numero.usuarios,4])) > 25){
    puntuacion = puntuacion + 1
  }

  if(as.integer(paste0(final[numero.usuarios,5])) >150){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[numero.usuarios,6])) >100){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[numero.usuarios,7])) >1000){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[numero.usuarios,8])) >100){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[numero.usuarios,9])) == 1){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[numero.usuarios,10])) > 4){
```

```
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[numero.usuarios,13])) == 2017){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[numero.usuarios,14])) > 4){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[numero.usuarios,15])) == 1){
    puntuacion = puntuacion + 1
  }
  rate <- c(rate,puntuacion)
  puntuacion = 0
  numero.usuarios = numero.usuarios +1
}
final2 <- cbind(final,rate)
write.csv(final2, file = "final2.csv", row.names=FALSE)

valoracion = vector()
numero.usuarios2 = 1
while(nrow(final2) >= numero.usuarios2){
  print(numero.usuarios2)
  for(z in 1:nrow(datos_recommender)){
    if(paste0(datos_recommender[z,1]) == paste0(final2[numero.usuarios2,1])){
      valoracion <- c(valoracion,final2[numero.usuarios2,16])
    }
  }

  numero.usuarios2 = numero.usuarios2 + 1
}

ffinal <- cbind(datos_recommender,valoracion)

write.csv(ffinal, file = "final.csv", row.names=FALSE)

# algoritmo ------------------------------------------------------------

library("recommenderlab")
tr<-read.csv("final.csv",header=TRUE, stringsAsFactors=FALSE)
tr[["lenguaje"]][is.na(tr[["lenguaje"]])] <- "null"
```

```
tr[is.na(tr)] <- 0
for(i in 1:nrow(tr)){
  if(tr[i,1] == tr[i,2]){
    tr[i,2] = "null"
  }
}
tr <- cbind(tr$lenguaje,tr$usuario,tr$valoracion)
tr <- data.frame(tr)

tr.matrix<- as(tr,"realRatingMatrix")
valoracion.lenguaje <- getData.frame(tr.matrix)

rec=Recommender(tr.matrix[1:nrow(tr.matrix)],method="IBCF")

usuario.recomendados = data.frame()
usuario.recomendados.finales = data.frame()
for(z in 1:nrow(tr.matrix)){
  recommended.user <- predict(rec, tr.matrix[z,], n=1000)
  recomendados.top5 = as(recommended.user, "list")
  usuario.recomendados <- data.frame(recomendados.top5)
    if(nrow(usuario.recomendados) !=0){
      for(p in 1:nrow(usuario.recomendados)){
        if(paste0(usuario.recomendados[p,1]) == "alex"){
          usuario.recomendados.finales <- data.frame(usuario.recomendados)
          break
        }
    }
    }
}
```

Apéndice B. Funcionalidad extra

B.1. Obtención de datos acerca de contribuciones

```
#*******************************************************************************
#
# obtención_datos_contribuciones.R
#
#*******************************************************************************
#
#
# AUTOR: Adrian Gutierrez Alvarez
#
#
# FECHA 28/06/2017
#
#
# DESCRIPCION método para la obtención y generación del diccionario de contribuciones
# además de obtener los datos del usuario con “issue”.
#
#
#*******************************************************************************/
library(jsonlite)
library(httpuv)
library(httr)
library(pipeR)
library(rlist)

lenguajes = vector()
usuario = vector()
contribuciones = vector()

oauth_endpoints("github")
```

```
myapp <- oauth_app(appname = "aplicacion_tfg",
                   key = "Introducir key",
                   secret = "introducir clave")

github_token <- oauth2.0_token(oauth_endpoints("github"), myapp)

gtoken <- config(token = github_token)


datos <-  read.csv("MyData3.csv", header = TRUE)

for(i in 1:nrow(datos)){

  req <- GET("https://api.github.com/users/"%>>%paste0(datos[i,1],"/events"), gtoken)

  stop_for_status(req)


  json1 = content(req)


  gitDF = jsonlite::fromJSON(jsonlite::toJSON(json1))

  git2 <-data.frame(gitDF$repo)

  # Hasta aquí se ha basado el código en: https://medium.com/towards-data-
science/accessing-data-from-github-api-using-r-3633fb62cb08

  if (length(git2$url)!= 0){
    repos <- GET(paste0(git2$url[1]), gtoken)
    if(repos$status_code != 404){
      stop_for_status(repos)

    json2 = content(repos)
    gitDF2 = jsonlite::fromJSON(jsonlite::toJSON(json2))
      if(length(gitDF2$language)== 0){
        lenguajes <- c(lenguajes,"NA")
      }
      lenguajes <- c(lenguajes,paste0(gitDF2$language))
      usuario <- c(usuario,paste0(datos$usuario[i]))
```

```
      contribuciones <- c(contribuciones,as.integer(nrow(git2)))
    }
  }
  else{
    lenguajes <- c(lenguajes,"NA")
    usuario <- c(usuario,paste0(datos$usuario[i]))
    contribuciones <- c(contribuciones,as.integer(nrow(git2)))
  }
  print(i)
}

final <- cbind(usuario,lenguajes,contribuciones)
final <- data.frame(final)
write.csv(final, file = "contribuciones.csv", row.names=FALSE)

datos<-read.csv("contribuciones.csv",header=TRUE, stringsAsFactors=FALSE)


oauth_endpoints("github")


myapp <- oauth_app(appname = "aplicacion_tfg",
                   key = "Introducir key",
                   secret = "introducir clave")


github_token <- oauth2.0_token(oauth_endpoints("github"), myapp)


gtoken <- config(token = github_token)
datos[is.na(datos)] <- "NA"

datos <- datos[order(datos$lenguajes, datos$contribuciones, decreasing = TRUE),]
row.names(datos) <- NULL


req <- GET("https://api.github.com/search/issues?q=type:pr+state:open+author:"%>>%
             paste0("al3x","&per_page=100&page=1"), gtoken)
```

```
  stop_for_status(req)


  json3 = content(req)

  Usuario_externo = jsonlite::fromJSON(jsonlite::toJSON(json3))

  enlace <- paste0(Usuario_externo$items$repository_url[1])

  req <- GET(enlace, gtoken)

  stop_for_status(req)

  json4 = content(req)

  Usuario_externo_lenguaje = jsonlite::fromJSON(jsonlite::toJSON(json4))
  lenguaje_externo <- paste0(Usuario_externo_lenguaje$language)

  if(lenguaje_externo == "null"){
    print("No reconocemos el lenguaje de su último issue")
  }else{

    contador = 0
    contador2 = 0
    resultado_final = vector()
    for(k in 1:nrow(datos)){
      if(paste0(datos$lenguajes[k]) == lenguaje_externo){
        while(contador < 5){
          contador2 = contador + k
          resultado_final <- c(resultado_final,paste0(datos$usuario[contador2]))
          contador = contador +1
        }
      }
    }
    resultado_final <- paste0(resultado_final)
    print("Hemos detectado que tiene un issue abierto hace poco tiempo, nuestro sistema
ha encontrado a:")
    for(i in 1:length(resultado_final)){
      print(resultado_final[i])
```

```
  }

}
```

B.2. Automatización del SR

```
#*********************************************************************************
#
# automatización_SR.R
#
#*********************************************************************************
#
#
# AUTOR: Adrian Gutierrez Alvarez
#
#
# FECHA 28/06/2017
#
#
# DESCRIPCION: Automatización mediante una función para el SR
#
#
#*********************************************************************************/
recomendaciones<-function(usuario){
library(jsonlite)
library(httpuv)
library(httr)
library(pipeR)
library(rlist)

vector.lenguaje.frecuencia = vector()
informacion.usuario = ""
vector.lenguaje.frecuencia2 = vector()
anio= vector()
mes = vector()
primer_leng = vector()
primer_gist = vector()
multi=vector()
company = vector()
id= vector()
lenguages <- matrix(ncol=2)
lenguajes <- matrix(ncol=1)
codigo_lenguaje= data.frame()
valoracion.lenguaje = data.frame()
lenguaje_codif=vector()
lenguaje_gist = vector()
```

```
   oauth_endpoints("github")


   myapp <- oauth_app(appname = "aplicacion_tfg",
                      key = "Introducir key",
                      secret = "introducir clave")
   github_token <- oauth2.0_token(oauth_endpoints("github"), myapp)


   gtoken <- config(token = github_token)

   req <- GET(paste0("https://api.github.com/users/",usuario), gtoken)

   stop_for_status(req)

   json1 = content(req)

   # Hasta aquí se ha basado el código en: https://medium.com/towards-data-
science/accessing-data-from-github-api-using-r-3633fb62cb08

   gitDF = jsonlite::fromJSON(jsonlite::toJSON(json1))


   req2 <- GET(gitDF$url[[1]], gtoken)
   stop_for_status(req2)
   json2 = content(req2)
   gitDF2 = jsonlite::fromJSON(jsonlite::toJSON(json2))


   if(length(gitDF2$company)== 0){
     gitDF2$company= "null"
   }
   if(length(gitDF2$name)== 0){
     gitDF2$name= "null"
   }



   if (length(gitDF2$location)!= 0){
     #calculamos la frecuencia de los lenguajes en los repos
     repos <- paste0(gitDF2$repos_url,"?per_page=100&page=%d"%>>%
sprintf(1:3),"&client_id=a3d164c4f98c9aea8355&client_secret=b1be4732c7136de6d614eba2420
25edbe98f5275") %>>%
       list.load("json") %>>%
       list.ungroup
     lenguaje.frecuencias = data.frame(repos %>>%
                          list.filter(!is.null(language)) %>>%
                          list.table(language) %>>%
                          list.sort(-.))
     #calculamos la frecuencia de los lenguajes en los usuarios
     repos2 <- paste0("https://api.github.com/users/",
gitDF2$login,"/starred?per_page=100&page=%d" %>>%
sprintf(1:3),"&client_id=a3d164c4f98c9aea8355&client_secret=b1be4732c7136de6d614eba2420
25edbe98f5275") %>>%
       list.load("json") %>>%
       list.ungroup
     lenguaje.frecuencias2 = data.frame(repos2 %>>%
                           list.filter(!is.null(language)) %>>%
                           list.table(language) %>>%
```

```
                        list.sort(-.))

    j = 1
    if(!is.null(lenguaje.frecuencias[j,2])){
      while(as.numeric(lenguaje.frecuencias[j,2]) >= 5 &&
paste0(lenguaje.frecuencias[j,2])!= "NA"){
        vector.lenguaje.frecuencia =
c(vector.lenguaje.frecuencia,paste0(lenguaje.frecuencias[[j,1]]))
        j = j+1
      }
    }


    k = 1
    if(!is.null(lenguaje.frecuencias2[k,2])){
      while(as.numeric(lenguaje.frecuencias2[k,2]) >= 5 &&
paste0(lenguaje.frecuencias2[k,2])!= "NA"){
        vector.lenguaje.frecuencia2 =
c(vector.lenguaje.frecuencia2,paste0(lenguaje.frecuencias2[[k,1]]))
        elementos.frecuencia = repos2 %>>% list.table(id) %>>%list.count()
        k = k+1
      }
    }


    vector.lenguaje.frecuencia = toString(vector.lenguaje.frecuencia)
    vector.lenguaje.frecuencia2 = toString(vector.lenguaje.frecuencia2)
    matrix.usuario =
matrix(c(gitDF2$login,gitDF2$location,gitDF2$name,gitDF2$company,gitDF2$public_repos,pa
ste0(vector.lenguaje.frecuencia[1:length(vector.lenguaje.frecuencia)]),elementos.frecue
ncia,paste0(vector.lenguaje.frecuencia2[1:length(vector.lenguaje.frecuencia2)]),gitDF2$
public_gists,gitDF2$followers,gitDF2$following,gitDF2$updated_at),nrow=1,ncol =
12,byrow=T)
    #dimnames(datos)<-
list(c("usuario","lugar","nombre","compania","repositorios","lenguajes","gists","seguid
ores","siguiendo","ult_actualizacion"))
    informacion.usuario = rbind(informacion.usuario,matrix.usuario)

  }


  colnames(informacion.usuario)=list("usuario","lugar","nombre","compania","repositorio
s","lenguajes","estrellas","lenguaje_estrellas","gists","seguidores","siguiendo","ult_a
ctualizacion")
  informacion.usuario = data.frame(informacion.usuario)
  format.git.date<- function(datestring) {
    date <- as.POSIXct(datestring, format = "%Y-%m-%dT%H:%M:%SZ",
                       tz = "GMT")
  }
  informacion.usuario$ult_actualizacion <-
as.character(format.git.date(informacion.usuario$ult_actualizacion))
  informacion.usuario <- informacion.usuario[-c(1),]

  separar = unlist(strsplit(paste0(informacion.usuario[1,6]),", "))
  separar2 = unlist(strsplit(paste0(informacion.usuario[1,8]),", "))
  fecha = unlist(strsplit(paste0(informacion.usuario[1,12]),"-"))


  anio <- c(anio,fecha[1])
  mes <- c(mes,fecha[2])
```

```
if(informacion.usuario[1,4] == "null"){
  company <- c(company,0)
}else{
  company <- c(company,1)
}

if(is.na(separar[1]) && length(primer_leng)!=0){
  primer_leng <- c(primer_leng,"null")
}else{
  primer_leng <- c(primer_leng,separar[1])
}


if(is.na(separar2[1]) && length(primer_gist)!=0){
  primer_gist <- c(primer_gist,"null")
}else{
  primer_gist <- c(primer_gist,separar2[1])
}

if(length(separar) > 1){
  multi<- c(multi,1)

}else{
  multi<- c(multi,0)
}


for(x in 1:length(separar)){
  datos <- c(paste0(informacion.usuario[1,1]),paste0(as.character(separar[x])))
  lenguages <- rbind(lenguages,datos)
  id <- c(id,1)
}



if(length(separar) !=0){

  for(i in 1:length(separar)){
    insertado = 0
    if(length(codigo_lenguaje)==0){
      codigo_lenguaje <- rbind(separar[i])
      insertado = 1
    }
    else{
      for(j in 1:length(codigo_lenguaje)){
        if(!is.na(separar[i])){
          if(separar[i]==codigo_lenguaje[j]){
            insertado= 1
          }
        }
      }
    }
    if(insertado !=1){
      codigo_lenguaje <- rbind(codigo_lenguaje,separar[i])
    }
  }

}
```

```
datos <- c(length(separar))
lenguajes <- rbind(lenguajes,datos)

for(l in 1:length(primer_leng)){
  for(m in 1:nrow(codigo_lenguaje)){
    if(primer_leng[l]==codigo_lenguaje[m,1]){
      lenguaje_codif <- c(lenguaje_codif,codigo_lenguaje[m,1])
    }
    if(is.na(primer_gist[l])){
      primer_gist[l] = "null"
    }
    if(primer_gist[l]==codigo_lenguaje[m,1]){
      lenguaje_gist <- c(lenguaje_gist,codigo_lenguaje[m,1])
    }
  }

}

# segunda parte procesamiento ---------------------------------------------


codigo_lenguaje <- rbind(codigo_lenguaje, "null")
k = 1
limite = 100
while(k <= length(codigo_lenguaje)){
  valoracion.lenguaje <- rbind(valoracion.lenguaje,limite)
  k = k+1
  limite = limite + 100
}
codigo_lenguaje <- cbind(codigo_lenguaje,valoracion.lenguaje)
colnames(codigo_lenguaje) = list("lenguaje","numero")

for(l in 1:length(primer_leng)){
  for(m in 1:nrow(codigo_lenguaje)){
    if(primer_leng[l]==codigo_lenguaje[m,1]){
      lenguaje_codif <- c(lenguaje_codif,codigo_lenguaje[m,2])
    }
    if(is.na(primer_gist[l])){
      primer_gist[l] = "null"
    }
    if(primer_gist[l]==codigo_lenguaje[m,1]){
      lenguaje_gist <- c(lenguaje_gist,codigo_lenguaje[m,2])
    }
  }

}

datos_recommender <- data.frame(lenguages)
colnames(datos_recommender) = list("usuario","lenguaje")

lenguajes <- lenguajes[-c(1),]

#formamos el diccionario con los datos

parte1 <- cbind(multi,lenguajes)

parte2 <- cbind(parte1,lenguaje_codif)
parte3 <- cbind(parte2,lenguaje_gist)
parte4 <- cbind(parte3,anio)
parte5 <- cbind(parte4,mes)
```

```
diccionario <- cbind(parte5,company)

final <- cbind(informacion.usuario,diccionario)
final <- final[,-c(4,6,8,12)]
final[is.na(final)] <- 0

datos_recommender <- data.frame(lenguages)
colnames(datos_recommender) = list("usuario","lenguaje")
datos_recommender <- datos_recommender[-c(1),]
rownames(datos_recommender) <- NULL

# valoracion de cada usuario ----------------------------------------------


rate= vector()
numero.usuarios = 1
puntuacion = 0
# recorremos el dicionario para valorar al usuario
while(nrow(final) >= numero.usuarios){


  if(as.integer(paste0(final[1,4])) > 25){
    puntuacion = puntuacion + 1
  }

  if(as.integer(paste0(final[1,5])) >150){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[1,6])) >100){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[1,7])) >1000){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[1,8])) >100){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[1,9])) == 1){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[1,10])) > 4){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[1,13])) == 2017){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[1,14])) > 4){
    puntuacion = puntuacion + 1
  }
  if(as.integer(paste0(final[1,15])) == 1){
    puntuacion = puntuacion + 1
  }
  rate <- c(rate,puntuacion)
  puntuacion = 0
  numero.usuarios = numero.usuarios +1
}
final2 <- cbind(final,rate)
```

```
valoracion = vector()
numero.usuarios2 = 1
while(nrow(final2) >= numero.usuarios2){
  print(numero.usuarios2)
  for(z in 1:nrow(datos_recommender)){
    if(paste0(datos_recommender[z,1]) == paste0(final2[numero.usuarios2,1])){
      valoracion <- c(valoracion,final2[numero.usuarios2,16])
    }
  }

  numero.usuarios2 = numero.usuarios2 + 1
}

ffinal <- cbind(datos_recommender,valoracion)




# Parte recomendación ----------------------------------------------------


library("recommenderlab")
tr<-read.csv("final.csv",header=TRUE, stringsAsFactors=FALSE)
tr <- merge.data.frame(tr,ffinal, all = TRUE)
tr[["lenguaje"]][is.na(tr[["lenguaje"]])] <- "null"
tr[is.na(tr)] <- 0
for(i in 1:nrow(tr)){
  if(tr[i,1] == tr[i,2]){
      tr[i,2] = "null"
  }
}
tr <- cbind(tr$lenguaje,tr$usuario,tr$valoracion)
tr <- data.frame(tr)

tr.matrix<- as(tr,"realRatingMatrix")
valoracion.lenguaje <- getData.frame(tr.matrix)

rec=Recommender(tr.matrix[1:nrow(tr.matrix)],method="IBCF")

usuario.recomendados = data.frame()
usuario.recomendados.finales = data.frame()
for(z in 1:nrow(tr.matrix)){
  recommended.user <- predict(rec, tr.matrix[z,], n=1000)
  recomendados.top5 = as(recommended.user, "list")
  usuario.recomendados <- data.frame(recomendados.top5)
  if(nrow(usuario.recomendados) !=0){
    for(p in 1:nrow(usuario.recomendados)){
      if(paste0(usuario.recomendados[p,1]) == usuario){
        usuario.recomendados.finales <- data.frame(usuario.recomendados)
        break
        }
      }
    }
}
#almacenamos los 5 primeros usarios para recomendar
    numero = 0
    numero2 = 1
    resultado = vector()
    while(numero < 5){
      if(paste0(usuario.recomendados.finales[numero2,1]) != usuario){
```

```
        resultado <- c(resultado,paste0(usuario.recomendados.finales[numero2,1]))
        numero = numero + 1
      }
      numero2= numero2 + 1
    }

    return(resultado)

}
```

B.3. Automatización de las contribuciones

```
#*********************************************************************************
#
# automatización_contrib.R
#
#*********************************************************************************
#
#
# AUTOR: Adrian Gutierrez Alvarez
#
#
# FECHA 28/06/2017
#
#
# DESCRIPCION: Automatización mediante una función para las contribuciones
#
#
#*********************************************************************************/
contribuciones<-function(usuario){
library(jsonlite)
library(httpuv)
library(httr)
library(pipeR)
library(rlist)

resultado_final = vector()
datos<-read.csv("C:/r/adri/contribuciones.csv",header=TRUE, stringsAsFactors=FALSE)


oauth_endpoints("github")

myapp <- oauth_app(appname = "aplicacion_tfg",
                   key = "Introducir key",
                   secret = "introducir clave")

github_token <- oauth2.0_token(oauth_endpoints("github"), myapp)


gtoken <- config(token = github_token)
```

```
datos[is.na(datos)] <- "NA"

datos <- datos[order(datos$lenguajes, datos$contribuciones, decreasing = TRUE),]
row.names(datos) <- NULL

# Obtenemos los issues del usuario a recomendar
req <- GET("https://api.github.com/search/issues?q=type:pr+state:open+author:"%>>%
             paste0(usuario,"&per_page=100&page=1"), gtoken)

stop_for_status(req)

json3 = content(req)

Usuario_externo = jsonlite::fromJSON(jsonlite::toJSON(json3))

# Hasta aquí se ha basado el código en: https://medium.com/towards-data-
science/accessing-data-from-github-api-using-r-3633fb62cb08

#comprobamos la existencia de issues
if(length(Usuario_externo$items)== 0){
  return("No existen issues abiertos")
}
else{
  enlace <- paste0(Usuario_externo$items$repository_url[1])

  req <- GET(enlace, gtoken)

  stop_for_status(req)


  json4 = content(req)

  Usuario_externo_lenguaje = jsonlite::fromJSON(jsonlite::toJSON(json4))
  lenguaje_externo <- paste0(Usuario_externo_lenguaje$language)

  if(lenguaje_externo == "null"){
    print("No reconocemos el lenguaje de su último issue")
  }else{

    contador = 0
    contador2 = 0
    resultado_final = vector()
    for(k in 1:nrow(datos)){
      if(paste0(datos$lenguajes[k]) == lenguaje_externo){
        while(contador < 5){
          contador2 = contador + k
          resultado_final <- c(resultado_final,paste0(datos$usuario[contador2]))
          contador = contador +1
        }
      }
    }
    resultado_final <- paste0(resultado_final)

     }
return(resultado_final)
  }
}
```

Apéndice C.
Interfaz gráfica

C.1. Implementación de la GUI

```
#*****************************************************************************
#
# gui.R
#
#*****************************************************************************
#
#
# AUTOR: Adrian Gutierrez Alvarez
#
#
# FECHA 28/06/2017
#
#
# DESCRIPCION: creación de la interfaz gráfica con RGTK2
#
#
#*****************************************************************************/
require("RGtk2")
#añadimos las funciones que recomiendan
source( "C:/r/adri/recomendador.R" )
source( "C:/r/adri/recomendador_grande.R" )



# implementacion ------------------------------------------------------

DoCalculation<-function(button)
{
  if ((TextoUsuario$getText())=="") return(invisible(NULL))


  tryCatch(
    if (TextoUsuario$getText()!=""){
      usuario <- TextoUsuario$getText()

      recomendados<-recomendaciones(usuario)

      recomendados<- paste0("Usuario: ",recomendados," | link:
      ","https://github.com/", recomendados)
```

```
  result$setText(recomendados[1])
  result2$setText(recomendadosm[2])
  result3$setText(recomendados[3])
  result4$setText(recomendados[4])
  result5$setText(recomendados[5])

  contribuidores <- contribuciones(usuario)
  if (length(contribuidores )!=0){
    window2 <- gtkWindow()
    window2["title"] <- "Issue Abierto "


    frame2 <- gtkFrameNew("Hemos detectado que tiene un issue abierto y creemos
que estas personas le pueden ayudar :")
    window2$add(frame2)

    box3 <- gtkVBoxNew()
    box3$setBorderWidth(30)
    frame2$add(box3)

    result5<- gtkEntryNew() #text field with result
    result5$setWidthChars(50)
    box3$packStart(result5)

    result6<- gtkEntryNew()
    result6$setWidthChars(50)
    box3$packStart(result6)

    result7<- gtkEntryNew()
    result7$setWidthChars(50)
    box3$packStart(result7)

    result8<- gtkEntryNew()
    result8$setWidthChars(50)
    box3$packStart(result8)

    result9<- gtkEntryNew()
    result9$setWidthChars(50)
    box3$packStart(result9)
    contribuidores <- paste0("Usuario: ",contribuidores ," | link:
","https://github.com/", contribuidores )
    result5$setText(contribuidores[1])
    result6$setText(contribuidores[2])
    result7$setText(contribuidores[3])
    result8$setText(contribuidores[4])
    result9$setText(contribuidores[5])

    buttonCancel = gtkButtonNewFromStock("gtk-close")
    gSignalConnect(buttonCancel, "clicked", window2$destroy)
    box3$packStart(buttonCancel,fill=F)
  }
}
else (result$setText(as.integer(eval(parse(text=TextToCalculate$getText())))),
error=function(e)
{
  ErrorBox <- gtkDialogNewWithButtons("Error",window, "modal","gtk-ok",
GtkResponseType["ok"])
  box1 <- gtkVBoxNew()
  box1$setBorderWidth(24)
```

```
      ErrorBox$getContentArea()$packStart(box1)

      box2 <- gtkHBoxNew()
      box1$packStart(box2)

      ErrorLabel <- gtkLabelNewWithMnemonic("Error, hay algo mal escrito o no existen
    recomendaciones")
      box2$packStart(ErrorLabel)
      response <- ErrorBox$run()


      if (response == GtkResponseType["ok"])
        ErrorBox$destroy()

    }
  )

}

#creamos la ventana donde mostramos la información

window <- gtkWindow()
window["title"] <- "Recomendador GitHub"


  frame <- gtkFrameNew("Recomendador de usuarios de GitHub")
  window$add(frame)

  box1 <- gtkVBoxNew()
  box1$setBorderWidth(30)
  frame$add(box1)

  label = gtkLabelNewWithMnemonic("Introduzca su usuario")
  box1$packStart(label)

  box2 <- gtkHBoxNew(spacing= 10)
  box2$setBorderWidth(24)

  TextoUsuario<- gtkEntryNew()
  TextoUsuario$setWidthChars(25)
  box1$packStart(TextoUsuario)

  label = gtkLabelNewWithMnemonic("Le recomendamos: ")
  box1$packStart(label)

  result<- gtkEntryNew()
  result$setWidthChars(50)
  box1$packStart(result)

  result2<- gtkEntryNew()
  result2$setWidthChars(50)
  box1$packStart(result2)

  result3<- gtkEntryNew()
  result3$setWidthChars(50)
  box1$packStart(result3)

  result4<- gtkEntryNew()
  result4$setWidthChars(50)
  box1$packStart(result4)
```

```
  result5<- gtkEntryNew()
  result5$setWidthChars(50)
  box1$packStart(result5)

  box2 <- gtkHBoxNew(spacing= 20)
  box2$setBorderWidth(24)
  box1$packStart(box2)

  #añadimos los botones de recomendar y cerrar la ventana.
  Recomendar <- gtkButton("Recomendar")
  box2$packStart(Recomendar,fill=F)

  gSignalConnect(Recomendar, "clicked", DoCalculation)

  buttonCancel2 = gtkButton("Cerrar")
  gSignalConnect(buttonCancel2, "clicked", window$destroy)
  box2$packStart(buttonCancel2,fill=F)

#código basado de: https://www.r-bloggers.com/playing-with-guis-in-r-with-rgtk2/
```