Universidad
Rey Juan Carlos

ESCUELA DE INGENIERÍA DE FUENLABRADA

GRADO EN INGENIERIA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

EVOLUCIÓN DE PROYECTOS FOSS EN GITHUB

Autor : Paula Sepúlveda Membrilla

Tutor : Dr. Gregorio Robles Martínez

Curso académico 2023/2024

Trabajo Fin de Grado

Evolución de proyectos FOSS en GitHub

Autor : Paula Sepúlveda Membrilla

Tutor : Dr. Gregorio Robles Martínez

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 202X, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 202X

Dedicado a

mi familia / mi abuelo / mi abuela

II

Agradecimientos

Aquí vienen los agradecimientos... Aunque está bien acordarse de la pareja, no hay que olvidarse de dar las gracias a tu madre, que aunque a veces no lo parezca disfrutará tanto de tus logros como tú... Además, la pareja quizás no sea para siempre, pero tu madre sí.

Resumen

Aquí viene un resumen del proyecto. Ha de constar de tres o cuatro párrafos, donde se presente de manera clara y concisa de qué va el proyecto. Han de quedar respondidas las siguientes preguntas:

- ¿De qué va este proyecto? ¿Cuál es su objetivo principal?
- ¿Cómo se ha realizado? ¿Qué tecnologías están involucradas?
- ¿En qué contexto se ha realizado el proyecto? ¿Es un proyecto dentro de un marco general?

Lo mejor es escribir el resumen al final.

Summary

Here comes a translation of the “Resumen” into English. Please, double check it for correct grammar and spelling. As it is the translation of the “Resumen”, which is supposed to be written at the end, this as well should be filled out just before submitting.

Índice general

Índice de figuras

Capítulo 1

Introducción

En este capítulo se presenta la motivación que ha impulsado esta investigación, proporcionando el contexto y las razones fundamentales para llevarla a cabo. A continuación, se define claramente el problema a resolver, especificando su alcance y relevancia, así como los objetivos que se desean alcanzar y las hipótesis planteadas al inicio del estudio.
Finalmente, se describe una visión general de cómo está organizado el documento y qué secciones lo componen.

1.1. Motivación

La inspiración que me ha llevado a realizar este proyecto proviene de mi profesor y tutor, Gregorio Robles, quién me ha motivado a adentrarme en el mundo del software libre (OSS, del inglés *Open Source Software*).
El término software libre hace referencia al código diseñado de manera descentralizada y colaborativa, siendo accesible para todo el público. Esto permite que cualquiera pueda utilizarlo, examinarlo, modificarlo y redistribuirlo como considere conveniente.
La elección de código abierto nos aporta beneficios tales como un bajo coste, gran flexibilidad para modificar el código fuente, así como un buen soporte proporcionado por parte de la comunidad. Sin embargo, no existe una clara distinción entre el proceso de desarrollo y el de mantenimiento, lo cual resulta interesante para la Ingeniería de Software al estudiar y explicar el funcionamiento de las interacciones que se llevan a cabo abiertamente.
Por lo tanto, la principal motivación de esta investigación es analizar el comportamiento de

los proyectos de código abierto, los cuales se desarrollan en una comunidad de usuarios que se comunican a través de distintas herramientas, teniendo como factor común Internet, lo que permite guardar el registro de las actividades realizadas a lo largo del tiempo.
Además, este proyecto me ha brindado la oportunidad de adquirir conocimientos sobre minería de datos y desarrollar habilidades para realizar un correcto análisis de los datos obtenidos.

1.2. Definición del problema

El software libre permite a los desarrolladores realizar un completo análisis cuantitativo del código y de todos los parámetros involucrados en su producción, debido a que están disponibles públicamente. Al contar con datos de acceso público del desarrollo de software, permite llevar a cabo estudios estadísticos para estudiar la evolución del software.
En todo este proceso, tiene gran importancia la utilización de un sistema de control de versiones, donde se permita rastrear los cambios realizados en el pasado. De esta manera, podemos obtener un análisis de la estructura histórica del código en base a sus distintas contribuciones, así como harían los arqueólogos estudiando una ciudad en base a sus distintas construcciones. Para software libre esto tiene vital importancia, ya que debido a la falta de diseño, el mantenimiento del código es una actividad relevante para evitar posible código obsoleto de desarrolladores que ya no participan en el proyecto.
Aunque la Ingeniería de Software es una disciplina que se ha consolidado a través de los años, actualmente existen muy pocos análisis empíricos sobre la arqueología del software. La falta de estudios junto con el atractivo hacia el desarrollo comunitario del software libre, así como la importancia para la Ingeniería de Software de modelar con datos empíricos, nos impulsa a llevar a cabo este proyecto.

1.3. Objetivos e hipótesis

Nuestra hipótesis es que, mediante un estudio empírico de las prácticas de desarrollo de software, la aplicación de minería de datos al proyecto, el análisis de su evolución a lo largo del tiempo y la correlación de datos, se pueden determinar y aplicar métricas que demuestren que la longevidad y el mantenimiento del código son heterogéneos y varían entre los diferentes

componentes de varios proyectos.

Por lo que, el objetivo del proyecto consiste en estudiar la evolución y propiedades de varios proyectos de software libre a través del estudio empírico, y en aplicar un análisis estadístico de su código fuente. Se estudia el comportamiento de los proyectos FOSS[1] más relevantes en GitHub, se identifican las fuentes de datos que ofrecen de manera pública y se presenta una metodología para el análisis de los datos extraidos, que nos aportan datos interesantes a la investigación basados en la cantidad de colaboradores o en los lenguajes de programación más utilizados. De esta manera, podemos conocer mejor el fenómeno del software libre, el proceso de creación de software y cómo se puede aplicar en cualquier otro entorno de desarrollo, se describe la mecánica de desarrollo de los movimientos de software libre.

Tenemos el objetivo de comprender el desarrollo de proyectos de software libre, así como la posibilidad de renovar la mecánica llevada a cabo mediante la aplicación de los resultados obtenidos en esta investigación.

1.4. Estructura de la memoria

A continuacion, describimos la estructura de la memoria, exponiendo el contenido de cada uno de los capítulos, proporcionando así una guía organizada del trabajo de fin de grado para una mejor lectura y comprension de este.

- **Capítulo 1: Introducción**

  En este capítulo se describe la motivación que me ha llevado a realizar la investigación, así como el contexto, la definición del problema y las distintas hipótesis que se han desarrollado. Además, se ha ofrecido una breve descripción de la estructura del proyecto.

- **Capítulo 2: Objetivos**

  En este capítulo se describe el objetivo principal de la investigación, así como los objetivos específicos necesarios que van a guiar nuestro proyecto. Incluye una planificacion temporal de los objetivos mencionados.

- **Capítulo 3: Estado del arte**

  En este capítulo se proporciona información detallada sobre el diseño, las características

[1] Free/Open Source Software

y los usos de cada una de las tecnologías y herramientas utilizadas en el proyecto.

- **Capítulo 4: Diseño e implementación**

  En este capítulo se detallan las fases que se han llevado a cabo para realizar el análisis de estudio. Se describe la arquitectura general del proyecto, la mecánica de obtención y almacenamiento de los datos, y su posterior análisis empírico.

- **Capítulo 5: Resultados**

  En este capítulo se describen los resultados obtenidos tras analizar los datos recopilados en el análisis, y se realiza una justificación de dichos resultados.

- **Capítulo 6: Conclusiones**

  En este capítulo se explican las conclusiones a las que se ha llegado, con su correspondiente justificación. Además, se mencionan los conocimientos adquiridos durante la duración de todo el grado universitario que se han aplicado en este proyecto, y los relacionados con el aprendizaje adquirido durante la realización de este estudio. Por último, se mencionan posibles mejoras que se pueden aplicar en el futuro.

Capítulo 2

Objetivos

En este capítulo, se definen los objetivos que guiarán la realización de este Trabajo de Fin de Grado (TFG). Establecer objetivos claros y bien definidos es fundamental para asegurar una dirección precisa y coherente a lo largo del desarrollo del proyecto.
Estos objetivos se dividen en dos categorías principales: el objetivo general, que representa la meta global del trabajo, y los objetivos específicos, que son metas más concretas y detalladas necesarias para alcanzar el objetivo general.
Además, se presenta una planificación temporal que muestra la organización de las tareas a lo largo del período de ejecución del proyecto, garantizando un avance ordenado y sistemático.

2.1. Objetivo general

El objetivo general del proyecto es investigar en profundidad la evolución y las características de proyectos de software libre alojados en la plataforma de GitHub mediante un estudio empírico detallado y la aplicación estadística de su código fuente. A través de este enfoque, se pretende obtener una comprensión íntegra de cómo el proyecto ha cambiado y se ha desarrollado a lo largo del tiempo, identificando patrones, tendencias y características del código que puedan influir en su calidad, mantenibilidad y eficiencia.
Esta investigación no solo implica un análisis histórico, sino que también incluye la aplicacion de diversas métricas para evaluar distintos aspectos del código fuente. Al combinar métodos empíricos con análisis estadísticos robustos, este proyecto busca proporcionar una visión detallada y objetiva del comportamiento evolutivo del software.

Los resultados obtenidos nos ofrecen conclusiones y recomendaciones para la mejora continua del desarrollo del software libre, contribuyendo al conocimiento en el campo de Ingeniería de Software que puede aplicarse en la toma de decisiones en proyectos futuros.

2.2. Objetivos específicos

Para lograr el objetivo general detallado en el proyecto se han establecido los siguientes objetivos específicos:

- **Definición del problema y objetivos**

  Consiste en buscar y evaluar información sobre el contexto de trabajo, realizando una estimación de los costes de desarrollo software. Se piensa como desarrollar la programación de manera eficiente, y a su vez se realiza una selección de distintos repositorios de GitHub que resultan de especial interés para nuestra investigación.

- **Programación**

  En esta fase del proyecto se desarrollan los scripts necesarios para extraer, almacenar y analizar los datos con el fin de conseguir los objetivos de esta investigación.

- **Recopilación de datos**

  En esta etapa se recopilan y almacenan los datos más relevantes para el posterior estudio del comportamiento de los repositorios seleccionados. También es importante realizar un preprocesado de los datos, lo que implica una normalización o estandarización de variables, así como una limpieza y transformación de datos según sea necesario.

- **Análisis estadístico**

  Tras la recopilación de los datos, se ajustan y configuran las métricas correspondientes para obtener distintas estadísticas sobre la evolución de los repositorios a lo largo del tiempo.

- **Evaluación e interpretación de los resultados**

  Una vez que se ha realizado un análisis estadístico, se realiza un análisis empírico y se representan de manera gráfica los resultados obtenidos. Seguidamente, se interpretan los resultados obtenidos y se derivan conclusiones.

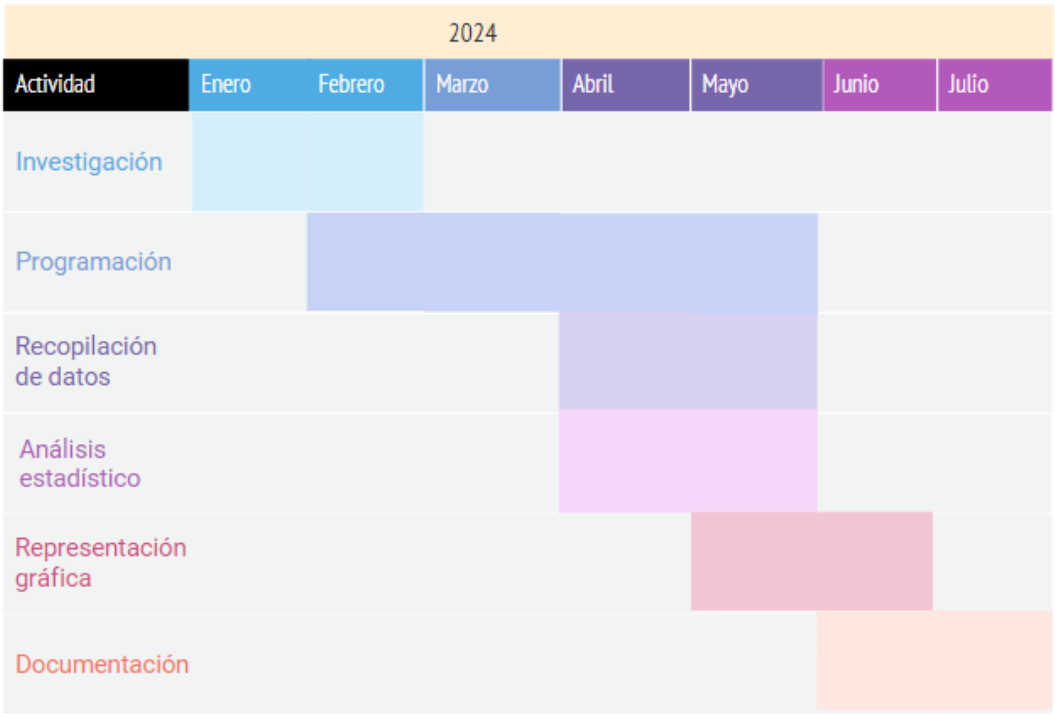

Figura 2.1: Diagrama de Gantt

- **Documentación**

  En esta etapa se elaboran la documentación y los informes necesarios para la correcta entrega del Trabajo de Fin de Grado (TFG).

2.3. Planificación temporal

El desarrollo de este proyecto abarca un período de siete meses, desde enero de 2024 hasta julio de 2024, con la realización de un trabajo diario que se lleva a cabo de manera constante. En la figura 2.1 se observa el tiempo empleado en las diferentes fases del proyecto, donde se puede observar que varias tareas se han solapado en algunos casos debido al avance en paralelo de distintas actividades.

- **Enero**

  En este mes comienzo el desarrollo de esta investigación, cuando mi tutor Gregorio Robles me sugirió la idea del proyecto. Comienzan las reuniones donde se definen los objetivos correspondientes, y empiezo a documentarme para obtener los conocimientos necesarios para la realización del proyecto.

Recopilo información acerca de los proyectos de código abierto y posibles repositorios de especial interés que son alojados en la plataforma de GitHub. A su vez, estudio la información que se puede obtener de las líneas de código mediante el comando *git blame*, y que análisis nos puede resultar de interés para estudiar la evolución de estos repositorios a lo largo de los años.

- **Febrero**

  Durante este mes sigo realizando la tarea de investigación, pero además trabajo en paralelo con la tarea de programación. Comienzo a desarrollar el código de programación basado en el comando *git blame*, mediante el cual obtengo información interesante acerca de los cambios realizados en un archivo específico línea por línea. El comando *git blame* nos proporciona información del autor de la línea de código, así como de la fecha del commit, del estado de esa línea y obviamente podemos ver también su contenido.

- **Marzo**

  En este mes me centro exclusivamente en la programación del código, desarrollando los scripts *init.py* y *git-blameall.py*. Estos scripts analizan un repositorio y un archivo respectivamente, con el objetivo de analizar línea a línea cada uno de los archivos de un repositorio, y guardan los datos recogidos en un archivo JSON.

  Al principio, se trabaja en realizar el análisis para un único archivo y finalmente se logra conseguir analizar un repositorio concreto. Además, se estudia la opción de tener que analizar las distintas versiones de un archivo o unicamente la versión final, quedandonos con la segunda opción al ser mucho más eficiente y conseguir obtener los datos que nos interesan.

- **Abril**

  Este mes tiene una gran carga de trabajo, al realizar en paralelo tres actividades distintas. Por una parte, sigo avanzando con la programación, mejorando los scripts *init.py* y *git-blameall.py* de manera que los datos pasan de guardarse en un archivo JSON a guardarse en tres tablas relacionadas entre sí: '*Repositories*', '*Files*' y '*Code*' respectivamente. Esto requiere de un estudio previo para obtener los conocimientos necesarios sobre la utilización de *MySQL* y el concepto de bases de datos relacionales. El hecho de almacenar los datos en bases de datos relacionales nos permite un posterior análisis de los datos de

manera mucho más eficiente.

- **Mayo**

  Durante estas semanas la carga de trabajo sigue siendo muy alta, desarrollando cuatro actividades de manera paralela. Termino de desarrollar los archivos de programación, incluido entre ellos el script *graphics.py* que tiene como finalidad poder obtener una representación gráfica de las métricas estudiadas. A su vez, termino de definir los índices y las métricas que se incluyen en este proyecto, para obtener estadísticas interesantes acerca de la evolución de cambios, número de autores o posible código obsoleto en un repositorio. Seguidamente, se recopilan los últimos datos de los repositorios elegidos para su análisis, todos ellos alojados en la plataforma de GitHub.

- **Junio**

  En esta etapa se elaboran las gráficas definitivas, y mientras tanto se empieza a escribir la memoria de nuestro proyecto.

- **Julio**

  En este último mes se termina de escribir la memoria y se realizan las correcciones oportunas por el tutor para poder realizar una correcta entrega del Trabajo de Fin de Grado (TFG).

Capítulo 3

Estado del arte

En este capítulo, se presentan las herramientas y librerías usadas en el Trabajo Fin de Grado. Esta exposicion nos da una visión de las tecnologías empleadas en el proyecto.

3.1. Tecnologías y herramientas

3.1.1. Python

Python[1] es un lenguaje de programación de alto nivel desarrollado por Guido Van Rossum a principios de 1989 en los Países Bajos. Se trata de un lenguaje ejecutado directamente por un intérprete, que no requiere de una compilación previa, lo que facilita la detección y el manejo de errores. Es un lenguaje con una sintaxis clara y sencilla, por lo que resulta bastante atractivo para los desarrolladores por su fácil lectura, escritura y comprensión. Además, Python se trata de un lenguaje multiplataforma, esto quiere decir que el mismo código puede utilizarse en distintos sistemas operativos al ser un lenguaje de código abierto, lo que proporciona bastante versatilidad a los desarrolladores.

Incluye una gran cantidad de bibliotecas que proporcionan códigos para la visualización de datos con distintos gráficos, la creación de matrices o el procesamiento de imágenes. Esto permite su amplia utilización en ámbitos muy distintos como las aplicaciones web, el desarrollo de software, la ciencia de datos o el machine learning (ML).

Ha experimentado un crecimiento significativo a lo largo de los años, pasando por distintas

[1] `https://www.python.org/`

versiones, convirtiendose en uno de los lenguajes más populares y utilizados en la actualidad.

3.1.2. GitHub

GitHub[2] es una plataforma de alojamiento de repositorios de código fuente que utiliza *Git* como sistema de control de versiones. Permite almacenar código y archivos en un servicio de la nube, de manera que los desarrolladores puedan colaborar en proyectos compartidos manteniendo un seguimiento de la evolución del proyecto.
Linus Torvalds, un programador finlandés con gran importancia dentro del software libre, creó en 2005 su propio sistema de control de versiones llamado *Git* para ser utilizado en proyectos comerciales y de software libre. Posteriormente, en 2008, varios desarrolladores fundaron la plataforma GitHub, ofreciendo una interfaz fácil de utilizar que ha contribuido a la popularización de *Git*.
Es un sistema de control de versiones eficiente, fiable y compatible que se ha convertido en el estándar por excelencia para el desarrollo de software.

3.1.3. MySQL

MySQL[3] es un sistema de gestión de bases de datos considerado como uno de los más populares junto a Oracle y Microsoft SQL Server, sobre todo para entornos de desarrollo web. Puede utilizarse en diferentes sistemas operativos con múltiples motores de almacenamiento para adaptarse a las necesidades de cada entorno. Sus puntos fuertes son la rapidez y la seguridad, ya que utiliza un sistema de contraseñas que permite la verificación basada en host.
Uno de sus grandes beneficios es que cuenta con una gran comunidad con la que intercambiar dudas y conocimientos. Además, es escalable y fácil de aprender por lo que se convierte en una de las bases de datos más utilizadas en la actualidad.

3.1.4. LaTeX

LaTex[4] es un sistema de composicion de textos o documentos formado por una colección de macros *Tex*. Fue desarrollado por Leslie Lamport en 1984, y en la actualidad se utiliza para

[2] `https://github.com/`
[3] `https://mysql.com`
[4] `https://es.overleaf.com/`

la generación de artículos y libros científicos que incluyen, entre otros elementos, expresiones matemáticas. Se utiliza para la composicion de tesis y libros técnicos, dado que la calidad tipográfica de los documentos realizados en LaTex se considera adecuadas a las necesidades de una editorial científica de primera línea, muchas de las cuales ya lo emplean.
Text es una mezcla entre procesador de textos y lenguaje de programación utilizado fundamentalmente para escribir documentos de contenido científico y de gran calidad de impresión. Fue desarrollado por Donald E. Knuth en 1978, y actualmente hay implementaciones para todo tipo de ordenadores. Es un sistema de tipografía muy popular en el entorno académico, especialmente entre las comunidades de matemáticos, físicos e informáticos.

3.2. Librerías

3.2.1. Pyodbc

Pyodbc[5] es una biblioteca de Python que nos sirve para tener la integración de la comunicación con bases de datos de una manera sencilla. En el proyecto se ha utilizado para conectar nuestros scripts de Python con la base de datos alojada en MySQL.
Su funcionamiento consiste en conectarse a una base de datos mediante el comando *connect()*, que nos devolverá una conexión. Una vez que tengamos la conexión, se crea un cursor mediante la función *cursor()*, con el cual podemos ejecutar *querys()* para trabajar con los datos obtenidos. Una vez que hayamos realizados las consultas oportunas, no hay que olvidarse de cerrar la conexión con la base de datos.

3.2.2. Subprocess

La biblioteca *subprocess*[6] permite ejecutar nuevos programas o comandos que se encuentran dentro de un script de Python a la vez que ejecutamos dicho script, es decir, ejecuta procesos en segundo plano. Una de las capacidades más útiles consiste en que permite al usuario controlar las entradas, salidas e incluso los errores que genera el proceso hijo desde dentro del código Python. Este modulo facilita la automatización de tareas y la integración de otros programas

[5] `https://pypi.org/project/pyodbc/`
[6] `https://docs.python.org/es/3/library/subprocess.html`

con el código de Python.

3.2.3. Os

La librería *os*[7] permite usar funcionalidades dependientes del sistema operativo, como indica su nombre, por lo que es una biblioteca de gran tamaño y tiene muchos métodos. Entre las funcionalidades más útiles se encuentran la de listar, crear y/o eliminar archivos y directorios, obtener el nombre de un archivo así como su directorio, obtener la extensión y tamaño de un archivo, así como obtener las fechas de creación, modificación y/o de acceso de un archivo. En general, esta librería facilita el trabajo con archivos y directorios independientemente de la plataforma, lo cual es bastante importante ya que debemos de tener en cuenta que los posibles usuarios de nuestro programa pueden tener distintos sistemas operativos.

3.2.4. Matplotlib

Matplotlib[8] es una librería muy completa de código abierto que se utiliza para crear visualizaciones estáticas, animadas e interactivas con Python. Ha sido desarrollada por John Hunter en 2002, con el objetivo inicial de visualizar las señales eléctricas del cerebro de personas epilépticas. Tras el fallecimiento de John Hunter, *matplotlib* se ha ido mejorando a lo largo del tiempo por numerosos contribuidores de la comunidad de software libre.
Se trata de una herramienta muy completa, que permite generar visualizaciones de datos muy detalladas. Es posible crear trazados, histogramas, diagramas de barra y cualquier tipo de gráfica para visualizar análisis estadísticos.
El módulo *Pyplot*[9] propone varias funciones sencillas para añadir elementos tales como líneas, imágenes o textos a los ejes de un gráfico. Su interfaz es muy intuitiva, lo que permite a los usuarios diseñar gráficos completamente personalizables con facilidad.

3.2.5. Pandas

Pandas es una librería de Python especializada en el manejo y análisis de estructuras de datos. Define nuevas estructuras de datos basadas en los arrays de la librería *NumPy* pero con

[7] `https://docs.python.org/es/3.10/library/os.html`
[8] `https://matplotlib.org/`
[9] `https://matplotlib.org/stable/tutorials/introductory/pyplot.html`

nuevas funcionalidades, nos permite leer y escribir fácilmente ficheros en bases de datos SQL y nos permite acceder a los datos mediante índices o nombres para filas y columnas. Nos permite trabajar con tres estructuras de datos diferentes: estructura de una dimensión denominadas *series*, estructura de dos dimensiones o tablas denominadas *DataFrame* y estructura de tres dimensiones o cubos llamado *panel*.
El nombre ≪*Pandas*≫ es en realidad una contracción del término ≪*Panel Data*≫ para series de datos que incluyen observaciones a lo largo de varios periodos de tiempo. Se creó como herramienta de alto nivel para el análisis en Python, y tiene la finalidad de evolucionar hasta convertirse en la biblioteca de manipulación de datos de código abierto más potente y flexible.

3.2.6. Chardet

Chardet consiste en una adaptación para Python del detector de codificación de caracteres universal C++ de Mozilla. La detección de la codificación de caracteres es en realidad una detección del lenguaje con dificultades, esto es especialmente útil cuando se trabaja con archivos de origen desconocido o múltiples fuentes que pueden tener diferentes codificaciones.
La forma más sencilla de utilizar esta librería es mediante la funcion *detect()*. Esta función toma un argumento de datos binarios y devuelve un diccionario que contiene la codificación de caracteres detectada junto con el nivel de confianza de la detección. En el caso de utilizar esta librería con archivos que contienen una gran cantidad de texto, lo recomendable es leer solo una parte del archivo y que se detenga tan pronto como tenga la confianza suficiente para determinar su codificación. Esta biblioteca es útil en situaciones donde se reciben archivos históricos de múltiples fuentes con codificaciones no uniformes, como es el caso de nuestro proyecto.

3.2.7. Pygments

Pygments consiste en una librería de resaltado de sintaxis escrita en Python. Es una herramienta poderosa y flexible para resaltar la sintaxis de código fuente en múltiples lenguajes de programación y formatos de salida, permite una personalización significativa para adaptarse a diferentes necesidades y preferencias.
De entre todas las funciones y herramientas que nos ofrece esta biblioteca, en nuestro proyecto se han utilizado la función *get_lexer_for_filename* y la clase *class pygment.lexer.name* con la

finalidad de adivinar el nombre del lexer, basado en el nombre y en el contenido del archivo. El *lexer*, o también denominado *analizador léxico* o *tokenizer*, es un componente que convierte una secuencia de caracteres en una secuencia de tokens, que se corresponden con unidades léxicas que representan estructuras significativas en el lenguaje de programación. Nos permite identificar y clasificar las diferentes partes del código fuente para poder aplicar resaltado de sintaxis adecuado o para poder identificar el lenguaje de programación.

Capítulo 4

Diseño e implementación

A continuación, se proporciona una visión detallada del desarrollo de este proyecto, destacando los aspectos tanto técnicos como metodológicos que lo forman. Se describen en detalle las fases del proyecto, así como las métricas escogidas con su debida justificación.
Exponer el diseño e implementación de nuestro proyecto permite a los lectores entender cómo se ha desarrollado la investigación, además de proporcionarles la capacidad de contribuir o ampliar el proyecto.

4.1. Arquitectura general

En la figura 4.1 se puede observar la arquitectura general del proyecto, que se compone de varias etapas interconectadas que permiten llevar a cabo la ejecución eficiente del mismo.

En la figura 4.2 se presenta un diagrama de funcionamiento que nos permite comprender el desarrollo de la programación.
Este diagrama consta del programa principal `init.py`, cuya función principal es clonar un

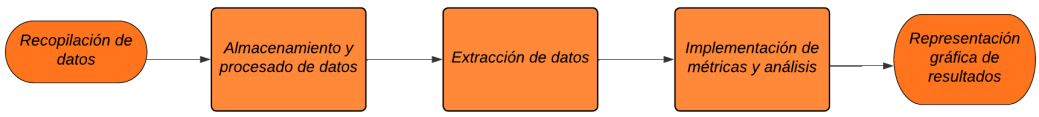

Figura 4.1: Diagrama arquitectura general

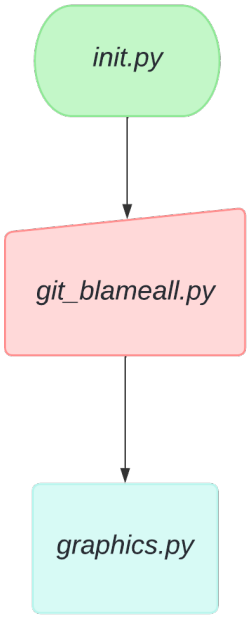

Figura 4.2: Diagrama de funcionamiento

repositorio de GitHub en nuestro directorio y poder recorrer cada uno de sus archivos y subdirectorios respectivamente.

Tras esto, se ejecuta el script `git_blameall.py` con la finalidad de analizar línea a línea el código de cada archivo, recopilando diversos datos que se almacenan en varias tablas de la base de datos.

Finalmente, se ejecuta el script `graphics.py` con el objetivo de generar distintas gráficas que muestran los resultados de los análisis estadísticos y la aplicación de las métricas estudiadas previamente.

4.2. Arquitectura específica

En esta sección, vamos a centrarnos en describir más a fondo el funcionamiento de cada uno de los scripts que conforman nuestro proyecto.

4.2.1. Programa inicial

El programa inicial se corresponde con el script `init.py`, ya que como indica el propio nombre, consiste en el script que inicia nuestro programa mediante el comando *python3 init.py repo-url 'name_urlclone'*. Se ejecuta a partir de la URL de un repositorio de GitHub, y entre sus diversas tareas esta la de ejecutar la función principal del programa `git_blameall.py`, que como se indica posteriormente se trata del programa principal. Sus funciones de manejo son las siguientes:

- `request_url()`. Al ejecutar el programa se pide introducir la URL de un repositorio de GitHub, a partir de la cual podemos obtener distintos parámetros de interés para nuestro proyecto. Se obtienen los valores del protocolo y procedencia del repositorio, de manera que mediante la función `check_url()` se comprueba que se utilice un protocolo 'https' y que el repositorio elegido sea propio de GitHub. Estos parámetros, junto con el usuario y el nombre del repositorio, nos permiten clonar el repositorio en nuestro directorio mediante la librería *subprocess* en la función `run_url()`. Esta función además, ejecuta las las funciones `get_directory()` y esta a su vez la función `get_path()`, que como el propio nombre indica obtienen el nombre del directorio y la ruta absoluta del directorio en cuestión. Se ha utilizado una ruta absoluta frente a una relativa porque nos aporta claridad para evitar conflictos entre directorios ambiguos dentro del mismo repositorio, además de proporcionarnos independencia frente al lugar en el que se ejecute el script.

- `remove_excluded()`. La finalidad de esta función consiste en clasificar archivos o directorios que no pueden ser soportados por el comando *git blame* para ser posteriormente eliminados del repositorio clonado, por lo que se descartan los archivos que tienen las siguientes extensiones [".png", ".jpg", ".jpeg", ".gif", ".pdf", ".exe", ".dll", ".ico", ".db", ".dat", ".class", ".o", ".pyc", ".mp3", ".mp4", ".wav", ".avi", ".bak", ".tmp", ".docx", ".pptx", ".zip", ".tar.gz", ".rar"] así como los siguientes directorios ["node_modules", "vendor", "Pods"].

  Partiendo de las listas mencionadas anteriormente, se construye una función booleana para clasificar cada archivo o directorio a ser excluido del análisis, mediante el comando `file_path.endwith()` se encuentran los archivos con la extensión correspon-

diente mientras que con la ayuda del separador de directorios específico de Windows `file_path.split(os.path.sep)` obtenemos los posibles directorios a descartar. Una vez que tenemos los archivos y/o directorios que tienen valor 'True', se realiza un recorrido ascendente por el árbol del repositorio clonado con la ayuda de `os.walk()` para eliminar cada objeto respectivamente. Al ejecutar el programa, se observan por pantalla los objetos que se han eliminado así como la cantidad de objetos, esto ayuda a comprobar que se realiza un análisis de la cantidad correcta de archivos y/o directorios.

- `remove_empty_files()`. Esta función tiene como finalidad eliminar archivos vacíos que se puedan encontrar en el directorio que se corresponde con el repositorio clonado, así como en todos sus subdirectorios. Toma como parámetro dicho directorio, y crea una lista inicialmente vacía que se utiliza para almacenar los nombres de los archivos eliminados. Mediante `os.walk()` se realiza un recorrido por todo el árbol del repositorio clonado e itera cada uno de sus archivos, para calcular su tamaño en bytes con el comando `os.path.getsize()`, y en caso de ser 0 se clasifica como un archivo vacío, y por lo tanto, se elimina mediante la función `os.remove()` y se añade a la lista de archivos eliminados. Finalmente, con la información incluida en la lista inicial, obtenemos información acerca de que cantidad de archivos se han eliminado y de cuales son sus rutas y nombres respectivamente.

- `detect_encoding()`. Esta función es una parte muy importante del código, ya que permite solucionar problemas de codificación al enfrentarse a cualquier tipo de archivo de datos. Tiene como objetivo adivinar el tipo de codificación del archivo pasado como parámetro, para ello realiza una lectura en bytes del archivo y mediante la librería *chardet* detecta la codificación del archivo.

- `convert_to_utf8()`. Esta función tiene como finalidad convertir en un archivo codificado en UTF-8 cada archivo que no esté codificado como tal, usando como parámetros el nombre del archivo a codificar y el tipo de codificación de dicho archivo. Se realiza una lectura en la codificación detectada anteriormente del archivo correspondiente, y se almacena su contenido en una variable. Este contenido se escribe en un archivo temporal, para posteriormente mediante la función `os.replace()` reescribir el archivo original con el contenido archivo temporal, realizando así una conversión a la codificación UTF-8

de manera segura.

- `get_table1()`. A continuación, nos centramos en las tareas relacionadas con la base de datos. La primera tarea consiste en verificar la existencia de la base de datos que vamos a utilizar, esto se ha podido comprobar a través de la conexión a una base de datos *'master'*, la cual nos proporciona la información necesaria, y en caso de que la base de datos denominada 'Analysis_Github_Repository' no exista, se procede a crearla. El paso siguiente consiste en crear la tabla denominada '*Repositories*', previa comprobación de su existencia.

  Siguiendo en el ámbito de las bases de datos, utilizando la función `get_table2()` se crea la tabla denominada 'Files' con las verificaciones necesarias y con una conexión previamente establecida a la base de datos 'Analysis_Github_Repository'.

- `read_directory()`. Esta función es la más importante de todo el script, ya que integra y unifica la mayoria de las funciones descritas anteriormente. En primer lugar, toma como parámetro el nombre del directorio que se va a analizar y crea una lista con todos sus archivos y subdirectorios. De esta lista, descarta el directorio '.git' para el análisis pero teniendo en cuenta que no se puede eliminar, al contener información de vital importancia sobre el repositorio clonado, incluyendo su historial de commits, su configuración así como sus objetos. Tras esto, el procedimiento a seguir consiste en recorrer el árbol del repositorio, clasificando cada objeto de la lista según sea un archivo o un subdirectorio.

  En caso de ser un archivo, se construye y ejecuta el comando *'git log'* para obtener el historial de commits del archivo. Si el comando falla, se imprime un mensaje de error y se continúa con el siguiente archivo, pero si el comando se ejecuta con éxito, se cuenta el número de commits del archivo, y en caso de tener un número de commits superior a 0, se ejecuta la función principal del script `git_blameall.py`. Mientras tanto, se ejecutan las funciones relacionadas con la codificación original del archivo, y se convierte a UTF-8 en caso de ser necesario. Así como también se adivina el lenguaje de programación del archivo mediante la librería *Pygments*. En caso de ser un subdirectorio, se muestra su nombre por pantalla y se invoca esta función recursivamente, con el objetivo de analizar todos los archivos del repositorio clonado.

  Mediante una conexión a la base de datos se realiza la inserción de datos obtenidos en es-

ta función. Primero se utiliza la función `insert_repo_data()`, que inserta en la tabla '*Repositories*' el nombre del repositorio clonado así como su índice correspondiente. Posteriormente se ejecuta la función `insert_files_data()` que se encarga de insertar en la tabla 'Files' el nombre, ruta, ID y número de commits de cada archivo respectivamente. También se ejecuta la función `insert_files_language()`, que inserta el lenguaje de cada archivo dentro de la tabla 'Files'.

4.2.2. Programa principal

El programa principal se corresponde con el script `git_blameall.py`, que consiste en el script que analiza cada fichero utilizando distintos comandos de git. Su objetivo principal es obtener datos interesantes del historial de cambios propio de cada archivo, que se utilizarán posteriormente para estudiar y analizar la evolución del repositorio. Sus funciones de manejo son las siguientes:

- `parse_chunk_header()`. En el contexto de Git y las diferencias (*diffs*) entre un archivo, un *chunk* se corresponde con la sección de un archivo que ha cambiado entre dos revisiones, por lo que esta es la forma que tiene Git de mostrar las posibles diferencias. Cada *chunk* contiene una cabecera que indica dónde se encuentra el cambio realizado y cuántas líneas se han añadido y/o eliminado, y también las líneas de código modificadas con prefijos que indican respectivamente si la línea ha sido añadida o eliminada.
  Esta función analiza la cabecera del *chunk* con el objetivo de extraer información sobre las líneas afectadas. Se obtiene una lista con los valores del número de la línea inicial en la versión original del archivo donde comienza el cambio (*'origL'*), el número de líneas eliminadas en la versión original del archivo (*'del_N'*), el número de la línea inicial en la nueva versión del archivo donde comienza el cambio (*'newL'*) y el número de líneas añadidas en la nueva versión del archivo (*'add_N'*). En caso de que no se puedan obtener los valores de (*'del_N'*) o (*'add_N'*), serán considerados '1' por defecto. En conclusión, esta función nos retorna las posiciones y números de líneas afectadas.

- `get_initial_version()`. El objetivo de esta función consiste en obtener el contenido inicial de un archivo en una revisión específica de Git, toma como parámetros el *hash* de la primera revisión del archivo y el nombre del archivo en cuestión. Cuando se

habla de *hash* se refiere al identificador de cada revisión/commit realizada sobre el archivo. Teniendo en cuenta estos parámetros construye el comando *git show* y se ejecuta, obteniendo como salida el contenido inicial del archivo, que almacena línea a línea en la lista *lines*.

- `find_index()`. Esta función tiene como parámetros una lista con todas las líneas del archivo inicial, cada una de ellas con su información correspondiente, además del número de líneas que se deben de avanzar desde la línea de referencia. El procedimiento de la función se basa en recorrer una a una las líneas que permanecen en la actualidad (vivas) hasta que el número de líneas a avanzar sea 0, teniendo en cuenta que se considera una línea viva aquella que no tiene una fecha de fin y que fue introducida en una revisión distinta a la actual. Tiene como objetivo encontrar el índice correcto de la posición donde se debe de insertar o eliminar alguna línea.

- `main()`. Esta es la función más importante de todo el script, donde se integran todas las anteriores. Tiene como objetivo analizar todas las revisiones de un archivo de un repositorio de Git y registrar información sobre los cambios producidos en una base de datos SQL Server. En primer lugar, construye y ejecuta el comando *git rev-list*, obteniendo como salida una variable denominada *hashes* con todas las revisiones que ha sufrido ese archivo y sus respectivos identificadores, ordenadas cronológicamente de la más antigua a la más actual. Seguidamente ejecuta el comando *git log* para obtener las fechas y los autores de cada una de las revisiones, que se guardan en la variable *date_author*. Realiza la comprobación de que el número de revisiones obtenido coincida con la longitud de *date_author*, y crea una lista de objetos *struct* denominada *revs* que contiene la información de cada revisión de la siguiente manera: (hash, fecha, autor). Según el procesamiento de líneas indicado, recorre cada línea de código de la versión inicial del archivo, y se guardan en la lista *ALL_LINES* como objetos de tipo *struct*.

  Una vez que se ha obtenido la versión inicial del archivo, se procesa cada una de sus revisiones desde la más reciente hasta la más antigua. Se utiliza el comando *git diff* para obtener las posibles diferencias entre revisiones, y se procesa cada una de sus líneas resultantes en función del tipo de línea obtenida: si es una cabecera de chunk (’@@’), se extrae la información correspondiente sobre el chunk llamando a la función `parse_chunk_header()`,

y si es una línea eliminada ('-') o añadida ('+'), encuentra el índice correspondiente en *ALL_LINES* usando la función `find_index()`, verifica la modificación en la revisión actual e inserta los cambios en la base de datos. También se insertan en la base de datos las líneas vivas que se encuentran en la versión más actual, es decir, las líneas que no se han modificado nunca desde la versión inicial del archivo.
En conclusión, esta función se centra en guardar las líneas que han sufrido cambios durante las revisiones así como aquellas que no se han cambiado desde la versión inicial.

- `print_so_far()`. La finalidad de esta función es manejar la conexión con la base de datos, verificar y crear la tabla 'Code' en caso de que no exista, e insertar cada una de las líneas mencionadas anteriormente en esta tabla.

4.2.3. Obtención de gráficos

– EXPLICACIÓN GRÁFICOS —

4.3. Fases del proyecto

En esta sección se desglosan las distintas fases del proyecto, describiendo el desarrollo de cada una de ellas.

4.3.1. Recopilación de datos

En esta etapa del proyecto, se tiene como objetivo identificar proyectos de software libre que resulten interesantes para estudiar su evolución. Por lo que se realiza una exhaustiva investigación, revisando gran variedad de artículos e información, para obtener una serie de repositorios que cumplan los siguientes criterios: que sean proyectos FOSS[1], que estén alojados en la plataforma de GitHub y que proporcionen datos que resulten interesantes para analizar su evolución a lo largo de los años.
Tras estudiar las distintas posibilidades, se han elegido los principales repositorios de los proyectos de la tabla X para estudiar su evolución a lo largo del tiempo. En la selección se han considerado factores como el número de colaboradores y/o su variación a lo largo de los años,

[1] Free/Open Source Software

la diversidad de lenguajes de programación, la relevancia del proyecto en la plataforma GitHub y en la actualidad, así como que sean tanto proyectos de nueva creación como proyectos de larga trayectoria.

HACER TABLA DE REPOS*

Una vez elegidos los repositorios a analizar, se lleva a cabo la recopilación de los datos, que se trata de una tarea crucial para cualquier proyecto de análisis o minería de datos. En esta fase, recogemos los siguientes datos para su posterior almacenamiento:

- *Repo_Name*. Como el propio nombre indica, se refiere al nombre de cada repositorio. De manera que se pueda realizar una clasificación de líneas o archivos según pertenezcan a un repositorio concreto.

- *Repo_ID*. Al realizar un análisis de varios repositorios, cada repositorio se identifica con un ID correspondiente. De esta forma, es más sencillo identificar los datos y podemos llevar a cabo la funcionalidad de tabla de datos relacional.

- *File_ID*. Al igual que cada repositorio se identifica con su propio ID, se hará lo mismo con cada archivo. De manera que se puedan clasificar las líneas de código pertenecientes a un archivo concreto.

- *File_Name*. Este valor se corresponde con el nombre del archivo que se va a analizar.

- *File_Path*. Este campo se corresponde con la ruta donde se encuentra el archivo que se está analizando, de manera que se pueda localizar en el repositorio clonado.

- *File_Language*. Este dato se corresponde con el lenguaje del archivo que se está analizando. El lenguaje se ha identificado mediante la librería Pygments, clasificandose como 'Archivo de datos' aquel archivo al que no se le ha podido identificar su lenguaje de programación, y que puede almacenar diversos datos como texto plano, datos binarios, etc.

- *Commits*. Este dato se corresponde con el número de revisiones que se ha realizado sobre el archivo que se está analizando. Cada 'commit' se corresponde con la acción que se realiza para guardar los cambios realizados en los archivos de un repositorio. El conjunto de todos estos 'commits' forman el historial de revisiones de cada repositorio, por lo que

el número de commits o revisiones nos puede indicar la frecuencia de actualizaciones de un proyecto.

- *Code_ID*. Cada línea que se guarda en la base de datos se identifica con su propio ID.
- *Author_Start*. Este valor se corresponde con el autor de la línea de código correspondiente, es decir, aquella persona que desarrolló esta línea por primera vez.
- *Date_Start*. Este dato se corresponde con la fecha de creación de la línea de código correspondiente.
- *Author_End*. Este valor se corresponde con el desarrollador que eliminó la línea de código correspondiente.
- *Date_End*. Este dato se corresponde con la fecha en la que la línea de código ha sido eliminada.
- *Longevity*. Teniendo en cuenta la fecha de creación y la posible fecha de fin de cada línea de código, se realiza un cálculo de los días de vida que tiene la línea de código. El resultado de este cálculo se guarda en esta variable.
- *Comment_Boolean*. Se clasifica como '1' la línea de código que se corresponde con un comentario de línea, teniendo en cuenta los distintos comentarios en los diversos lenguajes de programación. Así como se identifica como '0' en el caso contrario, es decir, en caso de no ser un comentario de línea.
- *Words_Count*. Este dato se corresponde con el cálculo de las palabras que forman cada línea de código.
- *Code*. En esta variable se inserta la línea de código en cuestión que se está analizando.

4.3.2. Almacenamiento de datos

Una vez que se recopilan los datos mencionados en el apartado anterior, la siguiente tarea consiste en su almacenamiento en una base de datos. Este almacenamiento se realiza en una base de datos relacional, que consiste en una colección de información que organiza datos en relaciones predefinidas, en la que los datos se almacenan en una o más tablas de columnas y

filas, lo que facilita su visualización y la comprensión de cómo se relacionan las diferentes estructuras de datos entre sí. Las tablas se relacionan con otras tablas mediante una relación de clave primaria o de clave foránea, estableciendo relaciones de uno a uno, uno a muchos, o muchos a muchos entre tablas. Una clave primaria es una columna o un conjunto de columnas en una tabla cuyos valores identifican de forma exclusiva una fila de la tabla, mientras que los valores de la clave foránea se corresponden con los valores de la clave primaria de otra tabla. Los principales beneficios de este modelo de base de datos son evitar la duplicidad de los registros y garantizar la integridad referencial de los datos, proporcionando así una manera intuitiva de representar los datos y un acceso fácil a datos relacionados. Este modelo de bases de datos garantiza las propiedades *ACID*, que consisten en atomicidad, que garantiza que los datos no se modifican en caso de producirse un error en alguna parte de la transacción; consistencia, que garantiza que los datos cumplan con las restricciones de integridad; aislamiento, permitiendo que las operaciones de realizadas sobre la base de datos sean indendientes y que no afecten las unas a las otras; y durabilidad, que proporciona la persistencia de las operaciones realizadas sobre los datos de manera que todos los cambios sean permanentes.

Las bases de datos relacionales se caracterizan por utilizar el lenguaje de consulta SQL, por lo que también se denominan bases de datos SQL. Mediante este lenguaje se realizan las operaciones básicas de interacción con la base de datos, definiendo a continuación las que han sido utilizadas en este proyecto:

- *CREATE DATABASE*. Se corresponde con la sentencia utilizada para crear la base de datos '*Analysis_GitHub_Repository*'.

- *CREATE TABLE*. Como el propio nombre indica, esta sentencia o query se utiliza para crear una tabla dentro de la base de datos. Se deben de especificar los campos que contenga la tabla, así como sus tipos, sus claves primarias o foráneas, o incluso se pueden restringir algunos campos para que sean válidos o no los valores *NULL*.

- *INSERT INTO*. Esta sentencia se utiliza para insertar valores en una tabla existente en la base de datos. Se deben de especificar los valores a insertar y el propio nombre de la tabla.

- *SELECT*. Esta sentencia aporta diferentes funcionalidades, siendo una de ellas la verificación de la existencia de una tabla de la base de datos, con el objetivo de poder crear la

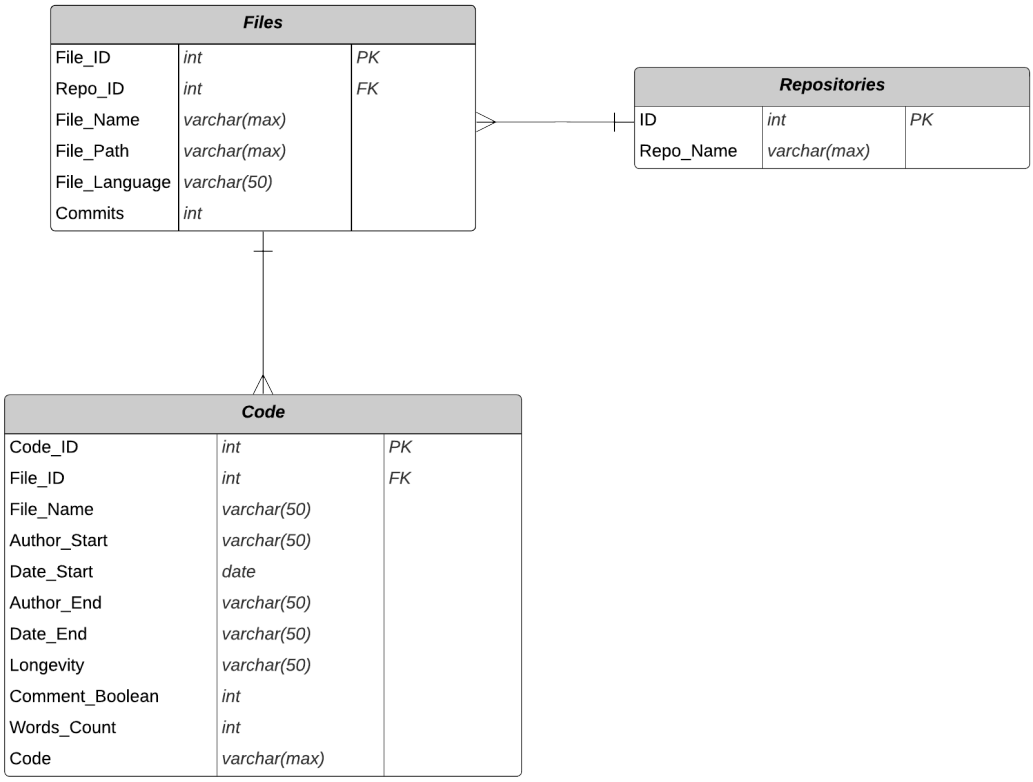

Figura 4.3: Diagrama de Entidad-Relación

tabla correspondiente en caso de ser necesario así como solamente tener acceso a ella en caso de ya estar creada. Por otro lado, se utiliza esta sentencia para obtener distintos datos de las tablas de la base de datos, en función de los campos que consideremos necesarios para calcular la métrica correspondiente y representar su respectivo gráfico resultante.

En la figura 4.3, se observa un diagrama de entidad relación (DER) que consiste en una herramienta para representar de manera simplificada cómo se ha organizado la información de la base de datos '*Analysis_GitHub_Repository*'. Esta base de datos se compone de tres tablas distintas denominadas '*Repositories*', '*Files*' y '*Code*'. De la tabla '*Repositories*' se establece una relación de uno a muchos, puesto que dentro de un repositorio se organizan diversos archivos, por lo que la clave primaria del ID correspondiente al repositorio pasa a ser la clave foránea de la tabla '*Files*'. Sucede lo mismo desde la tabla '*Files*' hacia la tabla '*Code*', ya que de un archivo se obtienen múltiples líneas de código, por lo que la clave primaria del ID correspondiente a cada archivo pasa a ser la clave foránea en la siguiente tabla.

4.3.3. Procesado de datos

Una vez recogidos y almacenados los datos, se realiza el preprocesamiento para prepararlos para el análisis. Esto puede implicar limpieza de datos (eliminación de valores atípicos, datos faltantes), normalización o estandarización de variables, selección de características relevantes y transformación de datos según sea necesario. Tenemos en cuenta diversas restricciones para que nuestro programa desarrolle su funcionamiento de manera eficiente. Teniendo en cuenta que el comando *git blame* no es compatible con cualquier tipo de archivo, ajustamos el código de manera que se eliminen del repositorio clonado los archivos con las siguientes extensiones [".png", ".jpg", ".jpeg", ".gif", ".pdf", ".exe", ".dll", ".ico", ".db", ".dat", ".class", ".o", ".pyc", ".mp3", ".mp4", ".wav", ".avi", ".bak", ".tmp", ".docx", ".pptx", ".zip", ".tar.gz", ".rar"]. También se eliminan los directorios ["node_modules", "vendor", "Pods"], que se corresponden con directorios donde se organizan y gestionan todas las dependencias del proyecto, como bibliotecas y módulos de terceros. Otra necesidad importante ha sido eliminar los archivos vacíos, es decir, aquellos que tienen un tamaño de 0 bytes. Estos archivos se utilizan comunmente para importar funciones y/o clases que se utilizan en otros módulos del proyecto, así como para marcar directorios como paquetes de Python, y producen errores en la ejecución del código al no disponer de líneas de código para analizar. El directorio '.git' también ha sido descartado para el análisis pero teniendo en cuenta que no se puede eliminar, ya que conserva toda la información acerca del historial de revisiones del repositorio.

Una vez que se realiza el análisis, se ha tenido en cuenta un manejo de todos los archivos en la codificación UTF-8, ya que permite un mayor manejo y procesado de los datos, así como menor dependencia de la detección automática consiguiendo así un menor flujo de trabajo. Por lo que inicialmente se realiza una detección automática de la codificación de todos los archivos, pero con la finalidad de obtener cada uno de ellos en codificación UTF-8. Además, se han descartado líneas con más de 255 caracteres así como líneas en blanco, con la finalidad de evitar errores al insertar los datos en la tabla denominada 'Code'.

También se realiza un procesado de datos teniendo en cuenta posibles funcionalidades a implementar en el futuro. Por lo que se identifica que cada línea de código sea un comentario o no, así como también se realiza un cálculo del número de palabras que tiene cada línea.

4.3.4. Implementación de métricas y análisis

A continuación, se desarrolla el objetivo más importante de este proyecto que consiste en el análisis de los datos. Mediante la implementación de distintas métricas, se pretenden estudiar distintos aspectos que resultan interesantes para analizar la evolución de los proyectos FOSS[2] a lo largo de los años. Tras la recopilación, revisión y limpieza de datos, se procede a la interpretación de los datos para la realización de un análisis estadístico, donde se han tenido en cuenta gran diversidad de factores como la antiguedad y/o tamaño del proyecto, el número de desarrolladores, la complejidad que desarrolla, etc. Se realizan distintos análisis estadísticos tanto para el conjunto de todos los proyectos, como para cada uno de ellos de manera individual. En total, se han analizado XXXXXXXX líneas de código de XXXX proyectos de software libre alojados en GitHub, y se les han aplicado las métricas que se presentan en los siguientes apartados.

Lenguajes de programación más utilizados

Se ha estudiado para cada uno de los repositorios analizados, los distintos lenguajes de programación utilizados por los desarrolladores del proyecto. De manera que, haciendo un cálculo con todos los archivos de los repositorios analizados se obtienen los porcentajes de los lenguajes de programación que tienen mayor presencia dentro de los proyectos de software libre y código abierto alojados en GitHub. Se han tenido en cuenta todo tipo de lenguajes de programación, que se han obtenido mediante la librería *Pygments*, y en caso de no corresponderse con ningún lenguaje conocido se ha clasificado como un archivo que contiene datos en un formato específico definido por el software que lo utiliza, y se ha denominado '*Archivo de datos*' o '*Archivo binario*'.

$$Porcentaje\,Lenguaje_i = \left(\frac{Archivos\,lenguaje_i}{Total\,de\,archivos}\right) \times 100$$

Líneas que permanecen en la actualidad

Resulta relevante estudiar el número de líneas que permanecen en la versión más actual del proyecto frente al momento de la creación del mismo, es decir, cuántas líneas permanecen intactas desde cualquier fecha pasada del proyecto. Esto puede aportar una estimación del esfuerzo de mantenimiento de cara a un futuro, debido a que un proyecto con una gran cantidad de líneas

[2]Free/Open Source Software

de código antiguas es más difícil de mantener, ya que puede que los autores hayan olvidado la razón de los cambios o incluso puede que ya ni siquiera formen parte del proyecto. Además, se puede observar qué proyectos se encuentran obsoletos en la actualidad, debido a que no hayan sufrido modificaciones en un período largo de tiempo, así como también se pueden observar proyectos con una tendencia de trabajo constante con la finalidad de mantener el proyecto actualizado.

Entonces, se desarrollan dos métricas, una de ellas para obtener el número de líneas permanentes en la actualidad a lo largo de los años en valores absolutos, y otra para obtener esta cantidad en porcentajes respecto al tamaño total del proyecto, y se aplican a cada uno de los repositorios analizados.

$$Cantidad\,de\,líneas\,vivas = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Líneas\,vivas\,año_i}{Líneas\,totales\,año_i}\right)$$

$$Porcentaje\,de\,líneas\,vivas = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Líneas\,vivas\,año_i}{Líneas\,totales\,año_i}\right) \times 100$$

Estabilidad del código

Medir la estabilidad del código es posible gracias a una métrica denominada *Tasa de Churn*. El análisis de esta métrica implica evaluar cuántas líneas de código han sido eliminadas en un período de tiempo específico. Proporciona una medida de la actividad y volatilidad del código, resulta ser una herramienta muy útil para comprender la dinámica de cambio de un código a lo largo del tiempo. La finalidad de utilizar esta métrica en este proyecto consiste en representar la *tasa de Churn* por cada archivo de los repositorios analizados, de manera que se compruebe así que archivos del repositorio se encuentran más obsoletos.

DUDA: APLICAR POR ARCHIVO O POR REPOS?¿?¿?

$$Tasa\,de\,Churn = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Líneas\,eliminadas\,año_i}{Líneas\,totales\,año_i}\right)$$

Frecuencia de cambios

Visualizar la frecuencia de cambios a lo largo del tiempo resulta útil para identificar períodos de actividad intensa o inactividad en el desarrollo del proyecto. Además, permite conocer las etapas del proyecto, ya que el desarrollo inicial se corresponde con la etapa donde se lleva a

cabo una alta frecuencia de cambios, pasando después a una etapa de estabilización con una frecuencia de cambios moderada, y siendo la etapa de mantenimiento aquella que tenga una baja frecuencia de cambios. Conocer las etapas del proyecto puede ser útil para realizar una planificación y estimación del trabajo de cara a un futuro, ya que un proyecto con cambios frecuentes y regular resulta menos costoso en términos de esfuerzo para implementar nuevas funcionalidades o mantenimientos.
La métrica utilizada para visualizar la frecuencia de cambios consiste en realizar un sumatorio del número de *commits* realizados en cada uno de los archivos para cada repositorio concreto.

$$Número\,de\,commits_i = \sum_{i=1}^{n} Commits\,archivo_i$$

Número de colaboradores

Otro dato relevante consiste en conocer la cantidad de colaboradores de un proyecto a lo largo de los años. Esto puede reflejar el crecimiento y la madurez del proyecto, ya que un aumento del número de desarrolladores a lo largo del tiempo puede indicar un proyecto en expansión y cada vez más relevante, así como el caso contrario se traduce en una reducción del interés o fases críticas del proyecto. En proyectos de código abierto, el la colaboración de múltiples desarrolladores indica un signo de confianza y relevancia en la comunidad. Por otro lado, un número alto de colaboradores se traduce en una diversidad de contribuciones, que aportan diferentes perspectivas, habilidades y conocimientos, lo que puede conducir a un código más robusto con funcionalidades más variadas.
Se calcula el número de colaboradores teniendo en cuenta el creador de cada línea de código respecto a un repositorio concreto a lo largo del tiempo.

$$Número\,de\,colaboradores_i = \sum_{i=1}^{n} Colaboradores\,repositorio_i$$

Evolución de la actividad de cada colaborador

Conocer la actividad de cada colaborador del proyecto nos puede resultar interesante para una estimación de costes, ya que evaluando la contribución individual se puede realizar una asignación de tareas más eficiente de acuerdo a las capacidades de cada colaborador. También resulta bastante útil conocer conocer en qué momento ha desarrollado cada colaborador mayor protagonismo para recurrir a él en caso de tener que realizar posibles modificaciones respecto

a algo concreto desarrollado en esa etapa del proyecto, esto nos proporciona una mayor gestión de los recursos y una mejor planificación. El hecho de visualizar las contribuciones individuales puede ayudar a mejorar el trabajo en equipo y el espíritu de colaboración.
La métrica para conocer la evolución de la actividad de cada colaborador del proyecto consiste en realizar un sumatorio del número de líneas creadas por cada desarrollador del código para conocer los valores absolutos. Mientras que comparando con el número total de líneas de todos los colaboradores del proyecto se calculan los valores relativos.

$$Actividad\,por\,colaborador = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Líneas\,vivas\,colaborador_i}{Líneas\,totales}\right)$$

$$Porcentaje\,por\,colaborador = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{Líneas\,vivas\,colaborador_i}{Líneas\,totales}\right) \times 100$$

4.3.5. Representación gráfica de resultados

Una vez que se han definido las métricas y los análisis estadísticos que resultan interesantes en esta investigación, se procede a una representación gráfica de los resultados.

Capítulo 5

Resultados

En este capítulo se incluyen los resultados de tu trabajo fin de grado.

Si es una herramienta de análisis lo que has realizado, aquí puedes poner ejemplos de haberla utilizado para que se vea su utilidad.

5.1. Descripción de los datos recopilados

5.2. Interpretación de los resultados

5.2.1. Lenguajes de programación más utilizados

5.2.2. Estabilidad del código

5.2.3. Frecuencia de cambios

5.2.4. Número de colaboradores

5.2.5. Evolución de la actividad de cada colaborador

Capítulo 6

Conclusiones

6.1. Consecución de objetivos

Esta sección es la sección espejo de las dos primeras del capítulo de objetivos, donde se planteaba el objetivo general y se elaboraban los específicos. Es aquí donde hay que debatir qué se ha conseguido y qué no. Cuando algo no se ha conseguido, se ha de justificar, en términos de qué problemas se han encontrado y qué medidas se han tomado para mitigar esos problemas. Y si has llegado hasta aquí, siempre es bueno pasarle el corrector ortográfico, que las erratas quedan fatal en la memoria final. Para eso, en Linux tenemos aspell, que se ejecuta de la siguiente manera desde la línea de *shell*:

```
aspell --lang=es_ES -c memoria.tex
```

6.2. Aplicación de lo aprendido

Aquí viene lo que has aprendido durante el Grado/Máster y que has aplicado en el TFG/TFM. Una buena idea es poner las asignaturas más relacionadas y comentar en un párrafo los conocimientos y habilidades puestos en práctica.

6.3. Lecciones aprendidas

Aquí viene lo que has aprendido en el Trabajo Fin de Grado/Máster.

6.4. Trabajos futuros

Ningún proyecto ni software se termina, así que aquí vienen ideas y funcionalidades que estaría bien tener implementadas en el futuro.

Es un apartado que sirve para dar ideas de cara a futuros TFGs/TFMs.

Apéndice A

Manual de usuario

Esto es un apéndice. Si has creado una aplicación, siempre viene bien tener un manual de usuario. Pues ponlo aquí.