



Universidad  
Rey Juan Carlos

GRADO EN INGENIERÍA EN TECNOLOGÍA DE LA  
TELECOMUNICACIÓN

Curso Académico 2019/2020

Trabajo Fin de Grado

APLICACIÓN DE BIG DATA A PROYECTOS DE  
INGENIERÍA EN GitHub

Autor : Carlos Morón Barrios

Tutor : Dr. Gregorio Robles



# Trabajo Fin de Grado

Aplicación de Big Data a Proyectos de Ingeniería en GitHub

**Autor :** Carlos Morón Barrios

**Tutor :** Dr. Gregorio Robles Martínez

La defensa del presente Trabajo Fin de Grado se realizó el día                      de  
de 2020, siendo calificada por el siguiente tribunal:

**Presidente:**

**Secretario:**

**Vocal:**

y habiendo obtenido la siguiente calificación:

**Calificación:**

Fuenlabrada, a                      de                      de 2020



*“Programar sin una arquitectura o  
diseño en mente es como explorar una  
gruta sólo con una linterna: no sabes  
dónde estás, dónde has estado ni  
hacia dónde vas”.*

*Danny Thorpe*



# Agradecimientos

Quiero agradecer a mis padres su apoyo incondicional, sin el cual habría sido casi imposible llegar hasta aquí. También quiero agradecer a mi hermano su ayuda y compañía, sobre todo en los momentos difíciles y complicados, que los ha habido.

La carrera no habría sido lo mismo sin mis compañeros, que han hecho más ligeros los madrugones para ir a estudiar. Aunque con el tiempo nos hemos desperdigado por los diferentes cursos y asignaturas, siempre hemos conseguido reunirnos para esas tapas y escapadas.

Gracias también a mis amigos porque sin ellos, nada sería lo mismo. Su apoyo e interés también es una razón para seguir adelante y poder compartir con ellos los buenos momentos.





# Resumen

El objetivo de este Trabajo Fin de Grado es hacer un estudio sobre las tecnologías Big Data y su aplicación al estudio de proyectos de ingeniería en grandes repositorios como el GitHub.

Github es un sitio que guarda una carpeta de documentos al que te conectas a través de un terminal de comando (o CMD, en Windows) o por medio de una aplicación de escritorio.

En el mundo actual centrado en el software, repositorios de software a gran escala, p. ej. SourceForge (más de 350.000 proyectos), GitHub (más de 250.000 proyectos) y Google Code (más de 250.000 proyectos) son la nueva biblioteca de Alejandría. Contienen una enorme cantidad de software e información sobre el software. Tanto los científicos como los ingenieros están interesados en analizar esta gran cantidad de información tanto por curiosidad como por probar sus hipótesis. Sin embargo, la extracción sistemática de datos relevantes de estos repositorios y el análisis de dichos datos para probar hipótesis es difícil.

En este trabajo se pretende analizar los repositorios de GitHub para seleccionar los proyectos de software de ingeniería e identificarlos.



# Summary

The objective of this project is to study Big Data technologies and their application to the study of engineering projects in large repositories such as GitHub.

GitHub is a site that stores a folder of documents that you connect to through a command terminal (or CMD, in Windows) or through a desktop application.

In today's software-centric world, large-scale software repositories, e.g. ex. SourceForge (more than 350,000 projects), GitHub (more than 250,000 projects) and Google Code (more than 250,000 projects) are the new library of Alexandria. They contain a huge amount of software and information about the software. Scientists and engineers are interested in analyzing this vast amount of information both out of curiosity and to test their hypotheses. However, the systematic extraction of relevant data from these repositories and analysis of such data to test hypotheses is difficult.

This work aims to analyze the GitHub repositories to select engineering software projects and identify them.



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Retos del Big Data . . . . .	2
1.1.1. Análisis de los datos . . . . .	2
1.2. Estructura de la memoria . . . . .	3
<b>2. Objetivos</b>	<b>5</b>
2.1. Objetivo general . . . . .	5
2.2. Objetivos específicos . . . . .	5
2.3. Planificación temporal . . . . .	6
<b>3. Estado del arte - Tecnologías relacionadas</b>	<b>7</b>
3.1. Contexto . . . . .	10
3.2. Definición de Git . . . . .	11
3.3. Funcionamiento de GitHub . . . . .	12
3.4. Marco de evaluación . . . . .	16
3.5. Fuentes de datos . . . . .	17
3.5.1. Metadatos . . . . .	17
3.5.2. Código fuente . . . . .	19
3.6. Dimensiones . . . . .	20
3.6.1. Comunidad . . . . .	22
3.6.2. Integración continua . . . . .	23
3.6.3. Documentación . . . . .	24
3.6.4. Historia . . . . .	28
3.6.5. Problemas . . . . .	29

3.6.6.	Licencia . . . . .	30
3.6.7.	Pruebas unitarias (Unit testing) . . . . .	31
3.7.	Herramientas estadísticas empleadas . . . . .	33
3.7.1.	Coefficiente de correlación de Spearman . . . . .	33
3.7.2.	Test de mannwhitneywilcoxon . . . . .	33
3.7.3.	Estadístico Delta de Cliff . . . . .	35
3.7.4.	Clasificador basado en la puntuación . . . . .	36
3.7.5.	Random Forest . . . . .	37
<b>4.</b>	<b>Diseño e implementación</b>	<b>39</b>
4.1.	Training Data Sets . . . . .	39
4.1.1.	Organización del conjunto de datos . . . . .	41
4.1.2.	Utilidad del conjunto de datos . . . . .	41
4.1.3.	Datos de instancias negativas . . . . .	42
4.1.4.	Resumen de los conjuntos de datos . . . . .	43
<b>5.</b>	<b>Resultados</b>	<b>49</b>
5.1.	Validación . . . . .	49
5.1.1.	Establecer la verdad fundamental . . . . .	51
5.1.2.	Validación interna . . . . .	51
5.1.3.	Validación externa . . . . .	53
5.2.	Predicción . . . . .	54
5.3.	Discusión . . . . .	55
5.4.	Puntos débiles del estudio . . . . .	59
<b>6.</b>	<b>Conclusiones</b>	<b>61</b>
6.1.	Consecución de objetivos . . . . .	61
6.2.	Aplicación de lo aprendido . . . . .	62
6.3.	Lecciones aprendidas . . . . .	62
6.4.	Futuros trabajos . . . . .	62
	<b>Bibliografía</b>	<b>63</b>

# Índice de figuras

1.1. Fases de análisis . . . . .	3
2.1. Planificación temporal del Trabajo Fin de Grado . . . . .	6
3.1. Imagen extraída de guides.github.com . . . . .	13
3.2. Imagen extraída de guides.github.com . . . . .	14
3.3. Imagen extraída de guides.github.com . . . . .	15
3.4. Imagen extraída de guides.github.com . . . . .	16
3.5. Métrica SLOC de un repositorio . . . . .	21
4.1. Número de repositorios en la organización agrupados por lenguajes de programación. . . . .	42
4.2. Número de repositorios en la utilidad del conjunto de datos agrupados por lenguajes de programación. . . . .	43
4.3. Distribución del número de puntuación de los repositorios; (a) en la organización y (b) utilidad del conjunto de datos . . . . .	44
4.4. Distribución de las dimensiones de repositorios en el conjunto de datos de la organización. . . . .	45
4.5. Distribución de las dimensiones obtenidas de los repositorios en el conjunto de datos de utilidad. . . . .	46
4.6. $\rho$ de Spearman entre pares de dimensiones en la organización (a) y conjuntos de datos de utilidad (b) con - (guión) representando las correlaciones estadísticamente insignificantes. . . . .	47
5.1. Distribución de las dimensiones de los repositorios en el conjunto de validación.	50

5.2. Número de repositorios obtenidos por los clasificadores basados en la puntuación y Random Forest agrupados por lenguajes de programación (ORGANIZACIÓN). . . . .	56
5.3. Número de repositorios obtenidos por los clasificadores basados en la puntuación y Random Forest agrupados por los lenguajes de programación (UTILIDAD). . . . .	57
5.4. Comparación de la distribución de las dimensiones de los repositorios con diferentes etiquetas de clasificación manual pero todas propiedad de organizaciones. . . . .	58
5.5. Comparación de la distribución de las dimensiones de los repositorios que contienen proyectos informáticos de ingeniería propiedad de organizaciones y usuarios. . . . .	60



# Capítulo 1

## Introducción

Big Data se ha convertido en la actualidad en un término muy usado y asociado a la gran avalancha de datos que se generan cada día, bien localmente en las empresas o disponibles en Internet [35].

La captura y posterior tratamiento de datos correspondientes a empresas tecnológicas e investigación en ingeniería puede revelar información valiosa para la organización relacionada con sus productos, descubrimiento clientes potenciales y grandes avances en una tecnología determinada.

Debido a la gran cantidad de datos que hay que almacenar y tratar, las tecnologías Big Data han estado aparentemente monopolizadas por grandes corporaciones, pero cada vez se avanza más en su aplicación en Universidades e instituciones con una gran I+D implantada, así como en los departamentos I+D de empresas tecnológicas (Pérez, 2017).

Debido al desarrollo tecnológico que se ha vivido en la última época con el nacimiento de las redes sociales, dispositivos móviles, sensores, añadir internet a un amplio abanico de cosas y sobre todo la llegada de la tecnología a los ciudadanos, se ha generado un gran volumen de datos (Big Data), así nace la necesidad de poder analizar dichos datos para transformarlos en información (Minelli et al., 2013). Ahora el problema reside en cómo analizar eficazmente los datos.

En el mundo actual centrado en el software, repositorios de software a gran escala, p. ej. SourceForge (más de 350.000 proyectos), GitHub (más de 250.000 proyectos) y Google Code (más de 250.000 proyectos) son la nueva biblioteca de Alejandría. Contienen una enorme cantidad de software e información sobre el software. Tanto los científicos como los ingenieros están interesados en analizar esta gran cantidad de información tanto por curiosidad como por probar sus hipótesis. Sin embargo, la extracción sistemática de datos relevantes de estos repositorios y el análisis de dichos datos para probar hipótesis es difícil.

## 1.1. Retos del Big Data

El concepto Big Data abarca diversas propiedades que es necesario tener claras. Debido a la gran cantidad de datos que se va a manejar, es necesario tener las herramientas necesarias para conseguir dar sentido a todos ellos y poder analizarlos, puesto que un gran porcentaje de esos datos (cerca del 85 %) no van a estar estructurados (Pérez, 2017).

Además, parte de esta enorme generación de datos se da en tiempo real, con lo que no se puede analizar con efectividad con las técnicas normales de procesamiento de datos.

Por otro lado, tenemos la problemática de quien puede realizar el tratamiento de los datos relacionamos con Big Data. Para solucionar este problema se deben formar profesionales especialistas en el análisis de los datos para que creen o usen herramientas de gestión de la información derivada del procesamiento de los datos. En este entorno es donde las tecnologías de la información deben dar el salto (C. N. & Batista, 2013). A modo de resumen podemos concluir que la tecnología asociada a Big Data y los objetivos de negocios deben coordinarse para ser capaces de obtener información de calidad.

### 1.1.1. Análisis de los datos

Los datos deben superar un análisis exhaustivo, para revelar la información que contienen y generar conocimiento. El valor potencial de los datos no solo radica en disponer de ellos sino en saber organizarlos y refinarlos para convertirlos en información relevante; el valor que se le da a los datos después del análisis es para incrementar la capacidad de innovar y obtener ventaja

sobre los competidores (C. N. & Batista, 2013).

Gestionar correctamente los datos debe generar jurisprudencia en las empresas, la ciudadanía y sobre todo en las administraciones ya sean públicas o privadas puesto que analizar correctamente los datos se debe convertir en un activo para la sociedad (C. N. & Batista, 2013). En este último trabajo (C. N. & Batista, 2013), se proponen las fases del análisis (figura 1.1).

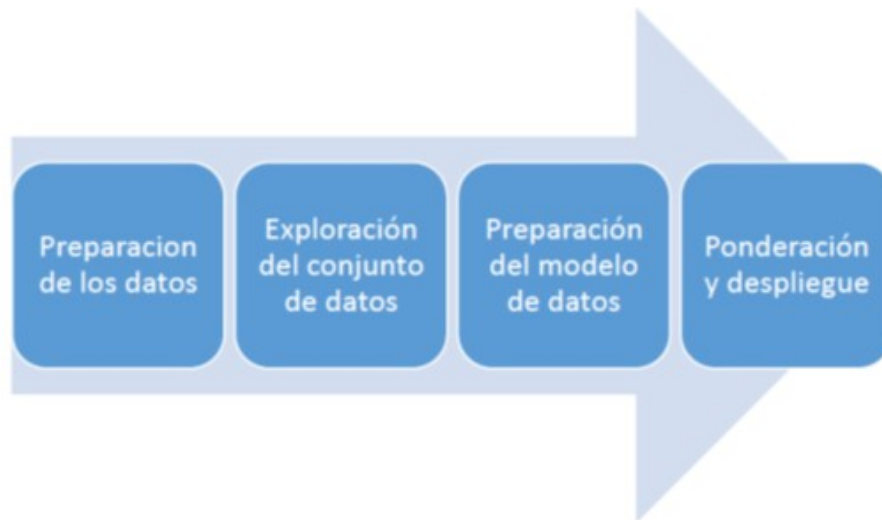


Figura 1.1: Fases de análisis

## 1.2. Estructura de la memoria

- En el capítulo 1 se hace una introducción al proyecto y a la importancia del manejo de Big Data.
- En el capítulo 2 se describen los objetivos : comenzamos presentando la noción de un proyecto de software de ingeniería en el capítulo 2. Luego, proponemos un marco de evaluación en la Sección 2.1 que tiene como objetivo poner en práctica la definición de un proyecto de software de ingeniería a lo largo de un conjunto de dimensiones.
- Describimos las diversas fuentes de datos utilizadas en nuestro estudio en el capítulo 3.
- En el capítulo 4, presentamos las siete dimensiones utilizadas para representar el repositorio de nuestro estudio.

- En el capítulo 5, proponemos dos variaciones a la definición de un proyecto de software de ingeniería, recopilamos un conjunto de repositorios que se ajustan a las definiciones y presentamos enfoques para construir clasificadores capaces de identificar otros repositorios que se ajusten a la definición de un proyecto de software de ingeniería.
- Los resultados de la validación de los clasificadores y su uso para identificar repositorios que se ajustan a una definición particular de un proyecto de software diseñado a partir de una muestra de 1,857,423 repositorios GitHub se presentan en el capítulo 6.
- Comparamos nuestro estudio con la literatura previa en el capítulo 7, discutimos los escenarios de investigación en los que el conjunto de datos y el clasificador podrían usarse en el capítulo 8, y se discuten los matices de ciertos repositorios y las amenazas a la validez en el capítulo 10 y concluimos el documento con el capítulo 11.

# Capítulo 2

## Objetivos

### 2.1. Objetivo general

Mi trabajo fin de grado consiste en identificar los proyectos de software de ingeniería de los repositorios de software libre alojados en la plataforma GitHub.

### 2.2. Objetivos específicos

Para conseguir el objetivo principal ha sido necesario realizar los siguientes logros:

- Conseguir y analizar la base de datos de GHTorrent.
- Seleccionar un número determinado de datos para identificar los campos de la base de datos.
- Poner las cabeceras correspondientes a cada fichero como se indica en el esquema de la página GHTorrent.
- Calcular los parámetros correspondientes a cada dimensión.
- Seleccionar los proyectos que se consideran ingeniería en función de las dimensiones elaboradas.
- Comprobar que los proyectos seleccionados son los correctos.

## 2.3. Planificación temporal

A continuación, detallo en la figura 2.1 la temporalidad de las diferentes fases de mi trabajo que empezó en enero de 2020.

Trabajo Fin de Grado	Ene	Feb	Mar	Abr	May	Jun	Jul	Ago	Sep	Oct	Nov	Dic
Planteamiento												
Base de datos GHTorrent												
Identificar campos												
Calcular dimensiones												
Seleccionar Proyectos												
Comprobación												
Escritura TFG												

Figura 2.1: Planificación temporal del Trabajo Fin de Grado

## Capítulo 3

### Estado del arte - Tecnologías relacionadas

Los repositorios de software contienen una gran cantidad de información sobre el código, las personas y los procesos que intervienen en el desarrollo de un producto de software. Suponiendo solo una exigua línea de 1K de código por proyecto, los repositorios SourceForge, GitHub y Google Code supondrían al menos 8,61 mil millones de líneas de código. El análisis retrospectivo de estos repositorios de software puede proporcionar información valiosa sobre la evolución y el crecimiento de los productos de software que contiene. Podemos rastrear ese análisis hasta la década de 1970, cuando Belady y Lehman (1976) propusieron las Leyes de evolución del software de Lehman. Hoy en día hay una fuerte inversión económica para el desarrollo de estos análisis, como el proyecto Boa (Dryer et al, [9]) que recibió más de 1.4 millones de dólares.

En este último trabajo Dyer y colaboradores plantean que tanto los científicos como los ingenieros están interesados en analizar esta gran cantidad de información tanto por curiosidad como por probar hipótesis tan importantes como: “¿cómo perciben y consideran las personas los posibles impactos de sus ediciones y las de los demás mientras escriben juntas? (Dourish and Bellotti, [8])”; “¿Cuál es la licencia de código abierto más utilizada? (Lerner and Tirole., [18])”; “¿Cuántos proyectos continúan usando estándares de cifrado DES (considerados inseguros)? (Landau, [16]) ”; “¿Cuántos proyectos de código abierto tienen una política de control de exportación restringida? (Goodman et al., [12])”; “¿Cuántos proyectos en promedio comienzan con una base de código existente de otro proyecto en lugar de cero? (Raymond, [32]) ”; “¿Con qué frecuencia los profesionales usan las características dinámicas de Javascript, p. Ej. eval? (Richards et al., [33])”; “¿Cuál es el tiempo promedio para resolver un error reportado como crítico? (Weiss et al.,[42])”.

Sin embargo, la barrera de entrada actual podría ser prohibitiva. Por ejemplo, para responder a las preguntas anteriores, un equipo de investigación necesitaría (a) desarrollar experiencia en el acceso programático a los sistemas de control de versiones, (b) establecer una infraestructura para descargar y almacenar los datos de los repositorios de software en poco tiempo, (c) programar una infraestructura en un lenguaje de programación completo como C ++, Java, C o Python para acceder a estos datos locales y responder a la hipótesis, y (d) mejorar la escalabilidad de la infraestructura de análisis para ser capaz de procesar datos a gran escala en un tiempo razonable. Estos cuatro requisitos aumentan sustancialmente el costo de la investigación científica. Hay cuatro problemas adicionales. Primero, los experimentos son a menudo irreproducibles porque replicar una configuración experimental requiere un esfuerzo enorme. En segundo lugar, la reutilización de la infraestructura experimental es típicamente baja porque la infraestructura de análisis no está diseñada de manera reutilizable. Después de todo, el foco del investigador original está en el resultado del análisis y no en la reutilización de la infraestructura de análisis. Por lo tanto, los investigadores tienen que replicar los esfuerzos de los demás. Tercero, los datos asociados y producidos por tales experimentos a menudo se pierden y se vuelven inaccesibles y obsoletos, porque no existe una depuración sistemática. Por último, pero no menos importante, construir una infraestructura de análisis para procesar datos a gran escala de manera eficiente puede ser muy difícil (Dean and Ghemawat, [7]; Pike et al., [29]; Chambers et al., [3]).

Por otro lado, los conocimientos adquiridos a través del análisis de los repositorios pueden afectar al proceso de toma de decisiones en un proyecto y mejorar la calidad del software que se está desarrollando. Un ejemplo de esto se puede ver en las recomendaciones hechas por Bird y colaboradores (Bird et al., 2011) en su estudio sobre los efectos de la propiedad del código en la calidad de los sistemas de software. Los autores sugieren que los esfuerzos de aseguramiento de la calidad deberían centrarse en aquellos componentes con muchos desarrolladores del código.

La riqueza de los datos y las posibles ideas que representa fueron los factores habilitantes en el inicio de todo un campo de investigación: los repositorios de software de minería (MSR). En los primeros días de repositorios de software de la minería, los investigadores tenían acceso limitado a los repositorios de software, que estaban alojados principalmente dentro de las organizaciones. Sin embargo, con la proliferación de repositorios de software de acceso abierto como GitHub, Bitbucket, SourceForge y CodePlex para el código fuente y Bugzilla, Mantis y



Trac para los errores, los investigadores ahora tienen una gran cantidad de datos para extraer y sacar conclusiones interesantes. Cada confirmación de código fuente contiene una gran cantidad de información que se puede utilizar para comprender el arte del desarrollo de software. Por ejemplo, Eick [10] se zambulló en el rico historial de compromisos (más de 15 años) de un gran sistema de conmutación telefónica para explorar la idea de la descomposición del código. Los repositorios de código fuente modernos ofrecen características que hacen que la gestión de un proyecto de software sea lo más fluida posible. Si bien la integración de características proporciona una mejor trazabilidad para los desarrolladores y gerentes de proyectos, también brinda a los investigadores una fuente de información única, autónoma, organizada y, lo que es más importante, de acceso público desde la cual extraer. Sin embargo, cualquiera puede crear un repositorio para cualquier propósito sin costo alguno. Por lo tanto, la calidad de la información contenida en las falsificaciones puede estar disminuyendo con la adición de muchos repositorios ruidosos, p. repositorios que contienen tareas de trabajo a domicilio, archivos de texto, imágenes o, lo que es peor, la copia de seguridad de una computadora de escritorio.

Kalliamvakou y col. (2014) identificaron este ruido como uno de los nueve peligros a tener en cuenta al extraer datos de GitHub para la investigación de ingeniería de software. La situación se agrava por el gran volumen de repositorios contenidos en estas falsificaciones. A partir de junio de 2016, solo GitHub alojó más de 38 millones de repositorios<sup>2</sup> y este número está aumentando rápidamente. Los investigadores han utilizado varios criterios para dividir las falsificaciones de software gigantescas en conjuntos de datos manejables para sus estudios. Por ejemplo, los investigadores han aprovechado filtros simples como la popularidad para eliminar repositorios ruidosos. Los filtros como la popularidad (medidos como el número de observadores o observadores de estrellas en GitHub, por ejemplo) son simplemente representantes y no pueden ser de propósito general ni representativos de un proyecto de software diseñado. La intuición al usar la popularidad como filtro es que se puede suponer que la popularidad está correlacionada con la calidad. Sin embargo, como Sajnani et al. (2014) han demostrado que la popularidad no está correlacionada con la calidad, cuestionando la utilidad del uso de filtros basados en la popularidad. Creemos que los investigadores que extraen repositorios de software no deberían tener que reinventar filtros para eliminar repositorios no deseados. Hay algunos ejemplos de investigación que adoptan el enfoque de desarrollar sus propios filtros para obtener un conjunto de datos para analizar:

- En un estudio de la relación entre lenguajes de programación y calidad de código, Ray et al. (2014) seleccionaron los 50 repositorios más populares (medidos por el número de estrellas) en cada uno de los 19 idiomas más populares.
- Bissyand´e y col. (2013b) eligió los primeros 100,000 repositorios devueltos por la API de GitHub en su estudio de la popularidad, interoperabilidad e impacto de los lenguajes de programación.
- Allamanis y Sutton (2013) eligieron 14.807 repositorios de Java con al menos una bifurcación en su estudio de aplicar el modelado de lenguaje a la minería de repositorios de código fuente.

Los sitios web del proyecto para GHTorrent (2016a) y Boa (Iowa State University 2016) enumeran más artículos que emplean diferentes esquemas de filtrado.

La suposición que se podría hacer es que los repositorios muestreados en estos estudios contienen proyectos de software diseñados. Sin embargo, las falsificaciones de código fuente están plagadas de repositorios que no contienen código fuente, y mucho menos un proyecto de software diseñado. Kalliamvakou et al. (2014) muestrearon manualmente 434 repositorios de GitHub y encontraron que solo el 63.4 % (275) de ellos eran para desarrollo de software; Los 159 repositorios restantes se utilizaron con fines experimentales, de almacenamiento o académicos, o estaban vacíos o ya no eran accesibles. La inclusión de repositorios que contengan dichos artefactos que no sean de software en los estudios dirigidos a proyectos de software podría llevar a conclusiones que pueden no ser aplicables a la ingeniería de software en general. Al mismo tiempo, no es posible seleccionar una muestra mediante investigación manual dado el gran volumen de repositorios alojados por estas falsificaciones de código fuente.

### 3.1. Contexto

Laplane (2007) define la ingeniería del software como “un enfoque sistemático para el análisis, diseño, evaluación, implementación, prueba, mantenimiento y reingeniería de software”. Un proyecto de software puede considerarse como “ingeniería” si hay evidencia perceptible de la aplicación de principios de ingeniería de software como diseño, prueba, mantenimiento,

etc. En líneas similares, se puede definir un proyecto de software como un proyecto que aprovecha las buenas prácticas de ingeniería del software en una o más de sus dimensiones, como la documentación, las pruebas y la gestión de proyectos (Munaiah et al., [23]).

Esa definición es lo suficientemente abstracta como para que se pueda adaptar a muchos casos. Por ejemplo, un estudio relacionado con las pruebas en proyectos de software podría definir un proyecto de software que aprovecha las prácticas de prueba del software. Más específicamente se puede definir un proyecto de ingeniería del software de dos maneras: (a) cuando es similar a los proyectos contenidos en repositorios propiedad de organizaciones populares de ingeniería de software como Microsoft, Apache, Amazon y Mozilla y (b) cuando es similar a los proyectos que tienen una utilidad de propósito general para usuarios que no sean los propios desarrolladores.

## 3.2. Definición de Git

Git es un software de control específico de versión de código abierto creado por el ingeniero Linus Torvalds en 2005, conocido por ser el creador del kernel de Linux. Esto quiere decir que si activamos el control sobre la carpeta donde está nuestro código el sistema se encargará de controlar los cambios en los archivos, con lo que tanto la base del código entero como su historial se encuentran disponibles en los ordenadores de todos los desarrolladores, lo cual permite un fácil acceso a las bifurcaciones y fusiones (Kinsta, 2020).

El principal objetivo de Git es llevar un estricto control de los cambios que varias personas realizan al tiempo sobre un archivo de computadora (Nextu, 2020).

Sus principales características son:

1. El diseño de Git se basa en BitKeeper y en Monotone.
2. Además de ofrecer apoyo al desarrollo no lineal, proporciona rapidez en la gestión de ramas y fusión de diferentes versiones.
3. Proporciona a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales.
4. Los almacenes de información pueden publicarse mediante HTTP, FTP, rsync o un protocolo nativo.

5. Los repositorios Subversion y svn se pueden usar directamente con git-svn.

Además, cuando se quieran marcar los cambios se realizará lo que se conoce como commit, que consiste en describir los cambios realizados, apuntándolos en este registro. De esta forma, podremos movernos entre los diferentes commits, por ejemplo, para volver a una versión anterior de nuestro proyecto.

También, los sistemas de control de versiones, entre los que se encuentra Git, son la base en la actualidad para los proyectos de equipo ya que podemos trabajar a la vez múltiples programadores en un mismo proyecto, incluso en un mismo archivo de una manera fácil (Openwebinars, 2020).

Por otro lado, con el portal Kinsta, el sistema de control de versiones permite a los desarrolladores administrar el código fuente de un programa y habilitarlo para que se hagan modificaciones a través de la bifurcación y la fusión. La bifurcación te permite crear una copia de una parte del código, para que los desarrolladores puedan modificarla de forma segura sin que los cambios afecten la versión original (Nextu, [25]). Esto evita que cualquier error afecte el software final. La fusión permite al desarrollador unir su versión de código al código fuente una vez que ha comprobado su buen funcionamiento, aunque posteriormente el sistema de control de versiones les permitirá a los administradores revertir cualquier cambio.

### 3.3. Funcionamiento de GitHub

GitHub es una plataforma que permite tener nuestros repositorios de Git en la nube. Esto permite centralizar el contenido del repositorio y colaborar con otros desarrolladores. Además, permite alojar programas o sistemas operativos para que la comunidad acceda a los códigos fuente y realice comentarios. Aunque el registro es opcional, es necesario crear una cuenta para iniciar proyectos y modificar copias de otros repositorios.

Dependiendo de las autorizaciones que otorgue el titular de la cuenta, los usuarios podrán realizar modificaciones que quedan registradas gracias al sistema de control de versiones Git.

GitHub es la plataforma de código abierto más utilizada del mundo, con más de 28 millones de desarrolladores según cifras de la compañía. El uso de GitHub es totalmente gratuito para los proyectos públicos y permite que varios colaboradores puedan trabajar en la modificación

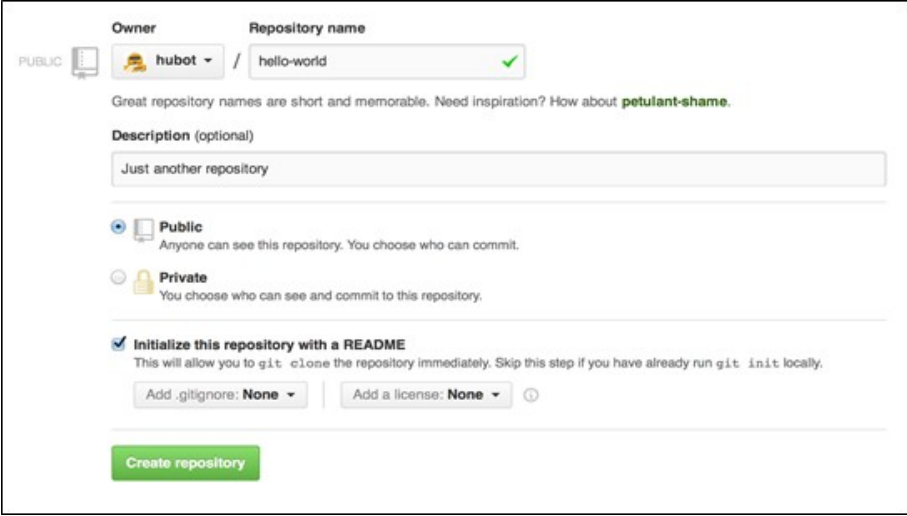
de un proyecto. Además, los usuarios también han aprovechado la plataforma para compartir canciones, recetas, tipografías, guías colaborativas o, incluso, escribir libros.

En junio de 2019 Microsoft anunció la compra de GitHub 7.500 millones de dólares y el pasado 26 de octubre se confirmó la autorización del parlamento europeo a dicha transacción.

Para manejar GitHub hay que seguir los siguientes pasos (Nextu, 2020):

#### 1. Crear un repositorio

- a) Hacer clic en la esquina superior derecha junto a tu foto de perfil y luego seleccionar New Repository.
- b) Nombrar tu repositorio, por ejemplo: “Prueba”.
- c) Escribir una breve descripción del proyecto.
- d) Seleccionar la opción: Initialize this repository with a README (Figura 3.1).



The screenshot shows the GitHub interface for creating a new repository. At the top, there's a 'PUBLIC' label and a 'hubot' user icon. The 'Repository name' field contains 'hello-world' with a green checkmark. Below this, a hint suggests repository names should be short and memorable, with 'petulant-shame' as an example. The 'Description (optional)' field contains 'Just another repository'. Under the 'Visibility' section, 'Public' is selected with a radio button, and 'Private' is unselected. Below this, the 'Initialize this repository with a README' checkbox is checked. A note explains that this will allow cloning the repository immediately. At the bottom, there are two dropdown menus: 'Add .gitignore: None' and 'Add a license: None'. A green 'Create repository' button is at the very bottom.

Figura 3.1: Imagen extraída de guides.github.com

#### 2. Crear una bifurcación

Por lo general, la rama principal en la que quedará alojado el código fuente original se denomina “master”, mientras que las bifurcaciones o copias que se creen para ser modificadas se llamarán “feature”.

- a) Dirígete a tu nuevo repositorio “Prueba”.

- b) Haz clic en el menú desplegable llamado branch: master.
- c) Escribe el nombre de una división en el cuadro de texto, por ejemplo, “readme-edits”.
- d) Selecciona la casilla azul Create Branch o presiona Enter en tu teclado (Figura 3.2).

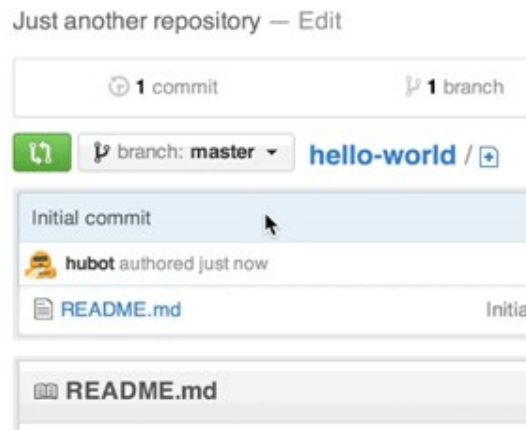


Figura 3.2: Imagen extraída de guides.github.com

### 3. Realizar cambios

En GitHub cada cambio que se realiza es denominado “Commits” y queda asociado a un “Commits message” para explicar el historial de ajustes que se hicieron. Esto permitirá que otros colaboradores comprendan los cambios.

- a) Haz clic en el archivo README.
- b) Haz clic en el ícono de lápiz ubicado en la esquina superior derecha del archivo para editar.
- c) En el editor, escribe un poco sobre tu perfil y lo que desees que otros usuarios conozcan de ti.
- d) Escribe un mensaje que explique los cambios que has realizado ara que los colaboradores comprendan los ajustes en el código.
- e) Haz clic en el botón Commit change (Figura 3.3).

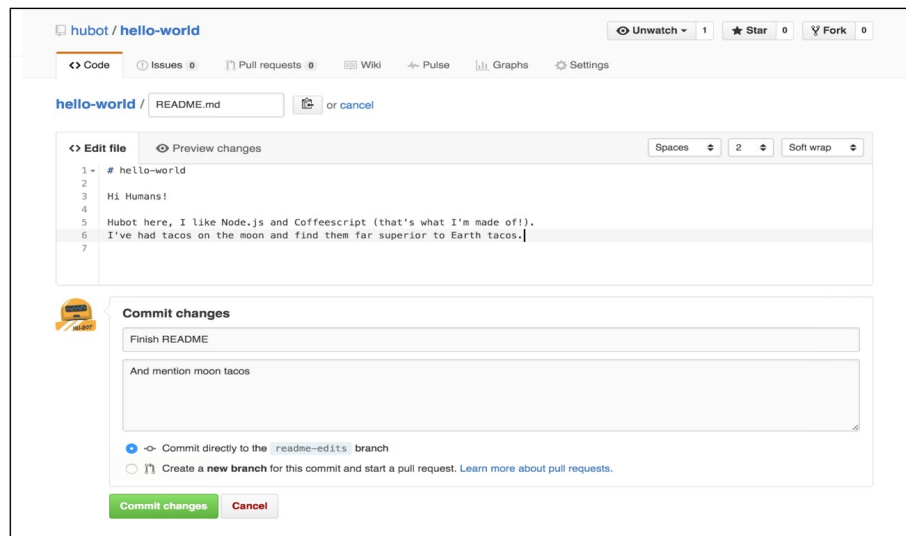


Figura 3.3: Imagen extraída de guides.github.com

#### 4. Abrir una solicitud de extracción (si modificaste el archivo de alguien más)

Una vez que se han terminado las modificaciones en un proyecto y se ha comprobado que funciona adecuadamente, se puede enviar una solicitud de extracción a un usuario, para que tenga en cuenta tu versión, la revise y la agregue a su código fuente o versión master.

- a) Haz clic en la pestaña Pull Request.
- b) Luego, haz clic en el botón verde New Pull Request.
- c) En el cuadro Example Comparisons, selecciona la división que creaste de del archivo README para comparar con el código original “master”.
- d) Revisa tus cambios y asegúrate de que sean los que deseas enviar. En verde aparecerán los elementos agregados y en rojos los eliminados.
- e) Haz clic en el botón verde Create Pull Request.
- f) Asigna un título a tu solicitud y escribe una breve descripción de tus cambios.

#### 5. Combinar la solicitud de extracción (si alguien más realizó modificaciones del proyecto)

- a) Haz clic en el botón verde de Merge Pull Request para combinar los cambios que realizaron a tu código con la versión “master” (Figura 3.4).
- b) Haz clic en Confirm merge.

- c) Ya que los cambios se han incorporado, elimina la división con el botón Delete branch en el cuadro púrpura.

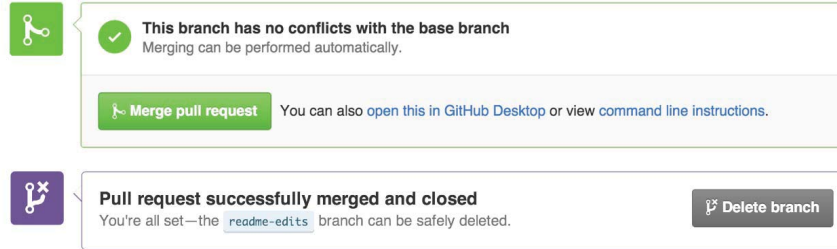


Figura 3.4: Imagen extraída de guides.github.com

### 3.4. Marco de evaluación

Para poner identificar los proyectos de ingeniería del software se necesita (a) identificar las prácticas esenciales de ingeniería del software que se emplean en el desarrollo y mantenimiento de un proyecto de software típico y (b) proponer medios para cuantificar la evidencia de su uso en un proyecto de software dado. El marco de evaluación es nuestro intento de lograr este objetivo.

El marco de evaluación, en su forma más simple, es una función de valor booleano definida como una función por partes como se muestra en la ecuación 3.1.

$$f(r) = \begin{cases} \text{verdadero si el repositorio } r \text{ contiene un proyecto de ingeniería} \\ \text{falso en cualquier otro caso} \end{cases} \quad (3.1)$$

El marco de evaluación no asume la implementación de la función de valor booleano,  $f(r)$ . En nuestra implementación del marco de evaluación, hemos elegido realizar  $f(r)$  de dos maneras: (a)  $f(r)$  como un clasificador basado en la puntuación y (b)  $f(r)$  como un clasificador Random Forest. En ambos casos, la implementación de la función,  $f(r)$ , se logra al expresar el repositorio,  $r$ , usando un conjunto de atributos cuantificables (llamados dimensiones) que creemos que son esenciales para razonar que un repositorio contiene un proyecto de ingeniería de software.



## 3.5. Fuentes de datos

En nuestro trabajo vamos a analizar las dos fuentes principales de datos que hemos utilizado: los repositorios de acceso público disponibles en GitHub y sus subsecciones.

### 3.5.1. Metadatos

Los metadatos de GitHub contienen una gran cantidad de información con la que se podrían describir varias características que se identifican en el código fuente. Por ejemplo, algunas de las piezas importantes de los metadatos son el lenguaje principal de implementación en un repositorio y los compromisos realizados por los desarrolladores en un repositorio. GitHub proporciona una REST API (GitHub Inc, 2016a) con la que se pueden obtener metadatos de GitHub a través de Internet. Existen varios servicios que capturan y publican estos metadatos de forma masiva, evitando los retardos de la API oficial. El proyecto GitHub Archive (GitHub Inc, 2016b) fue creado para este propósito y almacena los eventos públicos en el cronograma de GitHub y los publica a través de Google BigQuery. Google BigQuery es un motor de consultas que admite construcciones similares a SQL para consultar grandes conjuntos de datos. Sin embargo, acceder al conjunto de datos de GitHub Archive a través de BigQuery conlleva un costo por terabyte de datos procesados. Afortunadamente, Gousios [13] tiene una solución de uso libre a través de su Proyecto GHTorrent. El proyecto GHTorrent proporciona un espejo fuera de línea escalable y consultable de todos los metadatos de Git y GitHub disponibles a través de la REST API de GitHub. Durante los últimos años, GitHub se ha convertido en el sitio de alojamiento de repositorios elegido para muchos proyectos de software de código abierto (OSS). Curiosamente, GitHub proporciona una REST API para su conjunto de datos completo, por lo que es un objetivo de investigación atractivo. El proyecto GHTorrent utiliza la API de GitHub para recopilar datos sin procesar y poder extraer, archivar y compartir metadatos. Los conjuntos de datos creados ya han sido utilizados en otros trabajos: análisis del modelo de desarrollo de extracción, análisis de compromisos de conducción y análisis de incentivos de prueba en sitios sociales (Gousios et al.,[14]) y son de libre acceso para los investigadores. GHTorrent fue diseñado para usar el almacenamiento en caché en exceso (para evitar solicitudes duplicadas) y también para ser distribuido (para permitir que múltiples usuarios recuperen datos en paralelo).

La API de GitHub admite dos tipos de consultas:

- Las consultas de recursos recuperan una instancia específica de un recurso. Según los mandatos de la arquitectura REST, la URL que identifica un recurso estático permanece constante después de que el recurso se ha inicializado.
- Las consultas de rango recuperan una lista de recursos, generalmente relacionados con un recurso dado. Por ejemplo, la cadena `/user/ followers` recupera los seguidores de un usuario, mientras que la cadena `/user/followers/commits` recupera la asignación para un repositorio. La paginación se usa para limitar la cantidad de datos por respuesta. Las consultas de rango no necesariamente devuelven la instancia de entidad completa para cada elemento, pero generalmente incluyen una URL donde se puede recuperar el elemento. Como resultado, una consulta de rango podría resultar en varias consultas de recursos.

Los resultados de la consulta de recursos se pueden almacenar en la memoria caché de manera muy eficiente, ya que, por definición, nunca cambian. Las consultas de rango son más difíciles de almacenar en caché, ya que su resultado puede cambiar a medida que el proyecto evoluciona (nuevos compromisos, nuevos seguidores, etc.); afortunadamente, por defecto, GitHub proporciona primero los resultados más nuevos, por lo que es suficiente pasar por las primeras páginas de resultados solo para recuperar los datos actualizados. Para almacenar en caché los resultados por entidad, GHTorrent utiliza una base de datos MongoDB, que ofrece el beneficio adicional de habilitar la consulta en los datos sin procesar. El almacenamiento en caché también se utiliza en la capa de solicitud HTTP; GHTorrent serializa automáticamente las respuestas HTTP en el disco. Esto evita la recuperación de páginas antiguas.

El algoritmo de duplicación se basa en un proceso de resolución de dependencia recursiva. Para cada elemento recuperable, especificamos un conjunto de dependencias, ya que lógicamente fluyen desde el esquema de datos. Por ejemplo, para recuperar un proyecto, es necesario recuperar primero el usuario propietario; del mismo modo, para recuperar una solicitud de extracción, el proyecto debe recuperarse primero. Si falla cualquier paso de la resolución de dependencia, todo el elemento se marca como no recuperado. El proceso fue diseñado desde el principio para ser idempotente: cada paso de la resolución de dependencia puede fallar, pero una vez que tiene éxito, siempre devolverá el mismo resultado. Esta elección de diseño es muy importante ya que hace que se agreguen los datos y los resultados de cada paso y, por lo tanto, que sean almacenables en caché.

El proyecto GHTorrent es similar al proyecto GitHub Archive (Grigorik, 2020), ya que ambos comienzan con el marco temporal de los eventos públicos de GitHub. Mientras que el proyecto GitHub Archive simplemente registra los detalles de un evento GitHub, el proyecto GHTorrent recupera exhaustivamente los contenidos del evento y los almacena en una base de datos. Además, los conjuntos de datos GHTorrent están disponibles para descargar, ya sea como volcados incrementales de MongoDB o como un único volcado de MySQL, lo que permite el acceso sin conexión a los metadatos. Se ha optado por utilizar el volcado de MySQL que se descargó y restauró en un servidor local. En el resto del documento, cada vez que se usa el término base de datos nos referimos a la base de datos GHTorrent.

El volcado de la base de datos utilizado en este estudio se lanzó el 1 de abril de 2015. El volcado de la base de datos contenía metadatos para 16,331,225 repositorios de GitHub. En este estudio, nos restringimos a los repositorios en los que el lenguaje principal es Java, Python, PHP, Ruby, C ++, C o C #. Además, no consideramos los repositorios que se han marcado como eliminados y los que están contenidos en otros repositorios. Los repositorios eliminados restringen la cantidad de datos disponibles para el análisis, mientras que los repositorios bifurcados pueden inflar artificialmente los resultados al introducir duplicados en la muestra. Con estas restricciones aplicadas, el tamaño de nuestra muestra se reduce a 2,247,526 repositorios.

Una limitación inherente de la base de datos es la obsolescencia de los datos. Puede haber repositorios en la base de datos que ya no existan en GitHub, ya que pueden haber sido eliminados, renombrados, privados o bloqueados por GitHub. Como resultado, solo consideramos los repositorios que estaban activos en el momento en que se ejecutó nuestro análisis.

### 3.5.2. Código fuente

Además de los metadatos sobre un repositorio, el código que contiene es una fuente importante de información sobre el proyecto. Los desarrolladores suelen interactuar con sus repositorios utilizando el cliente git o la interfaz web GitHub. Los desarrolladores también pueden usar la REST API de GitHub para interactuar mediante programación con GitHub.

Se va a utilizar GitHub con el fin de obtener una copia del código fuente para cada repositorio. No se puede utilizar la REST API de GitHub para recuperaciones instantáneas del repositorio, ya que la API utiliza internamente el comando git archive para crear esas instantáneas. Como resultado, las instantáneas pueden no incluir archivos que los desarrolladores hayan mar-

cado como irrelevantes para un usuario final (como los archivos de prueba unitaria). Como se quieren examinar todos los archivos de desarrollo en nuestro análisis, utilizamos el comando `git clone` para asegurarnos de que todos los archivos se descargan.

Como ya se ha mencionado, los metadatos utilizados en este estudio están actualizados a partir del 1 de abril de 2015. Para sincronizar el repositorio con los metadatos, hay que restablecer el estado del repositorio a una fecha pasada (Munaiah et al. [22]), de forma que para cada repositorio evaluado en la base de datos, recuperamos la fecha de la confirmación más reciente en el repositorio. Posteriormente se identifica el *SHA* de la última confirmación realizada en el repositorio antes del final del día, identificado por fechas y usando el comando `git log -l -before = "date 11:59:59"`. Para los repositorios sin confirmaciones registradas en la base de datos, utilizamos la fecha en la que se lanzó el volcado de metadatos GHTorrent, es decir, 01-04-2015. Con el *SHA* de confirmación apropiado identificado, el estado del repositorio clonado se restablece utilizando el comando `git reset -duro {SHA}`.

### 3.6. Dimensiones

En este apartado se van a analizar las distintas dimensiones utilizadas para representar un repositorio en relación con lo descrito en los anteriores apartados. De esta forma, se va a representar un repositorio mediante las siguientes siete dimensiones:

1. Comunidad, como evidencia de colaboración.
2. Integración continua, como evidencia de calidad.
3. Documentación, como evidencia de mantenibilidad.
4. Historia, como evidencia de evolución sostenida.
5. Cuestiones, como evidencia de la gestión del proyecto.
6. Licencia, como prueba de responsabilidad.
7. Pruebas unitarias, como prueba de calidad.

A la hora de seleccionar las dimensiones, además de tener en cuenta y como prioritario todo lo relacionado con la ingeniería del software era primordial, hay que considerar también

otros aspectos como son la precisión en la medida y la simplicidad. Además, el algoritmo que se debe utilizar para medir esas dimensiones tiene que ser lo suficientemente genérico como para incluir los distintos lenguajes de programación utilizados en los proyectos de software y lo suficientemente específico como para producir resultados significativos. Hay que tener en cuenta que, aunque la lista no es exhaustiva ni objetiva, en la evaluación no se tienen en cuenta ni las diferentes dimensiones ni la forma en que se utilizan las dimensiones para determinar si un repositorio contiene un proyecto de ingeniería del software.

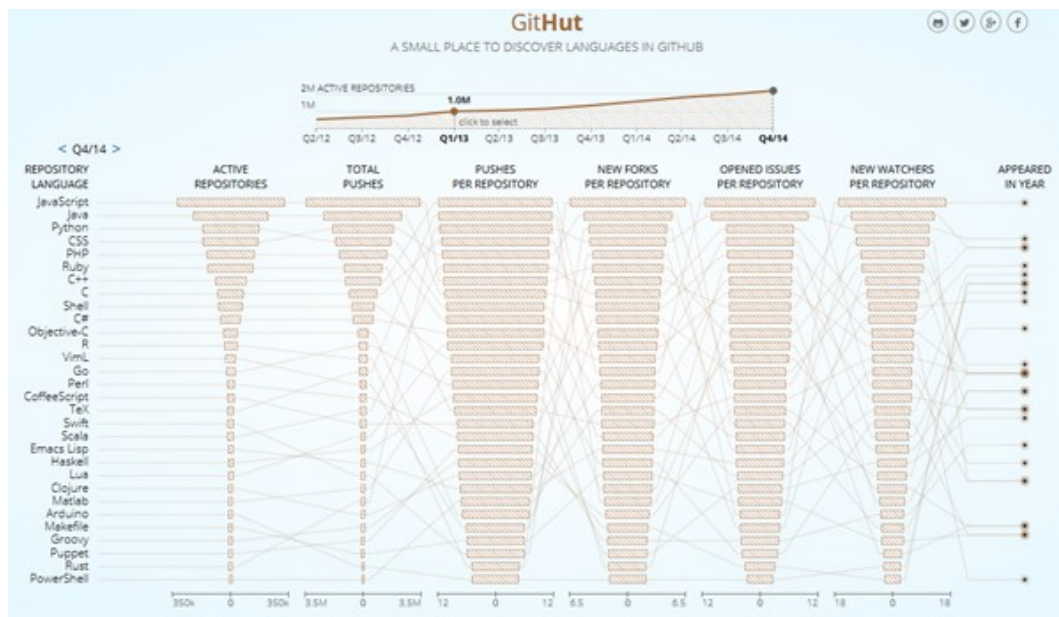


Figura 3.5: Métrica SLOC de un repositorio

En las próximas subsecciones, se van a describir las dimensiones con mayor detalle y se profundizará en el atributo de un proyecto de software que representa la dimensión, en la métrica para cuantificar la dimensión y en la metodología para recopilar la métrica de un repositorio de código fuente. El proceso de recopilación de la métrica de una dimensión puede necesitar una o ambas fuentes de datos. Además de las siete dimensiones enumeradas anteriormente, es necesario tener en cuenta el tamaño del repositorio, utilizando la métrica de las líneas de código fuente (SLOC), ya que hay que tener en cuenta como dicho tamaño influye en las otras dimensiones. En este trabajo se ha utilizado la utilidad Perl (Danial, [5]) para recopilar la métrica SLOC de un repositorio (Figura 3.5).

Se ha utilizado una herramienta de código abierto llamada Reaper, que es capaz de recopilar la métrica para cada una de las dimensiones de un repositorio de código fuente dado. El código

fuelle de Reaper está disponible en GitHub en <https://github.com/RepoReapers/reaper>. En su versión actual sus capacidades tienen las siguientes restricciones:

- El repositorio debe ser de acceso público en GitHub.
- El lenguaje principal del repositorio debe ser Ruby, Python, PHP, Java, C ++, C o C #.

Se han elegido estos lenguajes porque son los más utilizados en GitHub (Zapponi, [45]).

Dos de las características principales de Reaper son su flexibilidad y extensibilidad, ya que es muy fácil agregar la posibilidad de poder analizar repositorios con un código fuente en otro lenguaje (por ejemplo, JavaScript).

### 3.6.1. Comunidad

El desarrollo e implantación masiva de internet ha contribuido, de una forma colaborativa y descentralizada, al desarrollo de una disciplina como la ingeniería del software y a la proliferación de software libre y desarrolladores de código abierto que colaboran en multitud de proyectos.

La presencia de un conjunto grande de desarrolladores de software de código abierto indica que hay alguna forma de colaboración y cooperación involucrada en el desarrollo del software y nos da cierta idea de que un repositorio puede contener un proyecto de ingeniería del software. En este sentido, Whitehead y col. [43] han planteado la hipótesis de que el desarrollo de un software que involucra a más de un desarrollador puede considerarse en primera instancia como ingeniería de software colaborativa. Además, en este sentido proponen una métrica para cuantificar la comunidad que ha contribuido en el código fuente de un repositorio. En este sentido, se plantea que la colaboración es una actividad central en la ingeniería de software, ya que todos los proyectos, excepto los más triviales, involucran a varios ingenieros que trabajan juntos. Por lo tanto, comprender la colaboración en ingeniería de software es importante tanto para ingenieros como para investigadores. Este capítulo presenta un marco para comprender la colaboración de ingeniería de software, centrado en tres ideas clave: (1) los modelos en abierto; (2) la gestión de proyectos de software; y (3) formas de distancia espacial, temporal y sociocultural en las vías de colaboración existentes. El análisis de las tendencias futuras resalta varias formas en que los ingenieros podrán mejorar la colaboración del proyecto, específicamente, los

entornos de desarrollo de software cambiarán a estar totalmente basados en la Web, lo que abrirá el potencial para la integración, una mayor participación de los usuarios finales en el desarrollo del proyecto, y mayor facilidad en la ingeniería del software global.

Además, se va a considerar como contribuyentes principales en un repositorio al conjunto más pequeño de contribuyentes cuya aportación al código fuente de un repositorio representa el 80 % o más de las contribuciones totales.

La noción de contribuyentes principales es fundamental en el software de código abierto, donde un conjunto de contribuyentes conduce un proyecto hacia un objetivo común. Mockus y col. [21] han aplicado este concepto en su estudio de las prácticas de desarrollo del software de código abierto. La definición de contribuyentes principales es la misma que la de los desarrolladores principales tal como la definen Syer et al. [36].

Se va a calcular las contribuciones totales contando el número de confirmaciones realizadas en un repositorio cuando se registra en la base de datos. Luego agrupamos las confirmaciones por autor y seleccionamos los primeros  $n$  autores para los cuales el número acumulado de confirmaciones representaba el 80 % de las contribuciones totales. El valor de  $n$  representa la métrica de los contribuyentes principales. Hay un problema en la implementación de esta métrica en Reaper. Utilizamos los datos de GHTorrent (GHTorrent, 2016a) para encontrar contribuyentes únicos de un repositorio. Sin embargo, GHTorrent tiene la noción de "usuarios falsos" que no tienen cuentas de GitHub (GHTorrent 2016b) pero publican sus contribuciones con la ayuda de usuarios reales de GitHub. Por ejemplo, un usuario falso introduce un código commit en un repositorio local de Git, luego un usuario real introduce esos códigos commit en el GitHub usando su cuenta. A veces, los usuarios reales y falsos pueden ser los mismos. Este es el caso cuando un desarrollador con una cuenta de GitHub se conecta con una dirección de correo electrónico secundaria. Esto tiende a inflar la métrica de contribuyentes principales para repositorios pequeños con un solo contribuyente real, y podría mejorarse en el futuro mediante la detección de las direcciones de correo electrónico similares.

### **3.6.2. Integración continua**

La integración continua (CI) es una práctica de ingeniería del software en la que los desarrolladores regularmente crean, ejecutan y prueban su código combinado con el código de otros desarrolladores. La integración continua se realiza para garantizar que la estabilidad del siste-

ma en su conjunto no se vea afectada por los cambios. Normalmente implica implementar y compilar el software, ejecutar pruebas unitarias automatizadas y analizar la calidad del sistema.

Con millones de desarrolladores que contribuyen a miles de repositorios de código fuente, la práctica de integrar continuamente los cambios asegura que el software contenido en estos repositorios de código fuente está en constante evolución y que es estable para el desarrollo y/o lanzamiento. El uso de la integración continua es una prueba más de que el proyecto de software podría considerarse un proyecto de ingeniería del software.

La métrica para la integración continua se puede definir como una función por partes como se muestra a continuación:

$$M_{ci}(r) = \begin{cases} 1 & \text{si el repositorio } r \text{ usa un servicio de la integración continua} \\ 0 & \text{en cualquier otro caso} \end{cases} \quad (3.2)$$

El uso de un servicio de integración continua se determina buscando un archivo de configuración (requerido por ciertos servicios de CI) en el repositorio de código fuente. Una limitación inherente de este enfoque es que solo admite la identificación de servicios de CI sin estado. La integración con servicios con estado como Jenkins, Atlassian Bamboo y Cloudship no se puede identificar, ya que puede que no haya rastros de la integración en el repositorio. Los servicios de integración continua actualmente soportados son: Appveyor, Hound, Travis CI, Shippable, MagnumCI, CircleCI, Solano y Wercker.

### 3.6.3. Documentación

Un documento de software puede describirse como cualquier documento destinado a comunicar información sobre el software (Forward and Lethbridge, [11]). Según Ambler [1], la documentación del software responde a tres necesidades: (i) contractual; (ii) respaldar un proyecto de desarrollo de software al permitir que los miembros del equipo conciban gradualmente la solución a implementar, y (iii) permitir que un equipo de desarrollo del software comunique los detalles de la implementación a lo largo del tiempo al equipo de mantenimiento.

La documentación generalmente sufre los siguientes problemas: inexistente o de baja calidad (Pressman, [30]; anticuado (Thomas and Tilley [38]; Tilley [40]; Tilley et al. [41]; Forward and Lethbridge [11]); sobre abundante y sin un objetivo definido (Forward and Lethbridge [11];



Lindvall et al. [19]); difícil de acceder (por ejemplo, cuando los documentos están dispersos en varias computadoras o en diferentes formatos: texto, diagramas) (Tilley [40]); falta de interés de los programadores (Pigoski, [28]; HCI <sup>1</sup>; Tilley and Muller [39]); y, difícil de estandarizar, debido, por ejemplo, a las especificidades del proyecto (Phoha [27]).

Recientemente, otros métodos proponen un enfoque para el desarrollo del software que eliminan la necesidad de la documentación como una ayuda para el desarrollo del software. Proponen una comunicación informal entre desarrolladores y usuario. Sin embargo, estos métodos no eliminan la necesidad de documentación como herramienta de comunicación a lo largo del tiempo, lo que permite a los desarrolladores comunicar información importante sobre un sistema a futuros mantenedores.

Una mejor definición de los documentos que necesitan los encargados del mantenimiento del software ya se ha considerado en otros estudios.

- Tilley en [41] y [40] enfatiza la importancia de un documento que describa la arquitectura jerárquica del sistema.
- Cioch y col. [4] diferencian cuatro etapas (desde el recién llegado, el primer día de trabajo; a experto, después de algunos años de trabajo en un sistema). Para cada etapa, proponen diferentes documentos: los recién llegados necesitan una visión general breve del sistema; los aprendices necesitan la arquitectura del sistema; después necesitan documentos orientados a tareas tales como descripción de requisitos, descripción de procesos, ejemplos, instrucciones paso a paso; Finalmente, los expertos necesitan documentación de bajo nivel, así como la descripción de requisitos y las especificaciones de diseño.
- Rajlich [31] propone una herramienta que permite recopilar notas sobre el dominio de la aplicación.
- Ambler [1] recomienda documentar las decisiones y una visión general del diseño: requisitos, reglas comerciales, arquitectura, etc.

---

<sup>1</sup>What to put in software maintenance documentation. Available on the Internet at: [http://www.hci.com.au/hcisite2/journal/ What to put in software maintenance documentation. htm](http://www.hci.com.au/hcisite2/journal/What%20to%20put%20in%20software%20maintenance%20documentation.htm), 20012002. Last accessed on May 27

- En un taller organizado por Thomas y Tilley en SIGDoc 2001 [38], afirman que “nadie sabe realmente qué tipo de documentación es realmente útil para los ingenieros de software para ayudar a la comprensión del sistema”.
- Forward y Lethbridge [11] en su encuesta a los desarrolladores, encontraron que los documentos de especificación son los más consultados, mientras que los documentos de calidad y bajo nivel son los menos consultados.
- Grubb y Takang [15], identifican algunas necesidades de información de las personas que se dedican al mantenimiento de acuerdo con sus actividades. Los gerentes necesitan información de soporte de decisiones, como el tamaño del sistema y el costo de la modificación. Los analistas deben comprender el dominio de la aplicación, los requisitos y tener una visión global del sistema. Los diseñadores necesitan comprensión arquitectónica (componentes funcionales y cómo interactúan) e información detallada del diseño (algoritmos, estructuras de datos). Finalmente, los programadores necesitan una comprensión detallada del código fuente, así como una vista de nivel superior (similar a la vista arquitectónica).
- Anquetil y col. [2] presenta una herramienta de re-documentación para automatizar parcialmente un proceso de re-documentación que se centra en la siguiente información: vista de alto nivel (con descripción de los requisitos), modelo de datos, referencias cruzadas entre funcionalidades, funciones y datos, reglas de negocio, descripción e interacción de subsistemas y comentarios.
- Finalmente, de acuerdo con Teles[37], los documentos que deben generarse al final de un proyecto XP son: historiales, pruebas, modelo de datos, modelo de clase, descripción del proceso comercial, manual del usuario y minutos del proyecto. Una conclusión que podemos extraer de esta breve revisión es que la arquitectura del sistema es un documento importante para el mantenimiento del software.

El mantenimiento se define tradicionalmente como cualquier modificación realizada en un sistema después de su entrega. Los estudios muestran que el mantenimiento del software es, con mucho, la actividad predominante en la ingeniería del software (90 % del costo total de un software típico (Pigoski, [28]; Seacord et al., [34])). Es necesario para mantener los sistemas de

software actualizados y útiles: cualquier sistema de software refleja el mundo en el que opera, cuando este mundo cambia, el software debe cambiar en consecuencia. La primera ley de Lehman de la evolución del software (ley del cambio continuo) (Lehman, [17]) es que “un programa que se utiliza sufre cambios continuos o se vuelve cada vez menos útil”. El mantenimiento es obligatorio, uno simplemente no puede ignorar las nuevas leyes o la nueva funcionalidad introducida por un concurrente. Los programas también deben adaptarse a nuevas computadoras (con mejores rendimientos) o nuevos sistemas operativos (Souza et al., [6]).

Uno de los principales problemas que afectan el mantenimiento del software es la falta de documentación actualizada. Debido a esto, los mantenedores a menudo deben trabajar desde el código fuente hasta la exclusión de cualquier otra fuente de información. Por ejemplo, un estudio (ver Pfleeger [26]; Pigoski [28]) informa que del 40 % al 60 % de la actividad de mantenimiento se emplea en estudiar el software para comprenderlo y cómo la modificación planeada puede ser implementada.

Los desarrolladores de software crean y mantienen diversas formas de documentación. Algunos forman parte de los archivos fuente, como los comentarios de código, y otros son parte de fuentes externas, como wikis, requisitos y documentos de diseño. Una función de la documentación es ayudar a la comprensión del software para fines de mantenimiento. Entre las muchas formas de documentación, se ha encontrado que los comentarios sobre el código fuente eran tan importantes, que solo los superaban el código fuente en sí (Souza et al. [6]). La presencia de documentación, en cantidad suficiente, indica que el autor pensó en el mantenimiento; esto sirve como evidencia parcial para determinar que es ingeniería del software.

En este estudio, nos limitamos a la documentación en forma de comentarios de código fuente. Proponemos una relación métrica de comentarios para cuantificar la extensión de la documentación del código fuente de un repositorio.

La relación de comentarios es la relación entre el número de líneas de código de comentarios (cloc) y el número de líneas de código fuente (sloc) que no están en blanco en un repositorio  $r$ .

$$M_d(r) = \frac{cloc}{sloc + cloc} \quad (3.3)$$

Utilizamos un cloc de la utilidad Perl (Danial [5]) para calcular las líneas de código fuente y las líneas de código de comentarios. Cloc devuelve líneas de código en blanco, comentarios y líneas de código fuente agrupadas por los diferentes lenguajes de programación en el reposito-

rio. Se agregan entonces los valores devueltos por `cloc` al calcular la relación de comentarios. Se puede observar que la proporción de comentarios solo cuantifica la extensión del código fuente en la documentación contenida en un repositorio. No se considerará la calidad, la antigüedad o la relevancia de la documentación. Además, solo se va a considerar una única fuente de documentación (comentarios de código fuente) al cuantificar esta dimensión, No se considerarán otras fuentes de documentación (externas) como wikis, documentos de diseño y cualquier archivo README asociado porque identificar y cuantificar estas fuentes externas puede no ser tan sencillo. Es posible que tengamos que aprovechar las técnicas de procesamiento del lenguaje natural para analizar estas fuentes de documentación externa.

### 3.6.4. Historia

Eick y col. (2001) han demostrado que el código fuente debe sufrir cambios continuos para impedir su obsolescencia y seguir siendo comercializable. Un cambio podría ser la corrección de errores, adición de características, mantenimiento preventivo, resolución de vulnerabilidades, etc. La presencia de un cambio sostenido indica que software se está modificando para garantizar su viabilidad.

En el contexto de un repositorio de código fuente, un commit es la unidad por la cual se puede cuantificar el cambio. Se propone por tanto que la frecuencia del commit sea una métrica de los cambios que sufre un repositorio. Con todo esto tendremos que la frecuencia del commit va a ser el número promedio de commit por mes.

$$M_h(r) = \frac{1}{m} \sum_{i=1}^m c_i \quad (3.4)$$

donde:

- $c_i$  es el número de commit para el mes  $i$
- $m$  es el número de meses entre el primer y el último commit del repositorio  $r$

Cada  $c_i$  se calculará contando el número de commit registrados en la base de datos para el mes  $i$ . Sin embargo,  $m$  se calculó como la diferencia, en meses, entre la fecha del primer commit y la fecha del último commit del repositorio. Si  $m$  da como resultado 0, el valor de la métrica debe ser 0 también.

### 3.6.5. Problemas

Con los años se han desarrollado gran cantidad de herramientas que sirven para simplificar la gestión de grandes proyectos de software. Estas herramientas se utilizan en algunas de las principales propiedades de la ingeniería de software, como la administración de requisitos, cronogramas, tareas, defectos y versiones. Se supone que un proyecto de software que emplea herramientas de gestión de proyectos es representativo de un proyecto de ingeniería del software. Por lo tanto, la evidencia del uso de herramientas de gestión de proyectos en un repositorio de código fuente puede indicar la presencia de un proyecto de ingeniería del software.

Aunque hay varias herramientas comerciales disponibles, no existe una forma unificada en la que estas herramientas se integren en un repositorio de código fuente. Los repositorios de código fuente alojados en GitHub pueden aprovechar una característica engañosamente nombrada de GitHub (problemas de GitHub) para administrar potencialmente todo el ciclo de vida de un proyecto de software. Decimos que se llama engañosamente porque un "problema" en GitHub puede estar asociado con una variedad de etiquetas personalizables que podrían alterar la interpretación del problema. Por ejemplo, los desarrolladores podrían crear historias de usuario como problemas de GitHub y etiquetarlas como User Story. La riqueza y flexibilidad de la función GitHub Issues ha impulsado el desarrollo de varios servicios de terceros como Codetree (Codetree Studios, 2016), HuBoard (HuBoard Inc, 2016), waffle.io (CA Technologies, 2016) y ZenHub (Zenhub, 2016). Estos servicios utilizan problemas de GitHub para respaldar la gestión del ciclo de vida de los proyectos.

En este proyecto se asume que el uso sostenido de la función GitHub Issues indica la gestión en un repositorio de código fuente. Por eso se propone como métrica la frecuencia del problema para cuantificar el uso sostenido de GitHub Issues en un repositorio. Así, la frecuencia del problema va a ser la media del número de problemas ocurridos por mes.

$$M_i(r) = \frac{1}{m} \sum_{i=1}^m s_i \quad (3.5)$$

(5) donde:

- $s_i$  es el número de problemas para el mes  $i$
- $m$  es el número de meses entre el primer y el último commit del repositorio  $r$

Cada  $s_i$  se calculará contando el número de problemas registrados en la base de datos para el mes  $i$ . Sin embargo,  $m$  se calculará como la diferencia, en meses, entre la fecha del primer commit y la fecha del último commit del repositorio. Si  $m$  es 0, el valor de la métrica debe ser 0 también.

Un inconveniente de este enfoque es que no permite descubrir otras herramientas de gestión de proyectos ya que los enlaces estructurados a estas herramientas pueden no existir en el código fuente del repositorio.

### 3.6.6. Licencia

La posibilidad que tenga un usuario de usar, modificar y / o redistribuir una parte de un software viene determinado por el tipo de licencia que acompaña al software. Las licencias son especialmente importantes en el contexto de los proyectos de código abierto como lo analiza un artículo del Software Freedom Law Center<sup>2</sup>, donde se destaca la necesidad de las licencias de software de código abierto y las mejores prácticas de las mismas.

Un software que no venga con licencia está protegido por defecto por las leyes de copyright, que establecen que el autor retiene todos los derechos sobre el código fuente (GitHub, 2016c). Sin embargo, aunque no existe obligación de incluir una licencia en un repositorio de código fuente, se considera una práctica recomendada. Además, los permisos de uso de GitHub pueden permitir que otros usuarios puedan copiar repositorios de acceso público. Por lo tanto, incluir una licencia en el repositorio dicta explícitamente los derechos del usuario para copias del repositorio, pero aunque dicha licencia no es suficiente para indicar que un repositorio contiene un proyecto de ingeniería del software de acuerdo con nuestra definición de la dimensión.

Podemos definir la métrica para la dimensión de licencia como una función por partes como se indica a continuación:

$$M_l(r) = \begin{cases} 1 & \text{si el repositorio } r \text{ tiene una licencia} \\ 0 & \text{en cualquier otro caso} \end{cases} \quad (3.6)$$

La presencia de una licencia en un repositorio de código fuente se evalúa utilizando la licencia API de GitHub. La licencia API identifica la presencia de licencias de código abierto

---

<sup>2</sup>Managing copyright information within a free software project software freedom law center. <http://softwarefreedom.org/resources/2012/managingcopyrightinformation>. Html. Acceso: 11-02-2020

populares mediante el análisis de archivos como LICENCIA y COPIA en la raíz del repositorio de código fuente.

La licencia API de GitHub es limitada, ya que no tiene en cuenta la licencia contenida en README.md o en los archivos de código fuente. Además, la API todavía está en situación de "vista previa del desarrollador", por lo que no es fiable.

Por otro lado, aunque cualquier mejora en las capacidades de la API se tendría en cuenta, se han superado algunas de las limitaciones analizando la licencia contenida en los archivos del repositorio de código fuente. Identificamos la información de la licencia buscando en los archivos del repositorio las 12 licencias de código abierto más populares en GitHub, como por ejemplo la "The MIT License (MIT)". Las 12 licencias elegidas fueron enumeradas por la licencia API de GitHub (<https://api.Github.com/licenses>). Hay que hacer notar que esta solución provisional puede tener algún problema en repositorios sin licencia propia que incluyan archivos de código fuente de bibliotecas externa, ya que si alguno de los archivos de código fuente de la biblioteca externa contiene extractos de la licencia, la dimensión de la licencia puede indicar falsamente que el repositorio contiene una licencia.

### **3.6.7. Pruebas unitarias (Unit testing)**

Un producto de ingeniería tiene una vida útil determinada y está sujeto a una serie de pruebas rigurosas para que funcione correctamente durante ese tiempo. Un producto de ingeniería del software no es diferente, ya que está sujeta también a pruebas rigurosas que respaldan la garantía del producto. La evidencia de las pruebas en este último caso implica que los desarrolladores han dedicado tiempo y esfuerzo para garantizar que el producto tenga el comportamiento previsto. Sin embargo, la presencia de pruebas no es una medida suficiente para concluir que el software funciona correctamente, ya que influye la idoneidad de esas pruebas. La adecuación de las pruebas contenidas en un proyecto de software puede medirse de varias maneras (Zhu et al. [46]).

Esencialmente, para obtener la métrica de estas pruebas se requiere la ejecución de las pruebas unitarias, que pueden requieren satisfacer todas las dependencias que el programa bajo prueba puede tener. Afortunadamente, existen medios para aproximar la adecuación de las pruebas en un proyecto de software mediante el análisis estático. Nagappan et al. [24] han utilizado el número de casos de prueba por línea de código fuente y número de afirmaciones por línea fuen-

te de código en la evaluación de la cantidad de prueba en proyectos Java. Además, Zaidman et al. [44] han demostrado que la cobertura de la prueba está correlacionada con el porcentaje de código de prueba en el sistema.

Proponemos una relación métrica, el análisis del cociente, para cuantificar la extensión de la prueba unitaria. Así, ese análisis el cociente será la razón entre el número de líneas fuente de código en los archivos de prueba y el número de líneas de código fuente en todos los archivos fuente.

$$M_u(r) = \frac{slotc}{sloc} \quad (3.7)$$

dónde,

- *slotc* es el número de líneas de código fuente en los archivos de prueba en el repositorio *r*.
- *sloc* es el número de líneas de código fuente en todos los archivos fuente en el repositorio

Para calcular el *slotc*, primero debemos identificar los archivos de prueba. Logramos esto buscando patrones específicos del lenguaje y del marco de pruebas en el repositorio. Por ejemplo, los archivos de prueba en un proyecto de Python que utilizan el marco de las pruebas unitarias pueden identificarse mediante la búsqueda de patrones de *import unittest* o de *import TestCase*.

Utilizamos *grep* para buscar y obtener una lista de archivos que contienen patrones específicos como el anterior. Luego usamos la herramienta *cloc* para calcular *sloc* de todos los archivos fuente en el repositorio y *slotc* de los archivos de prueba identificados. Ocasionalmente, un proyecto de software puede usar marcos de prueba de unidades múltiples, como por ejemplo un proyecto de aplicación web de Django puede usar el programa *unittest* de Python y la extensión de *unittest django.test* de Django. Para tener en cuenta este escenario, acumulamos los archivos de prueba identificados usando patrones para marcos de prueba de unidades específicas de múltiples idiomas antes de calcular *slotc*.

La multitud de marcos de pruebas unitarias disponibles para cada uno de los lenguajes de programación considerados hace que el enfoque sea limitado en sus capacidades. Actualmente se admiten 20 marcos de prueba unitarios. Los marcos de prueba unitarios actualmente admitidos son: NUnit, Visual Studio Testing y xUnit para C #; Boost, Catch, googletest y Stout gtest



para C++; PHPUnit para PHP; clar, GLib Testing y picotest para C; JUnit y TestNG para Java; django.test, nose y unittest para Python; y pruebas de unidad minitest, RSpec y Ruby para Ruby. En escenarios en los que no podemos identificar un marco de prueba de unidad, recurrimos a considerar todos los archivos en directorios llamados prueba, pruebas o especificaciones como archivos de prueba.

## 3.7. Herramientas estadísticas empleadas

En este apartado se resumen las herramientas estadísticas que se han utilizado en el desarrollo de este trabajo.

### 3.7.1. Coeficiente de correlación de Spearman

El coeficiente de correlación de Spearman ( $\rho$ ) es una medida no paramétrica de la correlación de rango (dependencia estadística del ranking entre dos variables). Se utiliza principalmente para el análisis de datos, así como para medir la fuerza y la dirección de la asociación entre dos variables clasificadas (Question Pro, 2020).

Con otras palabras, es una medida de la correlación (la asociación o interdependencia) entre dos variables aleatorias (tanto continuas como discretas). Para calcular  $\rho$ , los datos son ordenados y reemplazados por su respectivo orden.

Dicho coeficiente  $\rho$  viene dado por la ecuación:

$$\rho = 1 - \frac{6 \sum D^2}{N(N^2 - 1)} \quad (3.8)$$

donde  $N$  es el número de parejas de datos y  $D$  es la diferencia entre los correspondientes estadísticos de orden de  $x - y$ .

### 3.7.2. Test de mannwhitneywilcoxon

Las pruebas no-paramétricas se necesitan cuando no se tienen información sobre la composición de los datos poblacionales, esto es no se tiene conocimiento sobre su distribución de probabilidad. Dentro de las pruebas no-paramétricas, una muy utilizada es la prueba **U de**

**Mann-Whitney** (también llamada de Mann-Whitney-Wilcoxon, prueba de suma de rangos Wilcoxon, o prueba de Wilcoxon-Mann-Whitney), que es una prueba no paramétrica aplicada a dos muestras independientes. Es, de hecho, la versión no paramétrica de la habitual prueba  $t$  de Student.

Esta prueba fue propuesta inicialmente en 1945 por Frank Wilcoxon para muestras de igual tamaño y extendido a muestras de tamaño arbitrario como en otros sentidos por Henry B. Mann y D. R. Whitney en 1947.

La prueba de Mann-Whitney se usa para comprobar la heterogeneidad de dos muestras ordinales. El planteamiento de partida es el siguiente:

- Las observaciones de ambos grupos son independientes
- Las observaciones son variables ordinales o continuas.
- Bajo la hipótesis nula, la distribución de partida de ambos grupos es la misma  $y$ ,
- Bajo la hipótesis alternativa, los valores de una de las muestras tienden a exceder a los de la otra:  $P(X > Y) + 0,05P(X = Y) > 0,05$ .

De esta forma, para calcular el estadístico  $U$  se asigna a cada uno de los valores de las dos muestras su rango para construir

$$U_1 = n_1 n_2 + \frac{n_1(n_1 + 1)}{2} - R_1 \quad (3.9)$$

$$U_2 = n_1 n_2 + \frac{n_2(n_2 + 1)}{2} - R_2 \quad (3.10)$$

donde  $n_1$  y  $n_2$  son los tamaños respectivos de cada muestra;  $R_1$  y  $R_2$  es la suma de los rangos de las observaciones de las muestras 1 y 2 respectivamente, definiéndose el estadístico  $U$  como el mínimo de  $U_1$  y  $U_2$ . Además, los cálculos tienen que tener en cuenta la presencia de observaciones idénticas a la hora de ordenarlas. No obstante, si su número es pequeño, se puede ignorar esa circunstancia.

Esta prueba calcula el llamado estadístico  $U$ , cuya distribución para muestras con más de 20 observaciones se aproxima bastante bien a la distribución normal. Así, la aproximación a la normal,  $z$ , cuando tenemos muestras lo suficientemente grandes viene dada por la expresión:

$$z = \frac{(U - m_U)}{\theta_U} \quad (3.11)$$

donde  $m_U$  y  $\theta_U$  son la media y la desviación estándar de  $U$  si la hipótesis nula es cierta, y vienen dadas por las siguientes fórmulas:

$$m_U = \frac{n_1 n_2}{2} \quad (3.12)$$

$$\theta_U = \sqrt{\frac{n_1 n_2 (n_1 + n_2 + 1)}{12}} \quad (3.13)$$

Esta prueba estadística es útil cuando las mediciones se pueden ordenar en escala ordinal (es decir, cuando los valores tienden a una variable continua, pero no tienen una distribución normal) y resulta aplicable cuando las muestras son independientes (Farfán, 2013; Farfán, 2014).

La forma de operar con esta prueba es la siguiente:

- Determinar el tamaño de las muestras ( $n_1$  y  $n_2$ ). Si  $n_1$  y  $n_2$  son menores que 20, se consideran muestras pequeñas, pero si son mayores que 20, se consideran muestras grandes.
- Arreglar los datos en rangos del menor al mayor valor. En caso de que existan ligas o empates de rangos iguales, se deberán detectar para un ajuste posterior.
- Calcular los valores de  $U_1$  y  $U_2$ , de modo que se elija el más pequeño para comparar con los críticos de U Mann-Whitney de la tabla de probabilidades asociadas con valores pequeños como los de U en la prueba de Mann-Whitney.
- En caso de muestras grandes, calcular el valor Z, pues en estas condiciones se distribuye normalmente.
- Decidir si se acepta o rechaza la hipótesis.

### 3.7.3. Estadístico Delta de Cliff

El estadístico Delta de Cliff permite cuantificar la magnitud de la diferencia entre dos grupos de observaciones que resultan incompatibles con el presupuesto de normalidad. El análisis de esta cuantificación complementa la interpretación del p-valor asociado a la correspondiente prueba de hipótesis. La aplicación de medidas del tamaño del efecto ha sido promovida, durante

las últimas décadas, tanto por metodólogos, como por instituciones líderes en las ciencias del comportamiento. El propósito de la presente contribución es presentar un programa denominado Cliffs Delta Calculator con capacidad para calcular y graficar el valor del tamaño del efecto no paramétrico para dos grupos de observaciones. Se ofrece una comparación de este programa con otras calculadoras comerciales y no comerciales actualmente disponibles. Se describe su funcionamiento, sus fundamentos matemáticos y se presentan y discuten dos ejemplos de aplicación en la investigación psicológica. Se concluye que este programa presenta algunas ventajas, en comparación con otras calculadoras comerciales disponibles. (Macbeth, G.; Razumiejczyk, E. Ledesma, R. [20]).

### 3.7.4. Clasificador basado en la puntuación

Clasificador basado en la puntuación es un enfoque personalizado para implementar el marco de evaluación que permite un control completo sobre la clasificación. En este enfoque, la función de valor booleano de (1) en la Sección 2.1 toma la forma mostrada en (14).

$$f(r) = \begin{cases} true & \text{if } score(r) \geq score_{ref} \\ false & \text{Otherwise} \end{cases} \quad (3.14)$$

$$score(r) = \sum_{d \in D} h_d(M_d, t_d) \times w_d \quad (3.15)$$

donde:

- $r$  es el repositorio a clasificar.
- $D$  es un conjunto de dimensiones a lo largo del cual se evalúa el repositorio,  $r$ .
- $M_d$  es la métrica que cuantifica la evidencia del repositorio,  $r$ , empleando una cierta práctica de ingeniería de software en la dimensión  $d$ . Por ejemplo, la proporción de líneas de comentario a líneas de origen cuantifica la documentación.
- $t_d$  es un umbral que debe ser satisfecho por la métrica correspondiente,  $M_d$ , para que el repositorio,  $r$ , sea considerado ingeniería en la dimensión  $d$ .
- $h_d(M_d, t_d)$  es una función heurística que evalúa a 1 si el valor métrico,  $M_d$ , satisface el umbral correspondiente,  $t_d$ , 0 en caso contrario.

- $w_d$  es el peso que especifica la importancia relativa de cada dimensión  $d$ .
- $score_{ref}$  es la puntuación de referencia i.e. la puntuación mínima a la que un repositorio debe evaluar para que se considere que contiene un proyecto de software diseñado.

### 3.7.5. Random Forest

Random Forest es un método versátil de aprendizaje automático capaz de realizar tanto tareas de regresión como de clasificación. También lleva a cabo métodos de reducción dimensional, trata valores perdidos, valores atípicos y otros pasos esenciales de exploración de datos. Es un tipo de método de aprendizaje por conjuntos, donde un grupo de modelos débiles se combinan para formar un modelo poderoso.<sup>3</sup>

En Random Forest se ejecutan varios algoritmos de árbol de decisiones en lugar de uno solo. Para clasificar un nuevo objeto basado en atributos, cada árbol de decisión da una clasificación y finalmente la decisión con mayor “votos” es la predicción del algoritmo. Las ventajas que tiene este algoritmo son las siguientes:

- Puede resolver ambos tipos de problemas, es decir, clasificación y regresión, y realiza una estimación decente en ambos casos.
- Unos de los beneficios que más llama la atención es el poder de manejar grandes cantidades de datos con mayor dimensionalidad. Puede manejar miles de variables de entrada e identificar las variables más significativas, por lo que se considera uno de los métodos de reducción de dimensionalidad. Además, el modelo muestra la importancia de la variable, que puede ser una característica muy útil.
- Tiene un método efectivo para estimar datos faltantes y mantiene la precisión cuando falta una gran proporción de los datos.

A su vez la desventaja que tiene este algoritmo es que en ocasiones se puede parecer este algoritmo a una caja negra, ya que se tiene muy poco control sobre lo que hace el modelo. En este caso, en el mejor de los casos, se pueden probar diferentes parámetros y datos aleatorios.

---

<sup>3</sup>AprendeIA (2020). <https://aprendeia.com/aprendizaje-supervisado-random-forest-classification/>. Acceso: 13-02-2020.



# Capítulo 4

## Diseño e implementación

En este trabajo se han utilizado dos formas distintas de implementar la función de valor booleano que se han aplicado y que representan el marco de evaluación dado por la ecuación 3.1. En ambos enfoques, el clasificador basado en la puntuación y el Training Data Sets, un repositorio está representado por las siete dimensiones (cuantificables) que se introdujeron en el capítulo anterior. Como en la sección 3.7.4 ya se ha descrito el clasificador basado en la puntuación, en este capítulo se va a detallar el Training Data Sets.

### 4.1. Training Data Sets

La función de valor booleano que representa el marco de evaluación es esencialmente un clasificador capaz de decidir si un repositorio contiene un proyecto de ingeniería del software o no. El clasificador se entrena utilizando un conjunto de repositorios (llamado training data set) que se han clasificado manualmente como que contienen proyectos de ingeniería del software de acuerdo con alguna definición específica de un proyecto de ingeniería del software.

En el contexto de nuestro estudio, se van a utilizar las dos definiciones siguientes de proyecto de ingeniería del software para demostrar la capacitación del clasificador:

- Organización: se dice que un repositorio contiene un proyecto de ingeniería del software si es similar a los repositorios propiedad de organizaciones populares de ingeniería de software.
- Utilidad: se dice que un repositorio contiene un proyecto de ingeniería del software si es

similar a los repositorios que tienen una utilidad de propósito general para los usuarios. Por ejemplo, un repositorio que contiene un plug-in de Chrome se considera que tiene una utilidad de propósito general, sin embargo, un repositorio que contiene una aplicación móvil desarrollado por un estudiante como un proyecto de curso no puede considerarse que tenga una utilidad de propósito general.

En las subsecciones que siguen, describimos el enfoque utilizado para identificar manualmente repositorios que se ajustan a cada una de las definiciones de un proyecto de ingeniería del software expuestas anteriormente. Estos repositorios componen los respectivos training data sets.

El training data sets se utiliza en el clasificador basado en la puntuación para determinar umbrales,  $t_d$ , y calcular la puntuación de referencia,  $score_{ref}$ . Para todos los repositorios en cada uno de los dos conjuntos de datos de entrenamiento (más los repositorios del conjunto de datos de instancias negativas), recopilamos los siete valores métricos. Los valores atípicos fueron eliminados usando el criterio de Peirce. Para las métricas de valor booleano, 1 (i.e. True) es el umbral. Para todas las demás métricas, se eligió el valor métrico mínimo distinto de cero como umbral correspondiente. El valor umbral correspondiente a cada una de las siete dimensiones, establecidos a partir de los repositorios de cada uno de los dos conjuntos de datos de formación, se muestran en el cuadro 2. En el cuadro 2 también figuran los pesos relativos que se ha utilizado en nuestro clasificador basado en la puntuación. Dichos pesos se pueden cambiar ya que el clasificador es personalizable. Los investigadores pueden alterar los pesos para ejercer un control más fino sobre la clasificación y seleccionar repositorios que se adapten a sus estudios individuales. Además, al decidir los pesos consideramos las limitaciones en la recolección del valor de la métrica asociada de un repositorio de código fuente. Por ejemplo, el enfoque para evaluar un repositorio de código fuente a lo largo de la dimensión comunitaria es más robusto y, por lo tanto, su peso es superior al de la dimensión de ensayo unitario, donde existen limitaciones inherentes debido a nuestro conjunto no exhaustivo de firmas marco. Los pesos y los umbrales del cuadro 2 se utilizaron para calcular las puntuaciones de todos los repositorios de la organización y los conjuntos de datos sobre servicios públicos. La distribución de las puntuaciones se muestra en la Fig. 6. Se determinó que la puntuación de referencia en la organización y utilidad del conjunto de datos sea de 70 y 30, respectivamente. El enfoque del clasificador basado en la puntuación es flexible y permite un control más fino sobre la clasificación. Los pesos permiten



al ejecutor definir explícitamente la importancia de cada dimensión. La expresión (14) puede adaptarse para implementar una variedad de clasificadores diferentes utilizando diferentes conjuntos de dimensiones,  $D$ , y métricas correspondientes, umbrales y pesos. Por ejemplo, si hay una necesidad de construir un clasificador que considere el sesgo de género en la aceptación de contribuciones en la comunidad de código abierto (como en el trabajo de Kofink 2015), uno podría introducir una nueva dimensión, digamos sesgo, definir una métrica para cuantificar el sesgo de género en un repositorio, identificar un umbral apropiado, y ponderar la dimensión en relación con otras dimensiones que puedan ser pertinentes para el estudio.

#### 4.1.1. Organización del conjunto de datos

El proceso de identificación de los repositorios que componen el conjunto de datos es sencillo. El paso preliminar consiste en examinar manualmente los repositorios de organizaciones como Amazon, Apache, Facebook, Google y Microsoft e identificar un conjunto de 1.000 repositorios. El procedimiento para la identificación manual de repositorios ha sido comprobar que el repositorio: (a) tiene licencia de código abierto, (b) usa comentarios para documentar el código, (c) usa integración continua y (d) contiene pruebas unitarias. Algunos ejemplos de repositorios que contienen proyectos de ingeniería del software son los siguientes: scrapy / scrapy, phalcon / incubator, JetBrains / FSharper y owncloud / calendar.

El conjunto de datos de la organización está disponible para su descarga como un archivo CSV (organization.csv) desde GitHub Gist <sup>1</sup>.

#### 4.1.2. Utilidad del conjunto de datos

A diferencia del proceso de identificación de los repositorios que componen el conjunto de datos de la organización, el proceso de identificación de los repositorios que componen el conjunto de datos de utilidad no fue trivial. Los repositorios que componían el conjunto de datos de utilidad fueron identificados mediante la evaluación manual de una muestra aleatoria de los 2.493.701 repositorios que fueron analizados por reaper. De manera similar al proceso de composición del conjunto de datos de la organización, utilizamos un conjunto de pautas para decidir si un repositorio debe ser incluido o no. Las directrices dictaban los diversos aspectos

---

<sup>1</sup>Accesible en <https://gist.github.com/nathanmunaiah/23dba27be17bbd0abc40079411dbf066>

que debían tenerse en cuenta al decidir si un repositorio tenía una utilidad general. Las directrices utilizadas aquí fueron más subjetivas que las utilizadas en el proceso de composición del conjunto de datos de la organización. Algunas de las directrices utilizadas fueron: a) el archivo contiene documentación suficiente para que el proyecto contenido en él pueda utilizarse en un contexto de fines generales; b) repositorio contiene una aplicación o servicio que es utilizado por o tiene el potencial de ser utilizado por personas distintas de los desarrolladores, (c) repositorio no contiene indicaciones que indiquen que el código fuente contenido en puede ser una asignación. El potencial de sesgo fue mitigado por dos autores que evaluaron independientemente la misma muestra aleatoria de repositorios. Los primeros 1.000 repositorios que se han incluido en este trabajo forman el conjunto de datos de la utilidad *set. veg/hyphy*, *seblin/launchit*, *smcameron/opencscad*, y *apanzerj/zit* donde aparecen por ejemplo repositorios incluidos en el conjunto de datos de utilidad que se sabe contienen proyectos de ingeniería del software.

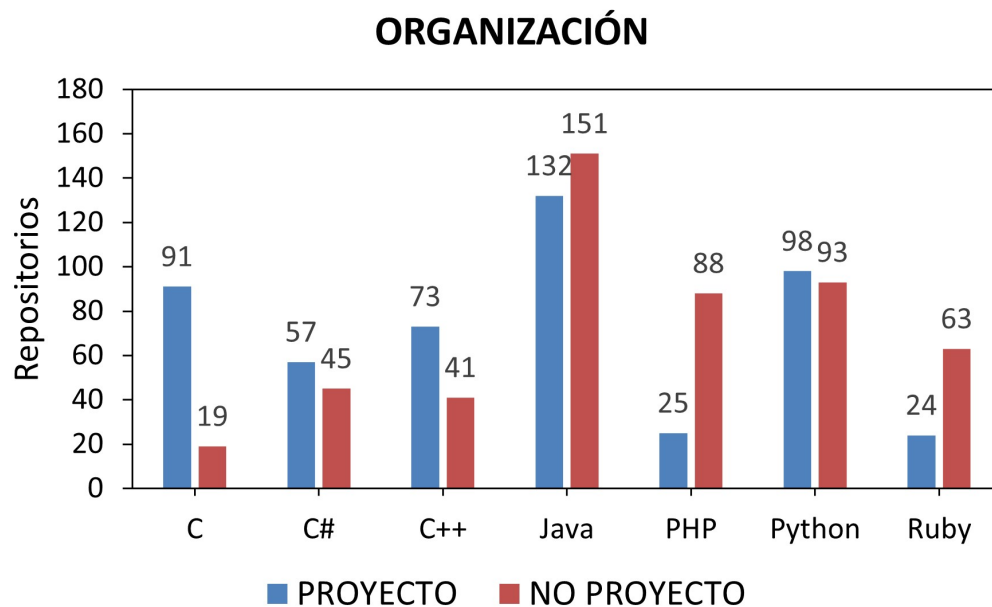


Figura 4.1: Número de repositorios en la organización agrupados por lenguajes de programación.

#### 4.1.3. Datos de instancias negativas

El conjunto de datos de instancias negativas es esencialmente una colección de repositorios que no se ajustan a ninguna de las dos definiciones (p.e. organización y utilidad) de un proyecto

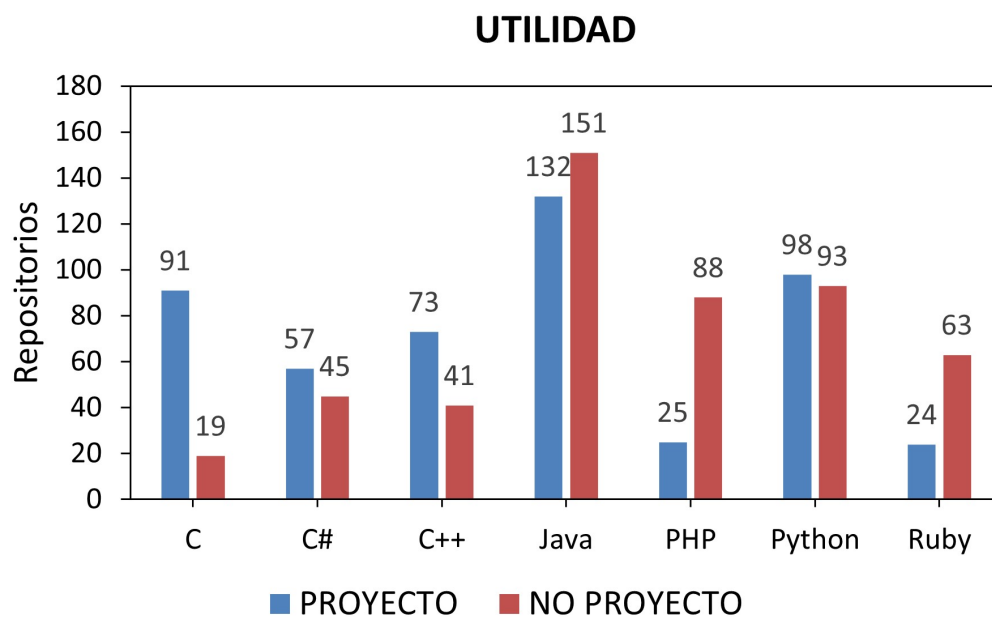


Figura 4.2: Número de repositorios en la utilidad del conjunto de datos agrupados por lenguajes de programación.

de software de ingeniería. Los repositorios que componen el conjunto de datos de instancias negativas se identificaron durante el proceso de composición del conjunto de datos de utilidad.

#### 4.1.4. Resumen de los conjuntos de datos

En esta sección, resumimos los datos de entrenamiento (training data sets) mediante diversas visualizaciones. Los repositorios que componen la organización y los conjuntos de datos de utilidad se etiquetan como “proyecto” y los repositorios que componen el conjunto de datos de instancias negativas se etiquetan como “no proyecto”. En las figuras 4.1 y 4.2 se muestra el número de repositorios de cada tipo (“proyecto” y “no proyecto”) agrupados por lenguaje de programación en la organización y conjuntos de datos de utilidad.

La distribución del número de repositorios de cada tipo (“proyecto” y “no proyecto”) en conjuntos de datos de organización y utilidad se muestran en la figura 4.3.

En las figuras 4.4 y 4.5 se muestran las distribuciones de las siete dimensiones obtenidas de los repositorios en la organización y los conjuntos de datos de utilidad, respectivamente.

He utilizado la correlación de Spearman Rank Co-efficient ( $\rho$ ) para evaluar la correlación entre las diferentes dimensiones de valor continuo, aunque también sirve para valores discre-

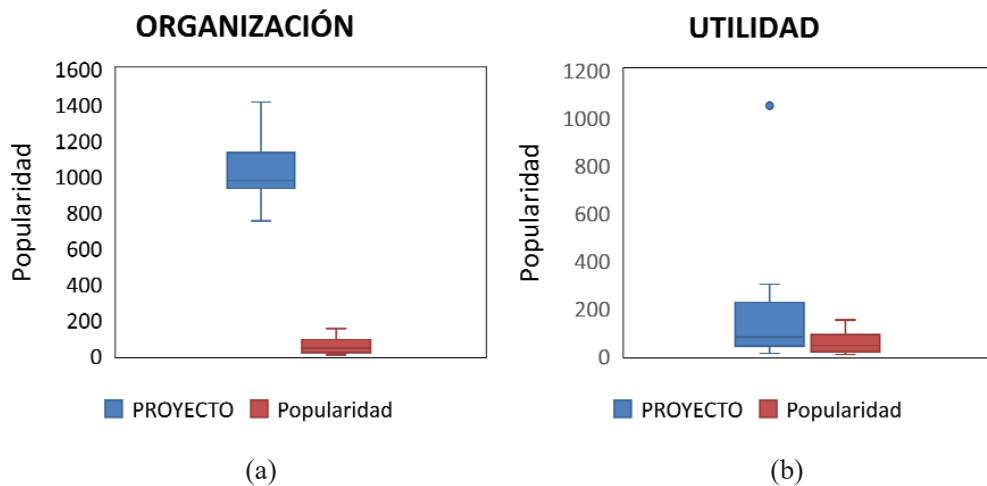


Figura 4.3: Distribución del número de puntuación de los repositorios; (a) en la organización y (b) utilidad del conjunto de datos

tos. En la figura 4.6 se muestran los valores de la  $\rho$  de Spearman, cuando es estadísticamente significativa en el valor  $p$  0.05, entre pares de dimensiones en los conjuntos de datos de organización y utilidad. También se pueden observar en dicha figura las correlaciones que no son estadísticamente significativas y se han representado con un guión. Como se ve en la figura 4.6, hay una correlación de moderada a fuerte entre las diversas dimensiones en el conjunto de datos de la organización. El tamaño del repositorio, en particular, está al menos moderadamente correlacionado con dos de las tres dimensiones de valor continuo. Para evaluar la relación entre el tamaño del repositorio y las dimensiones de valor binario (integración continua y licencia) se ha utilizado la prueba no paramétrica de Mann-Whitney-Wilcoxon (MWW) y el tamaño del efecto  $\eta^2$  de Cliff. En la figura 4.6 también se muestran los resultados del análisis de asociación en los conjuntos de datos de organización y utilidad. Como se ve en dicha figura, a excepción de la dimensión de integración continua en el conjunto de datos de utilidad, el tamaño del repositorio es estadísticamente significativo ( $p$ -valor  $\leq 0.05$ ) asociado con las dimensiones de valor binario con tamaño de efecto medio. Los resultados de la asociación indican que es más probable que un repositorio más grande tenga una integración y/o licencia continuas. Sin embargo, creemos que dicha relación puede crear algo de confusión, ya que algunas organizaciones pueden tener una licencia en todos los repositorios. Además, como señala Rosenberg (1997), la utilidad de SLOC como métrica es como covariable de otras métricas.

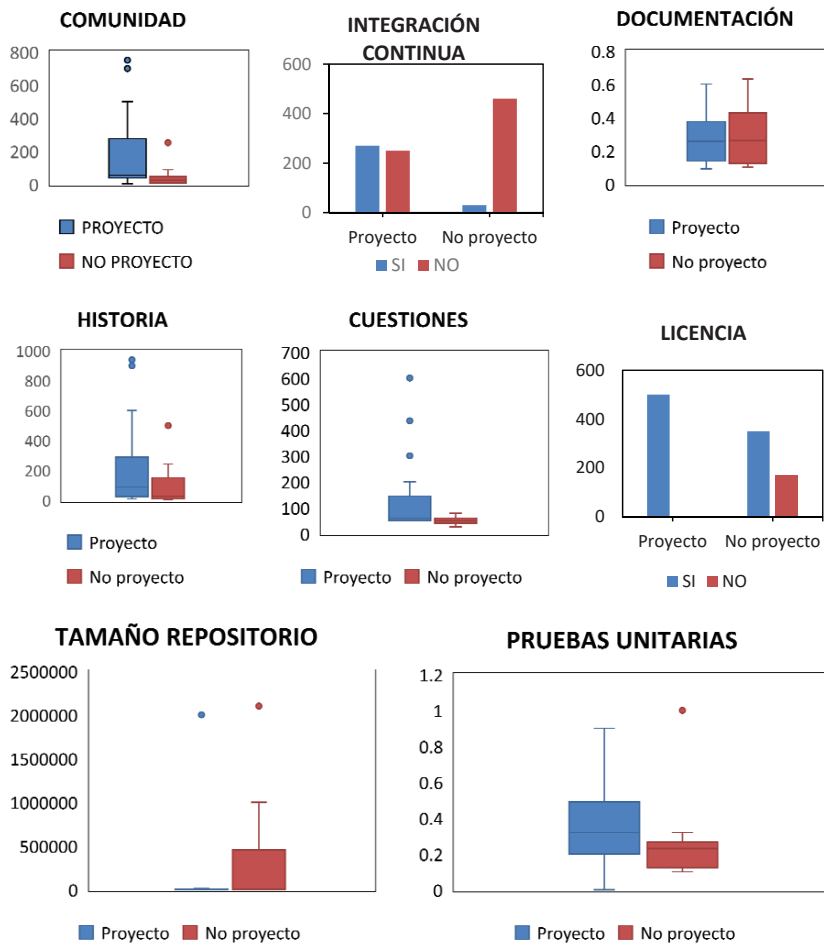


Figura 4.4: Distribución de las dimensiones de repositorios en el conjunto de datos de la organización.

También se ha tenido en cuenta la correlación entre el tamaño del repositorio y las diversas dimensiones propuestas en nuestro estudio al incluir el tamaño del repositorio como una característica tanto en los clasificadores basados en puntajes como en Random Forest.

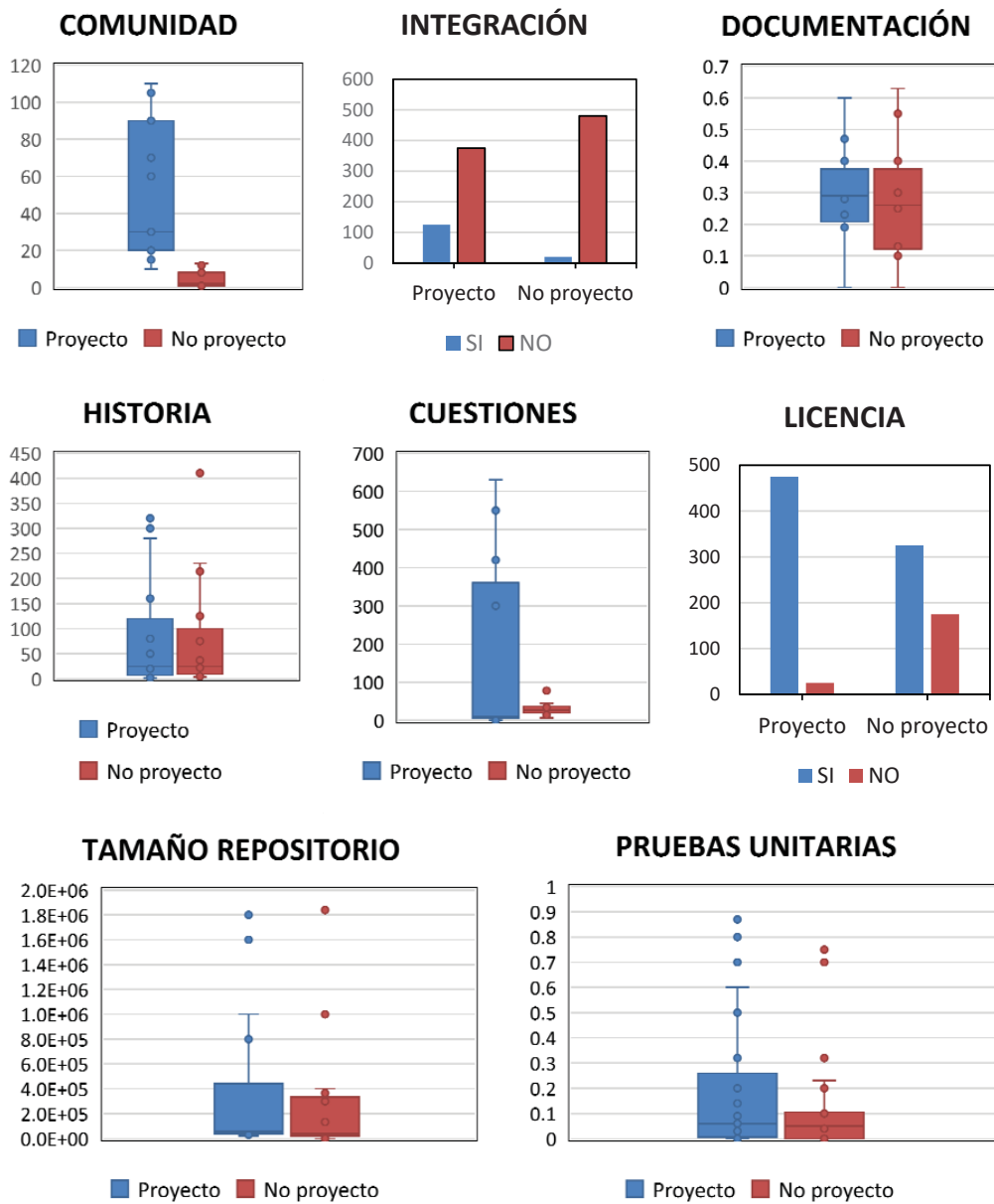


Figura 4.5: Distribución de las dimensiones obtenidas de los repositorios en el conjunto de datos de utilidad.

	Organización					
Pruebas Unitarias						1.00
Tamaño del Repositorio					1.00	0.1429
Cuestiones				1.00	0.2381	0.2619
Historia			1.00	0.5000	0.0952	0.3571
Documentación		1.00	0.5714	0.4286	-	0.1429
Comunidad	1.00	-	-	-	0.6190	0.1429
	Comunidad	Documentación	Historia	Cuestiones	Tamaño del Repositorio	Pruebas Unitarias

(a)

	Utilidad					
Pruebas Unitarias						1.00
Tamaño del Repositorio					1.00	-
Cuestiones				1.00	0.3333	0.0714
Historia			1.00	0.0476	0.5952	-
Documentación		1.00	-	0.5238	0.0476	-
Comunidad	1.00	0.5476	0.2381	0.6905	0.0476	-
	Comunidad	Documentación	Historia	Cuestiones	Tamaño del Repositorio	Pruebas Unitarias

(b)

Figura 4.6:  $\rho$  de Spearman entre pares de dimensiones en la organización (a) y conjuntos de datos de utilidad (b) con - (guión) representando las correlaciones estadísticamente insignificantes.





# Capítulo 5

## Resultados

En este capítulo se presentan los resultados de la validación de los clasificadores y los resultados de la aplicación de los clasificadores para identificar (o predecir) proyectos de software de ingeniería en una muestra de 2.316.524 de los 2.634.807 repositorios de GitHub que estaban activos en el momento en que se realizó el análisis. Como se han utilizado dos clasificadores (basado en la puntuación y Random Forest) y dos conjuntos de datos diferentes (organización y utilidad), el análisis de validación y predicción se ha repetido cuatro veces.

### 5.1. Validación

En esta sección, se presenta el enfoque y los resultados de la validación de los clasificadores basados en la puntuación y Random Forest entrenados con conjuntos de datos de organización y utilidad. La validación se ha realizado en un conjunto de 300 repositorios llamado conjunto de validación, para el que se estableció manualmente la verdadera clasificación. Se consideró la validación desde dos perspectivas: interna, en la que se validaba el rendimiento de los propios clasificadores, y externa, en la que se comparaba el rendimiento de los clasificadores con el de un esquema de clasificación que utilizaron Ray y colaboradores como criterio (Ray et al. 2014), como por ejemplo, el tamaño del proyecto, tamaño del equipo y tamaño del commit. Se utilizó tasa de falsos positivos (FPR), tasa de falsos negativos (FNR), precisión, memoria y F-medida para evaluar el rendimiento de la clasificación.

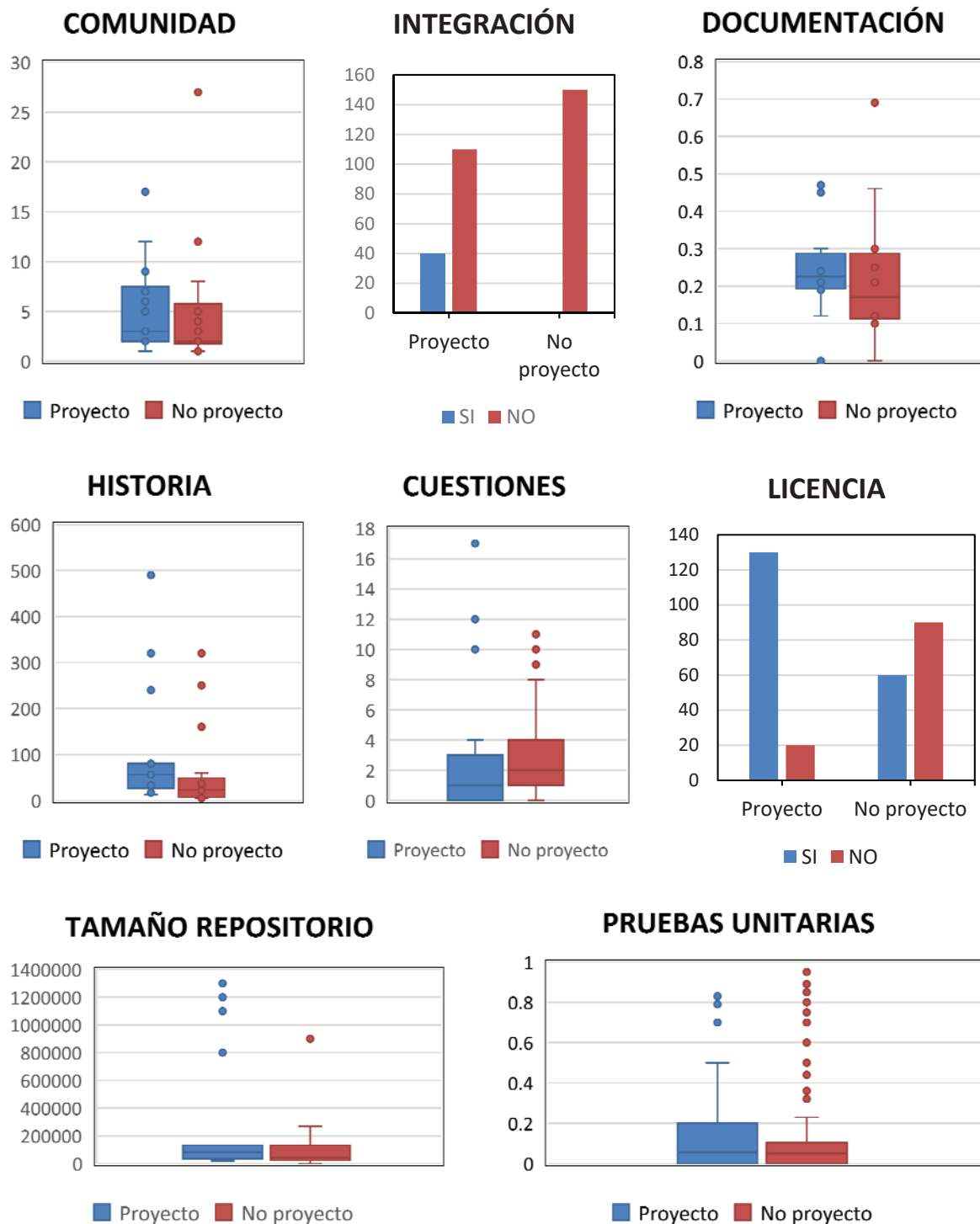


Figura 5.1: Distribución de las dimensiones de los repositorios en el conjunto de validación.

CLASIFICADOR	FPR	FNR	PRECISIÓN	LLAMADA	F-MEDIDA
BASADO EN PUNTUACIÓN	20 %	37 %	73 %	60 %	66 %
RANDOM FOREST	5 %	58 %	85 %	43 %	57 %

Cuadro 5.1: Resultados de los clasificadores Basado en Puntuación y Random Forest testeados con los datos de organización

### 5.1.1. Establecer la verdad fundamental

La evaluación del trabajo de cualquier clasificador normalmente implica el uso del clasificador para evaluar un conjunto de muestras para las que se conoce la clasificación de la verdad fundamental. En líneas similares, para evaluar el trabajo de los clasificadores basados en la puntuación y Random Forest, compusimos manualmente un conjunto de 300 repositorios, de los cuales se sabe a ciencia cierta que 150 de ellos tienen ingeniería del software y los restantes no.

En la figura 5.1 se muestra la distribución de las siete dimensiones recogidas de los repositorios en el conjunto de validación. Como se ve en la figura, los repositorios que contienen proyectos de software de ingeniería tienden a tener valores medianos más altos en casi todas las dimensiones.

### 5.1.2. Validación interna

En este tipo de validación, se evalúa el trabajo de los clasificadores basados en la puntuación y de Random Forest que han sido entrenados mediante la utilización de los conjuntos de datos de organización y utilidad en el contexto del conjunto de validación. Conjunto de datos de organización. En este estudio se muestra el trabajo de los clasificadores basados en la puntuación y Random Forest entrenados con un conjunto de datos de organización.

Claramente, en el cuadro 5.1 se puede apreciar que el clasificador basado en la puntuación tiene mejores resultados que el clasificador Random Forest en términos de medida F. Si se desea una tasa de falsos positivos más baja, el clasificador Random Forest puede ser más adecuado ya que tiene una tasa de falsos positivos considerablemente más baja que el clasificador basado en la puntuación. Ahora se muestran algunos ejemplos de repositorios del conjunto de datos de la organización que fueron clasificados erróneamente por clasificadores basados en puntajes y

Random Forest.

1. Falsos positivos: se pueden dar casos en los que aparentemente el repositorio evaluado supere el umbral de casi todas las dimensiones, pero que realmente no sea ingeniería del software. Munaiah y colaboradores (2017) ponen un ejemplo muy ilustrativo: `software-engineering-amsterdam/sea-of-ql`. Comprobaron que, aunque el repositorio recibió una puntuación de 95 (cerca de una puntuación perfecta de 100), no tenía una licencia real, sino que se había identificado una licencia en archivos de biblioteca que pueden haber sido incluidos en el repositorio de código fuente y que consistía en un espacio de colaboración donde todos los estudiantes matriculados en un curso en particular desarrollan sus proyectos de software individuales.
2. Falsos negativos: se pueden producir cuando un desarrollador construye un repositorio sin utilizar varias de las dimensiones requeridas en nuestro análisis. Así, puede no conseguir el umbral mínimo (70) para considerarse ingeniería por no haber utilizado todas las dimensiones.

Conjunto de datos de utilidad. En este estudio se muestra el rendimiento de los clasificadores basados en la puntuación y Random Forest entrenados con un conjunto de datos de utilidad.

CLASIFICADOR	FPR	FNR	PRECISIÓN	LLAMADA	F-MEDIDA
BASADO EN PUNTUACIÓN	76 %	1 %	52 %	98 %	68 %
RANDOM FOREST	20 %	17 %	80 %	86 %	83 %

Cuadro 5.2: Resultados de los clasificadores Basado en Puntuación y Random Forest testeados con los datos de utilidad.

Claramente, el modelo de Random Forest funciona mejor que el modelo basado en la puntuación. Como se puede observar en el cuadro 5.2 hay una gran tasa de falsos positivos del clasificador basado en la puntuación. La gran tasa de falsos positivos indica que el clasificador puede haber clasificado casi todos los repositorios como que contienen un proyecto de software de ingeniería. Igual que vimos anteriormente, en el conjunto de datos de utilidad también se pueden dar falsos positivos y negativos (Munaiah et al., 2017).

UMBRAL	FPR	FNR	PRECISIÓN	LLAMADA	F-MEDIDA
1.000	0 %	100 %	NA	0 %	NA
500	0 %	100 %	NA	0 %	NA
50	0 %	85 %	100 %	16 %	28 %
10	1 %	64 %	94 %	30 %	45 %

Cuadro 5.3: Rendimiento del clasificador basado en la popularidad (puntuación de estrellas) en función del umbral mínimo exigido.

### 5.1.3. Validación externa

En esta perspectiva de validación, el rendimiento de los clasificadores basados en la puntuación y Random Forest se compara con el de los clasificadores basados en citas utilizados en la literatura anterior (Ray et al. 2014). Anteriormente señalamos que la popularidad de un repositorio es un criterio potencial para identificar un conjunto de datos para estudios de investigación. La intuición es que los repositorios populares contendrán software real que a la gente le gusta y usa (Jarczyk et al. 2014). Por ejemplo, los artículos de Ray et al. (2014) sobre lenguajes de programación y calidad de código, y Guzmán et al. (2014) sobre el análisis de sentimiento de comentarios de compromiso, utilizan el número de citas como forma de seleccionar proyectos para sus estudios. Estos documentos utilizan los proyectos con estrellas en varios idiomas, que están destinados a ser extremadamente populares. El repositorio mongodb/mongo utilizado en el conjunto de datos por Ray et al. (2014), por ejemplo, tiene más de 8.927 estrellas.

En este estudio se muestra el resultado de clasificadores basados en la puntuación y Random Forest usando conjuntos de datos de organización y utilidad, respectivamente. Ahora usamos el clasificador basado en estrellas para clasificar los repositorios del conjunto de validación. Al usar un clasificador basado en estrellas, Ray et al. (2014) ordenaron y seleccionaron los 50 repositorios principales en cada uno de los 19 idiomas populares. Se ha aplicado el mismo esquema de filtrado a una muestra de 2.316.524 repositorios GitHub y se ha establecido el número mínimo de popularidad en 1.000. En otras palabras, un repositorio se clasifica como que contiene un proyecto de software de ingeniería (basado en la popularidad) si tiene 1.000 o más estrellas. También se evaluaron otros umbrales (500, 50 y 10) para la popularidad. En los casos en que el clasificador no produjo clasificaciones positivas (p.ej. tanto verdadero positivo como falso positivo son ceros), la precisión y la F-medida no pueden calcularse.

DATOS	CLASIFICADOR	Nº REPOSITORIOS	PORCENTAGE
-2*ORGANIZACIÓN	BASADO EN PUNTUACIÓN	244.624	10,56 %
	RANDOM FOREST	145.246	6,23 %
	BASADO EN PUNTUACIÓN	1.645.196	71,02 %
-2*UTILIDAD	RANDOM FOREST	544.615	23,51 %

Cuadro 5.4: Número de repositorios que contienen un proyecto de software de ingeniería en función de los clasificadores utilizados con los datos pertenecientes a organización y utilidad.

Como se observa en el cuadro 5.3, con un umbral elevado (1.000 y 500) el clasificador basado en popularidad clasifica erróneamente todos los repositorios que contienen proyectos informáticos de ingeniería. A medida que bajamos el umbral, el rendimiento mejora. La limitación más llamativa del clasificador basado en popularidad son los bajos porcentajes de memoria. Mientras que un repositorio con un gran número de estrellas es probable que contenga un proyecto de software diseñado, lo contrario no siempre es cierto. Los resultados de la validación indican que, al utilizar el clasificador basado en popularidad, se pueden estar excluyendo un gran conjunto de repositorios que contienen proyectos de software de ingeniería pero que pueden no ser populares. Por el contrario, los clasificadores basados en la puntuación y Random Forest que están orientados a la organización y utilidad de los datos, funcionan mucho mejor en términos de memoria y logran un nivel aceptable de precisión.

## 5.2. Predicción

En esta sección, se presentan los resultados de la aplicación de los clasificadores basados en la puntuación y Random Forest para identificar proyectos de software de ingeniería en una muestra de 2.316.524 repositorios GitHub. En el cuadro 5.4 se pueden observar el número de repositorios tanto de los datos de Organización como de Utilidad que tienen un proyecto de software de ingeniería cuando se usan los clasificadores Random Forest y el Basado en Puntuación. En dicha tabla se puede observar que el número de repositorios que tienen una utilidad de propósito general obtenido por el clasificador basado en la puntuación es considerablemente alto, lo cual podría deberse al número bajo de proyectos analizados del conjunto de datos utilizados (Munaiah et al., 2017). Una agrupación de resultados más detallada se puede obser-

var en las figuras 5.2 y 5.3, donde se pueden ver el número de repositorios por lenguaje de programación para los dos conjuntos de datos (organización y utilidad).

## 5.3. Discusión

En este estudio, se ha intentado identificar los repositorios que contienen proyectos de software de ingeniería de acuerdo con dos definiciones diferentes del término. La aplicación de una de las definiciones incluía la capacitación de dos clasificadores que utilizaban repositorios en el conjunto de datos de la organización. Se podría suponer que el resultado de la aplicación de estos clasificadores puede ser igualado porque todos los repositorios de cualquier organización en GitHub contienen un proyecto de software de ingeniería. Sin embargo, esto no siempre es así.

El conjunto de validación contiene 300 repositorios de los cuales 90 pertenecen a organizaciones. Como ya se vió, se eligieron 150 que contienen proyectos de software de ingeniería y los 150 restantes no contienen proyectos de software de ingeniería.

En la figura 5.4 se muestra una comparación entre la distribución de las siete dimensiones recogidas de repositorios propiedad de organizaciones, pero con diferentes etiquetas de clasificación manual. Como se puede ver en esta figura, la diferencia en la distribución de las dimensiones proporciona pruebas cualitativas que respaldan la idea de que no todos los repositorios de propiedad de las organizaciones son similares entre sí. En líneas similares, comparamos la distribución de las siete dimensiones recogidas de repositorios conocidos por contener proyectos de software de ingeniería, pero con el subgrupo de organizaciones y usuarios.

La comparación se muestra en la figura 5.5, donde las medianas de la mayoría de las dimensiones son comparables entre los repositorios de propiedad de los usuarios y los de propiedad de las organizaciones. En este estudio se puede observar, un número considerable de repositorios clasificados como que contienen proyectos de software de ingeniería que son propiedad de usuarios individuales. Por otra parte, un número considerable de repositorios clasificados como que no contenían un proyecto de software de ingeniería eran propiedad de organizaciones. El filtrado de repositorios basado únicamente en que el propietario es una organización puede dar lugar a la exclusión de repositorios potencialmente pertinentes, de propiedad de los usuarios, o a la inclusión de repositorios que pueden no contener proyectos informáticos de ingeniería o

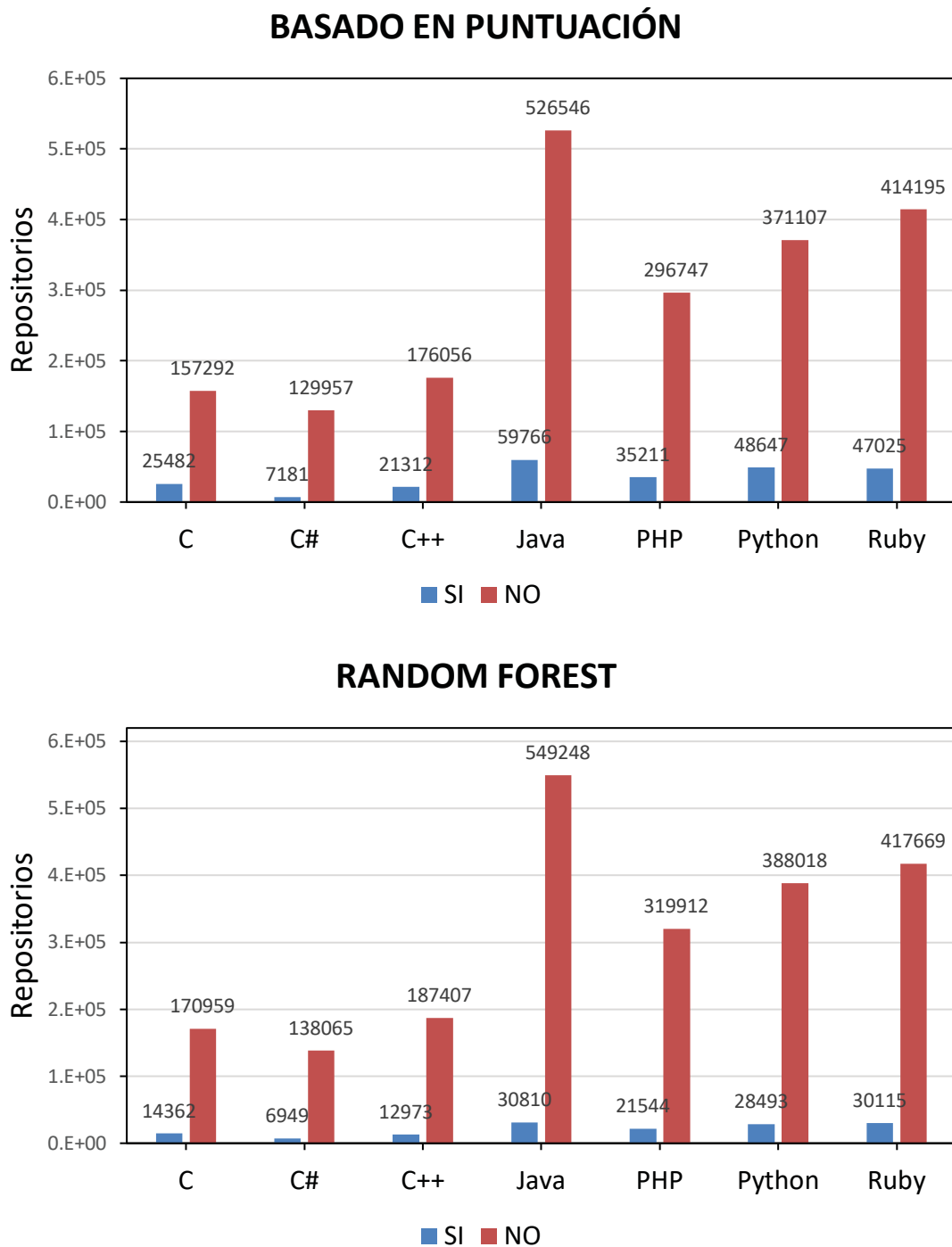


Figura 5.2: Número de repositorios obtenidos por los clasificadores basados en la puntuación y Random Forest agrupados por lenguajes de programación (ORGANIZACIÓN).



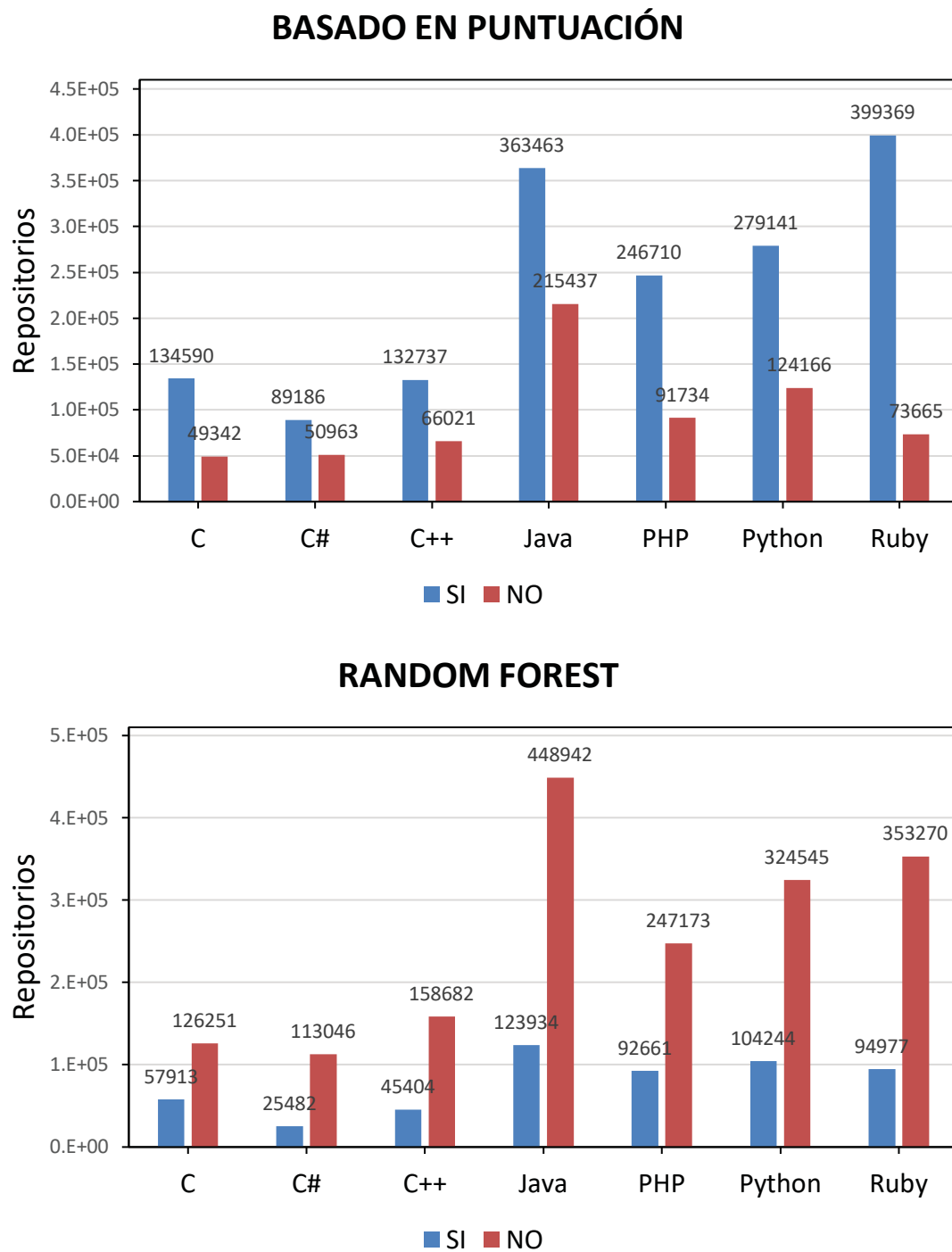


Figura 5.3: Número de repositorios obtenidos por los clasificadores basados en la puntuación y Random Forest agrupados por los lenguajes de programación (UTILIDAD).

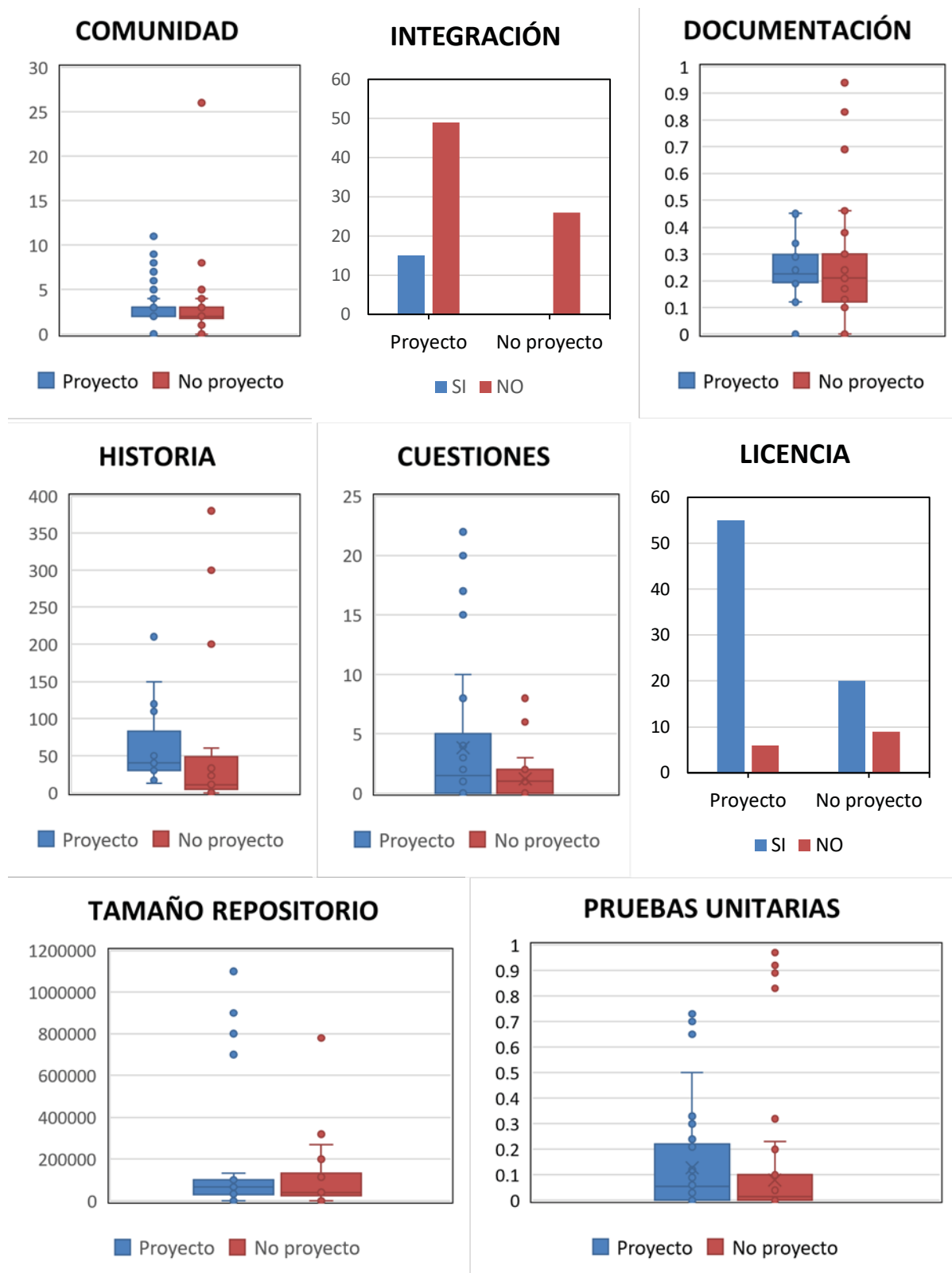


Figura 5.4: Comparación de la distribución de las dimensiones de los repositorios con diferentes etiquetas de clasificación manual pero todas propiedad de organizaciones.

ambos.

## 5.4. Puntos débiles del estudio

Las dimensiones utilizadas para representar repositorios de código fuente en el modelo de clasificación son subjetivas. Además de las dimensiones, los umbrales y pesos utilizados en el clasificador basado en la puntuación también son subjetivos. Aunque creemos que las ponderaciones y las dimensiones que se han utilizado son aceptables en el contexto de este estudio, sin embargo, se pueden utilizar esquemas de ponderación alternativos para mitigar parte de la subjetividad empleada. Algunos de estos enfoques alternativos se pueden orientar en el uso de algoritmos de aprendizaje automáticos para evaluar la importancia de las dimensiones utilizando repositorios en un conjunto de datos de entrenamiento, una ponderación uniforme entre dimensiones, o un esquema de ponderación basado en la popularidad.

Al describir las dimensiones medidas por Reaper en la Sección 3.6, se describen las limitaciones en dicho estudio para recopilar la métrica de dimensiones de un repositorio. Estas limitaciones pueden llevar a la inducción de sesgos en los repositorios seleccionados.

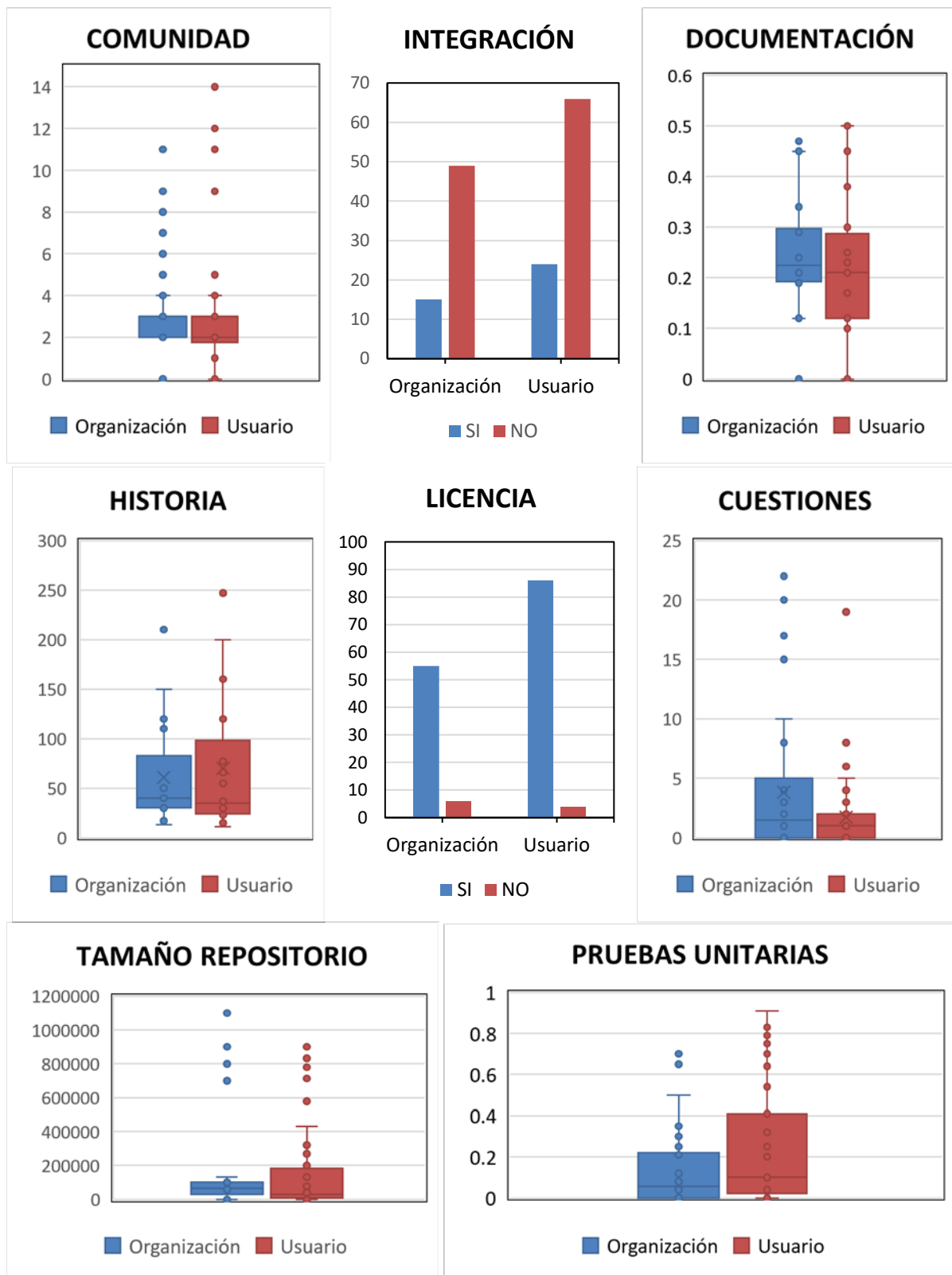


Figura 5.5: Comparación de la distribución de las dimensiones de los repositorios que contienen proyectos informáticos de ingeniería propiedad de organizaciones y usuarios.

# Capítulo 6

## Conclusiones

### 6.1. Consecución de objetivos

El objetivo principal de este trabajo era entender los elementos que constituyen un proyecto de software de ingeniería para poder identificar en repositorios de GitHub dichos proyectos de los que no son de ingeniería. Se han propuesto siete elementos, llamados dimensiones: comunidad, integración continua, documentación, historia, cuestiones, licencias y pruebas unitarias, para poder identificar dichos proyectos y se han realizado dos conjuntos de repositorios, cada uno de los cuales correspondía a una definición diferente de un proyecto de software de ingeniería que estaba formado y entrenado por el clasificador basado en la puntuación y por el clasificador Random Forest.

Aquí tengo que puntualizar que se han logrado tanto el objetivo principal, como los específicos planteados. Los clasificadores se utilizaron para identificar todos los repositorios de la muestra de 2.316.524 repositorios de GitHub que eran similares a los que se ajustan a las definiciones del proyecto de software de ingeniería. Nuestro modelo Random Forest ha dado el mejor resultado: predijo que el 23,51 % de 2.316.524 repositorios GitHub contienen proyectos de software de ingeniería.

Aunque el final ha sido bueno, tengo que decir que el proceso ha sido complicado, ya que, dado el gran volumen de repositorios tratados, conseguir abrir el fichero de datos ha sido difícil, así como relacionar los distintos archivos disponibles entre sí, ya que venían todos sin la cabecera y ésta ha sido necesario construirla en base a un archivo pdf explicativo en forma de diagrama disponible para los usuarios.

## 6.2. Aplicación de lo aprendido

Para la realización de este trabajo me han sido de gran ayuda varias asignaturas del Grado en Ingeniería en Tecnologías de las Telecomunicaciones tales como Estadística, ya que la base de los métodos estadísticos utilizados la he aprendido en dicha asignatura. Por otro lado, el análisis de los repositorios se ha llevado a cabo en Python, con lo que la asignatura de Servicios y Aplicaciones Telemáticas cursada en Grado me ha resultado imprescindible para el desarrollo de todo el trabajo. Además, la asignatura Ingeniería de Sistemas de Información me ha resultado muy útil en el manejo de la base de datos.

## 6.3. Lecciones aprendidas

En este trabajo he aprendido que Excel no permite abrir documentos con un número de filas superior a 1.048.576, lo cual me ha complicado en exceso el tratamiento de los datos. Además, para el análisis de estos he tenido que aprender a manejar la distribución Anaconda Phyton y la aplicación Jupyter Notebook, que es una aplicación web que sirve a modo de puente constante entre el código y los textos explicativos.

## 6.4. Futuros trabajos

Como ya adelanté al final del capítulo anterior, tanto las dimensiones utilizadas para representar repositorios de código fuente en el modelo de clasificación, como los umbrales y pesos utilizados en el clasificador basado en la puntuación son subjetivos. Aunque el resultado es razonablemente bueno, en futuros trabajos se podrían utilizar esquemas de ponderación alternativos para mitigar parte de esa subjetividad. Además, sería bueno probar otros enfoques alternativos como el uso de algoritmos de aprendizaje automáticos para evaluar la importancia de las dimensiones utilizando repositorios en un conjunto de datos de entrenamiento, una ponderación uniforme entre dimensiones, o un esquema de ponderación basado en la popularidad.

# Bibliografía

- [1] S. W. Ambler. Agile/lean documentation, May 2020.  
<http://www.agilemodeling.com/essays/agileDocumentation.htm>.
- [2] N. Anquetil, K. M. Oliveira, A. dos Santos, P. da Silva jr, L. C. de Araujo jr, and S. Vieira. A tool to automate re-documentation. In *Forum of the CAISE, Conference on Advanced Information Systems Engineering (CAiSEâ05)*, 2005.
- [3] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. *ACM Sigplan Notices*, 45(6):363–375, 2010.
- [4] F. A. Cioch, M. Palazzolo, and S. Lohrer. A documentation suite for maintenance programmers. In *1996 Proceedings of International Conference on Software Maintenance*, pages 286–286. IEEE Computer Society, 1996.
- [5] A. Danial. Cloc—count lines of code. *Open source*, 2009.
- [6] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75, 2005.
- [7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.
- [8] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 107–114, 1992.

- [9] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431. IEEE, 2013.
- [10] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.
- [11] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, 2002.
- [12] S. Goodman, P. Wolcott, and G. Burkhart. *Building on the basics: an examination of high-performance computing export control policy in the 1990s*. Center for International Security & Cooperation, 1995.
- [13] G. Gousios. The ghtorrent dataset and tool suite. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 233–236. IEEE, 2013.
- [14] G. Gousios, M. Pinzger, and A. Van Deursen. An exploration of the pull-based software development model. In *ICSE*, volume 10, pages 2568225–2568260, 2013.
- [15] P. Grubb and A. A. Takang. *Software maintenance: concepts and practice*. World Scientific, 2003.
- [16] S. Landau. Standing the test of time: The data encryption standard. *Notices of the AMS*, 47(3):341–349, 2000.
- [17] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.
- [18] J. Lerner and J. Tirole. Some simple economics of open source. *The journal of industrial economics*, 50(2):197–234, 2002.
- [19] M. Lindvall, V. Basili, B. Boehm, P. Costa, K. Dangle, F. Shull, R. Tesoriero, L. Williams, and M. Zelkowitz. Empirical findings in agile methods. In *Conference on extreme programming and agile methods*, pages 197–207. Springer, 2002.



- [20] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma. Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10(2):545–555, 2011.
- [21] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd international conference on Software engineering*, pages 263–272, 2000.
- [22] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.
- [23] C. C. Munaiah N, Kroh S and N. M. Home of the reporeapers, Feb. 2020.  
<https://reporeapers.github.io>.
- [24] N. Nagappan, L. Williams, J. Osborne, M. Vouk, and P. Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. In *16th IEEE international symposium on software reliability engineering (ISSRE'05)*, pages 10–pp. IEEE, 2005.
- [25] Nextu. ¿qué es github?, Feb. 2020.  
<https://www.nextu.com/blog/que-es-github>.
- [26] S. L. Pfleeger and J. M. Atlee. *Software engineering: theory and practice*. Pearson Education India, 1998.
- [27] V. Phoha. A standard for software documentation. *Computer*, 30(10):97–98, 1997.
- [28] T. M. Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.
- [29] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.
- [30] R. S. Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.
- [31] V. Rajlich. Incremental redocumentation using the web. *IEEE Software*, 17(5):102–106, 2000.

- [32] E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.
- [33] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *European Conference on Object-Oriented Programming*, pages 52–78. Springer, 2011.
- [34] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.
- [35] A. Sirera Martínez. *Trabajo fin de grado: Estudio sobre uso de Big Data en pymes*. Universitat Oberta de Catalunya, 2015.
- [36] M. D. Syer, M. Nagappan, A. E. Hassan, and B. Adams. Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 283–297, 2013.
- [37] V. M. TELES. Extreme programming: aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade novatec editora. *São Paulo*, 2004.
- [38] B. Thomas and S. Tilley. Documentation for software engineers: what is needed to aid system understanding? In *Proceedings of the 19th annual international conference on Computer documentation*, pages 235–236, 2001.
- [39] S. Tilley and H. Müller. Info: a simple document annotation facility. In *Proceedings of the 9th annual international conference on Systems documentation*, pages 30–36, 1991.
- [40] S. R. Tilley. Documenting-in-the-large vs. documenting-in-the-small. In *CASCON*, pages 1083–1090. Citeseer, 1993.
- [41] S. R. Tilley, H. A. Müller, and M. A. Orgun. Documenting software systems with views. In *Proceedings of the 10th annual international conference on Systems documentation*, pages 211–219, 1992.

- [42] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 1–1. IEEE, 2007.
- [43] J. Whitehead, I. Mistrík, J. Grundy, and A. Van der Hoek. Collaborative software engineering: concepts and techniques. In *Collaborative Software Engineering*, pages 1–30. Springer, 2010.
- [44] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. Mining software repositories to study co-evolution of production & test code. In *2008 1st international conference on software testing, verification, and validation*, pages 220–229. IEEE, 2008.
- [45] C. Zapponi. Github-programming languages and github, 2016.
- [46] H. Zhu, P. A. Hall, and J. H. May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.