Aplicación para el análisis de comunidades en GitHub basándose en grafos de interés

TITULACIÓN: Grado en Ingeniería Informática

AUTOR: Alejandro Betancor del Rosario

TUTORIZADO POR:
Nelson Monzón López

14 de junio de 2022

Agradecimientos

Agradecer a mi familia y amigos por haberme apoyado y haberme enseñado valores como la disciplina y el disfrutar de lo que haces, virtudes que me han llevado hasta aquí, a mi pareja que ha estado a mi lado en los momentos difíciles, a mi hermana por escucharme y aconsejarme, y al profesor Nelson Monzón López, por guiarme, enseñarme y apoyarme durante la realización de este proyecto con el que finalizo mis estudios de Grado.

Gracias

Resumen

GitHub es, probablemente, la plataforma de desarrollo colaborativo más popular para el control de versiones y gestión de código fuente basado en GIT. Entre otras características, permite a sus usuarios marcar con una estrella los repositorios que les generen mayor interés. Estos usuarios reciben el nombre de "stargazers" y, gracias a este sistema, podemos recolectar información relevante acerca de la comunidad de profesionales que se forma alrededor de un proyecto.

Por ejemplo, permite detectar cuales son los usuarios más relevantes asociados al repositorio, otros tipos de proyectos (repositorios) que les interesen, lenguajes de programación más utilizados, etc. Esta información, y las distintas relaciones que se generan entre los "stargazers", permiten entender objetivos, intereses y gustos de una determinada comunidad profesional de desarrollo de software.

A tenor de lo expuesto, este Trabajo Fin de Título propone una aplicación de escritorio que sirva como herramienta para desarrolladores e investigadores a la hora de realizar análisis acerca de dichas comunidades y proyectos extrayendo los datos de la API de la plataforma y relacionándolos mediante grafos de interés donde los nodos representan los *stargazers* y repositorios que les interesan y las aristas, las relaciones entre ambos, para luego aplicar algoritmos como *PageRank* que permitan obtener información sobre la relevancia de los usuarios y repositorios dentro la comunidad, entre otros datos.

Abstract

GitHub is probably the most popular collaborative development platform for version control and source code management based on GIT. Among other features, allows its users to mark with a star the repositories that generate them major interest. These users are called *stargazers* and, thanks to this system, we can collect relevant information about the community of professionals that forms around a project.

For example, it allows detecting which are the most relevant users associated with the repository, other types of projects (repositories) that interest them, most used programming languages, etc. This information, and the different relationships that are generated between the *stargazers*, allow us to understand objectives, interests and tastes of a certain professional software development community.

This Final Degree Project proposes a desktop application that serves as a tool for developers and researchers when carrying out analyzes about these communities and projects extracting the data from the platform's API and relating them through graphs of interest where the nodes represent the *stargazers* and repositories that interest them and the edges, the relationships between them, to then apply algorithms such as *PageRank* that allow obtaining information about the relevance of users and repositories within the community, among other data.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Motivación

Durante la última década, distintas organizaciones se han percatado del impacto que tienen las redes sociales en nuestras vidas y las implicaciones de estas en cuestiones del ámbito económico, político y sanitario. La mayoría de estudios se centran en el análisis de las comunidades de usuarios que se forman en las redes sociales más conocidas, como Facebook o Instagram, sin embargo, también existen **redes sociales de trabajo** como es el caso de GitHub, de las que se puede extraer información relevante.

GitHub es una plataforma de alojamiento de repositorios y control de versiones, donde los programadores pueden subir el código que desarrollan y trabajar en colaboración para mejorarlo. Mediante el control de versiones distribuido (Git) la plataforma permite a los desarrolladores rastrear y gestionar los cambios en el código fuente, ayudando a que los equipos trabajen de forma más rápida e inteligente. Este código es alojado en un repositorio, el cual, es una ubicación o ruta única donde se almacena la información del proyecto. Otras de las funcionalidades más importantes incluyen los *branchs*, es decir, ramas que son creadas a partir de la rama principal del proyecto y que permiten a los desarrolladores hacer pruebas sin afectar al código principal o los *pull request*, los cuales, permiten realizar una validación del código por otros desarrolladores antes de subirlo al proyecto.

Estas características han impulsado a GitHub convirtiéndola en la plataforma líder en el sector, con una media de 1.6 repositorios creados por minuto en 2018. El número total desarrolladores alcanzá los 73 millones con más de 16 millones de nuevos usuarios en 2021. De acuerdo con las cifras públicadas, se espera que el número total de usuarios escale a más de 100 millones en 2025.

Este trabajo tiene como objetivo crear una herramienta eficiente y de fácil uso que permita analizar dichos usuarios y repositorios. El método empleado para ejecutar dicho análisis consiste en extraer los *stargazers* de un proyecto en específico, extraer los últimos repositorios que les ha interesado, unirlos en un grafo de interés y aplicar algoritmos como *PageRank* de tal modo que se pueda obtener cuáles de esos repositorios y usuarios son los más relevantes, además, de otros datos como cuáles son los tópicos, lenguajes de programación y licencias en los que más están

interesados los desarrolladores.

1.2. Descripción del proyecto

La aplicación dispondrá de una interfaz por línea de comandos (CLI) donde el usuario deberá introducir dos parámetros, el nombre completo del repositorio, y el *token* de autenticación de su cuenta de GitHub. Una vez introducida esta información, la aplicación extraerá los datos necesarios de la API REST de la plataforma, empleando técnicas de programación asíncrona para agilizar dicho proceso. Estos datos son usados posteriormente para generar un grafo directo donde los nodos representan los usuarios/repositorios y las aristas representan las relaciones existentes entre usuarios que se siguen y repositorios que le gustan a los usuarios. Una vez generado el grafo, se dispondrán de múltiples opciones entre las que se encuentran guardar el grafo o elegir que información quiere obtener de este. Al indicar que información se quiere extraer, por ejemplo, licencias más utilizadas por los repositorios que siguen, se mostrará por pantalla las diez más usadas, además, se generan una serie de gráficas representando los resultados.

1.3. Objetivos del proyecto

Se plantean alcanzar los siguientes objetivos en el proyecto.

- Profundizar conocimientos en el lenguaje de programación Python.
- Familiarizarse en la creación y análisis de grafos de interés con un gran número de nodos.
- Aprender nuevas tecnologías, extender conocimientos en ingeniería del software y aplicar técnicas de desarrollo nuevas para el alumno.
- Familiarizarse con metodologías ágiles, aplicando un flujo de trabajo orientado a técnicas de integración y entrega continua.

1.4. Competencias específicas

Las competencias específicas de mención que cubre el alumno con la realización de este proyecto son:

1. TI02: Capacidad para seleccionar, diseñar, desplegar, integrar, evaluar, construir, gestionar, explotar y mantener las tecnologías de hardware, software y redes, dentro de los parámetros de coste y calidad adecuados.
   Se cubre esta competencia, ya que, uno de los objetivos fundamentales durante la creación del proyecto, fue desarrollar un código limpio y mantenible.
2. TI03: Capacidad para emplear metodologías centradas en el usuario y la organización para el desarrollo, evaluación y gestión de aplicaciones y sistemas basados en tecnologías

de la información que aseguren la accesibilidad, ergonomía y usabilidad de los sistemas. **El desarrollo de la aplicación se realizó utilizando una metodología ágil, por lo que, se puede confirmar que se cumple con dicha competencia**.

1.5. Presupuesto

En la siguiente sección se valorarán los costes que supone el proyecto.

1.5.1. Coste hardware

Recurso	Coste
Ordenador portátil	900€.
Total	900€

Cuadro 1.1: Coste hardware

1.5.2. Coste personal

Personal	Nº horas	Coste/Hora	Coste
Alumno	300	14€	4200€
Tutor	20	30€	600€
Total			4800€

Cuadro 1.2: Coste personal

1.5.3. Presupuesto final

Concepto	Coste
Coste hardware	900€.
Coste Personal	4800€
Presupuesto total	5700€

Cuadro 1.3: Coste total

Capítulo 2

Estado del arte

2.1. Medidas de centralidad *Personalized PageRank*

2.1.1. *Random Walk*

El algoritmo de *PageRank* está basado en los *random walks* [35] o caminos aleatorios, por ende, para entender este algoritmo correctamente, primero se ha de explicar dicho modelo matemático. Dado un grafo $G = (V, E)$ comenzando en uno de sus nodos, se avanza aleatoriamente hacia uno de sus vecinos, una vez se encuentra en este, avanza hacia otro de sus vecinos y sigue iterando sucesivamente hasta alcanzar la convergencia, es decir, hasta que no haya cambios significativos en el valor de los nodos. Esta idea es definida como *random walker*. Por ejemplo, supongamos un grafo formado por los nodos $n1, n2, n3, n4$ y $n5$ donde el *random walker* comienza en el nodo $n1$ 2.1.

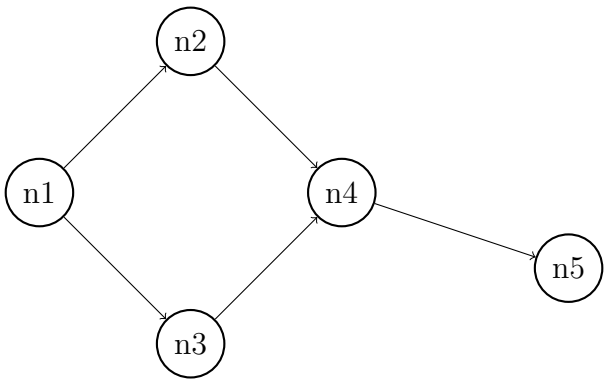

Figura 2.1: Grafo de ejemplo formado por cinco vértices.

La posición del *random walker* puede ser representada mediante un vector de probabilidad $\vec{v} = (1, 0, 0, 0)$, en este caso muestra el estado en el cual el *random walker* reside en $n1$, luego el estado del vector de probabilidad para el caso en el que se desplaza a $n2$ o $n3$ será $\vec{v} = (0, 1/2, 1/2, 0)$.

$$n_{ij} = \begin{cases} \dfrac{1}{|O(n_i)|'} & \text{if j} \in O(n_i) \\ 0 & otherwise \end{cases} \tag{2.1}$$

Donde $O(n_i)$ son los vecinos salientes del vértice i. A partir de la anterior fórmula se puede comprobar que cumple con las propiedades de Márkov, es decir, el valor del vector de estado del siguiente paso del *random walker* depende del valor del vector de estado actual.

2.1.2. *PageRank*

El algoritmo de *PageRank* [19] se apoya de un *surfero web* imaginario que se mueve mediante enlaces a través de los distintos sitios, luego la popularidad de un nodo dependerá de la probabilidad de que el surfero visite dicho nodo. Esta suposición puede ser interpretada dentro del modelo del *random walker*, por ende, el valor de *PageRank* de un nodo dependerá del número de nodos que enlacen con él y valor de *PageRank* de cada uno de esos sitios además del número de enlaces salientes que contengan.

$$PR(i) = \sum_{j=1}^{n} \frac{PR(j)}{C(j)} \tag{2.2}$$

Esta fórmula esta incompleta, ya que se pueden dar otras dos situaciones, que dos o más nodos se referencien entre ellos formando un bucle, también denominado *spider trap problem* o que un nodo no tenga enlaces salientes, *death nodes*. La solución a este problema se denomina *teleportation*, el cual introduce la variable d denominada factor de amortiguación, que representa la probabilidad de que el *random walker* salte a otro nodo. De este modo todos los nodos del grafo se encontrarían interconcetados.

$$PR(i) = \frac{(1-d)}{N} + d \sum_{j=1}^{n} \frac{PR(j)}{C(j)} \tag{2.3}$$

El valor que generalmente tiene el factor de amortiguación es 0.85. N es el número total de nodos del grafo.

2.1.3. *Personalized PageRank*

La versión personalizada del algoritmo de *PageRank* [19] sustituye $\frac{1}{N}$ por el vector de personalización $\vec{v}$.

$$PR(i) = (1-d)\vec{v} + d\sum_{j=1}^{n}\frac{PR(j)}{C(j)} \tag{2.4}$$

Los valores del vector de personalización suelen estar definidos entre 0 y 1, en el caso de los sitios web, determina si un sitio es relevante para un usuario o si por el contrario no le interesa.

2.2. *Extreme Gradient Boosting*

XGBoost [7] es una implementación *open source* del algoritmo de *gradient boosting trees*, el cual es un algoritmo de aprendizaje supervizado, que intenta predecir una variable objetivo con alta presición, realizando una combinación entre las estimaciones de conjuntos de modelos más simples y débiles.

Los árboles de regresión son empleados en modelos de *gradient boosting* para regresión, donde cada árbol asigna un punto de datos a una de sus hojas que contiene una puntuación continua. *XGBoost*, busca minimizar una función objetivo que combina una función de pérdida basada en la diferencia entre el resultado predicho y el objetivo, y un término de regularización. Durante el entrenamiento se procede iterativamente, agregando nuevos árboles que predicen los residuos/errores del árbol anterior que a su vez combinan el resultado con el de otros árboles anteriores para realizar la predicción final.

2.2.1. Elementos del aprendizaje supervizado

Antes de entrar a explicar los árboles en profundidad, será necesario explicar algunos conceptos acerca del aprendizaje supervizado. *XGBoost* es usado para resolver este tipo de problemas donde se requiere generar un modelo a partir de un conjunto de datos de entrenamiento x_i que sea capas de predecir unos valores objetivo y_i. Estos problemas pueden ser de regresión, clasificación o de ranking. Los datos con los que trabaja el algoritmo son estructurados y tabulares.

El **modelo** [5] hace referencia a la estructura matemática en la que a partir de una entrada de datos x_i es realizada una predicción y_i, un ejemplo pueden ser los modelos líneales representado con $\hat{y}_i = \sum_j \theta_j x_{ij}$, donde θ representa los **parámetros** los cuales no han sido determiandos y deben de obtenerse a partir de los datos . Dependiendo de si se requiere para regresión o clasificación, el valor de predicción puede tener distintas interpretaciones.

Para poder entrenar el modelo, de tal modo que se encuentren los mejores parámetros θ para el conjunto x_i e y_i, será necesario definir una **función objetivo** [6] para medir que tan bien se ajusta el modelo a los datos de entrenamiento.

$$obj(\theta) = L(\theta) + \Omega(\theta) \tag{2.5}$$

donde $L(\theta)$ representa la **función de pérdida** y $\Omega(\theta)$ el término de regularización. La función de pérdida mide como de predictivo es el modelo respecto a los datos de entrenamiento. Una elección común para L en problemas de regresión, es el error cuadrático medio.

$$L(\theta) = \sum_i (y_i - \hat{y}_i)^2 \tag{2.6}$$

En problemas de clasificación binaria, suele ser utilizado el *logistic loss*.

$$L(\theta) = \sum_i [y_i ln(1 + e^{-\hat{y}_i}) + (1 - y_i) ln(1 + e^{\hat{y}_i})] \tag{2.7}$$

El término de regularización permite controlar la complejidad del modelo de tal modo que se pueda evitar el sobreajuste, que es el efecto causado por el sobreentrenamiento con unos datos para los que se conoce el resultado.

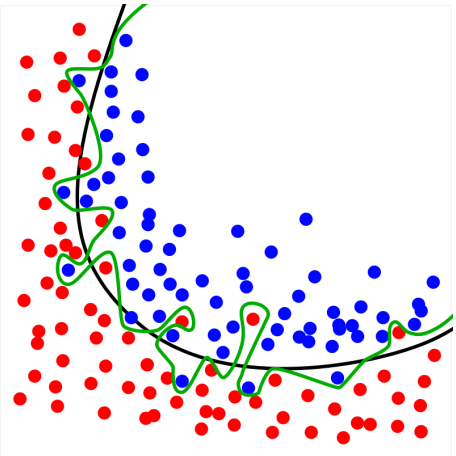

Figura 2.2: Ejemplo sobreajuste. Figura extraida de [34]

En la figura 2.2, se puede apreciar que la línea verde se ajusta mejor a los datos, no obstante si se utilizara otro *dataset* la línea negra se ajustaría mejor debido a que la verde está demasiado ajustada al conjunto de datos anterior. Este problema es el que intenta evitar la regularización, buscando generar modelos **predictivos** y **simples**, es decir, modelos con un sesgo y varianza compensados.

2.2.2. Conjunto de árboles de decisión

Los conjuntos de árboles de decisión, son un modelo formado por árboles de regresión y clasificación (CART), los cuales son diferentes de los árboles de decisión en los que la hoja contiene valores de decisión. En un CART, a cada hoja se le aporta una puntuación real, aportando mejores interpretaciones y permitiendo un enfoque unificado basado en principios para la optimización. Generalmente, se usa un conjunto de árboles y se suman sus predicciones resultantes ya que un solo árbol no es suficiente para realizar el entrenamiento.

Siguiendo el modelo, el valor de la predicción de cada árbol se suma al de los demás para obtener la puntuación final.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i) \tag{2.8}$$

$f_j(x_i)$ es la predicción de cada árbol para los datos y k representa el número total de árboles. Luego, la función objetivo a optimizar sería:

$$obj(\theta) = \sum_{i}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{k} \omega(f_k) \tag{2.9}$$

$\omega(f_k)$ representa la complejidad de árbol f_k.

2.2.3. *Tree Boosting*

Como se pudo observar con anterioridad cada función f_k, contiene la estructura del árbol junto con la puntuación de sus hojas. Aprender todos los árboles a la vez no es posible, por ello se aplica una estrategia conocida como *additive training* [3], donde cada árbol añadido intenta corregir los errores del anterior.

$$\begin{aligned}
&\hat{y}_i{}^{(0)} = 0 \\
&\hat{y}_i{}^{(1)} = f_1(x_i) = \hat{y}_i{}^{(0)} + f_1(x_i) \\
&\hat{y}_i{}^{(2)} = f_1(x_i) + f_2(x_i) = \hat{y}_i{}^{(1)} + f_2(x_i) \\
&\quad ... \\
&\hat{y}_i{}^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i{}^{(t-1)} + f_t(x_i)
\end{aligned}$$

Luego, se requiere usar en cada paso, el árbol que optimiza la función objetivo de tal modo que reemplazando la función de pérdida, quedaría de la siguiente manera:

$$obj(\theta) = \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(x_i)) + \omega(f_t) + constant \tag{2.10}$$

Si se utilizara como función de pérdida el error cuadrático medio, se sustituiría $l(y_i, \hat{y_i}) = (y_i - \hat{y_i})^2$ y quedaría de la siguiente forma:

$$obj(\theta) = \sum_{i=1}^{n} [2(\hat{y_i}^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2] + \omega(f_t) + constant \tag{2.11}$$

Cuando se utilizan otras funciones de pérdida diferentes al error cuadrático medio, la ecuación puede complicarse considerablemente, por ello se aplica la serie de *Taylor* de segundo orden.

$$obj(\theta) = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \omega(f_t) + constant \tag{2.12}$$

Donde g_i y h_i representan:

$$g_i = \partial_{\hat{y_i}^{t-1}} l(y_i, \hat{y}^{t-1}) \tag{2.13}$$
$$h_i = \partial^2_{\hat{y_i}^{t-1}} l(y_i, \hat{y}^{t-1}) \tag{2.14}$$

Esta sería la meta de optimización para cada nuevo árbol, siendo una de las ventajas principales de esta definición, que los valores de la función objetivo solo dependen de g_i y h_i.

2.2.4. Complejidad de modelo

La complejidad del modelo [8] dependerá del número de parámetros libres que este contenga, a más parámetros libres, mayor complejidad. La regularización funciona como restricción a dichos parámetros de tal modo que se pueda obtener un modelo más simple. Por ello la complejidad del modelo o factor de regularización vendría representada por la siguente fórmula:

$$\omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} \omega_j^2 \tag{2.15}$$

T es el número de hojas del árbol y ω_j es la puntuación de cada una de esas hojas, a más hojas, mayor será el número de parámetros libres y entre mayor sean los pesos, el modelo será más complejo (similar a *ridge regression* [1]).La definición de la complejidad de un modelo no está estandarizada, sin embargo esta es la más utilizada.

[1]https://www.mygreatlearning.com/blog/what-is-ridge-regression/

2.2.5. Puntuación de la estructura

Tras analizar la complejidad del modelo, se puede reformular la función objetivo de la siguietne manera:

$$obj^{(t)} = \sum_{j=1}^{T} [(\sum_{i \epsilon I_i} g_i)\omega_j + \frac{1}{2}(\sum_{i \epsilon I_i} h_i + \lambda)\omega_j^2] + \gamma T \tag{2.16}$$

$I_j = i|q(x_i) = j$ es el conjunto de instancias de la hoja j, es decir todos los datos que pertencen a dicha hoja, donde $q(x_i)$ es la función que mapea una instancia x_i a su número de hoja. La función objetivo queda como la suma de T cuadrática independientes, pudiéndose reescribir de la siguiente forma.

$$obj^{(t)} = \sum_{j=1}^{T} [G_j\omega_j + \frac{1}{2}(H_j + \lambda)\omega_j^2] + \gamma T \tag{2.17}$$

Donde G_j y H_j representan.

$$G_j = \sum_{i \epsilon I_i} g_i \tag{2.18}$$

$$H_j = \sum_{i \epsilon I_i} h_i \tag{2.19}$$

El peso óptimo de cada hoja se verá representado por:

$$\omega_j^* = -\frac{G_j}{H_j + \lambda} \tag{2.20}$$

Por ende, la función objetivo para un determinado árbol será:

$$obj^* = -\frac{1}{2}\sum_{j=1}^{T} \frac{G_j}{H_j + \lambda} + \gamma T \tag{2.21}$$

2.2.6. Esctructura del árbol

Al haber muchas estructuras posibles para el árbol, *XGBoost* lo hará crecer de forma *greedy* usando una función de ganancia [4] que define a partir de que atributo se realizará la separación, dependiendo de si el costo de agregar dicha hoja, es mayor que la ganancia que se puede obtener al realizar la separación. En caso de serlo no se añadiría la rama, esto es conocido como **pruning** o poda. La fórmula de la ganancia es la siguiente.

$$Gain = \frac{1}{2}[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}] - \gamma \tag{2.22}$$

Donde el primer término, representa el puntaje del hijo izquierdo, el segundo término el puntaje del hijo derecho, el tercer término, el puntaje de la hoja original y gamma, el costo de agregar otra hoja.

Por último, se ordenarían los atributos según su valor y se buscaría la separación de mayor ganancia. Esto es en el caso de variables numéricas, si estás fueran categóricas, primero se emplearía *one-hot encoding* para convertirlas en variables binarias.

2.3. Mining the Network of the Programmers: A Data-Driven Analysis of GitHub

En el siguiente artículo científico [15], realizado en 2017 por miembros de la escuela de ciencias de la computación de la universidad de Fundan [2]en china, crean un grafo de interés a partir de un *dataset* de más de 2 millones de usuarios, luego realizan un análisis del mismo, investigan acerca de los patrones de comportamiento de los usuarios, haciendo énfasis en los *commits* y finalmente, aplican técnicas de *machine learning* para extraer con alta precisión los usuarios más relevantes de la red.

Attribute	Definition	Value
Nodes	Number of nodes.	2,006,356
Edges	Number of edges.	10,034,342
Nodes with 0 in-degree	Number of nodes which do not have edges pointing to them.	555,231
Nodes with 0 out-degree	Number of nodes with no edges pointing to others.	517,422
Nodes with positive in&out-degree	Number of nodes whose both in and out-degree is more than 0.	934,228
Directed edges	Number of directed edges.	10,034,342
Undirected edges	Number of edges in the undirected version of the network	9,070,211
Mutual edges	Number of edge pairs between two nodes	1,928,262
Diameter of graph	The length of the longest "shortest path" in a graph.	19

Figura 2.3: Información acerca del grafo, los vértices y las aristas. Extraído de [15].

La recolección de datos se efectuó usando un sistema distribuido formado por 21 nodos en un centro de datos de Amazon Web Services en Estados Unidos. Uno de los nodos es el planificador y los demás son trabajadores. El planificador se encargaría de mantener una base de datos donde se guardan los datos recolectados por los trabajadores. El sistema de recolección distribuido, selecciona un usuario aleatoriamente y aplica el algoritmo de búsqueda en anchura para las páginas de perfil de usuarios y la lista de usuarios que siguen y de seguidores. En total se tardaron 11 días en recopilar todos los datos.

[2]https://www.fudan.edu.cn/en/

El siguiente paso que realizaron consistió en generar un grafo directo $G = (V, E)$, donde las aristas representan la relación entre usuarios que se siguen. Por la forma en la que se extrajo la información, todos los usuarios se ven de alguna manera conectados. Posteriormente, analizaron los patrones en los *commits*, dándose cuenta de que la mayor cantidad de contribuciones se hacen de Lunes a Jueves y observando que la productividad cae considerablemente el fin de semana incluyendo los Viernes, además, se observó que este patrón varía considerablemente dependiendo del país, por ejemplo, la participación de los usuarios de nacionalidad china en los proyectos cae considerablemente durante febrero debido al año nuevo chino. En resumen, el análisis temporal y espacial del comportamiento del usuario revela ciertos hechos del mundo real.

Luego, aplican el algoritmo de *Personalized PageRank*, al grafo generado anteriormente con la intención de extraer los usuarios más relevantes de la red. Añadieron las siguientes tres métricas para efectuar la clasificación, número de contribuciones en el último año, número de repositorios que les interesan, y el *strike period*, es decir, el periodo de trabajo continuo más largo en el último año. Del resultado final se extrajeron los 2000 usuarios que más participan en la plataforma. Uno de los mayores problemas, es que este algoritmo requiere iterar sobre todo el grafo y por ende el consumo de recursos es muy alto, por ello, decidieron crear un modelo de *machine learning* que facilitara el trabajo. La creación del dataset se realizó con el 1 % de los usuarios más importantes, es decir, unos 20.000 y otros 20.000 usuarios aleatorios del resto de usuarios no relevantes.

Se añadieron catorce características en el análisis, extraídas del perfil de cada usuario y para la creación del modelo se empleó el algoritmo de *Gradient Boosting*. El resultado final muestra que a partir del comportamiento de los usuarios y características extraídas de su perfil se pueden obtener los usuarios relevantes de GitHub con alta precisión.

Feature	Explanation
repos	#Repositories the user has
forks	Sum of #forks for original repos
max-forked-repo	#Forks of repo with highest #forks
account age	Days since registration
starred-repos	#Repos that have been starred
streak	Length of streak period last year
contributions	#Contributions to any repos
starred	Total #stars the user gets
stars	Total #stars the user gives to others
organizations	#organizations the user joins
curr-streak	Length of current streak period
recent-contributions	#contributions in last year
location	Whether the user has location info
email	Whether the user has public email

Figura 2.4: Características de los usuarios añadidas al análisis. Extraído de [15].

2.4. Network Structure of Social Coding in GitHub

El trabajo de investigación publicado en [30] fue desarrollado por un equipo formado por miembros de la *Singapore Management University* [3]en Singapur y el *Laboratoire Bordelais de Recherche en Informatique* [4]en Francia y su principal objetivo es, estudiar la relación entre usuarios y proyectos, identificando cuáles de estos usuarios y repositorios son relevantes. El número de usuarios con los que se realizó el estudio fue de 30.000 y de proyectos 100.000 en total.

Se construyeron dos grafos diferentes, el primero únicamente formado por proyectos, donde cada vértice representa un proyecto y dos de ellos tendrán una relación cuando dos usuarios trabajen a la vez en el mismo, también asignan un peso a dicha arista, el cual, corresponde al número de usuarios que participan en ambos proyecto. El segundo grafo estaría formado únicamente por los desarrolladores, en caso de que dos usuarios participen en un proyecto en común, se crea una relación entre ellos, asignando como peso de la arista el número de proyectos que ambos desarrolladores tienen en común.

La primera cuestión a resolver es saber como de fuertes son las relaciones entre los distintos proyectos. Se generaron un total de 1.116.533 aristas, lo cual significa, que al menos existe el mismo número de parejas de usuarios trabajando en un mismo proyecto, luego, hallaron que el diámetro completo del grafo es 9, y que la media del camino más corto entre nodos es 3.7. Estos números son más bajos que los encontrados en otro tipo de redes, lo que implica que las redes de proyectos están más interconectadas que las redes formadas por personas. Respecto a la red formada únicamente por usuarios, el número de aristas es de 23.678.445, revelando el número mínimo de desarrolladores que comparten el mismo proyecto. En este caso, el diámetro del grafo resulta en 5 y la media del camino más corto es 2.47, comparando este resultado con el de otras redes como Facebook, se puede concluir que plataformas como GitHub potencian la colaboración entre desarrolladores.

Finalmente, aplican el algoritmo de *PageRank* en ambos grafos, de tal modo que extraen los usuarios y repositorios más importantes de ambas redes.

2.5. ¿Qué aporta este proyecto?

La principal diferencia entre este Trabajo Fin de Grado y los estudios mostrados con anterioridad, es que, este proyecto tiene como objetivo ser una **herramienta**, en los casos anteriores el objetivo era realizar un estudio acerca de una comunidad de usuarios y proyectos en un momento en específico, sin embargo, con este proyecto se pretende que desarrolladores y grupos de investigación puedan realizar estudios de la comunidad que se forma alrededor de un repositorio en GitHub en cualquier momento y sin tener que emplear gran cantidad de recursos. Otro de los aspectos que diferencian este proyecto con respecto a los estudios anteriores, específicamente el de la universidad de Fundan, es que los datos son extraídos de la API de GitHub en vez de utilizar técnicas de *web scraping*, esto limita la velocidad de extracción de los datos debido a

[4]https://www.smu.edu.sg/
[4]https://www.labri.fr/en

los *rate limits* establecidos por la API, sin embargo, el programa no se verá afectado en el caso de que haya cambios en la estructura de la web de la plataforma y además, el alcance de la información a extraer es mucho mayor.

Otro punto que diferencia este trabajo, es que se busca maximizar la cantidad de información extraída de dicha comunidad, extrayendo datos acerca de los proyectos, lenguajes o tópicos que les interesa a los usuarios, además de aplicar algoritmos como *PageRank* o su versión personalizada para extraer que usuarios y repositorios son los más relevantes dentro de la comunidad. También se desarrollará un modelo de *machine learning* que identifique los repositorios más importantes, reduciendo el consumo de recursos.

Capítulo 3

Recursos utilizados

3.1. Tecnologías y herramientas

3.1.1. Python

Python [32] es un lenguaje de programación interpretado, interactivo y orientado a objetos. Entre las ventajas de este lenguaje de programación se encuentran su sencilla sintaxis que facilita la lectura del código y por ende su mantenimiento, un intérprete que dispone de un modo interactivo, el cual permite escribir instrucciones y una amplia variedad de librerías. Estas características han hecho que Python sea líder en campos como la ciencia de datos y que tenga un gran uso en el desarrollo de aplicaciones web en el lado del servidor.

Figura 3.1: Simbolo de python.

El proyecto está escrito en su totalidad en este lenguaje debido a que dispone de librerías que facilitan la creación de grafos, la extracción de información o la representación de gráficas.

3.1.2. Docker

Docker [11] es un proyecto de código abierto que permite empaquetar software en unidades estandarizadas llamadas *contenedores* que incluyen las librerías, herramientas del sistema y código necesario para que ejecute la aplicación.

Esta tecnología divide los procesos y los ejecuta independientemente, apoyándose de algunas funciones del kernel de Linux como los grupos de control y los espacios de nombres ya que, utiliza el núcleo de dicho sistema operativo. Los contenedores tienen como objetivo ejecutar varios procesos y aplicaciones por separado para aprovechar mejor la infraestructura y mantener la seguridad que se obtendría con los sistemas individuales.

Figura 3.2: Logo Docker.

La librería utilizada para generar los grafos está contenida en una imagen de Docker Arch GNU/Linux por ello fue necesario contenerizar el proyecto.

3.1.3. Visual Studio Code

Visual Studio Code [33] es un editor de código fuente gratuito y de código abierto elaborado por Microsoft e incluye herramientas de depuración, control integrado de Git, finalización inteligente de código, refactorización de código y resaltado de sintaxis. Está desarrollado con Electron, un framework que permite el desarrollo de aplicaciones de escritorio usando tecnologías web.

Figura 3.3: Logo Visual Studio Code.

Este es el IDE seleccionado para el desarrollo del proyecto, debido a la experiencia previa del alumno con dicha herramienta y la existencia de plugins que facilitan el desarrollo de software en contenedores de docker.

3.1.4. SonarQube

SonarQube [27] es una plataforma *open source* para la inspección continua de la calidad del código, para realizar revisiones automáticas con análisis estático de código para detectar, *bugs* y *code smells* en más de 17 lenguajes de programación distintos. También ofrece reportes sobre código duplicado, estilo de programación, tests, cobertura de código, complejidad de código, comentarios y recomendaciones de seguridad.

Figura 3.4: Logo SonarQube.

Esta herramienta se emplea en la última etapa del proyecto para mantener un código limpio y evitar errores.

3.1.5. Overleaf

Overleaf [10] es un editor online en la nube y colaborativo que permite escribir texto en LaTex, ques un sistema de composición de texto ampliamente utilizando en la redacción de documentos científico-técnicos. Overleaf ha sido la herramienta utilizada para redactar este documento de TFG y, dada su naturaleza *cloud*, ha permitido la colaboración entre estudiante y profesor.

Figura 3.5: Logo Overleaf.

3.2. Librerías

3.2.1. graph-tool

Graph-tool [20] es un módulo de Python para la manipulación y el análisis de grafos. La principal diferencia con respecto a otras librerías de Python que disponen de funcionalidades parecidas es que el núcleo de las estructuras de datos y los algoritmos están implementados en C++.

Figura 3.6: Logo Graph-tool.

Este módulo será empleado para la creación del grafo y la aplicación de los algoritmos.

3.2.2. asyncio

Asyncio [21] es un módulo de Python que permite la creación de programas concurrentes mediante programación asíncrona. Es usado en múltiples proyectos de Python para dar un alto rendimiento a redes, servidores web, librerías para realizar conexiones con bases de datos, etc.

Permite la creación de corrutinas, tareas y futuros, necesarios en el proyecto para la extracción concurrente de información de la API de GitHub.

3.2.3. aiohttp

Cliente/Servidor HTTP asíncrono para el módulo asyncio de Python. Esta librería permite realizar llamadas HTTP asíncronas a las API de GitHub.

Figura 3.7: Logo Aiohttp.

3.2.4. matplotlib

Matplotlib [29] es una librería que permite la creación de gráficas a partir de datos contenidos en listas de Python o arrays en el módulo matemático NumPy. Entre algunos de los gráficos

que permite generar se encuentran:

- Diagramas de barras
- Histogramas
- Diagramas de sectores
- Diagramas de caja y bigotes.
- ...

Figura 3.8: Logo Matplotlib.

3.2.5. XGBoost

XGboost [7] es una librería de código abierto de una implementación del algoritmo supervizado *gradient boosting*. Algunas de sus principales ventajas son su alta flexibilidad, empleo de procesamiento paralelo, soporta regularización y es más rápido que *gradient boosting*.

Esta librería es utilizada para generar el modelo de *machine learning* con el que extraer el repositorio más relevante.

Figura 3.9: Logo XGBoost.

3.2.6. skitic learn

Skitic learn [31] es una de las librerías más conocidas para la ciencia de datos en Python, cuenta con múltiples algoritmos de clasficación, clustering y regresión, entre otros.

Está librería es empleada en el trabajo para poder aplicar validación cruzada.

3.2.7. pytest

Pytest [12] es un framework para el desarrollo de test en Python. Permite escribir varios tipos de test entre los que se incluyen, test unitarios, test de integración y test funcionales.

3.2.8. argparse

Argparse [9] facilita la creación de interfaces de línea de comandos. El programa define los argumentos que requiere y *argparse* los analizará fuera de *sys.argv*. El módulo genera automáticamente mensajes de ayuda, de uso y errores cuando el usuario introduce un argumento erróneo.

El trabajo se apoya de esta librería para pasar por parámetros el nombre completo del repositorio y el token de autenticación de GitHub.

3.2.9. prompt toolkit

Python Prompt Toolkit [28] es una librería para la creación de aplicaciones por línea de comandos y terminales potentes e interactivas. Algunas de sus funcionalidades incluyen el resaltado de texto, edición de entrada multilínea, finalización avanzada de código, seleccionar texto para copiar o pegar, soporte de cursor o sugerencias automáticas.

Capítulo 4

Metodología empleada

4.1. Scrum

Scrum[24] es un marco de gestión de proyectos ágil que aplica un conjunto de buenas prácticas a la hora de trabajar colaborativamente, de tal modo, que se puedan abordar problemas complejos y a la vez entregar productos del máximo valor posible. Dicha metodología aplica un sistema iterativo e incremental para el desarrollo, entrega y mantenimiento de los productos, dando mayor importancia a la interacción entre personas y priorizando el trabajo que tiene más valor para el cliente. Estas características aportan flexibilidad y rapidez, dos cualidades necesarias en entornos de desarrollo donde los proyectos tienen requisitos inestables. Algunas de las ventajas que aportan dicha metodología son:

- Aporta autonomía y responsabilidad, ya que, implica a todas las partes de un proyecto, de tal modo que se potencia la confianza junto con el crecimiento personal y profesional.
- Permite el desarrollo de productos mínimos viables, capaces de cumplir con las garantías necesarias, sin tener la necesidad de que este completamente acabado.
- Proporciona *feedbacks* rápidos y precisos, que permiten que el equipo conozca la situación del proyecto en todas sus etapas y propongan soluciones rápidamente.
- Aprendizaje continuo durante todo el desarrollo del proyecto. Al disponer de iteraciones de corta duración, se puede adquirir un aprendizaje, el cual, puede ser aplicado en las siguientes iteraciones.
- Como el proyecto está dividido en múltiples entregas, los márgenes de errores son más pequeños y por ende las fechas de entregas finales se ajustan mucho más a lo planificado.
- Dividir el proyecto en iteraciones permite dimensionarlo más fácilmente.

4.1.1. Eventos

Los *eventos* [22] o reuniones dentro del marco Scrum son:

Sprint: es una un ciclo o iteración que se realizará dentro de un proyecto Scrum. Se realizan en periodos cortos de tiempo de entre dos y cuatro semanas, donde el equipo de desarrollo trabaja para sacar un incremento desplegable y entregable del producto.

Planificación del Sprint: es la reunión de trabajo que se efectúa previamente a cada Sprint, donde se determina cuál va a ser el objetivo del Sprint y que tareas son necesarias para conseguirlo.

Scrum diario: durante esta reunión diaria de máximo 15 minutos de duración, el equipo de desarrollo pone en común sus avances y dificultades, además de organizar un plan para la próxima jornada.

Revisión del sprint: durante esta reunión el equipo de desarrollo presenta al cliente el trabajo terminado y en caso de ser necesario puede ser modificada la pila de producto.

Retrospectiva del sprint: Esta reunión tiene como objetivo mejorar la forma en la que el Scrum Team desempeña su trabajo. Los aspectos a analizar son los obstáculos aparecidos durante la iteración y las relaciones interpersonales entre las personas implicadas. En un Sprint de un mes debe durar 3 horas como máximo, uno de tres semanas deberá durar 2,25 horas y así sucesivamente.

Este proyecto está dividido en cuatro Sprints de aproximadamente un mes de duración, al principio de cada Sprint se realizó una reunión en la que se definieron las funcionalidades a implementar en la iteración, la reunión del Scrum diario se realizó dos veces a la semana con el tutor por cuestiones de disponibilidad, al final de cada Sprint se realizaron las reuniones referentes a la revisión y retrospectiva, del Sprint con el objetivo de valorar el estado del TFG y analizar problemas y aspectos mejorables con respecto a la siguiente iteración. En la figura 4.1 se ofrece un diagrama donde se pueden ver varios de estos conceptos y su relación.

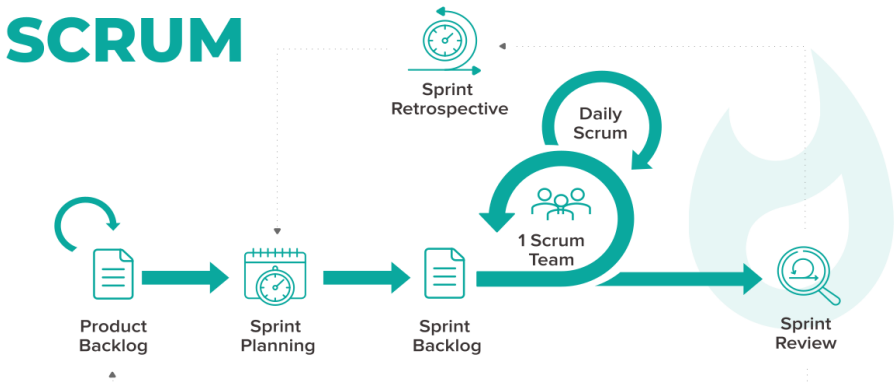

Figura 4.1: Representación de distintos conceptos del marco de trabajo Scrum. Extraido de:[13].

4.1.2. Artefactos

Los *artefactos* [16] del marco Scrum son:

Pila de producto: lista ordenada que contiene todos los requisitos necesario para que el producto cumpla con las necesidades de los clientes. Estos requisitos deben de ser simples, deben de estar estimados y ordenados por priordad del cliente.La pila varía conforme avanza el desarrollo del producto, gracias a esto, refleja aquello que el producto necesita incorporar para adecuarse a las circunstancias, en todo momento.

Pila de Sprint: Esta pila está conformada por un subconjunto de historias de usuario escogidas de la pila de producto para ser abordadas en el periodo de la iteración. Cada historia debe registrar, una descripción breve, el desarrollador asignado y el esfuerzo pendiente para terminarla. Permite descomponer el proyecto en unidades de tamaño adecuado para determinar el avance diario e identificar riesgos sin tener que aplicar procesos de gestión complejos.

Incremento: está constituido por la suma de todos los elementos de la pila de producto completados durante los incrementos realizados. De manera ideal, en el marco Scrum, cada elemento de la pila del producto representa una funcionalidad, no un trabajo interno y se produce un incremento en cada iteración/Sprint.

4.1.3. Roles

Los *roles* [17] en el marco Scrum son los siguientes:

Propietario del producto: es el encargado de tomar las decisiones del cliente. Recae en una única persona, con el objetivo de simplificar la comunicación y la toma de decisiones. *Decide* como será el producto final en última instancia, el orden de los incrementos, el contenido de la pila del producto, la prioridad de cada historia de usuario y *conoce* el plan del producto, sus posibilidades y el plan de inversión.

Desarrollador: Son los profesionales que participan en el Sprint con el objetivo de generar un incremento. El número recomendado de desarrolladores por equipo está entre 3 y 9 miembros y se busca que sea un equipo *multifuncional*, es decir, todos los miembros trabajan con responsabilidad compartida.

Scrum Master: se asegura de que se cumplen las reglas del marco de trabajo Scrum, proporcionando la asesoría y formación necesaria a los desarrolladores y al propietario del producto, revisando y validando la pila del producto, moderando las reuniones, gestionando dinámica de grupo, resolviendo problemas en los Sprints y realizando una mejora continua de las prácticas Scrum en la organización.

En este proyecto no hay unos roles claramente definidos, ya que, la idea del TFG fue definida por el alumno quién también la desarrolla.

4.1.4. Kanban

Kanban [26] es un sistema de información que permite al equipo de desarrollo mantener un equilibrio entre la disponibilidad de cada miembro del equipo y el trabajo a realizar. La posición de cada tarjeta en el tablero refleja el estado en el que se encuentra el trabajo al que hace referencia, los estados mínimos son *pendiente*, *en curso* y *hecho*, luego, se pueden añadir y modificar los estados dependiendo de las necesidades de cada equipo. Todo esto permite gestionar el flujo de trabajo, de tal modo que, se pongan de manifiesto cualquier incidencia, se genere un avance continuo del trabajo, evitando la ley de Parkinson [1]y se favorezca la comunicación directa. Durante la realización de este TFG se ha utilizado el tablero de la plataforma Trello para implementar dicha práctica de gestión visual (ver figura 4.2)

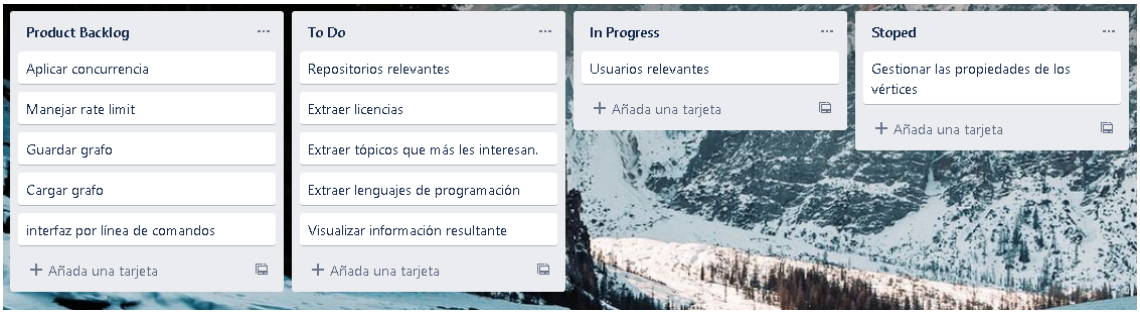

Figura 4.2: Tablero Trello.

4.1.5. Historias de usuario

Las historias de usuario [18] son una explicaciones generales de las funcionalidades del producto, escritas desde la perspectiva del usuario final. Son escritas por el *product owner* y el lenguaje usado al describir cada historia debe explicar el objetivo de manera sencilla, sin entrar en detalles. Los beneficios de trabajar con historias son:

- Centran la atención del usuario, ya que, mantiene al equipo centrado en solucionar problemas de usuarios reales.
- Permiten la colaboración, definiendo adecuadamente los objetivos.
- Impulsan soluciones creativas, fomentando el pensamiento crítico y creativo.
- Aumentan la motivación del equipo.

Generalmente, las historias de usuario siguen la siguiente estructura:

yo como [perfil], deseo [intención], para [objetivo]

[1]La ley de Parkinson afirma que el trabajo se expande hasta llenar el tiempo disponible para que se termine.

Esta es la plantilla comunmente empleada, sin embargo, las funcionalidades de algunos proyectos están ocultas al usuario final y la estructura mostrada anteriormente no puede ser aplicada de manera efectiva. Ejemplos de proyectos que se pueden ver afectados por esto, pueden ser los centrados en ciencia de datos, *machine learning* o que trabajan únicamente con el lado servidor de una aplicación.

Ante dicha situación y la solución que se consideró más adecuada para este caso es usar el patrón de las **tres Ws** [14]. La estructura en inglés sería *what*, *why* y *who will benefit*, en español sería de la siguiente forma.

¿Qué?: [funcionalidad]. ¿Por qué?: [razón] y ¿Quién se beneficia?: [perfil]

Por último, comentar que las historias deben de contener criterios de validación detallados que dejen claro, cuál debería de ser el resultado y cómo evaluarlo.

Capítulo 5

Desarrollo

5.1. Sprint Zero

5.1.1. Estudio de las herramientas y familizarización con el problema

Extracción de información de la API REST de GitHub

Token OAuth [1]

Todos los datos necesarios para la creación del grafo se extraen de la API REST de GitHub la cual permite interactuar con la plataforma, mediante el uso de métodos HTTP. Algunas de las acciones permitidas pueden ser crear repositorios o extraer información acerca de un usuario u organización, entre otras cosas. Si el usuario no está autenticado, la plataforma solo permite realizar 60 llamadas al servidor por hora, por ello será necesario autenticarse para poder realizar más llamadas. La manera más sencilla y segura es usando un *token OAuth*[2], el cual, se puede crear desde la cuenta personal del usuario (ver figura 5.1). Una vez autenticado, el número máximo de llamadas por hora pasaría a ser 5000.

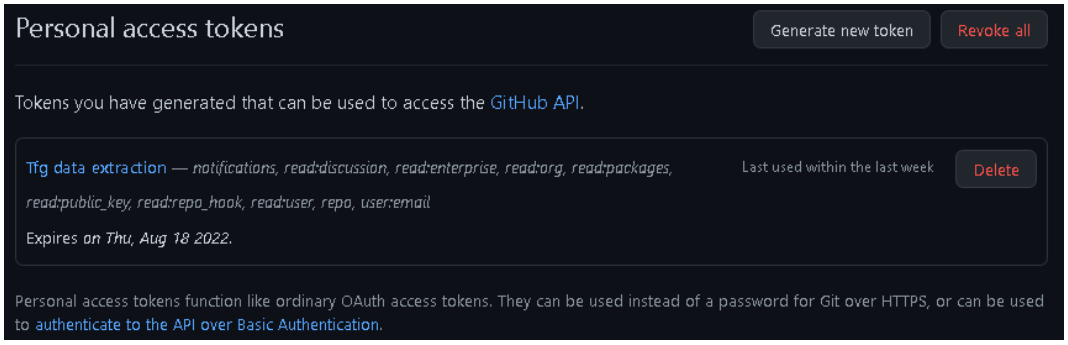

Figura 5.1: Creación token OAuth.

[1] https://docs.github.com/en/rest/guides/getting-started-with-the-rest-api#authentication
[2] https://en.wikipedia.org/wiki/OAuth

Rate Limits [3]

Uno de los aspectos más relevantes a tener en cuenta a la hora de trabajar con la API, es el *rate limit*, que de alcanzarse significa que se ha efectuado el número máximo de llamadas. El estado del *rate limit* se puede comprobar en las cabeceras HTTP de las respuestas:

- *x-ratelimit-limit*: número máximo de llamadas que se puede realizar por hora.
- *x-ratelimit-remaining*: número de llamadas restantes que se puede efectuar actualmente.
- *x-ratelimit-reset*: tiempo en formato UTC hasta que el *rate limit* actual se resetee.

También se puede dar el caso en el que se alcance el *secondary rate limit*, el cual, se da en situaciones en las que el usuario crea contenido rápidamente, llama concurrentemente a la API o pide mucha información de alto coste computacional. El mayor problema con esta limitación es que no existe ningún tipo de indicador que nos comunique cuando se va a alcanzar. Una vez alcanzado, se devuelve junto con la respuesta la cabecera HTTP *Retry-After*, que indica el tiempo que debe transcurrir hasta que se puedan realizar más llamadas a la API.

Paginación [4]

Una de las herramientas usadas por la API para evitar saturarse con llamadas que piden mucha información es la *paginación*. Por ejemplo, en el caso de llamar a una lista pública de repositorios, la API paginará por defecto la respuesta en listas de 30 ítems, por ende, cada vez que se requiera extraer otros treinta elementos será necesario realizar una llamada. Este valor puede ser modificado hasta un número de 100 ítems por página y la información acerca de la página extraída y el número total de páginas a se encuentra en la cabezera HTTP *Link*. Es necesario entender este aspecto de la API para poder consumir toda la información que se requiere de manera eficiente y sin equivocaciones.

```
Link: <https://api.github.com/search/code?q=addClass+user%3Amozilla&page=2>; rel="next",
  <https://api.github.com/search/code?q=addClass+user%3Amozilla&page=44>; rel="last"
```

En el ejemplo anterior se puede comprobar en $rel =$ "*next*" que la siguiente página es la número 2 lo cual tiene sentido, ya que la primera llamada será a la primera página, luego en $rel =$ "*last*", se puede observar que el número total de páginas es 44.

Librerías para interactuar con la API

Para la extracción de datos desde la API de GitHub, en un principio se utilizó PyGithub [5], una librería de Python que permite manejar los recursos de la plataforma mediante el uso de scripts del lenguaje. La principal razón por la que se escogió está herramienta, es su fácil uso, ya que, no es necesario indicar la ruta al *end point* de la API, ni manejar su paginación, entre otras razones. Sin embargo, conforme se avanzó en el proyecto se encontraron una serie de problemas que obligaron a buscar otra herramienta:

- La manera con la que maneja la paginación desemboca en la realización de llamadas extras a la API, lo que ralentiza la aplicación considerablemente [6]

[3] https://docs.github.com/en/rest/overview/resources-in-the-rest-api#rate-limiting
[4] https://docs.github.com/en/rest/guides/traversing-with-pagination

- La libería sufre problemas de mantenimiento por la falta de desarrolladores trabajando en ella [7]

La librería sustituta fue Request [8], el módulo más empleado de Python para la realización de llamadas HTTP/1.1. Su principal desventaja respecto a PyGithub, es que es más compleja de usar, sin embargo, no sufre de problemas de mantenimiento y permite controlar en detalle las llamadas.

No obstante, uno de los objetivos de este proyecto es obtener el mayor rendimiento disminuyendo al mínimo posible el tiempo invertido en la extracción de los datos y como solución, se encontró la **concurrencia**. Había dos vías para aplicarla, mediante multihilos o programación asíncrona, tras realizar un estudio comparativo, se decidió implementar la segunda opción debido a las limitaciones [1] de los multihilos en Python y es que las interacciones del GIL en CPython limita la ejecución a un solo hilo a la vez. Las librerías escogidas fueron Asyncio que permite la programación concurrente utilizando la sintaxis async/await y la librería aiohttp un cliente HTTP asíncrono.

Creación del grafo

Con los datos extraídos se pretende construir un grafo directo $G = (V, E)$ donde los vértices están conformados por los *stargazers* y los repositorios que les interesan. En el caso de que un usuario siga a otro usuario, se crea una relación directa entre ellos, del mismo modo, si a un usuario le interesa un repositorio, se crea una relación entre ellos 5.11. Las propiedades de los vértices de los repositorios serán su nombre, el número de *stargazers*, el lenguaje de programación, el número de *forks*, la fecha de creación y los topicos y en el caso de los *stargazers* únicamente el nombre, esto es así debido a que para extraer más información sobre cada usuario sería necesario realizar como mínimo una llamada extra por cada uno de ellos y por ende, se consideró adecuado acotar la información extraída.

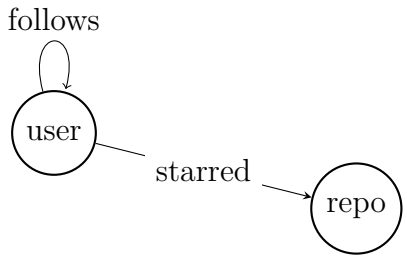

Figura 5.2: Esquema del grafo.

[7]https://pygithub.readthedocs.io/en/latest/introduction.html
[7]https://github.com/PyGithub/PyGithub/issues/1701
[7]https://github.com/PyGithub/PyGithub/issues/2178
[8]https://docs.python-requests.org/en/latest/

La librería empleada es *Graph-tool*, un módulo de Python para la manipulación y el análisis estadístico de grafos. La razón porque la que se escogió frente a otras librerías como networkx o igraph, es su rendimiento, ya que, el núcleo de las estructuras y los algoritmos están implementados en C++. En la documentación de dicho módulo se puede encontrar un ejemplo que muestra la diferencia en el tiempo de ejecución entre los tres módulos, aplicando diferentes algoritmos a un grafo directo de 39.796 vértices y 301.498 aristas 5.3.

Algorithm	graph-tool (16 threads)	graph-tool (1 thread)	igraph	NetworkX
Single-source shortest path	0.0023 s	0.0022 s	0.0092 s	0.25 s
Global clustering	0.011 s	0.025 s	0.027 s	7.94 s
PageRank	0.0052 s	0.022 s	0.072 s	1.54 s
K-core	0.0033 s	0.0036 s	0.0098 s	0.72 s
Minimum spanning tree	0.0073 s	0.0072 s	0.026 s	0.64 s
Betweenness	102 s (~1.7 mins)	331 s (~5.5 mins)	198 s (vertex) + 439 s (edge) (~ 10.6 mins)	10297 s (vertex) 13913 s (edge) (~6.7 hours)

Figura 5.3: Tabla comparativa del rendimiento de las librerías [9].

Otras de las razones por las que se escogió este módulo es porque implementa varios algoritmos, entre ellos, *Personalized Page Rank*, e incluye una serie de funcionalidades muy útiles como poder guardar el grafo en distintos formatos, cargarlo o visualización de grafos, entre otras cosas.

5.1.2. Configuración del entorno

Graph-tool está contenida en una imagen de docker Arch GNU/Linux por lo que es necesario contenerizar la aplicación para poder utilizar dicho módulo. El primer paso consiste en instalar la versión de escritorio de docker, que permite construir y compartir aplicaciones contenerizadas y microservicios. Una vez instalado y configurado en el sistema, dentro del proyecto se debe crear un fichero *dockerfile* [10], el cual, es un documento de texto que contiene ordenes que un usuario puede ejecutar desde la interfaz de línea de comandos para construir una imagen, luego con el comando *docker build* se automatiza el proceso de construcción ejecutando varias órdenes al mismo tiempo 5.4.

```
FROM tiagopeixoto/graph-tool

WORKDIR /app
COPY . .

RUN pacman -S --noconfirm python-pip
RUN pip3 --no-cache-dir install -r requirements.txt

CMD ["python3", "src/main.py"]
```

Figura 5.4: Comandos dentro del dockerfile para la creación del contenedor.

[9] https://graph-tool.skewed.de/performance

- **FROM**: Especifica la imagen padre a partir de la que se construirá la imágen del proyecto
- **WORKDIR**: Indica cual va a ser el directorio de trabajo.
- **COPY**: Copia el fichero indicado en el interior imagen.
- **RUN**: Crea y arranca el contendor para ejecutar los parámetros establecidos.
- **CMD**: Establece valores predeterminados para un contenedor en ejecución.

Para evitar tener que construir nuevamente la imagen cada vez que se realiza un cambio en el código, se instaló la extensión de Visual Studio Code *Remote - Container* [11], la cual, permite usar los contenedores de docker como un entorno de desarrollo. La extensión genera un fichero *devcontainer.json* que indica a VS Code como crear o acceder al contenedor de desarrollo.

5.1.3. Planificación del Sprint 1

El primer paso consiste en calcular el factor de foco, dicho valor representa el porcentaje de tiempo en el que las personas que forman parte de un equipo de desarrollo son productivas y están totalmente comprometidas con la producción del software. Para este primer Sprint el factor de foco será de 0.85 y se irá modificando en cada iteración en caso de que no se estimen correctamente las historias.

Esta iteración durará cuatro semanas, trabajando 12 horas semanales, lo que da como resultado un **tiempo ideal**[12]de 48 horas de trabajo en un mes. Suponiendo que cuatro horas de trabajo equivalen a un *punto de historia* entonces se obtiene una velocidad de $48/4 = 12$ puntos a partir del tiempo ideal, luego, aplicando el factor de foco, se obtendría el número de puntos de historias a repartir en el Sprint $12 * 0{,}85 = 10{,}2$.

La pila del Sprint 1 quedaría de la siguiente manera.

ID	Nombre	Horas	Puntos	Prioridad
HU01	Extraer datos de la API	12	3	1
HU02	Generar grafo *stargazers*	12	3	1
HU03	Añadir relaciones usuarios	6	1.5	1
HU04	Añadir repositorios que les interesan	6	1.5	1
HU05	Gestionar propiedades de los vértices	4.8	1.2	1

5.2. Sprint 1. Extracción de datos y creación del grafo

Durante la realización de este Sprint la lógica de extracción de datos de la API y creación del grafo, están unidas en una misma clase. La librería usada en un principio para la extracción era

[10]https://docs.docker.com/engine/reference/builder/
[11]https://docs.docker.com/engine/reference/builder/
[12]https://www.scrummanager.net/bok/index.php?title=Velocidad,_trabajo_y_tiempo

PyGithub, sin embargo, debido a los problemas comentados con anterioridad, fue sustituida por el módulo de Request. Un ejemplo del funcionamiento de dicha librería puede ser el siguiente.

```
from github import Github

g = Github("Oauth_token", per_page=100)
repository = g.get_repo(full_name_repository)
```

El primer paso fue crear la clase *interest graph*, cuyo constructor requiere dos parámetros, el nombre completo del repositorio (autor/repositorio) y el token de autenticación del usuario 5.5.

```
def __init__(self, full_name_repository: str, token: str) -> None:
    self.g = Graph(directed=True)
    self.session = requests.Session()
    self.session.headers['Authorization'] = 'token %s' % token
    self.full_name_repository = full_name_repository
    self.set_graph_properties()
```

Figura 5.5: Constructor de la clase interestGraph, donde se crea el grafo directo, el objeto *session* asignándole una cabecera y se crean las propiedades de los distintos elementos del grafo.

Al generar un objeto de dicha clase se genera un grafo directo, un objeto de la librería Request denominado *session* el cual permite persistir la información en cada llamada a la API, en este caso, el header HTTP *Authorization* con el token de autenticación de su cuenta y por último, se inicializan las propiedades de los vértices y aristas del grafo 5.6.

```
def set_graph_properties(self) -> None:
    self.__v_name: VertexPropertyMap = self.g.new_vertex_property("string")
    self.__v_is_user: VertexPropertyMap = self.g.new_vertex_property("bool")
    self.__v_is_repo: VertexPropertyMap = self.g.new_vertex_property("bool")
    self.__v_repo_st: VertexPropertyMap = self.g.new_vertex_property("int32_t")
    self.__v_repo_forks: VertexPropertyMap = self.g.new_vertex_property("int32_t")
    self.__v_repo_date: VertexPropertyMap = self.g.new_vertex_property("string")
    self.__v_repo_lang: VertexPropertyMap = self.g.new_vertex_property("string")
    self.__e_relation: EdgePropertyMap = self.g.new_edge_property("string")
    self.__v_repo_topics: VertexPropertyMap = self.g.new_vertex_property("vector<string>")
    self.__v_repo_license: VertexPropertyMap = self.g.new_vertex_property("string")
    self.__v_no_main: VertexPropertyMap = self.g.new_vertex_property("bool")
```

Figura 5.6: Mapas de propiedades de los vértices y aristas.

Una vez creado el objeto, al llamar al método *create graph* se incluyen en el grafo los vértices y las aristas representando los usuarios, repositorios y las relaciones entre ambos. A continuación, se realizará una explicación del algoritmo implementado para la creación del grafo.

El primer paso consiste en generar el vértice que represente al repositorio que queremos analizar, luego se extraen los *stargazers* en una lista y se recorren. En caso de que el vértice del usuario no exista en el grafo, se genera dicho vértice y se crea una relación con el repositorio principal, en el supuesto de que ya exista, se crea únicamente la relación con el repositorio 5.7.

```python
def create_graph(self) -> None:
    main_vertex: Vertex = self.create_main_vertex()

    stargazers: list = self.request_api(None, self.__MAX_NUMBER_ITEMS, "stargazers", True)
    for stargazer in stargazers:
        stargazer_vertex: list = find_vertex(self.g, self.__v_name, stargazer['login'])
        if stargazer_vertex:
            new_stargazer_vertex: Vertex = stargazer_vertex[0]
            self.create_edge(relationship.STARRED, new_stargazer_vertex, main_vertex)
        else:
            new_stargazer_vertex: Vertex = self.create_stargazer_vertex(stargazer)
            self.create_edge(relationship.STARRED, new_stargazer_vertex, main_vertex)

        self.add_follower_relationship(stargazer, new_stargazer_vertex, stargazers)
        self.add_starred_repos(stargazer, new_stargazer_vertex, main_vertex)
```

Figura 5.7: Creación de los vértices referentes a los *stargazers*, empleando métodos auxiliares para generar las relaciones entre usuarios y los repositorios.

Posteriormente, se añaden las relaciones entre usuarios, para ello, se extraen todos los seguidores de un usuario en una lista y se recorre. En caso de que alguno de los seguidores este en la lista de los *stargazers* y su vértice exista en el grafo, entonces, se crea una relación entre dicho seguidor y el usuario, si el seguidor existe en la lista de *stargazers*, pero su vértice no ha sido creado todavía, se crea el vértice de dicho seguidor y la relación con el usuario y en el supuesto de que el seguidor no exista en la lista de *stargazers* no se crea la relación entre ambos y se pasa a la siguiente iteración del bucle 5.8.

```python
def add_follower_relationship(self, stargazer: json, new_vertex: Vertex, stargazers: list) -> None:
    followers: list = self.request_api(stargazer, self.__MAX_NUMBER_ITEMS, "followers", False)

    for follower in followers:
        try:
            stargazers.index(follower)
            follower_vertex: list = find_vertex(self.g, self.__v_name, follower['login'])
            if follower_vertex:
                self.create_edge(relationship.FOLLOWS, follower_vertex[0], new_vertex)
            else:
                follower_vertex = self.create_stargazer_vertex(follower)
                self.create_edge(relationship.FOLLOWS, follower_vertex, new_vertex)
        except:
                pass
```

Figura 5.8: Método para crear las relaciones entre los *stargazers*.

Finalmente, se añaden repositorios que siguen los usuarios, para ello, se extraen los últimos 20 repositorios a los que les haya dado me gusta el stargazer en una lista y se recorre. Si el vértice de dicho repositorio ya existe, se comprueba que no sea el repositorio principal y en caso de no serlo, se genera una relación entre el usuario y el repositorio, en el supuesto de que el vértice del repositorio no exista todavía, se crea el vértice y se genera la relación con el usuario 5.9.

```python
def add_starred_repos(self, stargazer: json, new_vertex: Vertex, main_vertex: Vertex) -> None:
    starred_repos: list = self.request_api(stargazer, self.__MAX_REPOS_STARGAZER, "starred", False)

    for starred in starred_repos:
        repeated_repos: list = find_vertex(self.g, self.__v_name, starred['name'])
        if repeated_repos:
            if repeated_repos[0] != main_vertex:
                self.create_edge(relationship.STARRED, new_vertex, repeated_repos[0])
        else:
            starred_repo: Vertex = self.create_repository_vertex(starred)
            self.create_edge(relationship.STARRED, new_vertex, starred_repo)
```

Figura 5.9: Método para incluir en el grafo los últimos 20 repositorios que siguen cada uno de los usuarios y las relaciones entre estos.

Se dispone de un método encargado de realizar las llamadas a la API, donde primero se construye la dirección *url* del *endpoint* al que se va a realizar la petición, esta dependerá de si se quiere extraer los *stargazers* del repositorio a analizar, los últimos repositorios a los que ha dado me gusta un usuario o sus seguidores e indicando el número de ítems que se desean obtener por página, 20 en el caso de los repositorios y 100 en el caso de los seguidores ya que se quieren extraer todos. Posteriormente, se efectúa la llamada, en el caso de obtener los repositorios, se devuelve el resultado, en cambio si se obtienen los stargazers o los seguidores, se comprueba si hay más páginas con datos, si es así, recorre un bucle con el número total de páginas, consumiéndolas y devolviendo el resultado 5.10.

```python
def request_api(self, stargazer: json, num_items: int, info: str, is_repo_url: bool) -> list:
    if is_repo_url:
        url: str = self.__API_URL+"repos/%s/%s?per_page=%d" % (self.full_name_repository, info, num_items)
    else:
        url: str = self.__API_URL+"users/%s/%s?per_page=%d" % (stargazer['login'], info, num_items)

    api_response = self.session.get(url , headers=self.session.headers)
    response_json: list = api_response.json()
    if info != "starred":
        pages_list: list = []
        num_pages: int =  self.get_number_pages(api_response)
        for page in range(2, num_pages+1):
            pages_list.append(self.consume_pages(url, page))

        consumed_pages = [item for page in pages_list for item in page]
        response_json.extend(consumed_pages)

    return response_json
```

Figura 5.10: Método para realizar las llamadas a la API, seleccionando una url dependiendo del *end point* al que se quiera realizar la petición y comprobando si hay que consumir varias páginas de datos.

El siguiente paso consistió en visualizar el grafo, de tal modo que se pudiera comprobar que el número de vértices y aristas generados es correcto y disponer de una representación visual del mismo. En el siguiente ejemplo el grafo corresponde al repositorio *marius92mc/github-stargazers*, en cuyo caso cuenta con 12 *stargazers* formando un grafo de 223 vértices y 228 aristas 5.11.

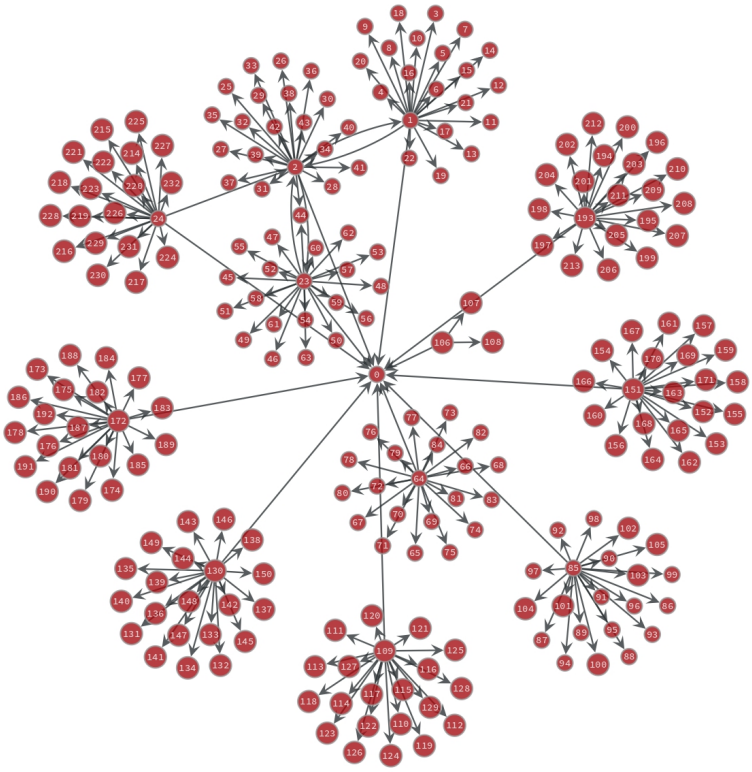

Figura 5.11: Ejemplo visualización grafo 12 stargazers, donde se puede apreciar las relaciones entre los usuarios y los repositorios que siguen.

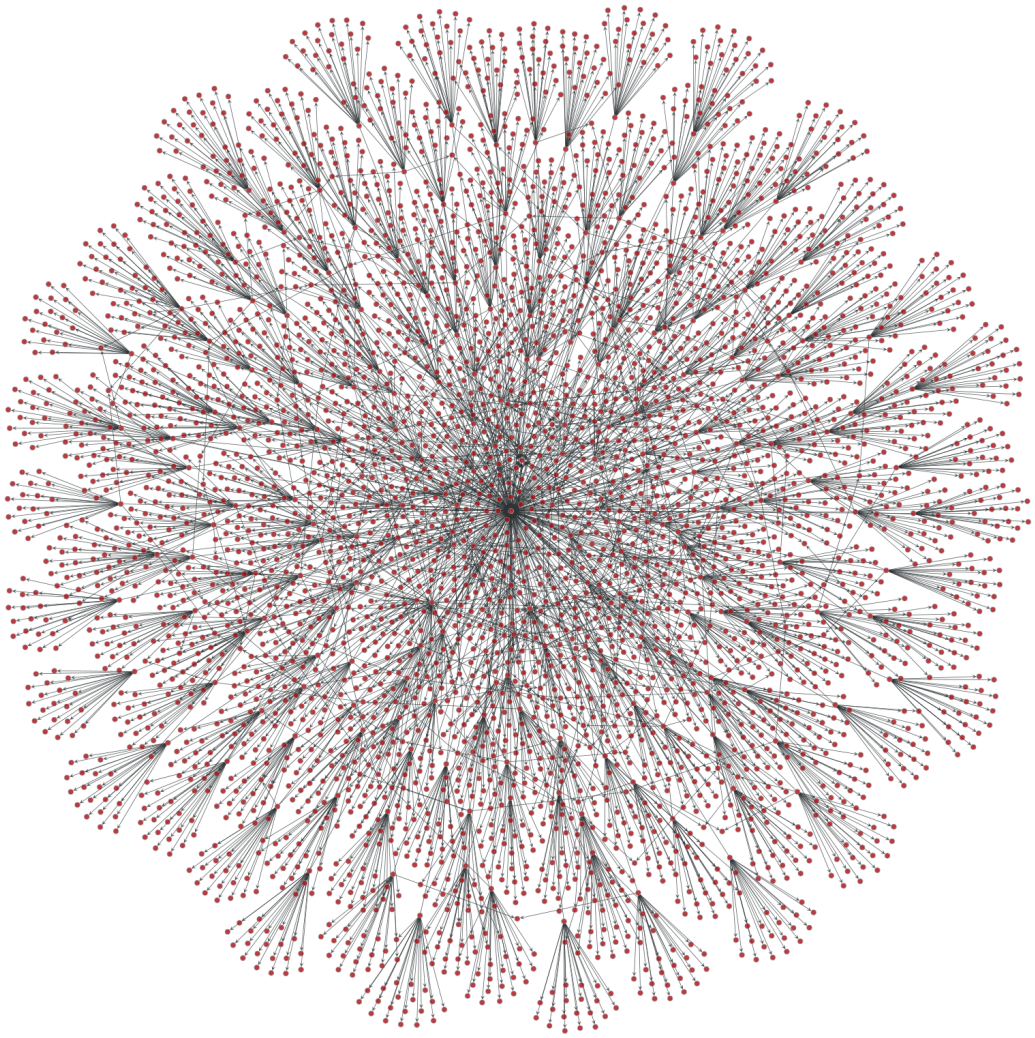

Figura 5.12: Ejemplo visualización grafo más de 200 stargazers.

5.2.1. Planificación del Sprint 2

Durante la realización del primer Sprint se tuvieron que realizar cambios en gran parte del código al sustituir la librería *PyGithub* por *Request*, esto provocó que la historia HU05 quedase inconclusa, lo que significa que se sobre estimo la capacidad para terminar las historias del primer Sprint. Se aplicará la siguiente fórmula para averiguar el reajuste a realizar al factor de foco.

$$\frac{\text{velocidad real}}{\text{velocidad ideal}} \tag{5.1}$$

Teniendo en cuenta que la velocidad real fue de 9, entonces $9/10{,}2 = 0{,}88$, luego, sumándolo al factor de foco actual da, 93.8 % . Teniendo en cuenta que este Sprint también es de un mes y el mismo número de horas semanales, la velocidad para el siguiente Sprint será de $12 * 0{,}93 = 11{,}16$

ID	Nombre	Horas	Puntos	Prioridad
HU05	Gestionar propiedades de los vértices	4.8	1.2	1
HU06	Usuarios relevantes	8	2	2
HU07	Repositorios relevantes	16	4	2
HU11	Visualizar información resultante	8	2	2
HU08	Extraer lenguajes de programación	4.5	1	2
HU09	Extraer tópicos	4.5	1	2
HU10	Extraer licencias	4.5	1	2

5.3. Sprint 2. Algoritmos y visualización de datos

Esta iteración se centra en la extracción de información relevante del grafo, para ello, se aplicaron algoritmos como *PageRank* o su versión personalizada, de tal modo que se pueda realizar una clasificación acerca de los usuarios y repositorios más relevantes, entre otros datos. Los ejemplos mostrados en este apartado fueron realizados con el repositorio *marius92mc/github-stargazers*.

Para poder obtener un ranking de los usuarios más importantes de la comunidad que se está analizando, se emplea el algoritmo de *PageRank* de tal modo que la relevancia de un usuario dependerá del número de usuarios que le siguen y la importancia de dichos usuarios. Primero se genera un sub grafo que contendrá únicamente los vértices que representen a los *stargazers*, seguidamente, se aplica el algoritmo a dicho sub grafo y se muestra el resultado por pantalla 5.13.

```python
def get_relevant_users(self) -> list:
    v_name: VertexPropertyMap = self.graph.get_name()
    v_is_user: VertexPropertyMap = self.graph.get_is_user()
    sub_graph = GraphView(self.graph.g, v_is_user)

    pr = pagerank(sub_graph)
    users: dict = {}
    for item in sub_graph.vertices():
        users[v_name[item]] = pr[item]

    ordered_users = dict(sorted(users.items(), key=lambda item: item[1], reverse = True))
    print("*** Most relevant users ***")
    self.print_map(ordered_users)
    return list(ordered_users.items())[:10]
```

Figura 5.13: Código Python relativo a la clasificación de usuarios acorde a su relevancia.

```
*** Most relevant users ***
marius92mc : 0.36902267037424935
ignacio-chiazzo : 0.185680972505183
adelinaenache : 0.185680972505183
abetancordelrosario : 0.028846153846153848
alecbw : 0.028846153846153848
EgorBu : 0.028846153846153848
ponnuchamy51 : 0.028846153846153848
robotsandcake : 0.028846153846153848
paper2code-bot : 0.028846153846153848
madsfjeder : 0.028846153846153848
```

Figura 5.14: Resultado de ejemplo usuarios.

Con respecto a la clasificación de los repositorios por relevancia, al disponer de más información sobre ellos, se puede realizar un ranking personalizado. Para ello se empleará *Personalized PageRank* donde se ha de incluir el vector de personalización indicando que vértices son relevantes, de este modo, la relevancia de un repositorio no solo depende del número de *stargazers* que dispongan en la comunidad o de la popularidad de estos, sino que también se pueden tener otros factores en cuenta a la hora de clasificarlos. Se suma 1 entre el número de vértices al valor de cada nodo en dicho vector , en caso de que cumpla las siguientes condiciones y 0 en caso contrario. 5.15:

- Debe tener más de 1000 stargazers
- Debe tener más de 100 forks
- Debe de haberse creado durante el último año

```
# Repositories with more than 1000 stargazers
starg = v_repo_st.a >= 1000
personalized_vector.a[starg] += 1/num_vertices

# Repositories with more than 100 forks
forks = v_repo_forks.a >= 100
personalized_vector.a[forks] += 1/num_vertices

# Repositories that have been created during the last year.
yearago = date.today() - timedelta(365)
for item in sub_graph.iter_vertices():
    year = v_repo_date[item][:4]
    month = v_repo_date[item][5:7]
    day = v_repo_date[item][8:10]
    if year:
        dateobj = date(int(year), int(month), int(day))
        res = yearago - dateobj
        if res.days < 0: personalized_vector[item] += 1/num_vertices

# Personalized pagerank
pr = pagerank(sub_graph, pers=personalized_vector)
```

Figura 5.15: Función que clasifica repositorios acorde a la relevancia de los mismos. Se puede apreciar el código empleado para establecer los valores del vector de personalización de acuerdo al número de condiciones que cumplan.

```
*** Most relevant repos ***
angago : 0.003573812642669135
react-redux : 0.003573812642669135
software-design-and-architecture-roadmap : 0.0027779401452619344
PythonDataScienceHandbook : 0.0027253853532625844
pyscript : 0.0026719180908070456
python-prompt-toolkit : 0.0026719180908070456
aiogithub : 0.0026719180908070456
Requester : 0.0026719180908070456
QuickOSM : 0.0026719180908070456
XGboost_Index-Enhancement-Strategy : 0.0026719180908070456
----------------------------------------
```

Figura 5.16: Resultado ejemplo repositorios.

La extracción de los tópicos 5.17 que más les interesa, los lenguajes de programación 5.18 y licencias 5.19 más usadas por los repositorios se obtuvo de una manera parecida, primero se crea un sub grafo donde los vértices son únicamente repositorios, luego se recorre dicho grafo usando una función de iteración rápida de la librería *graph-tool*, en cada iteración se actualiza los valores de un diccionario que guarda las ocurrencias de un tópico/lenguaje/licencia y finalmente se ordena el resultado y se muestra por pantalla.

```python
def get_topics(self) -> dict:
    v_repo_topics: VertexPropertyMap = self.graph.get_repo_topics()
    v_is_repo: VertexPropertyMap = self.graph.get_is_repo()
    sub_graph = GraphView(self.graph.g, v_is_repo)

    topics: dict = {}
    for repo  in sub_graph.iter_vertices():
        for topic in v_repo_topics[repo]:
            if topic in topics:
                topics[topic] += 1
            elif topic != 'None':
                topics[topic] = 1

    ordered_topics = dict(sorted(topics.items(), key=lambda item: item[1], reverse = True))
    print("*** Topics ***")
    self.print_map(ordered_topics)
    return list(ordered_topics.items())[:10]
```

Figura 5.17: Extracción de los tópicos que más les interesan a los usuarios.

```python
def get_languages(self) -> dict:
    v_repo_lang: VertexPropertyMap = self.graph.get_repo_lang()
    v_is_repo: VertexPropertyMap = self.graph.get_is_repo()
    sub_graph = GraphView(self.graph.g, v_is_repo)

    languages: dict = {}
    for repo in sub_graph.iter_vertices():
        if v_repo_lang[repo] in languages:
            languages[v_repo_lang[repo]] += 1
        elif v_repo_lang[repo] != 'None':
            languages[v_repo_lang[repo]] = 1

    ordererd_languages = dict(sorted(languages.items(), key=lambda item: item[1], reverse = True))
    print("*** Languages ***")
    self.print_map(ordererd_languages)
    return list(ordererd_languages.items())[:10]
```

Figura 5.18: Lenguajes de programación más usados en los repositorios que les interesan.

```python
def get_licenses(self) -> dict:
    v_repo_license = self.graph.get_repo_license()
    v_is_repo: VertexPropertyMap = self.graph.get_is_repo()
    sub_graph = GraphView(self.graph.g, v_is_repo)

    licenses: dict = {}
    for repo in sub_graph.iter_vertices():
        if v_repo_license[repo] in licenses:
            licenses[v_repo_license[repo]] += 1
        elif v_repo_license[repo] and  v_repo_license[repo] != "Other":
            licenses[v_repo_license[repo]] = 1

    ordered_licenses = dict(sorted(licenses.items(), key=lambda item: item[1], reverse = True))
    print("*** Licenses ***")
    self.print_map(ordered_licenses)
    return list(ordered_licenses.items())[:10]
```

Figura 5.19: Licencias que más les interesan.

```
*** Licenses ***
MIT License : 92
Apache License 2.0 : 25
BSD 3-Clause "New" or "Revised" License : 11
GNU General Public License v3.0 : 7
Mozilla Public License 2.0 : 3
GNU Affero General Public License v3.0 : 2
Eclipse Public License 1.0 : 1
Creative Commons Attribution 4.0 International : 1
BSD 2-Clause "Simplified" License : 1
The Unlicense : 1
----------------------------------------
```

Figura 5.20: Ejemplo licencias que más les interesa.

La siguiente funcionalidad implementada en el Sprint fue la creación de gráficas a partir de los resultados, de tal modo que sirvan de apoyo para su compresión. Al extraer algún dato del grafo, se genera una serie de gráficas representando la información.

En la gráfica de barras horizontal, en el caso de que se extrajeran los lenguajes de programación que más les interesa a los usuarios, el eje y estaría conformado por los lenguajes y el eje x representaría el número de repositorios que utilizan dicho lenguaje 5.21.

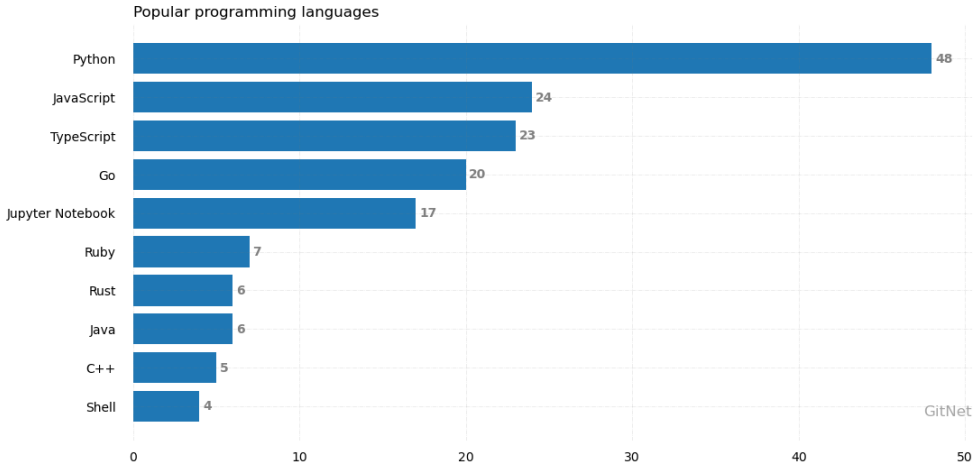

Figura 5.21: Gráfica de barras horizontal donde se muestra un ranking de lenguajes de programación por número de repositorios que lo usan.

En el caso de la gráfica circular, los distintos sectores representarían los lenguajes e irían acompañados del porcentaje de repositorios que usan dicho lenguaje 5.22.

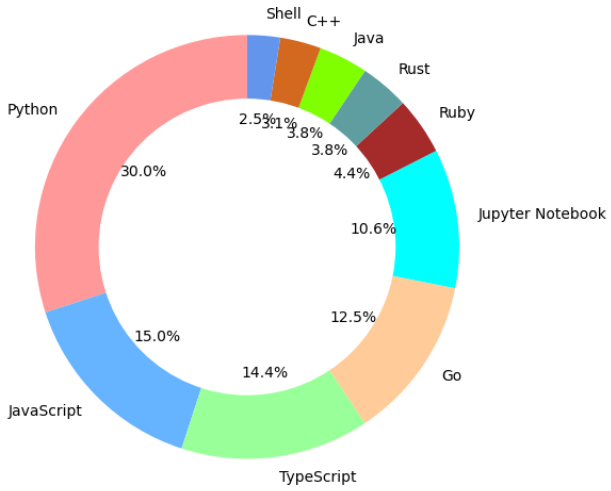

Figura 5.22: Gráfica circular donde se muestran los lenguajes de programación más usados junto con el porcentaje que representan dentro de la comunidad.

Posteriormente, se incluyó una gráfica de regresión que representa la relación entre los *forks* y el número de *stargazers* en los repositorios más importantes de la comunidad.

5.3.1. Planificación del Sprint 3

El segundo Sprint terminó dentro de los plazos establecidos, consumiendo todas las historias de usuarios, por ende, no será necesario realizar ajustes respecto al factor de foco, sin embargo, el número de horas invertidas en el trabajo se verán reducidas debido a prácticas y exámenes finales de otras asignaturas. El número de horas de trabajo semanal pasan a ser 11, por lo que, teniendo en cuenta que este Sprint también dura un mes, la velocidad será de $44/4 = 11$, luego, aplicando el factor de foco, resultaría en $11 * 0{,}93 = 10{,}23$.

ID	Nombre	Horas	Puntos	Prioridad
HU12	Aplicar concurrencia	12	3	4
HU13	Manejar rate limits	8	2	4
HU14	Guardar grafo	4.68	1.11	5
HU15	Cargar grafo	4.68	1.11	5
HU16	Interfaz por línea de comandos	12	3	4

5.4. Sprint 3. CLI y programación asíncrona

Esta tercera iteración se centró en la mejora de la extracción de información de la API de GitHub, con el objetivo de mejorar el rendimiento de la aplicación y en el desarrollo de una interfaz de línea de comandos que facilitara su uso.

Respecto a la mejora del rendimiento, se decidió utilizar técnicas de programación asíncrona a la hora de realizar las llamadas a la API, para ello fue necesario separar la lógica de creación del grafo y manejo de las llamadas en dos clases separadas, *interestGraph* y *dataExtraction*. En la primera, los datos no se extraerían de la API conforme se fuera necesitando, sino que dispondría de una serie de variables globales que serían inicializadas apoyándose de diferentes métodos de la clase *dataExtraction*, de este modo primero se extraía la información y luego se crearía el grafo 5.23.

```
__main_repository: json
__stargazers: list = []
__stargazers_starred_repos: list = []
__stargazers_followers: list = []
```

Figura 5.23: Variables globales donde se guardará la información extraída de la API para generar el grafo.

```
def extract_data_from_api(self):
    self.__stargazers_starred_repos, self.__stargazers_followers = self.extract_data.fetch_data()
    self.__main_repository = self.extract_data.get_main_repo()
    self.__stargazers = self.extract_data.get_stargazers()
```

Figura 5.24: Inicialización variables globales con los datos extraídos de la API.

El constructor 5.24 de la clase solo requiere un objeto de tipo *dataExtraction*, el método encargado de realizar las llamadas a la API sería eliminado de la clase y los métodos encargados de la creación de los vértices y las aristas, quedarían de la siguiente manera.

```
def create_graph(self) -> None:
    main_vertex: Vertex = self.create_main_vertex()

    for index, stargazer in enumerate(self.__stargazers):
        stargazer_vertex: list = find_vertex(self.g, self.__v_name, stargazer['login'])
        if stargazer_vertex:
            new_stargazer_vertex: Vertex = stargazer_vertex[0]
            self.create_edge(relationship.STARRED.value, new_stargazer_vertex, main_vertex)
        else:
            new_stargazer_vertex: Vertex = self.create_stargazer_vertex(stargazer)
            self.create_edge(relationship.STARRED.value, new_stargazer_vertex, main_vertex)

        self.add_follower_relationship(new_stargazer_vertex, self.__stargazers, index)
        self.add_starred_repos(new_stargazer_vertex, main_vertex, index)
```

Figura 5.25: Método para añadir los *stargazers* refactorizado.

```python
def add_follower_relationship(self, new_vertex: Vertex, stargazers: list, index: int) -> None:
    followers: list = self.__stargazers_followers[index]

    for follower in followers:
        try:
            stargazers.index(follower)
            follower_vertex: list = find_vertex(self.g, self.__v_name, follower['login'])
            if follower_vertex:
                self.create_edge(relationship.FOLLOWS.value, follower_vertex[0], new_vertex)
            else:
                follower_vertex = self.create_stargazer_vertex(follower)
                self.create_edge(relationship.FOLLOWS.value, follower_vertex, new_vertex)
        except:
                pass
```

Figura 5.26: Añadir relaciones entre usuarios refactorizado.

```python
def add_starred_repos(self, new_vertex: Vertex, main_vertex: Vertex, index: int):
    starred_repos: list = self.__stargazers_starred_repos[index]

    for starred in starred_repos:
        repeated_repos: list = find_vertex(self.g, self.__v_name, starred['name'])
        if repeated_repos:
            if repeated_repos[0] != main_vertex:
                self.create_edge(relationship.STARRED.value, new_vertex, repeated_repos[0])
        else:
            starred_repo: Vertex = self.create_repository_vertex(starred)
            self.create_edge(relationship.STARRED.value, new_vertex, starred_repo)
```

Figura 5.27: Añadir repositorios que le interesan a los usuarios refactorizado.

Como se puede apreciar, uno de los beneficios resultantes de separar ambas lógicas es que se reduce considerablemente el código de la clase y los métodos, además de mejorar la legibilidad del código y respetar el principio de responsabilidad única, encargándose cada clase de una funcionalidad del programa.

La clase encargada de la extracción concurrente de información de la API es *dataExtraction*, el constructor de dicha clase requiere dos parámetros, el nombre completo del repositorio y el token de autenticación de la cuenta de GitHub. El método principal de la clase es *fetch data*, el cual ejecuta el método asíncrono *fetch repo and stargazers*, retornando dos listas, la primera formada por listas con todos los seguidores de cada uno de los *stargazers* y la segunda formada por listas con los últimos 20 repositorios a los que han dado me gusta los usuarios 5.28.

```python
def fetch_data(self) -> Tuple[list,list]:
    stargazers_followers_list, stargazer_starred_repos_list = asyncio.run(self.fetch_repo_and_stargazers())
    return stargazers_followers_list, stargazer_starred_repos_list
```

Figura 5.28: Método *fetch data* donde se llama a un método asíncrono que devuelve una tupla con las listas de seguidores de los usuarios y los repositorios que les interesan.

El método asíncrono *fetch repo and stargazers* 5.29 crea un objeto *ClientSession* de la librería *aiohttp* al que se le pasa por parámetro la cabecera HTTP *Authorization* con el token de autenticación del usuario y se establece que no tendrá *timeout*, ya que el valor predeterminado son 5 minutos y trás discurrir dicho tiempo, la sesión caducará. Al igual que el objeto *session* de la librería Request, permite persistir la información en cada llamada a la API. Los siguientes dos pasos que se realizan son extraer el repositorio que se quiere analizar y los *stargazers* del mismo, para luego extraer concurrentemente todos los repositorios y seguidores de los usuarios. Para realizar esta labor, primero se crean dos listas donde se guardarán los *tasks* las cuáles son *futures* que ejecutan una corrutina, luego se recorre la lista de *stargazers*, guardando las distintas *tasks*. Dichas corrutinas corresponden al método *request api* que efectúa las llamadas a la API de GitHub. Una vez termina el bucle, el método *gather* ejecuta concurrentemente las *tasks* y retornando su resultado, este será una lista formada por listas con los seguidores de los *stargazers* y una lista formada por listas de los repositorios que les interesa.

```python
async def fetch_repo_and_stargazers(self) -> Tuple[list, list]:
    session_timeout = aiohttp.ClientTimeout(total=None)
    async with aiohttp.ClientSession(headers=self.headers, timeout=session_timeout) as session:
        await self.fetch_main_repo(session)
        self.__stargazers = await self.request_api(None, self.__MAX_NUMBER_ITEMS, api_data.STARGAZERS.value, True, session)

        starred_tasks = []
        follower_tasks = []
        for stargazer in self.__stargazers:
            follow_task =  asyncio.ensure_future(self.request_api(stargazer,
                                                                  self.__MAX_NUMBER_ITEMS,
                                                                  api_data.FOLLOWERS.value,
                                                                  False,
                                                                  session))
            follower_tasks.append(follow_task)
            starred_task = asyncio.ensure_future(self.request_api(stargazer,
                                                                  self.__MAX_REPOS_STARGAZER,
                                                                  api_data.STARRED.value,
                                                                  False,
                                                                  session))
            starred_tasks.append(starred_task)
        starred_results = await asyncio.gather(*starred_tasks)
        follower_results = await asyncio.gather(*follower_tasks)

        return starred_results, follower_results
```

Figura 5.29: Creación *tasks* para la extracción concurrente de datos de la API.

Request api 5.30, es el método encargado de efectuar las llamadas a la API de GitHub, al igual que en el Sprint 1, primero construye la dirección *url* al *endpoint* de la API, luego realiza la llamada, si la solicitud ha tenido éxito, entonces revisa si hay paginación y retorna el resultado, en caso contrario, es decir, se ha alcanzado alguno de los *rate limits* y el *flag unlock* está abierto, entonces, se cierra dicho *flag*, se guarda el *task* actual en una variable global, y se duerme dicho *task* hasta que pase el tiempo indicado por los *rate limits*, una vez pasado ese tiempo, llama recursivamente al método repitiendo la llamada. Como se han alcanzado los límites de llamadas de API, la respuesta del resto de *tasks* al realizar la llamada será ”403”, por ende, se les indica con el método *wait for* que esperen hasta que dicha tarea termine de ejecutarse, para luego llamar recursivamente al método y repetir la llamada.

```python
async def request_api(self, stargazer: json, num_items: int, info: str, is_repo_url: bool, session) -> list:
    if is_repo_url:
        url: str = self.__API_URL+"repos/%s/%s?per_page=%d" % (self.full_name_repository, info, num_items)
    else:
        url: str = self.__API_URL+"users/%s/%s?per_page=%d" % (stargazer['login'], info, num_items)

    async with session.get(url , headers=session.headers) as api_response:
        response_json: list = await api_response.json()
        if api_response.status == self.__OK_STATUS_CODE:
            response_json = await self.check_pagination(session, info, api_response, response_json, url)
        elif api_response.status == self.__RATE_LIMIT_STATUS_CODE and self.__UNLOCK:
            self.__UNLOCK = False
            self.__TASK = asyncio.current_task()
            await self.sleep_execution(api_response)
            response_json = await self.request_api(stargazer, num_items, info, is_repo_url, session)
        else:
            await asyncio.wait_for(self.__TASK, timeout=None)
            response_json = await self.request_api(stargazer, num_items, info, is_repo_url, session)
        return response_json
```

Figura 5.30: Realización de las peticiones a la API. El método es una refactorización de la versión anterior aptada a la concurrencia.

La paginación se maneja de la misma manera que el Sprint 1, con la diferencia de que se asigna una tarea por cada página y una vez guardadas todas las tareas en una lista, se consumirían todas las páginas concurrentemente, reduciendo el tiempo de ejecución del programa.

El tiempo que debe dormir un *task* o tarea antes de seguir ejecutándose, se obtiene con el método *sleep execution*, el cual, dependiendo de la cabecera identifica que *rate limit* se ha alcanzado y extrae el tiempo que es necesario parar la tarea antes de realizar otra llamada. En el supuesto de haberse alcanzado el *rate limit* principal, es decir, se han realizado 5000 llamadas a la API, obtiene en formato UTC el tiempo que debe descansar en la cabecera *X-RateLimit-Reset*, luego lo convierte a segundos y duerme la tarea, en caso de haberse alcanzado el *secondary rate limit* se obtiene el valor de la cabecera *Retry-After* y se duerme la tarea. Finalmente, se desbloquea el *flag unlock* y continua con la ejecución del programa 5.31.

```python
async def sleep_execution(self, api_response: aiohttp.ClientResponse) -> None:
    if 'Retry-After' in api_response.headers:
        print("Secondary rate limit exceeded")
        sleep_time = api_response.headers['Retry-After']
        if sleep_time == '60': await asyncio.sleep(int(sleep_time))
    else:
        print("Rate limit exceeded")
        utc_reset_time = datetime.fromtimestamp(int(api_response.headers['X-RateLimit-Reset']))
        sleep_time: float = (utc_reset_time - datetime.utcnow() + timedelta(0, 5)).total_seconds()
        print(sleep_time)
        await asyncio.sleep(sleep_time)

    self.__UNLOCK = True
```

Figura 5.31: Método encargado de dormir los *tasks* en caso de alcanzar alguno de los *rate limits*.

Otra de las principales funcionalidades implementadas durante esta iteración fue la creación de una interfaz por línea de comandos que facilitara el uso de la aplicación. Mediante el uso de la librería *argparse* 5.32, la cual pide al usuario que al ejecutar la aplicación pase dos parámetros, el nombre completo del repositorio y el token de autenticación de su cuenta personal, en caso de faltar alguno de los dos, muestra un mensaje indicando los argumentos válidos y en caso de introducir la opción *-h* se mostrará texto de ayuda indicando los argumentos a introducir y una explicación de cada uno de ellos. También se mostrarán mensajes de error en el caso de que el repositorio indicado no exista o el token de autenticación no sea válido o haya espirado 5.33. La opción *–load* se explicará más adelante.

```python
def start_cli(self) -> None:
    parser = argparse.ArgumentParser(description="GitNet")
    group = parser.add_argument_group('group')
    group.add_argument("-r", "--repository", help="Enter full name of the repository (author/repo)")
    group.add_argument("-t", "--token", help="Enter OAuth GitHub token")
    parser.add_argument("-l", "--load", help="Enter the name of a graph file")
    args = parser.parse_args()
    self.process_arguments(args.repository, args.token, args.load)
```

Figura 5.32: Uso librería argparse.

```
[root@8cb3b179d0f5 GitNet]# /bin/python /workspaces/GitNet/src/main.py  -a
usage: main.py [-h] [-r REPOSITORY] [-t TOKEN] [-l LOAD]
main.py: error: unrecognized arguments: -a
[root@8cb3b179d0f5 GitNet]#
```

Figura 5.33: Mensaje en caso de introducir un argumento erróneo.

```
[root@8cb3b179d0f5 GitNet]# /bin/python /workspaces/GitNet/src/main.py  -h
usage: main.py [-h] [-r REPOSITORY] [-t TOKEN] [-l LOAD]

GitNet

options:
  -h, --help            show this help message and exit
  -l LOAD, --load LOAD  Enter the name of a graph file

group:
  -r REPOSITORY, --repository REPOSITORY
                        Enter full name of the repository (author/repo)
  -t TOKEN, --token TOKEN
                        Enter OAuth GitHub token
```

Figura 5.34: Ejecución del comando –help.

Tras ejecutar la aplicación 5.35, el usuario deberá esperar un tiempo hasta que se genere el grafo, en caso de llegar a alguno de los *rate limits*, se mostrará un mensaje por pantalla con el tiempo en segundos que debe esperar. Una vez creado el grafo, entra en juego la librería *prompt toolkit*, que permite a la aplicación seguir ejecutándose como un REPL [13].

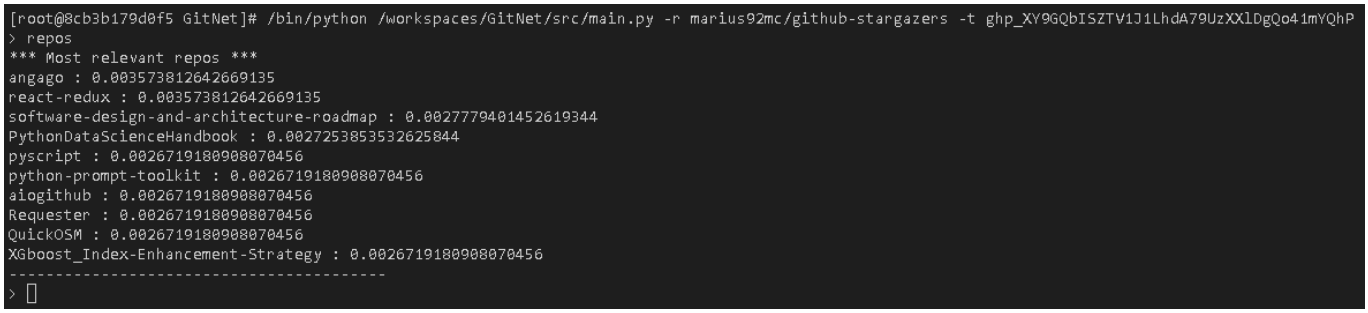

Figura 5.35: Ejemplo uso de la interfaz línea de comandos.

En caso de que el usuario introduzca un comando erróneo, se mostrará por pantalla un mensaje indicándole que use el comando *help* 5.36, que muestra los argumentos válidos.

Figura 5.36: Comando help.

[13] https://en.wikipedia.org/wiki/Read-eval-print_loop

Para implementar esta sección del programa, primero se genera un objeto de tipo *PromptSession*, luego se itera en un bucle infinito y en cada iteración se ejecuta el método *session.prompt()*, el cual permite al usuario introducir un parámetro por consola, para posteriormente procesarlo. Seguidamente, se comprueba que el texto introducido se encuentra entre las opciones del programa, es caso afirmativo, se ejecuta el método que corresponde a dicha función y en el caso de que el parámetro sea *save*, se guarda el grafo con el formato indicado, en el caso de que el parámetro sea incorrecto saldrá un mensaje en pantalla indicándolo. Finalmente, se muestra en pantalla el número de vértices y aristas del grafo y se despide la aplicación.

```
def repl(self) -> None:
    session = PromptSession()
    while True:
        try:
            text = session.prompt('> ')
            if text in self.__OPTIONS:
                method = getattr(self, self.__OPTIONS[text])
                method()
                print("----------------------------------------")
            elif text.split(" ")[0] == 'save' and len(text.split()) == 2:
                self.save_graph(text)
            else:
                print("""Invalid argument. Write 'help' to see valid arguments.""")
        except KeyboardInterrupt:
            break
        except EOFError:
            break
    print(self.graph.g.num_vertices)
    print('GoodBye!')
```

Figura 5.37: Implementación código del REPl. Se comprueba si el argumento introducido es válido, en caso, verdadero se ejecuta el método correspondiente, en caso contrario, aparece un mensaje de error.

Las dos últimas funcionalidades implementadas durante la iteración, fueron guardar y cargar el grafo. Para su desarrollo se creó una clase *manageGraph*, donde se implementarían todas las funcionalidades requeridas para el manejo del grafo. La librería *graph-tool* tiene una función que permite guardar los grafos en distintos formatos, gt, graphml, xml ,dot y gml. Los únicos formatos de ficheros que preservan en perfecto estado los mapas internos de propiedades son gt y graphml 5.38.

```
> save gt
Graph have been saved correctly in /data folder
>
```

Figura 5.38: Línea de ejecución para la guardar los grafos, indicando el formato.

La carga del grafo se implementó como un argumento opcional al ejecutar la aplicación –*load*, al que se le pasa como parámetro el nombre del fichero donde se ha guardado el grafo. Al ejecutar la aplicación, la carga del grafo se realiza mediante el uso del comando *load graph* de la libería *Graph-tool* 5.39. Una vez cargado el usuaro podrá ejecutar el comadno *draw* para visualizarlo.

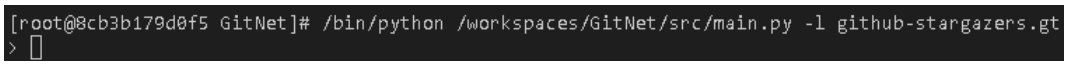

Figura 5.39: Línea de ejecución para la carga de los grafos de la aplicación.

5.4.1. Planificación del Sprint 4

No hubo problemas a la hora de terminar todas las historias de tercer Sprint, esto demuestra que la velocidad calculada para la iteración fue la correcta. No obstante, para este cuarto Sprint de tres semanas de duración se dispone de más tiempo para realizar las tareas debido a que gran parte de las entregas ya fueron realizadas y por ende, la carga de trabajo se redujo considerablemente. En esta iteración se dispondrán de 14 horas de trabajo semanal. De este modo el tiempo ideal será de $14 * 3 = 56$, a partir de este tiempo se extraen $56/4 = 14$ puntos de historia. Por último aplicando el factor de foco $14 * 0{,}93 = 13{,}02$ puntos de historias a repartir en el Sprint.

ID	Nombre	Horas	Puntos	Prioridad
HU17	Limpieza datos	20	5	5
HU18	Creación del modelo limits	20	5	5
HU19	Integrar el modelo	12	3	5

5.5. Sprint 4.*Extreme Gradient Boosting*

Este Sprint está centrado en la creación de un modelo de *machine learning* empleando el algoritmo de *XGBoost* para analizar y extraer los repositorios más importantes dentro de la comunidad. Para ello, el primer paso se centro en el *datacleaning* o limpieza de datos, proceso consistente en la depuración de datos mal formateados, incompletos o incorrectos.

Tras ejecutar la aplicación y extraer la información de la API, se genera una lista en la que se guardan los últimos 20 repositorios en los que están interesados cada uno de los usuarios. Dicha lista es usada posteriormente para generar el *dataset* con el que se entrenará el modelo. En este caso el repositorio escógido para el análisis es alicevision/AliceVision que cuenta con más de 2.2k *stargazers* y permite recolectar un conjunto de datos de 17.484 repositorios. Una vez obtenida la información y guardada en un *dataFrame* 5.40, se incluyen exclusivamente las columnas relevantes para el entrenamiento.

full_name	created_at	stargazers_count	forks_count
joshje/Responsive-Enhance	2012-02-02T23:27:02Z	380	26
WordPress/WordPress	2011-12-01T07:05:17Z	16281	11253
YirsisHertz/codigos-curso-react	2021-04-02T20:50:57Z	47	53

Figura 5.40: Dataset antes de transformar los datos.

La primera columna *full name* hace referencia al nombre del repositorio, posteriormente será extraída del *dataset*, la segunda columna *created at*, muestra la fecha en la que fue creado el repositorio, en este caso al encontrarse ante un dato categórico, será necesario sustituir dichas fechas por un 0 en caso de que el repositorio no haya sido creado en el último año o un 1 en caso contrario, la tercera columna *stargazers count*, representa el número de seguidores que tienen los repositorio y la cuarta columna *forks count* muestra el número de *forks* de cada repositorio.

Luego, se añadieron dos columnas más, *count*, que representa el número de usuarios dentro de la comunidad que siguen al repositorio, se obtuvo contando el número de veces que se repetía un repositorio en el *dataFrame*, guardando los resultados en la columna y borrando los duplicados y la columna *is relevant*, que indica que si un usuario el relevante o no. Para generar esta última columna, primero se aplicó el algoritmo de *Personalized PageRank* al grafo, una vez obtenidos y guardado los resultados en una lista, se extrae el 10 % de los repositorios más relevantes y posteriormente se asigna dentro del *dataset* el valor 1 a los repositorios relevantes y 0 al 90 % restante (ver figura 5.41).

full_name	created_at	stargazers_count	forks_count	open_issues_count	count	is_relevant
alicevision/AliceVision	0	2185	651	78	470	1
lucidrains/DALLE2-pytorch	1	5539	320	15	39	1
weiaicunzai/pytorch-cifar100	0	2778	910	44	2	1
ity-Face-Recognition-Dataset	0	98	32	2	1	0

Figura 5.41: Dataset de entrenamiento, resultante de realizar el *datacleaning*.

Para evitar tener un *dataset* demasiado desbalanceado, se escoge aleatoriamente 1.748 repositorios entre los no relevantes, la misma cantidad que los repositorios relevantes. El proceso de procesamiento de los datos se desarrolló en un método donde se crea el *dataset* de entrenamiento, apoyándose de métodos auxiliares encargados de incluir la columna *is relevant* y de transformar los datos de la columna *created at* 5.42.

```
def create_dataframe(self, graph: interestGraph) -> Tuple:
    stargazers_starred: list = graph.get_stargazers_starred_repos()
    starred_repos: list = [repo for stargazer in stargazers_starred for repo in stargazer]

    df = pd.DataFrame.from_records(starred_repos)
    df.drop(labels=df.columns.difference(['full_name',
                                          'created_at',
                                          'stargazers_count',
                                          'forks_count'
                                          ]),
                                           axis=1,
                                           inplace=True)
    df['count'] = df.groupby('full_name')['full_name'].transform('count')
    df = df.drop_duplicates()
    self.set_created_at_value(df)

    self.create_training_dataset(graph, df)

    names = df.pop("full_name")
    df.to_csv(f"data/temp")

    return df, names
```

Figura 5.42: Método principal para crear el dataframe.

Tras crear el conjunto de datos datos que se usará para entrenar el modelo, el siguiende paso consite en apoyarse de la validación cruzada [25] para encontrar los hiperparámetros que mejor se ajusten al mismo. Primero se extrae la columna *is relevant* ya que será el *target* u objetivo a predecir del modelo, luego se divide el *dataset* de la siguiente manera, 50 % para el subconjunto de entrenamiento, 25 % de pruebas con el que se seleccionará el mejor de los modelos creados y 25 % de validación, para comprobar que el modelo no está sobreajustado 5.43.

```
X_train, X_test, y_train, y_test= train_test_split(X, y, test_size= 0.5, random_state=1)

X_valid, X_test, y_valid, y_test= train_test_split(X_test, y_test, test_size= 0.5, random_state=1)
```

Figura 5.43: Separación del dataset inicial en subconjuntos.

El siguiente paso, consiste en crear el modelo, en este caso de clasificación, luego se asginan los hiperparámetros con los que se quiere entrenar el modelo y se crea un objeto *GridSearchCV* 5.44 con el que se realizará la validación cruzada y al que se le pasan por parámetros, el modelo, los hiperparámetros, el número de *splits* que en este caso serán 5, y el *scoring*, que es la estrategia utilizada para medir la actuación de los modelos sobre el conjunto de test. La métrica seleccionada en este caso es *average precision* [14]ya que indica si un modelo es capaz de identificar los casos positivos, en este caso repositorios relevantes, sin marcar muchos ejemplos negativos como positivos, es decir, falsos positivos.

[14]https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html

```
xgb = xgboost.XGBClassifier()

parametres = {  'objective': ['binary:logistic'],
                'learning_rate': [0.25],
                'n_estimators': [95, 100, 150],
                'reg_alpha': [0, 0.5, 1],
                'gamma': [0, 0.5, 1],
                'early_stopping_rounds': [20, 30, 50],
                'eval_metric': ['aucpr'],
                'max_depth': [3, 4, 5]
                }

fit_params = { 'eval_set': [(X_test, y_test)]}

clf =  GridSearchCV(xgb, parametres, cv=5, scoring='average_precision')
clf.fit(X_train, y_train, **fit_params)
```

Figura 5.44: Identificación del mejor modelo mediante validación cruzada.

Después de realizar varias pruebas con diferentes hiperparámetros, la mayor puntuación alcanzada con el conjunto de validación fue de un 85.28 % de *average precision* para el mejor de los modelos. Una vez se disponen de los hiperparámetros óptimos para el modelo, se entrenara al mismo, para posteriormente guardarlo en formato *json*. Algunos de los hiperparámetros modificados fueron:

- *objective*: define la función objetivo del modelo, debido a que el problema propuesto es de clasificación binaria se escogió *binary:logistic*.
- *learning rate*: es una técnica para ralentizar el entranamiento en los modelos de *gradient boosting*, aplicando un factor (en este caso 0.25) a las correcciones de los nuevos árboles que se agregan al modelo.
- *nestimators*: es el número de árboles que generará el modelo. Se usaron 100 en el modelo de este trabajo.
- *eval metric*: es una métrica que evalúa el modelo en cada iteración. Se escogió *aucpr*.
- *max depth*: determina la profundidad de cada árbol del modelo, un valor pequeño puede ayudar a controlar el sobreajuste. En el caso del modelo generado, la profundidad de los árboles es de 5.

Otros de los hiperparámetros con los que se realizarón pruebas fueron *gamma* que establece si se realizará una partición de una hoja a partir de la reducción de pérdida mínima, *min child weight* que establece que si el peso de instancia de un nodo hoja resultante de la partición del árbol, es menor que dicho valor, entonces no se incluirá dicha partición o *reg alpha* que representa el factor de regularización L1, sin embargo, en el mejor de los modelos el valor de estos hiperparámetros no variaba respecto a su valor *default*.

Una vez se dispone del modelo, el siguiente paso es implementarlo dentro de la aplicación. Para ello se creó otro argumento en la interfaz por línea de comandos *repos xgboost*.Tras ejecutar dicho argumento, se transforma la lista de los repositorios de la comunidad que se está analizando en un *dataFrame* con la misma estructura que se comentó anteriormente, con la diferencia de que no cuenta con la columna *is relevant* ya que son los datos a predecir.

Se desarrollo un método en la clase *dataProcessing* 5.45, encargado de cargar el modelo y realizar la predicción. Una vez obtenidos los resultados, el número de repositorios que el algoritmo a clasificado como relevantes, puede ser bastante elevado, sobretodo cuando se analizan comunidades grandes, por ello, se ordenan los repositorios según el número de seguidores que contengan dentro de la comunidad y se extraen los 30 primeros, de este modo se acota considerablemente el resultado final.

```python
def xgboost_repos(self, df: pd.DataFrame, names_column: pd.DataFrame) -> None:
    model = xgboost.XGBClassifier()
    model.load_model("xgb_model/model.json")
    result = model.predict(df)

    df['is_relevant'] = result.tolist()

    df.insert(loc=0, column='full_name', value=names_column)
    df.sort_values(by=['count'], inplace=True, ascending=False)

    df = df[df.is_relevant != 0]
    df = df.dropna()
    df = df.head(30)

    df.to_csv("xgb_model/result", index=False)
    print("The result has been saved in the 'xgb_model' folder")
```

Figura 5.45: Método para obtener los repositorios más relevantes con XGBoost.

Por último, se devolvería un fichero en formato *csv* con los resultados 5.46.

	full_name	created_at	stargazers_count	forks_count	count	is_relevant
0	digital-standard/ThreeDPoseUnityBarracuda	0	1061	193	393	1
1	BandaiNamcoResearchInc/Bandai-Namco-Research-M...	1	2269	283	32	1
2	generative-design/Code-Package-p5.js	0	17472	3674	32	1

Figura 5.46: Resultado de la predicción.

5.6. Análisis calidad del código

Unos de los principales objetivos durante el desarrollo de este proyecto ha sido crear un código limpio que, cumpliendo con las mejores prácticas de programación de software, sea por tanto mantenible, simple y consistente. Para asegurar que todo esto se cumpla, se introdujo en la última etapa del desarrollo del producto la herramienta **SonarQube** [27], una plataforma de código abierto para la inspección continua de la calidad del código, permitiendo detectar errores y vulnerabilidades. Tras realizar un primer análisis de la aplicación usando dicha herramienta, se obtuvo el siguiente resultado (ver figura 5.47):

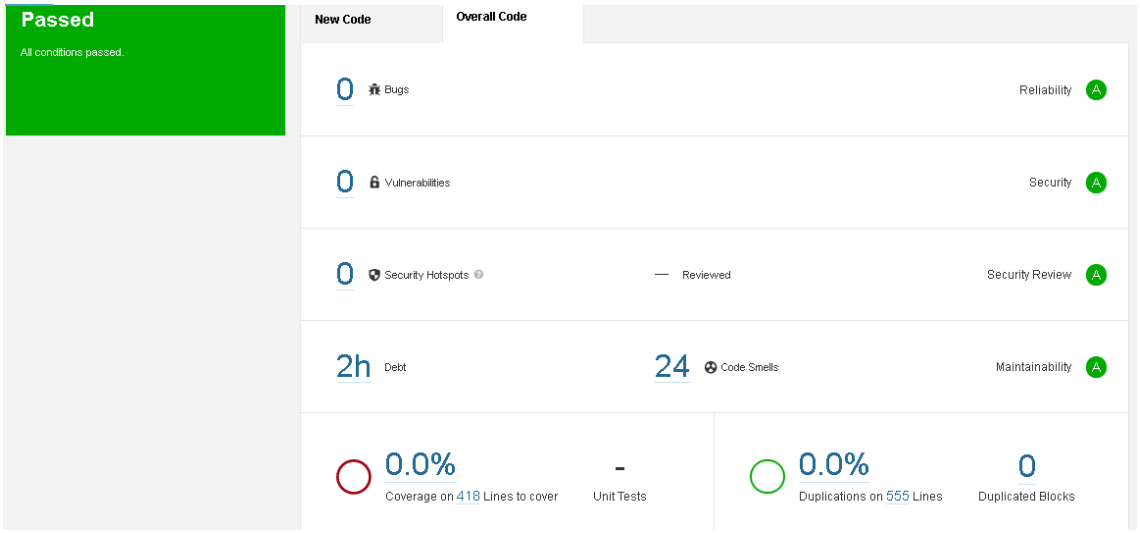

Figura 5.47: Análisis con SonarQube.

El análisis se realizó empleando el *quality gate* por defecto *sonar way* 5.48 el cual, establece que la cobertura del código debe de ser mayor al 80.0 %, el código duplicado no puede superar el 3.0 %, las notas respecto a *mantainability* y *reliability* debe de ser máxima, no debe de haber *security hotspots reviewed* y nota respecto a la seguridad de la aplicación debe de ser máxima. Respecto al *quality profile*, se empleó el que viene por defecto para Python, con 158 reglas activas.

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A

Figura 5.48: Sonar way

Reliability: esta nota está asociada al número de bugs del proyecto y la importancia de cada uno de ellos, como en este caso no ha encontrado ninguno, la nota es máxima.

Security: esta nota está asociada al número de vulnerabilidades encontradas en el programa, como no ha encontrado ninguna, se puntúa la aplicación con la nota máxima.

Security Review: está calificación está basada en el porcentaje de puntos de acceso de seguridad revisados, también se obtiene la nota máxima.

Maintainability: se basa en la cantidad de *code smells*, es decir, partes del código que presentan problemas de diseño y legibilidad, la deuda técnica, es decir, la relación entre el costo de desarrollo del software y el costo de arreglarlo, con base en el costo de tiempo de los problemas y el tiempo estimado para escribir el número líneas de código para corregirlos.

Se puede apreciar que el proyecto pasó el *quality gate* por defecto, que son un conjunto de reglas mínimas que debe pasar un proyecto, además, no se encontró código duplicado en la aplicación. Sin embargo, la deuda técnica es de 2 horas y se encontraron 24 *code smells* 5.49.

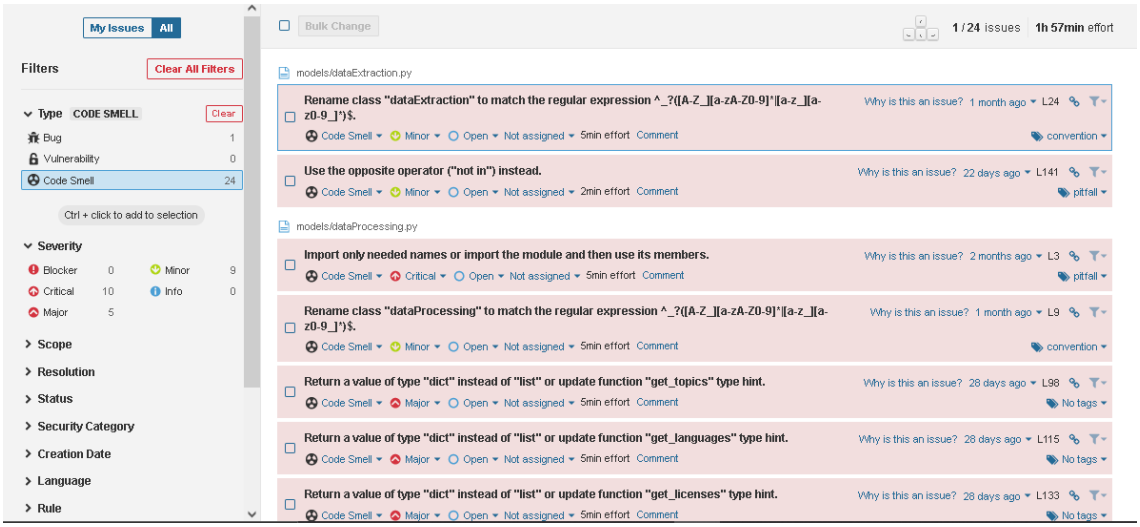

Figura 5.49: Code smells del programa.

Las causas de gran parte de los *code smells* son las siguientes:

- Rename class ”dataExtraction”to match the regular expression ...: “El código no cumple con la convención *snakecase* aceptada por la PEP-8, cuando la clase es usada principalmente como un invocable“.
- Import only needed names or import the module and then use its members: “llamar a un módulo de un a librería usando * puede generar conflictos entre nombres definidos localmente e importados, reduce la legibilidad del código y dificulta la depuración, ya que desordena el espacio de nombres local“.
- Return a value of type dict instead of list or update function get topics type hint: “El tipo de objeto devuelto por el método, no concuerda con el que se indica en la declaración del mismo“.

- Remove the unused function parameter title: “No se utiliza dicho parámetro en el método por ende hay que eliminarlo“.

El siguiente paso consistió en corregir gran parte de los *code smells* existentes 5.50:

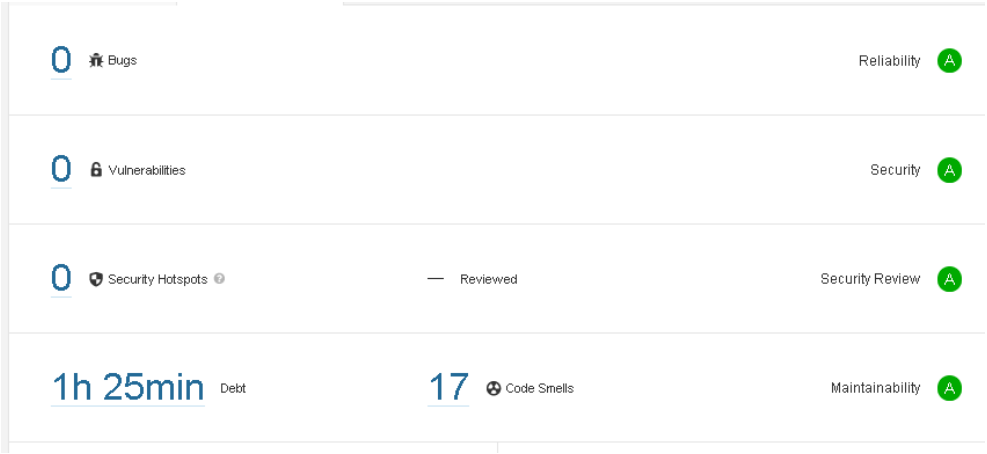

Figura 5.50: Segundo análisis SonarQube.

El tiempo de deuda técnica se redujo considerablemente gracias a la refactorización del código. El tiempo restante que se puede observar, corresponde al cambio de nombre de clases y métodos, ya que, no cumplen el estándar *snake case*, sin embargo, se consideró que los nombres actuales se ajustan adecuadamente al funcionamiento de los métodos y clases y por ello, se decidió no modificarlas.

Otros de los parámetros que SonarQube permite analizar son la *cyclomatic complexity*, calculada basándose en el número de caminos a través del código, es decir, cada vez que el flujo de control de una función se divide, el contador de complejidad se incrementa en 1. Este cálculo varía ligeramente según el lenguaje de programación. La *cognitive complexity*, representa la dificultad para entender un flujo de control (ver figura 5.51).

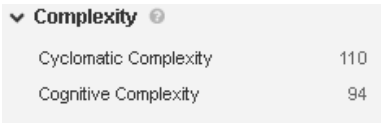

Figura 5.51: Complejidad del código.

Capítulo 6

Conclusiones y trabajo futuro

Durante la realización de este Trabajo Fin de Grado, se ha desarrollado una aplicación que permite el análisis de las comunidades de usuarios que se forman alrededor de un proyecto en la plataforma de desarrollo colaborativo GitHub, generando grafos de interés entre usuarios y repositorios y visualizando los resultados finales en gráficas.

Recalcar el uso del marco de trabajo Scrum para el desarrollo ágil del proyecto cuyos elementos princiaples han sido adaptados a las circunstancias propias de un TFT. Desde una fase inicial en el proyecto, se organizó las tareas mediante un tablero Kanban, una serie de historias de usuario definiendo las funcionalidades del software y realizando una planificación antes de cada iteración, de tal modo que, se obtuviera una mejora continua en cada Sprint. Además, se realizaron varias reuniones con el tutor en las que se revisó el estado del proyecto y se tomaron decisiones acerca de como abordar los diferentes problemas que surgieron durante la etapa de desarrollo.

Destacar el empleo de concurrencia para la extracción de datos de la plataforma, lo que supuso un verdadero reto a la hora de manejar los diferentes límites establecidos por la API y la extracción de datos paginados, pero que optimizó notablemente el tiempo necesario en la etapa de recolección y el empleo del algoritmo de *Personalized PageRank*, diseñado y empleado en el pasado por *Google*, que en el caso particular de este proyecto, permite clasificar los distintos usuarios y repositorios dentro las comunidades por relevancia. Referente a esto último, también se ha de recalcar la creación de un modelo de *machine learning* empleando el algoritmo *XGBoost* con el objetivo de obtener los repositorios más relevantes, consumiendo menos recursos en comparación con *Personalized PageRank*.

Este proyecto ha permitido al alumno aplicar distintos conocimientos adquiridos a lo largo de la carrera en campos como la teoría de grafos, algoritmia o el desarrollo de software de calidad, además de reforzar habilidades vinculadas al aprendizaje autónomo y la resolución de problemas. Gracias a la buena organización surgida del empleo de metodologías ágiles durante el desarrollo, se alcanzaron los objetivos planteados para el proyecto. No obstante, siempre hay puntos de mejora y nuevas funcionalidades de trabajo futuro, como pueden ser:

- Generar grafos únicamente de usuarios de tal modo que se pueda extraer más datos de los mismos, para poder realizar un análisis más profundo.

- Crear una interfaz web con el *framework* Angular que permita realizar las mismas acciones que el CLI y muestre las gráficas generadas.

Apéndice A

Programación Asíncrona

A.1. Asyncio

Asyncio [21] es un módulo de Python para el desarrollo de aplicaciones concurrentes utilizando corrutinas, la principal diferencia con respecto a otros módulos como *threading* que implementa la concurencia a través de varios hilos o *multiprocessing* que la implementa usando procesos del sistema, asyncio emplea un único hilo y proceso, cambiando de tareas explícitamente en momentos óptimos. Donde mejor encaja es en el manejo de operaciones de entrada, salida y código de red estructurado de alto nivel. La librería proporciona un conjunto de APIs de alto nivel:

- Permite ejecutar corrutinas de Python, y disponer de total control en la ejecución.
- Crear redes E/S y comunicación entre procesos (IPC).
- Permite controlar los subprocesos.
- Sincronización de código concurrente.

Adicionalmente, incluye APIs de bajo nivel para los desarrolladores de *frameworks* y librerías:

- Crear y manejar el bucle de eventos, que proveen APIs asíncronas para redes, ejecuta subprocesos y gestiona señales del sistema operativo, entre otras cosas.
- Implementa protocolos eficiente usando transportes.
- Código de sintaxis async/await y librería puente basada en retrollamadas.

Los programas [23] creados empleando otro tipo de modelo de concurrencia, están desarrollados de manera lineal, confiando en hilos o la gestión de subprocesos al sistema operativo o al tiempo de ejecución del lenguaje para según corresponda, realizar el cambio de contexto, en cambio, las aplicaciones basadas en asyncio requieren que el código se encargue explícitamente de los cambios de contexto. Para comprender adecuadamente este modelo de concurrencia, primero se han de explicar ciertos conceptos interrelacionados.

El **bucle de eventos** es un objeto, encargado de manejar de manera eficiente los eventos del sistema, cambios de contexto y eventos de entrada salida, forma parte del marco central de asyncio. Existen diferentes implementaciones del bucle de eventos, de tal modo que se puede escoger una en específico dentro de la aplicación, maximizando las capacidades de cada sistema operativo, lo cual es muy útil es sistemas como Windows, donde algunos tipos de bucles permiten soportes para procesos externos de tal modo que se optimiza el rendimiento de algunas E/S.

Las aplicaciones interactúan con el bucle de eventos para registrar código que se requiere ejecutar, autorizando al bucle de eventos a que ejectúe las llamadas al código cuando estén disponibles los recursos. Un ejemplo puede ser un servidor red donde se abren los conectores y se registran de tal modo que se obtengan avisos en caso en que ocurra algún evento. El bucle de eventos avisa al código cuando hay que leer datos o si hay una nueva conexión entrante. El código cederá el control de nuevo tras un breve periodo de tiempo, cuando en el contexto actual no se pueda realizar más trabajo, por ejemplo si desde un conector no hubiesen más datos a leer, el servidor devolverá el control al bucle de eventos.

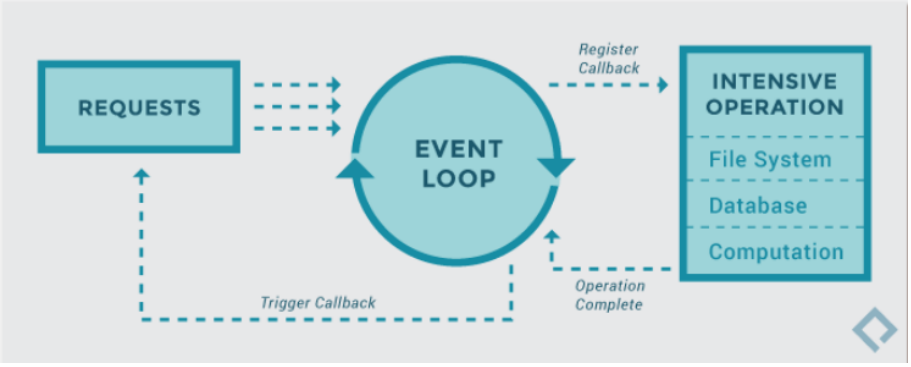

Figura A.1: Funcionamiento bucle de eventos [1].

Las **corrutinas** de Python son funciones especiales que transfieren el control al bucle de eventos sin perder su estado actual. Son similares a las funciones generadoras, de hecho en las versiones anteriores a Python 3.5 eran utilizados para implementar corrutinas. La librería también permite escribir código usando funciones *callback* en vez de crear corrutinas directamente, proporcionando capas de abstracción basadas en clases para transportes y protocolos. Los cambios implícitos de contexto de la implementación de hilos de Python son reemplazados por los cambios explícitos cuando las corrutinas entran en el bucle de eventos.

Los **futuros** [2] o *futures* son estructuras de datos que ha diferencia de las corrutinas, cuando se espera por estos, no se ejecuta el código, de tal modo que puede considerarse que representa un proceso en curso en otro lugar y que puede o no haberse terminado todavía. La librería también incluye primitivas de concurrencia como pueden ser semáforos o cerrojos.

Los **tasks** son objetos de tipo *future*, que envuelven y gestionan la ejecución de una corrutina. Las tareas se pueden programar en el bucle de eventos para que ejecuten cuando estén disponibles los recursos necesarios y generar un resultado que puedan usar otras corrutinas. Los *tasks* son parecidos a las *promises* en JavaScript.

[1] https://eng.paxos.com/python-3s-killer-feature-asyncio

A.2. Comparativa rendimiento

La siguiente tabla muestra una comparativa del tiempo aproximado que se tarda en extraer la información de la API y generar el grafo, empleando distintas librerías. El tiempo que tarda en generar el grafo es muy bajo debido al rendimiento de la librería *graph-tool* y el reducido número de nodos y aristas, ya que el repositorio examinado solo dispone de 12 stargazers.

Librería	Tiempo
PyGithub	20 segundos
Request	10 segundos
aiohttp	1.5 segundos

Cuadro A.1: Comparación rendimiento librerías.

Como se puede apreciar, la mejora de la velocidad de la aplicación es considerable, en el caso de repositorios formados por miles de stargazers puede significar pasar de varias horas de ejecución a tan solo par de minutos.

Apéndice B

Pila del producto

Historia de usuario-01	Extraer datos de la API
Prioridad	1
Tiempo estimado	12
Descripción	¿Qué?: Extraer *stargazers* y repositorios de la API de GitHub ¿Por qué?: Para generar el grafo de interés con dichos datos ¿Quién se beneficia?: El usuario final
Validación	Disponer todos los datos sobre los usuarios y repositorios guardados en variables. En caso de alcanzar algún límite de la API, termina la ejecución y muestra un mensaje. En caso de no existir el repositorio o que el token sea incorrecto, terminar aplicación y mostrar mensaje

Cuadro B.1: Historia de usuario 01

Historia de usuario-02	Generar grafo *stargazers*
Prioridad	1
Tiempo estimado	12
Descripción	¿Qué?: Crear algoritmo para generar grafo directo con los *stargazers* y repositorios. ¿Por qué?: disponer del grafo al que se le realizará posteriormente un análisis ¿Quién se beneficia? El usuario final
Validación	Comprobar que no hay vértices, ni aristas duplicadas Comprobar que el número de aristas del nodo principal es el correcto

Cuadro B.2: Historia de usuario 02

Historia de usuario-03	Añadir relaciones usuarios
Prioridad	
Tiempo estimado	
Descripción	¿Qué?: Ampliar el algoritmo para la creación del grafo, incluyendo las relaciones entre usuarios ¿Por qué?: Establecer las relaciones entre vértices para luego poder extraer información del grafo aplicando algoritmos. ¿Quién se beneficia?: El usuario final
Validación	Comprobar que no existen vértices sin aristas Comprobar que las relaciones entre vértices son las correctas

Cuadro B.3: Historia de usuario 03

Historia de usuario-04	Añadir repositorios que les interesan
Prioridad	1
Tiempo estimado	6
Descripción	¿Qué?: Ampliar el algoritmo de creación del grafo añadiendo repositorios que les interesan ¿Por qué?: para analizar que le interesa a los usuarios de dicha comunidad ¿Quién se beneficia?: El usuario final
Validación	El número de repositorios a crear por usuario no debe de ser mayor a 20. Comprobar que todos los repositorios tienen aristas entrantes

Cuadro B.4: Historia de usuario 04

Historia de usuario-05	Gestionar propiedades de los vértices
Prioridad	1
Tiempo estimado	4.8
Descripción	¿Qué?: Crear los mapas de propiedades de los elementos del grafo ¿Por qué?: para guardar las propiedades de los repositorios y los usuarios ¿Quién se beneficia? El usuario final
Validación	Comprobar que cada vértice tiene las propiedades adecuadas Comprobar que las aristas tienen las propiedades adecuadas

Cuadro B.5: Historia de usuario 05

Historia de usuario-06	Usuarios relevantes
Prioridad	2
Tiempo estimado	8
Descripción	¿Qué?: Aplicar *PageRank* al sub grafo de usuarios ¿Por qué?: para obtener los usuarios relevantes de la comunidad ¿Quién se beneficia?: El usuario final
Validación	Se mostrará una lista de los 10 usuarios más relevantes, junto con su valor de *PageRank*.

Cuadro B.6: Historia de usuario 06

Historia de usuario-07	Repositorios relevantes
Prioridad	2
Tiempo estimado	16
Descripción	¿Qué?: Aplicar *Personalized PageRank* al grafo ¿Por qué?: para obtener los repositorios más relevantes de la comunidad, teniendo en cuenta las características del repositorio ¿Quién se beneficia? El usuario final
Validación	Se mostrará en pantalla los 10 repositorios más relevantes, junto con su valor de *PageRank*

Cuadro B.7: Historia de usuario 07

Historia de usuario-08	Extraer lenguajes de programación
Prioridad	2
Tiempo estimado	4.5
Descripción	¿Qué?: Extraer los lenguajes de programación usados por los repositorios del grafo ¿Por qué?: obtener cuáles son los lenguajes en los que están interesados los usuarios actualmente ¿Quién se beneficia?: El usuario final
Validación	Se mostrarán los 10 lenguajes que más les interesa. Comprobar que solo se muestran lenguajes de programación

Cuadro B.8: Historia de usuario 08

Historia de usuario-09	Extraer tópicos
Prioridad	2
Tiempo estimado	4.5
Descripción	¿Qué?: Extraer los tópicos de los repositorios del grafo ¿Por qué?: obtener cuáles son los tópicos en los que están interesados los usuarios actualmente ¿Quién se beneficia?: El usuario final
Validación	Se mostrarán los 10 tópicos que más les interesa Comprobar que solo se muestran los tópicos

Cuadro B.9: Historia de usuario 09

Historia de usuario-10	Extraer licencias
Prioridad	2
Tiempo estimado	4.5
Descripción	¿Qué?: Extraer las licencias de los repositorios del grafo ¿Por qué?: obtener cuáles son las licencias en los que están interesados los usuarios actualmente ¿Quién se beneficia?: El usuario final
Validación	Se mostrarán las 10 licencias que más les interesa Comprobar que solo se muestran las licencias

Cuadro B.10: Historia de usuario 10

Historia de usuario -	Visualizar información resultante
Prioridad	2
Tiempo estimado	8
Descripción	¿Qué?: Visualizar los resultados con la librería matplotlib ¿Por qué?: facilitar la comprensión de los resultados al usuario ¿Quién se beneficia?: Al usuario final
Validación	Comprobar que las gráficas muestran todos los datos Comprobar que los datos mostrados son correctos

Cuadro B.11: Historia de usuario 11

Historia de usuario-12	Aplicar concurrencia
Prioridad	4
Tiempo estimado	12
Descripción	¿Qué?: Obtener datos de la API concurrentemente ¿Por qué?: para mejorar el rendimiento del programa ¿Quién se beneficia?: El usuario final
Validación	Comprobar que se extraen los datos correctamente de la API Comprobar que los datos quedan guardados en variables Comprobar que no se ha duplicado información durante la extracción

Cuadro B.12: Historia de usuario 12

Historia de usuario-13	Manejar rate limits
Prioridad	4
Tiempo estimado	8
Descripción	¿Qué?: Poner la aplicación en un estado de descanso al alcanzar los *rate limits* ¿Por qué?: para evitar que termine de ejecutar la aplicación durante la extracción de datos de la API ¿Quién se beneficia?: El usuario final
Validación	Comprobar que al alcanzar el *rate limit* el programa descansa el tiempo establecido indicado por la API Comprobar que al alcanzar el *secondary rate limit* la aplicación descansa un minuto Comprobar que todas las corrutinas siguen ejecutando tras terminar el estado de descanso

Cuadro B.13: Historia de usuario 13

Historia de usuario-14	Guardar grafo
Prioridad	5
Tiempo estimado	4.68
Descripción	¿Qué?: Guardar el grafo en el formato indicado ¿Por qué?: poder persistir el grafo generado ¿Quién se beneficia?: El usuario final
Validación	Comprobar que el grafo se ha guardado en el formato indicado

Cuadro B.14: Historia de usuario 14

Historia de usuario-15	Cargar grafo
Prioridad	5
Tiempo estimado	4.68
Descripción	¿Qué?: Cargar el grafo guardado con anterioridad ¿Por qué?: para poder acceder a la información que guarda dicho grafo ¿Quién se beneficia?: El usuario final
Validación	Comprobar que el grafo se puede cargar Comprobar el correcto estado de los mapas de propiedades del grafo

Cuadro B.15: Historia de usuario 15

Historia de usuario-16	Interfaz por línea de comandos
Prioridad	4
Tiempo estimado	12
Descripción	¿Qué?: Crear un CLI ¿Por qué?: para facilitar al usuario el uso de la aplicación y que pueda elegir entre las distintas opciones que ofrece la aplicación ¿Quién se beneficia?: El usuario final
Validación	Comprobar que están validados los argumentos Mostrar mensaje de error en caso de introducir un argumento erróneo. Comprobar que todas las funcionalidades de la aplicación ejecutan adecuadamente

Cuadro B.16: Historia de usuario 16

Historia de usuario-17	Limpieza de datos
Prioridad	5
Tiempo estimado	20
Descripción	¿Qué?: Crear el dataset de entramiento ¿Por qué?: para entrenar el modelo de machine learning ¿Quién se beneficia?: El usuario final
Validación	Comprobar que no hay celdas con valores nulos o vacías El número de filas debe coincidir con el número de repositorios extraídos

Cuadro B.17: Historia de usuario 17

Historia de usuario-18	Crear el modelo
Prioridad	5
Tiempo estimado	20
Descripción	¿Qué?: Crear el modelo de machine learning ¿Por qué?: poder extraer repositorios más relevantes eficientemente ¿Quién se beneficia?: El usuario final
Validación	El fichero resultante debe contener los 10 mejores resultados de PageRank

Cuadro B.18: Historia de usuario 18

Historia de usuario-19	Integrar modelo
Prioridad	5
Tiempo estimado	12
Descripción	¿Qué?: Integrar modelo XGBoost en la aplicación ¿Por qué?: para poder aplicar el modelo a la comunidad que se esté analizando ¿Quién se beneficia?: El usuario final
Validación	Debe de generarse un fichero en la carpta *xgb model* con el resultado

Cuadro B.19: Historia de usuario 19

Bibliografía

[1] Anderson, J. (2022). An Intro to Threading in Python. https://realpython.com/intro-to-python-threading/.

[2] cloudfit-public docs (2018). Python asyncio part 2 – awaitables, tasks, and futures. https://bbc.github.io/cloudfit-public-docs/asyncio/asyncio-part-2.html. [Online; accessed: 25-May-2022].

[3] developers, X. (2016a). Additive training. https://xgboost.readthedocs.io/en/stable/tutorials/model.html#additive-training. [Online; accessed: 25-May-2022].

[4] developers, X. (2016b). Learn the tree structure. https://xgboost.readthedocs.io/en/stable/tutorials/model.html#learn-the-tree-structure. [Online; accessed: 25-May-2022].

[5] developers, X. (2016c). Model and parameters. https://xgboost.readthedocs.io/en/stable/tutorials/model.html#model-and-parameters. [Online; accessed: 25-May-2022].

[6] developers, X. (2016d). Objective function: Training loss + regularization. https://xgboost.readthedocs.io/en/stable/tutorials/model.html#objective-function-training-loss-regularization. [Online; accessed: 25-May-2022].

[7] developers, X. (2021). Xgboost. https://xgboost.readthedocs.io/en/stable/. [Online; accessed: 25-May-2022].

[8] Du, H. (2016). Definition of model complexity in xgboost. https://stats.stackexchange.com/questions/229557/definition-of-model-complexity-in-xgboost. [Online; accessed: 25-May-2022].

[9] Foundation, P. S. (2001). argparse. https://docs.python.org/3/library/argparse.html. [Online; accessed: 25-May-2022].

[10] Hammersley, J. and Lees-Miller, J. (2013). Overleaf. https://en.wikipedia.org/wiki/Overleaf. [Online; accessed: 25-May-2022].

[11] Hat, R. (2018). ¿qué es docker? https://www.redhat.com/es/topics/containers/what-is-docker. [Online; accessed: 25-May-2022].

[12] holger krekel and pytest-dev team. (2015). Pytest. https://docs.pytest.org/en/7.1.x/. [Online; accessed: 25-May-2022].

[13] Joseph (2020). Que es SCRUM. https://inleggo.training/2020/06/que-es-scrum/.

[14] Korvyakov, A. (2019). How to structure machine learning work effectively. https://www.tomtom.com/blog/automated-driving/structuring-machine-learning/.

[15] Ma, Y., Li, H., Hu, J., Xie, R., and Chen, Y. (2017). Mining the network of the programmers: A data-driven analysis of github. https://www.researchgate.net/publication/319602891_Mining_the_Network_of_the_Programmers_A_Data-Driven_Analysis_of_GitHub. [Online; accessed: 25-May-2022].

[16] Manager, S. (2012). Artefactos. https://www.scrummanager.net/bok/index.php?title=Artefactos.

[17] Manager, S. (2021). Roles. https://www.scrummanager.net/bok/index.php?title=Roles.

[18] Menzinsky, A., López, G., Palacio, J., Ángel Sobrino, M., Álvarez, R., and Rivas., V. (2020). Historias de Usuario. https://scrummanager.net/files/scrum_manager_historias_usuario.pdf.

[19] Park, S., Lee, W., Choe, B., , and goo Lee, S. (2017). Random walks. https://www.researchgate.net/publication/337177619_A_Survey_on_Personalized_PageRank_Computation_Algorithms/fulltext/5dca00b892851c818046d84a/A-Survey-on-Personalized-PageRank-Computation-Algorithms.pdf?origin=publication_detail. [Online; accessed: 25-May-2022].

[20] Peixoto, T. P. (2016a). graph-tool. efficient network analysis. https://graph-tool.skewed.de/. [Online; accessed: 25-May-2022].

[21] Peixoto, T. P. (2016b). graph-tool. efficient network analysis. https://graph-tool.skewed.de/. [Online; accessed: 25-May-2022].

[22] Pérez, A. (2021). Las 5 etapas en los "sprints" de un desarrollo scrum. https://www.obsbusiness.school/blog/las-5-etapas-en-los-sprints-de-un-desarrollo-scrum.

[23] Schmidt, E. R. (2018). asyncio — e/s asíncrona, bucle de eventos y herramientas de concurrencia. https://rico-schmidt.name/pymotw-3/asyncio/index.html. [Online; accessed: 25-May-2022].

[24] Schwaber, K. and Sutherland, J. (2017). La guía de scrum. la guía definitiva de scrum: las reglas del juego. https://scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-Spanish-European.pdf. [Online; accessed: 25-May-2022].

[25] Scikit-learn (2016). Cross-validation: evaluating estimator performance. https://scikit-learn.org/stable/modules/cross_validation.html. [Online; accessed: 25-May-2022].

[26] Scrum Manager (2014). Tableros kanban: conceptos. https://www.scrummanager.net/bok/index.php?title=Tableros_kanban:_conceptos. [Online; accessed: 25-May-2022].

[27] Sentrio (2021). Qué es sonarqube: Verifica y analiza la calidad de tu código. https://sentrio.io/blog/que-es-sonarqube/. [Online; accessed: 25-May-2022].

[28] Slenders, J. (2014). Python prompt toolkit. https://python-prompt-toolkit.readthedocs.io/en/stable/. [Online; accessed: 25-May-2022].

[29] team, T. M. D. (2003). Matplotlib: Visualization with python. https://matplotlib.org/. [Online; accessed: 25-May-2022].

[30] Thung, F., Tegawend, nad David Lo1, F. B., and Jiang, L. (2013). Network structure of social coding in github. https://ieeexplore.ieee.org/abstract/document/6498480. [Online; accessed: 25-May-2022].

[31] Wikipedia (2013). Scikit-learn. https://es.wikipedia.org/wiki/Scikit-learn. [Online; accessed: 25-May-2022].

[32] Wikipedia (2016a). Python. https://es.wikipedia.org/wiki/Python. [Online; accessed: 25-May-2022].

[33] Wikipedia (2016b). Visual studio code. https://es.wikipedia.org/wiki/Visual_Studio_Code. [Online; accessed: 25-May-2022].

[34] Wikipedia (2022). Overfitting. https://en.wikipedia.org/wiki/Overfitting. [Online; accessed: 25-May-2022].

[35] Xia, F., Member, S., Liu, J., Nie, H., Fu, Y., Wan, L., and Kong, X. (2017). Random walks. https://arxiv.org/pdf/2008.03639.pdf. [Online; accessed: 25-May-2022].