UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Servicio de análisis de cambios en repositorios de código para la identificación de fallos potenciales

Diego Fermín Sanz Alonso

Mayo de 2019

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Servicio de análisis de cambios en repositorios de código para la identificación de fallos potenciales

Autor: Diego Fermín Sanz Alonso

Directores: Pablo Bermejo López

Luis de la Ossa Jiménez

Mayo de 2019

Resumen

Durante el desarrollo de una aplicación software en equipo, es muy común utilizar servicios de control de versiones como GitHub o BitBucket. Cuando un miembro del equipo aporta un incremento en alguna de las ramas del repositorio, esto lanza un proceso automático de Integración (CI) y Prueba Continua (CT). Cuando el proceso de build en CI o durante las pruebas en CT es posible que se produzca un error, entonces decimos que la pipeline se ha roto, y todo el equipo debe parar su trabajo y no enviar ningún incremento nuevo al repositorio hasta que el programador arregle el código que ha roto el proceso de CI o CT.

Este TFM asume que algunos de los commits de código que acaban rompiendo el pipeline pueden tener características que los diferencian de los cambios que no, como por ejemplo el número de archivos actualizados, o el Volumen de código añadido. Por ello, se pretende mostrar al usuario la información más relevante para la detección de fallos, a modo de advertencia, antes de enviar el nuevo incremento al repositorio compartido. Así, se podría evitar alertar o molestar a todo el equipo involucrado.

A partir de todo ello se desplegará un servicio al que el usuario pueda subir la última versión de su repositorio local y compararlo con el último push realizado al repositorio compartido online, de manera que sepa si dicho commit es probable que falle durante el build y las pruebas o no.

En este trabajo se explicarán todas las fases de creación del servicio, desde la recogida y el análisis de los datos hasta la implementación y el despliegue del servicio en la nube. Además, durante el desarrollo del proyecto se seguirá una metodología Scrum y se cumplirá con la Primera Vía de la filosofía DevOps, buscando entregar valor al cliente lo más rápido posible.

Agradecimientos

Agradecimientos

Índice de contenido

Índice de figuras

CAPÍTULO 1
INTRODUCCIÓN

1.1 MOTIVACIÓN

Durante el desarrollo de una aplicación software en equipo, es muy común utilizar servicios de control de versiones como GitHub[1] o BitBucket[2]. A partir de estos repositorios, el equipo de desarrollo puede obtener una copia con el código en un momento determinado y trabajar sobre él, para posteriormente subir sus cambios al repositorio remoto, con el fin de que los demás tengan acceso a dichos cambios. Sin embargo, en función de la complejidad del proyecto, el tamaño del equipo y la agilidad que se quiera llevar en el proyecto, la integración del código mediante las técnicas tradicionales puede conllevar conflictos y generar problemas que supongan tiempo de desarrollo perdido.

Por ello, para evitar estos contratiempos y a la vez asegurar la calidad del código, se realiza lo que se conoce como Integración Continua [1], que consiste en que, cada vez que un desarrollador añada una funcionalidad nueva, esta se compile, se realice el build y se ejecuten un conjunto de pruebas automáticas con el fin de determinar si el funcionamiento de todos los módulos del proyecto es el esperado, de manera que siempre se esté trabajando sobre una versión estable del proyecto.

Si se sigue esta filosofía, cuando un miembro del equipo aporta un incremento en alguna de las ramas del repositorio, esto lanza un proceso automático de Integración (CI) y Prueba Continua (CT). Cuando se lanza el proceso de build en CI o durante las pruebas en CT es posible que se produzca un error, entonces decimos que la pipeline se ha roto, y todo el equipo debe parar su trabajo y no enviar ningún incremento nuevo al repositorio hasta que el programador arregle el código que ha roto el proceso de CI o CT.

Este TFM asume que algunos de los commits de código que acaban rompiendo el pipeline pueden tener características que los diferencian de los cambios que no, como por ejemplo el número de archivos actualizados, o el volumen de código añadido [2]. Por ello,

[1] https://github.com/
[2] https://bitbucket.org/

se pretende mostrar un análisis descriptivo de las clases modificadas en un commit para que el usuario pueda decidir a raíz de estos datos si prefiere volver a revisar su código o si detecta, por ejemplo, que no cumple con los estándares de calidad del equipo. Asimismo, vamos a analizar si es posible crear un modelo que aprenda y pueda avisar al programador de estos 'commits smells', a modo de advertencia, antes de enviar el nuevo incremento al repositorio compartido. Así, se podría evitar alertar o molestar a todo el equipo involucrado.

Diversos artículos que apoyan esta teoría suelen basarse en entrevistas o encuestas para obtener feedback sobre los resultados de los commits [3]. En nuestro caso, utilizaremos la información de los propios commits, realizando un análisis estático del código modificado para poder extraer sus características y predecir si es probable que tenga fallos.

A partir de todo ello desplegará un servicio al que el usuario pueda subir la última versión de su repositorio local y compararlo con el último *push* realizado al repositorio compartido online, de manera que sepa si dicho commit es probable que falle durante el *build* y las pruebas o no. Los fallos que se intentarán detectar, por tanto, son exclusivamente los fallos de carácter funciónal, ya que los fallos de compilación se detectarían automáticamente antes de que comience a hacerse el *build*.

1.2 MÉTODO Y FASES DE TRABAJO

Las fases de trabajo se pueden resumir en 4 fases principales, que son:

1. Adquisición de los datos.
2. Realizar un análisis estático de los cambios de código en un repositorio.
3. Tratar de crear un modelo predictivo a partir de la información extraída.
4. Creación de un servicio a partir las dos fases anteriores.

El desarrollo del proyecto seguirá el framework de gestión de desarrollo Scrum [4], siguiendo recomendaciones para su adaptación en proyectos de Ciencia de Datos [5]. Además, el marco de trabajo cumplirá con la Primera Vía de DevOps [1]. Esta parte de la filosofía DevOps busca agilizar lo máximo posible el flujo de trabajo para así entregar valor a los clientes cuanto antes. Para ello, se aplicarán los siguientes puntos:

- Control de versiones, no solo del código fuente y sus dependencias, sino también de las bases de datos y el modelo creado, siguiendo la actual corriente ‘Machine Learning is code’ [6] [7].
- CI enlazada con el repositorio de control de versiones, y CT.

- En el repositorio siempre se encuentra la última versión de los cuadernos Jupyter con los que realizamos los distintos análisis de datos y creación de modelos, tal y como se aconseja en [5].
- Posible despliegue automático (CD) una vez creado el modelo.

1.3 COMPETENCIAS

En la Tabla 1 se pueden ver las competencias adquiridas en el máster que se abordan en este proyecto.

Tabla 1: Justificación de las competencias abordadas en el TFM

Competencias	**Justificación**
[CE4] Capacidad para modelar, diseñar, definir la arquitectura, implantar, gestionar, operar, administrar y mantener aplicaciones, redes, sistemas, servicios y contenidos informáticos.	Se ha desarrollado un servicio potencialmente desplegable para advertir al programador de posibles fallos en sus commits. Este desarrollo se ha abarcado desde el diseño hasta la implementación.
[CE8] Capacidad para analizar las necesidades de información que se plantean en un entorno y llevar a cabo en todas sus etapas el proceso de construcción de un sistema de información.	Para elaborar un modelo que advierta sobre posibles fallos en los commits, es necesario tanto la recolección como el estudio de datos a través de diversas fuentes para comprender el problema.
[CE12] Capacidad para aplicar métodos matemáticos, estadísticos y de inteligencia artificial para modelar, diseñar y desarrollar aplicaciones, servicios, sistemas inteligentes y sistemas basados en el conocimiento.	A partir de los datos recogidos se ha tratado de crear un modelo de predicción, entrenarlo con la información recabada y poder predecir futuros casos.

1.4 ESTRUCTURA DE LA MEMORIA

Este trabajo se dividirá en un total de 5 capítulos, incluyendo este capítulo introductorio. (CAMBIARLO CUANDO TENGAMOS ESTRUCTURA FINAL)

El segundo capítulo hablará del estado del arte. En este capítulo entraremos más en detalle en el problema concreto que queremos abordar: avisar al programador de posibles

fallos de código en un commit antes de realizar ningún tipo de compilación o prueba automática. Describiremos algunas de las vías de estudio que se han realizado para resolver este problema, así como la fuente de la cuál se han extraído los datos que usaremos de cara a la predicción de fallos.

En el siguiente capítulo se abordará la creación del servicio. En este capítulo explicaremos en qué consiste, en qué fase de trabajo de un equipo de desarrollo se puede utilizar y qué información se muestra al usuario. Además, explicaremos los diferentes modelos de predicción que hemos creado para tratar de analizar de manera automática si un cambio de código puede tener fallos o no.

El cuarto capítulo explica cómo se ha creado el servicio y qué métodos se han llevado a cabo para crear una pipeline de CI/CD activa cada vez que hay un incremento funcional de código.

En el quinto y último capítulo se extraerán diferentes conclusiones tras el diseño e implementación de este servicio. Adicionalmente, se presentarán diversas mejoras que pueden desarrollarse en trabajos futuros.

Por último, se incluirá un apéndice en el que se detalle la metodología que hemos utilizado, la cual se basa en Scrum pero ha tenido que ser ligeramente adaptada debido a las características propias de un trabajo de fin de máster y al al contexto de la ciencia de datos, algo diferente a la construcción ágil de software por incrementos.

CAPÍTULO 2
ESTADO DEL ARTE

En este capítulo se explicarán los componentes imprescindibles en el desarrollo ágil de software durante los últimos años, como son los sistemas de control de versiones y las técnicas de Integración Continua, Entrega Continua y Despliegue Continuo. A continuación, se detallarán diferentes estudios que giran en torno al análisis de código y la predicción de fallos en función de los cambios en el mismo. Finalmente, se mostrará la fuente a partir de la cual hemos realizado nuestros análisis, explicando cómo se han conseguido.

2.1 SISTEMAS DE CONTROL DE VERSIONES

En términos generales, los proyectos de desarrollo software son realizados por más de una persona, en ocasiones alejados geográficamente. E incluso si estamos en el mismo centro de trabajo, es recomendable que cada persona pueda desarrollar el proyecto de manera independiente, y posteriormente tener la capacidad de combinar el trabajo del equipo de una manera ágil. Esto es controlado por el proceso conocido como control de versiones.

Los sistemas de control de versiones son herramientas software que ayudan a un equipo software a gestionar los cambios de código a través del tiempo [8]. Estos sistemas monitorizan los cambios de código que se realizan en cada modificación, de manera que el programador puede deshacer sus cambios si ha detectado un error, así como comparar versiones previas del código sin tener que interrumpir el trabajo de todos sus compañeros.

Estos sistemas, por tanto, tienen tres características fundamentales que los hacen casi imprescindible para el flujo de trabajo de casi cualquier proyecto software [8]. En primer lugar, mantienen un registro a largo plazo del histórico de cada archivo, lo que nos ayuda a tener una visión general de los cambios del proyecto. Esta característica es muy útil para volver a versiones anteriores con el fin de detectar errores en la aplicación, o para poder arreglar errores en versiones anteriores del proyecto.

Otra de las ventajas del control de versiones es que permiten diversificar el desarrollo, mediante un proceso conocido como ramificación, en el que un miembro del equipo puede, a partir de una versión específica, trabajar en una nueva funcionalidad o arreglar un fallo sin

tener que preocuparse de que otro miembro del equipo esté realizando cambios de manera simultánea. Una vez se realizan los cambios en una rama, se debe hacer el proceso de fusión, que consiste en unir los cambios y verificar que los cambios de las dos ramas que se intentan unir no tienen conflictos. Un conflicto es una modificación de la misma parte del código por parte de dos ramas diferentes. Si esto ocurre, es necesario resolver el conflicto manualmente, decidiendo de qué manera se combinan ambas modificaciones.

La tercera ventaja más importante de estas herramientas es la trazabilidad. Al llevar un registro de cada cambio de software, es muy sencillo conectar esta información con herramientas de gestión de proyectos y de gestión de *bugs* o errores, con el fin de poder documentar el propósito de cada cambio, de manera que es posible tener una visión más global de los cambios realizados sin tener que leer el código fuente para entender las nuevas funcionalidades o los *fix* realizados.

Dentro de los sistemas de control de versiones, existen dos grupos bien diferenciados [9]. Por un lado, tenemos los sistemas de control de versiones centralizados. En este tipo de herramientas se mantiene una copia central del proyecto en un único repositorio, y los cambios que realizan los desarrolladores lo aplican sobre esa versión central. Uno de los sistemas más conocidos es Subversion (SVN) [10]. Sin embargo, estos sistemas tienen un problema cuando dos desarrolladores realizan cambios simultáneos, ya que uno puede sobreescribir el trabajo del otro antes de que el resto del equipo pueda visualizar esos cambios.

Por ello, son muchos más frecuentes los sistemas de control de versiones distribuidos. Como se puede ver en la Imagen 2.1 [9], la principal diferencia con los anteriores radica en que no hay un repositorio central para los cambios de información. En su lugar, cada desarrollador tiene su propio repositorio (que es una copia de alguna versión del repositorio principal) y realiza los cambios a partir de él. De esta manera no se sobreescribe el trabajo de otros, ya que cada uno tiene su propia copia local, y una vez se quiere llevar el trabajo al repositorio principal se puede comprobar si hay conflictos, código fuente cambiado por dos o más personas, para poder realizar los cambios adicionales pertinentes. Existen muchos

sistemas distribuidos de control de versiones, aunque el más conocido y usado con diferencia es Git [11].

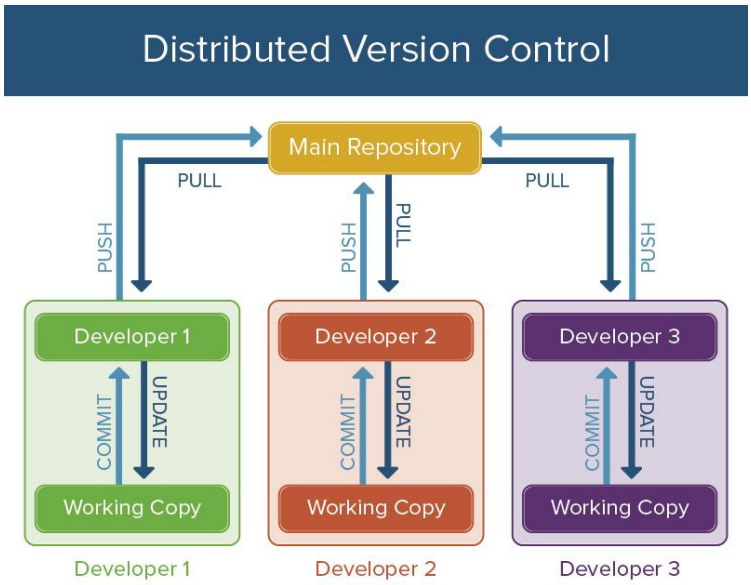

Imagen 2.1: Esquema general de un control de versiones distribuido

2.2 Integración Continua, Entrega Continua y Despliegue Continuo

Los sistemas de control de versiones han ayudado mucho a la agilidad de proyectos software, ya que todos los miembros del equipo pueden realizar modificaciones de manera simultánea. Sin embargo, existe un problema derivado de esta concurrencia: no siempre sabemos qué cambios ha realizado la otra persona. Por lo tanto, al margen de los errores propios del desarrollo software que existen desde siempre, con este modelo podemos encontrar errores debido a cambios que realiza un miembro del equipo sobre una fracción del código de la que se nutren parte de los cambios añadido por otro miembro.

Al margen de este problema surge la necesidad de disponer de herramientas que garanticen la calidad del software de manera automática, ya que los proyectos tienden a crecer y hacer esto de manera manual sería cada vez más complicado. A raíz de esta problemática, en las prácticas modernas de dearrollo se han incorporado tres conceptos o técnicas: Integración Continua, Entrega Continua y Despliegue Continuo [12].

La Integración Continua o *Continuous Integration* (*CI*) es la solución más directa al problema planteado anteriormente. Esta técnica trata de hacer que los desarrolladores incorporen sus cambios en la rama principal con la mayor frecuencia posible. Cuando un

desarrollador hace los cambios, estos se validan creando una compilación y sometiéndola a pruebas automatizadas.

La Entrega Continua o *Continuous delivery* (*CD*) es una extensión de la Integración Continua, y amplía la automatización a la preparación de la aplicación del despliegue, de manera que siempre se pueda desplegar una versión estable de la aplicación.

El último concepto relacionado con estas prácticas es la que se conoce como Despliegue Continuo o *Continuous Deployment* (*CD*). Aquí se da un paso más en la automatización, ya que los proyectos que aplican Despliegue Continuo, cada vez que los cambios de un desarrollador pasan todas las pruebas del pipeline o flujo de trabajo, se despliegan de manera automática para que sean visibles de manera inmediata para el cliente final.

En la Imagen 2.2 [12] se aprecia la diferencia entre estos tres términos. La Integración Continua se refiere únicamente a validar los cambios de manera automática, mientras que los otros dos conceptos son más amplios e incluyen los distintos procesos de preparación del despligue. La Entrega Continua cede el despligue final a una decisión manual, mientras que el Despliegue Continuo automatiza incluso ese último paso.

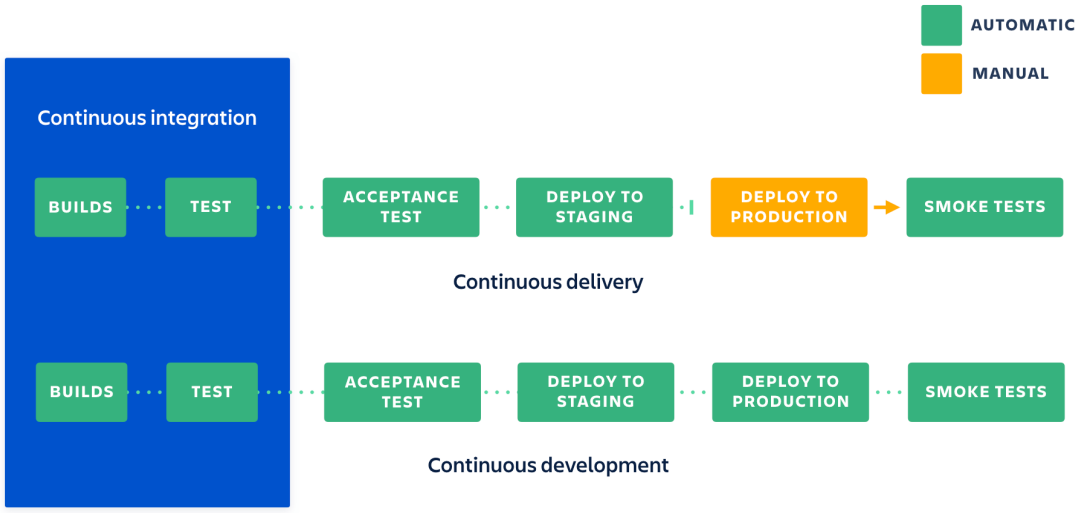

Imagen 2.2: Diferencia entre Integración Continua, Entrega Continua y Despliegue Continuo

Aunque gracias a estas técnicas, que cada vez se implantan en más empresas, es posible mantener la calidad del software sin mermar notablemente la agilidad de los proyectos, surge un problema derivado de las pruebas automáticas. Si se se sigue esta filosofía, cuando un miembro del equipo aporta un incremento en alguna de las ramas del repositorio, esto lanza

un proceso automático de Integración Continua. Cuando se lanza el proceso de build en CI es posible que se produzca un error. Es entonces cuando decimos que la pipeline se ha roto, y todo el equipo debe parar su trabajo y no enviar ningún incremento nuevo al repositorio hasta que el programador arregle el código que ha roto el proceso de CI. Además, en proyectos grandes el proceso de compilación y aceptación de las pruebas automáticas puede requerir un tiempo destacable, por lo que sería conveniente saber de antemano si es probable que los cambios contienen errores.

Debido a esto, una vía de investigación que puede beneficiar todavía más a la agilidad de los proyectos software modernos consiste en, una vez realizados los cambios por parte de un desarrollador, y antes de lanzar el proceso de CI, tener alguna manera de saber si es posible que existan fallos para revisarlos antes de lanzar estos procesos automáticos. Una de las alternativas para ello podría ser el análisis del código modificado, con el fin de extraer diferentes métricas y poder hacernos una idea de si los cambios pueden provocar interrupciones del pipeline.

2.3 Predicción de fallos mediante análisis de código

El análisis de los cambios de código en los equipos de desarrollo es un campo que se ha estudiado desde perspectivas muy diversas. Sin embargo, en términos generales podemos afirmar que este análisis es meramente exploratorio, llegando a distintas conclusiones a partir de encuestas o de la observación de distintos proyectos para ver sus puntos en común.

Existen estudios, por ejemplo, de la relación entre los patrones que se aprecian en los commits de los desarrolladores con la evolución del código de un proyecto. En el artículo [13] se determinan cuatro métricas para medir la actividad respecto a los commits y la evolución del código: cambios del commit, tiempo entre commits, autor del cambio y la evolución del código. A partir de estas cuatro métricas, extraen diferentes conclusiones, como qué cambios en clases que se utilizan en muchas partes del código provocan cambios en un número de archivos mayor.

Uno de los activos más importantes que podemos tener para tratar de resolver nuestro problema es la cantidad de datos que tenemos de manera abierta en repositorios públicos como GitHub. Proyectos como GHTorrent [14] se han encargado de recoger información relativa a los repositorios públicos de GitHub y la información de los usuarios durante más de un año. Esta base de datos presenta más de 900GB de información, y 10GB de metadatos, incluyendo casi 800.000 usuarios, más de 1,3 millones de repositorios y casi 30 millones de commits, y con el tiempo esta cifra aumenta, ya que se sigue actualizando con los eventos más recientes [15].

Sin embargo, necesitamos también información relativa a cuándo el estado del código en un commit concreto ha provocado un fallo o no. Existen otros artículos que analizan la categorización de los *issues*[1] de GitHub a partir de las etiquetas asociadas en los distintos proyectos [16]. No obstante, en [16] se destaca que, aunque el uso de etiquetas favorece la resolución de issues en un repositorio, este sistema de etiquetado es muy poco utilizado, y la manera en que la usan los repositorios varía en cada caso, lo que hace complicado su análisis.

Otra posible vía para tratar de encontrar los commits que han provocado un fallo de código son las técnicas de Integración Continua (CI). Cuando un desarrollador realiza un cambio de código y existe alguna herramienta de CI, automáticamente se lanzan una serie de pruebas tras la subida de ese nuevo código. Si las pruebas fallan, entonces decimos que la pipeline se ha roto, y todo el equipo debe parar su trabajo y no enviar ningún incremento nuevo al repositorio hasta que el programador arregle el código que ha roto el proceso de CI. Por ello, sería posible recoger información de bases de datos de los resultados de las pruebas de Integración Continua y cruzarlos con la información de los commits. Una base de datos válida para este tipo de estudios es TravisTorrent [17], que recopila el resultado de las pruebas realizadas por Travis-CI [18], un servicio de Integración Continua. En esta base de datos se puede observar el resultado de las pruebas y los commits que intervienen en cada subida de código, por lo que se podría cruzar con la información disponible en GHTorrent para estudiar qué clases han provocado fallos en cada subida de código. No obstante, esta vía de estudio tiene dos problemas principales: la enorme cantidad de información que se manejan en ambos casos; y los pocos proyectos que aparecen en ambas bases de datos para poder cruzar la información.

Existen otros estudios que hacen un análisis que buscan el mismo objetivo que el que nos hemos planteado. Un ejemplo lo encontramos en [19], donde se explica cómo se estudian varios proyectos de GitHub y, a partir de la información presente en los distintos commits de cada uno de ellos, se puede dilucidar qué clases han tenido bugs y en cuáles no se han detectado errores hasta ahora. A partir del estudio hecho en este artículo se ha generado una base de datos pública en la que se presentan las distintas clases de un conjunto de proyectos calsificadas en función de si presentan bugs o no [20].

Entrando un poco más en profundidad en [19] [20], el proceso que los autores han seguido para obtener la base de datos de *bugs* tiene varias partes. En primer lugar, se han seleccionado los proyectos, un total de 13 diferentes, todos ellos en lenguaje Java y en general con una gran cantidad de código, ya que son los más útiles para este tipo de análisis. Además, se han buscado proyectos con un número adecuados de commits que utilicen la etiqueta bug propia de GitHub, para poder diferenciar los cambios de código relacionado con errores y los que no, así como para permitir referenciar qué parte del código se ha modificado para solucionar un error. Dentro de los proyectos que cumplían estos requisitos, se han priorizado los que estaban en activo en el momento del análisis.

Una vez seleccionados los proyectos, mediante la API de GitHub se guardaron sus datos. Estos incluyen los usuarios que colaboran en el repositorio, los issues abiertos y cerrados y los datos de todos los commits. Entre todos los commits guardados se realizó un filtrado para seleccionar solo los relacionados con errores, mediante el uso de la etiqueta bug.

Por otra parte, se realizó una descarga de los repositorios en sus distintas versiones o releases. Para cada una de ellas, se realizó un análisis de código estático mediante SourceMeter [21]. Por último, se cruzó esta información con los commits extraídos en el paso anterior, para saber qué partes del código se modificaron para solucionar la incidencia.

Tras todo este análisis, los autores crearon en [20] una base de datos pública con versiones de un conjunto de proyectos en intervalos de 6 meses, y tras descartar aquellas versiones en las que no había errores suficientes. Para cada una de ellas tenemos las distintas clases que han sido modificadas entre versiones, y el número de *bugs* presentes en cada una de ellas. A partir de los datos que estos autores nos proporcionan, es posible intentar crear un modelo de predicción que aprenda a partir de estos datos para poder predecir en futuros commits de cualquier proyecto si un commit puede contener algún error.

2.4 Conclusiones

Tras estudiar diversas fuentes de estudio de análisis estático de código, y de búsqueda de fallos potenciales a partir de cambios en el mismo, se ha encontrado una fuente de datos que incluye diferentes métricas de código estático, combinado con la presencia de errores o no, discriminando por cada clase del proyecto. A partir de estos datos, se realizará el análisis posterior y se seleccionará qué información se muestra al usuario en el servicio desarrollado.

CAPÍTULO 3
SERVICIO DE ANÁLISIS

En esta sección vamos a explicar el servicio creado y en qué fase del flujo estándar de trabajo se introduce para facilitar la tarea a los equipos de trabajo de desarrollo software. Además, se explicará qué procedimiento se ha escogido para seleccionar la información que se va a mostrar al usuario, para finalmente explicar los distintos modelos de predicción que hemos intentado crear para realizar esta búsqueda de fallos potenciales de manera automática.

3.1 DESCRIPCIÓN DEL SERVICIO

En la Imagen 3.1 se puede observar el flujo de trabajo típico para las empresas que siguen la filosofía *DevOps* y aplican técnicas automáticas de Integración Continua, en el círculo verde, y de despliegue continuo, en el círculo azul [22]. En general, los desarrolladores suben sus cambios al repositorio del equipo. A continuación, si se utilizan técnicas de Integración Continua o CI se compila el proyecto y se pasan una serie de pruebas automatizadas. Si todas las pruebas son correctas, acaba la fase de CI, y a partir de aquí podemos, o bien desplegar manualmente en el momento que deseemos, si seguimos las tendencias clásicas, o bien comenzar el proceso de Despliegue Continuo o CD, que comienza las operaciones necesarias para desplegar una nueva versión visible para el usuario final.

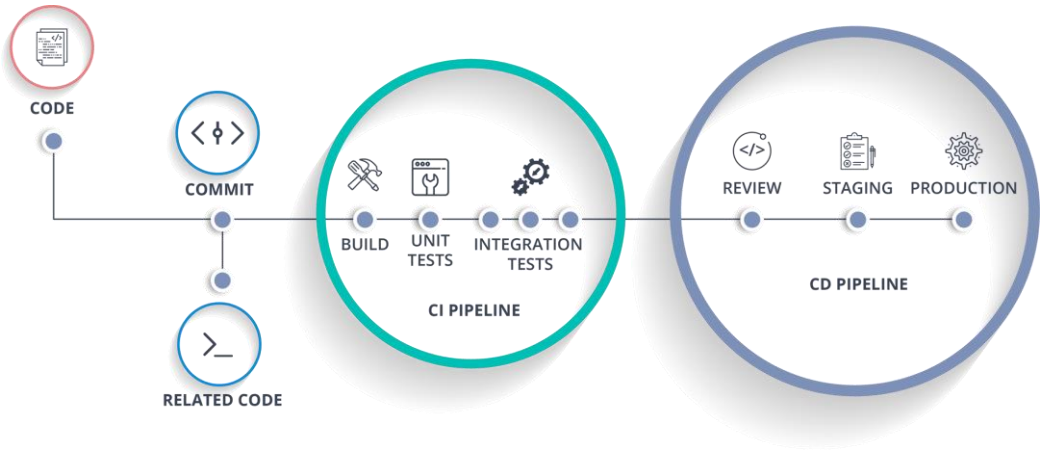

Imagen 3.1: Flujo de trabajo común - Sin utilizar nuestro servicio

Sin embargo, el flujo anterior tiene un inconveniente, y es que generalmente no hay un solo desarrollador intentando subir sus cambios para añadir nuevas funcionalidades o arreglar fallos, sino que será un equipo entero el que trate de enviar sus modificaciones. Si uno de estos envíos falla, entonces se dice que la *pipeline* se ha roto, y todo el equipo debe parar su trabajo y no enviar ningún incremento nuevo al repositorio hasta que el programador arregle el código que ha roto el proceso de CI. Por ello, sería conveniente tener información adicional antes de lanzar este proceso, con el fin de saber antes de romper el flujo de trabajo de todo el equipo que las modificaciones incorporadas pueden provocar fallos.

Teniendo en cuenta este problema, nuestro servicio se incorporaría justo antes de lanzar el proceso de Integración Continua. En la Imagen 3.2 podemos ver el nuevo flujo de trabajo, con la inclusión de nuestro servicio en color verde. Los desarrolladores, una vez tengan sus cambios, podrán utilizar el servicio para recibir información acerca del código que han modificado. Nuestro servicio comprueba qué partes del proyecto se han modificado y ofrece distintas métricas calculadas a partir de un análisis estático del código que ha sufrido cambios. Tras ver estos datos, los desarrolladores pueden continuar el flujo de trabajo y subir sus cambios para comenzar el proceso de Integración Continua, o si observan que los datos son diferentes a los valores usuales o a los aceptados en los estándares de calidad de la empresa, revisar sus cambios sin interrumpir al resto del equipo.

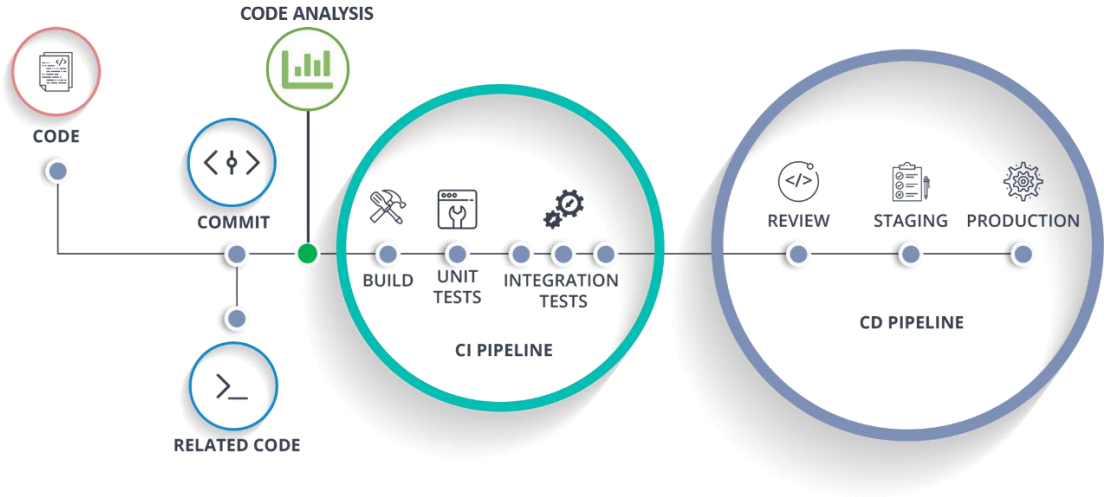

Imagen 3.2: Flujo de trabajo común - Utilizando nuestro servicio

En la Imagen 3.3 se puede apreciar más detalladamente en qué situación se puede utilizar nuestro servicio. El servicio presenta también algunas limitaciones. Por ejemplo, como se aprecia en el primer elemento del diagrama, el código analizado solo puede provenir del lenguaje de programación Java, ya que es el mejor estructurado y el que resulta más sencillo para analizar estáticamente. Además, otro requisito de nuestro servicio es que el proyecto debe estar alojado en GitHub, ya que el servicio estudia, a partir de su sistema de control de versiones, qué código se ha modificado.

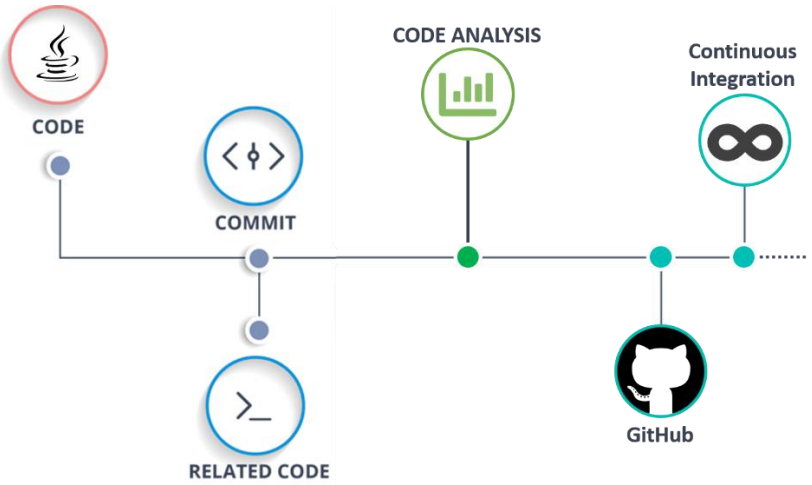

Imagen 3.3: Contexto de uso de nuestro servicio

Este servicio se proporciona vía web, de manera que cualquier usuario pueda enviar el estado de un repositorio e iniciar un análisis. En la Imagen 3.4 se puede ver la pantalla inicial del servicio, en la que se pide el repositorio de GitHub en el que se está trabajando.

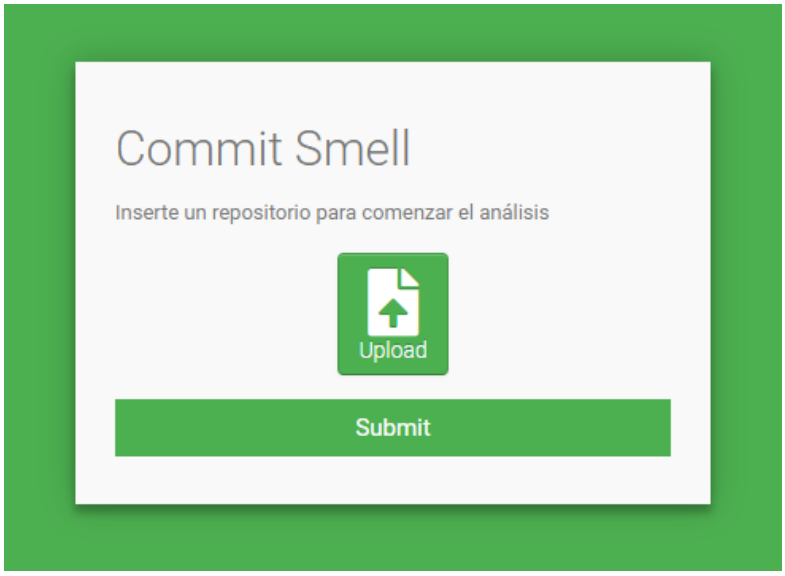

Imagen 3.4: Pantalla inicial del servicio

Tras introducir el repositorio comprimido, comenzará el procesamiento de los cambios realizados y se mostrarán una serie de métricas que nos pueden orientar en una posible detección de fallos. En la Imagen 3.5 se puede ver una pantalla con unos resultados de ejemplo. El análisis resultante consiste en una serie de 10 métricas que suelen estar relacionadas con la presencia de *bugs*, de manera que el usuario puede ver si se presenta alguna anomalía.

Class	CBO	NLE	RFC	CXMR	WMC	DMR	CPMR	TNLA	WI	SMR
Hammer	0	1	5	0	6	13	0	2	13	0
User	0	1	9	0	15	19	0	5	19	0
AppTest	0	2	1	0	3	5	0	0	5	0

Información sobre las métricas:

- **CBO**: *Coupling Between Object classes*, número de usos de otras clases. Un valor muy alto de este valor indica que es muy dependiente de otros módulos, y por tanto más difícil de testear y utilizar, además de muy sensible a cambios. Si tiene un valor alto quizás debería revisar sus cambios.
- **NLE**: *Nesting Level Else-If*, grado de anidamiento máximo de cada clase (bloques de tipo if-else-if cuentan como 1 nivel)
- **RFC**: *Response set For Class*: combinación de número de métodos locales y métodos llamados de otras clases.
- **CXMR**: *Complexity Metric Rules*, violaciones en las buenas prácticas relativas a métricas de complejidad. Si es distinto de 0, quizás deba revisar sus cambios.
- **WMC**: *Weigthed Methods per Class*, número de caminos independientes de una clase. Se calcula como la suma de la complejidad coclomática de los métodos locales y bloques de inicialización.
- **DMR**: *Documentation Metric Rules*, violaciones de buenas prácticas relativas a la cantidad de comentarios y documentación.
- **CPMR**: *Coupling Metric Rules*, violaciones en las buenas prácticas relativas al acoplamiento de las clases. Si es distinto de 0, quizás deba revisar sus cambios.
- **TNLA**: *Total Number of Local Attributes*, número de atributos locales de cada clase.
- **WI**: *Warning Info*, advertencias de tipo *WarningInfo* en cada clase
- **SMR**: *Size Metric Rules*, violaciones en las buenas prácticas relativas al tamaño de las clases. Si es distinto de 0, quizás deba revisar sus cambios.

Imagen 3.5: Pantalla de resultados

3.2 SELECCIÓN DE MÉTRICAS A MOSTRAR

Uno de los puntos fuertes del servicio desarrollado es la selección de las métricas a mostrar al usuario. Ofrecer una cantidad excesiva de métricas podría saturar al usuario y dificultar el uso del servicio. Por ello, se ha optado por estudiar qué características son las más útiles de cara a una posible detección de fallos.

La selección de variables se ha realizado a partir de [20], en la que disponíamos de una clasificación de las clases de un conjunto de proyectos, indicándonos para cada una de ellas distintas métricas de código estático junto con el número de bugs que presentaban. En total tenemos un total de 106 métricas diferentes, por lo que es conveniente seleccionar un subconjunto de estas. Un análisis más exhaustivo se puede encontrar en el Capítulo 5.

Se ha realizado un análisis exploratorio de cada una de estas métricas, para ver la distribución de cada una de ellas a lo largo de las clases recogidas en [20]. Hemos descartado variables que tuvieran una correlación muy fuerte con otras, ya que reflejaban prácticamente la misma información, así como aquellas que no tuvieran una variabilidad notable en sus valores. Siguiendo esos criterios se han logrado eliminar hasta 35 métricas del conjunto de 106 métricas que disponíamos.

Una vez realizado esta primera valoración de las métricas, y puesto que el objetivo de este servicio es prevenir la presencia de *bugs*, hemos ordenado las métricas restantes en función de la correlación de cada una con respecto al número de bugs de cada clase. Finalmente, se han descartado algunas más por representar información similar a otras métricas de las que disponíamos, para mostrar un total de 10 métricas que pueden ser indicativas de la presencia de errores en los cambios de código de los desarrolladores.

3.3 MODELO DE PREDICCIÓN

Al tener en [20] una base de datos en la que se indica la presencia de errores en cada una de las clases de una serie de proyectos, se ha valorado la posibilidad de elaborar un modelo de predicción que, a partir de estos datos, fuera capaz de predecir si los cambios de código que se envían a nuestro servicio tienen una probabilidad alta de presentar fallos o no.

En primer lugar, se ha intentado crear un modelo cuya salida pueda ser interpretable, con el objetivo de poder explicar al usuario las razones por las que el modelo ha predecido que los cambios pueden contener fallos. Por ello, se ha intentado crear un árbol de decisión, ya que con este modelo podemos saber qué variables ha tomado en cuenta el modelo y a partir de qué valores considera que la probabilidad de presentar fallos es alta; y un modelo de regresión lineal con regularización Lasso, ya que en este modelo podemos saber la importancia de cada variable para predecir el valor final de la variable clase.

Tabla 2: Resultados modelos predictivos interpretables

	Árbol de decisión	***Regresión Lasso***
Precisión	*0,31*	*0,19*
Recall	*0,29*	*0,46*
F1-Score	*0,3*	*0,27*
ROC	*0,63*	*0,69*

Sin embargo, los resultados obtenidos no han sido positivos, ya que los modelos tenían un alto porcentaje de acierto sobre el conjunto de entrenamiento, es decir, sobre los datos que utilizamos para aprender, pero no sobre el conjunto de test, aquel que apartamos del entrenamiento del modelo para estudiar el rendimiento real del mismo. Se ha producido, por tanto, un fenómeno conocido como sobreajuste, el cual no nos permite concluir que el modelo es capaz de predecir cambios de código en nuevos repositorios. En la Tabla 2 se muestran los resultados sobre el conjunto de test, en términos de precisión, recall, F1-Score y área bajo la curva.

Tras los resultados de los modelos anteriores, se optó por crear modelos de predicción más potentes pero que careciesen de interpretabilidad. En concreto, se utilizó un modelo conocido como *Random Forest*, que consiste en utilizar un conjunto de modelos de árbol de decisión y combinar la salida de todos ellos, y un modelo denominado *XGBoost* [23], un algoritmo que aplica la técnica del gradiente descendiente de manera optimizada para permitir utilizar grandes conjuntos de datos. De igual manera, se intentó crear un modelo máquina vector soporte o *Support Vector Machine* (SVM), ya que este modelo no tiende a sobreajustar, como sí lo han hecho el árbol de decisión y la regresión Lasso.

Tabla 3: Resultados modelos de predicción no interpretables

	SVM	*Random Forest*	*XGBoost*
Precisión	*0,43*	*0,6*	*0,67*
Recall	*0,05*	*0,03*	*0,02*
F1-Score	*0,09*	*0,06*	*0,03*
ROC	*0,52*	*0,52*	*0,5*

No obstante, los resultados son similares a los modelos anteriores. En la Tabla 3 se desglosan los resultados sobre el conjunto de test. En consecuencia, los modelos generados no son tan buenos como los que se comentan en [20], por lo que se ha incluido en el servicio como una funcionalidad en fase beta, pero centrando los resultados en la selección de métricas mencionada en la sección anterior. En la Imagen 3.6 tenemos una imagen de la parte del servicio relativa al uso de este modelo de predicción. Esta funcionalidad se encuentra en la parte final de la pantalla y solo se mostrará si el usuario está interesado.

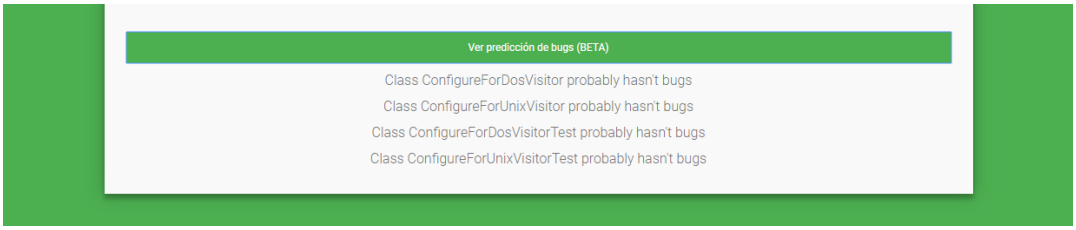

Imagen 3.6: Pantalla de predicción de fallos - versión Beta

3.4 CONCLUSIONES

Tras realizar distintos estudios sobre la creación de modelos de predicción que indiquen de manera automática si hay una alta probabilidad de tener fallos potenciales, hemos visto que los resultados no son los esperados. En consecuencia, este servicio se

centrará en mostrar al usuario métricas claves para la identificación de estos fallos, de manera que pueda observar de manera muy rápida si hay algún valor fuera de lo común o de los estándares de calidad del proyecto.

Se recomienda el uso de este servicio cuando se utilizan técnicas de CI, justo antes de subir los cambios al repositorio online, con el fin de poder detectar fallos antes de realizar la compilación y las pruebas automáticas y, por lo tanto, de romper el flujo de trabajo, aunque se puede utilizar en proyectos que no apliquen Integración Continua.

CAPÍTULO 4
METODOLOGÍA

Para este proyecto se seguirá una metodología ágil como Scrum. Sin embargo, en proyectos de Ciencia de Datos es complicado seguir la metodología clásica que propone Scrum. Uno de los puntos discordantes entre la gestión ágil y la ciencia de datos es que normalmente se necesita más de un sprint para tener un producto final. Esto se debe a que la ciencia de datos abarca multitud de etapas: recolección de datos, estudio de los mismos, creación de modelos, exposición de los resultados, etc. Por ello, en este proyecto no se ha seguido Scrum siguiendo las recomendaciones tradicionales de este framework, sino que se ha seguido la adaptación de Scrum para proyectos de Ciencia de Datos de [5]. Se explicará esta adaptación y se comentarán las diferencias con Scrum.

4.1 ADAPTACIÓN DE SCRUM

En este proyecto se seguirá la adaptación de Scrum de [5], pero se seguirán los principios básicos de la metodología tradicional. Dentro de esta metodología, los tutores ejercerán los papeles de Scrum Master, mientras que el alumno ejercería las funciones del equipo de desarrollo y del Product Owner. El Product Owner es el encargado de crear y ordenar por prioridad las tareas del Product Backlog, así como de mantener el Burndown Chart, y por ende es la persona con una visión más global del proyecto, por lo que el alumno es el que puede hacer mejor este papel. Los tutores, por su parte, al tomar el papel de Scrum Masters serán los encargados de asegurarse de que se siguen las prácticas descritas en la metodología y de eliminar cualquier impedimento que pueda ir surgiendo, así como de facilitar las reuniones y velar por no sobrepasar una duración máxima. Debido a esta elección en la metodología:

- Se realizarán Sprints quincenales, juntando la Sprint Review y la Retrospectiva del sprint que termina, con el Sprint Planning del nuevo sprint.
- Cada uno de los sprints quedará documentado, con la fecha de las reuniones y un resumen de los temas tratados en las mismas.
- Se mantendrá un dashboard online para dar transparencia al backlog del producto

y del sprint.

Sin embargo, al tratarse de un proyecto de ciencia de datos, existen algunas diferencias que convienen recalcar. En la adaptación de [5] se muestran un total de 7 principios de una metodología ágil en un proyecto de ciencia de datos, y todos estos principios se han tratado de respetar durante el desarrollo del proyecto. A continuación, se enumerarán estos principios, se describirán brevemente y se explicará cómo se ha aplicado este principio en nuestro proyecto.

4.1.1 Iterar, iterar, iterar

El primer principio es el de iterar continuamente. Este principio destaca que, en la ciencia de datos, no se suele tener un modelo con unos resultados óptimos en la primera iteración, por lo que se recomienda tratar de mejorar continuamente el desarrollo realizado.

En este trabajo, en una versión muy temprana del servicio se utilizó un modelo predictivo muy básico, para posteriormente intentar conseguir otro con mejores resultados y poder susituir al previo.

4.1.2 Mostrar salidas intermedias

El segundo principio dice que se deben mostrar resultados intermedios, aunque no estén completos. En un proyecto de ciencia de datos es común acabar un sprint sin un resultado tangible. Sin embargo, si no podemos mostrar nada no estamos siendo ágiles, por lo que se recomienda mostrar los avances de cada iteración, aunque no estén completos.

En este proyecto, se han mantenido dos libretas Jupyter, una para el análisis de los datos y otra para la creación del modelo predictivo, y se ha ido compartiendo la nueva información subiendo al repositorio remoto cada cambio realizado.

4.1.3 Prototipar experimentos en lugar de implementar tareas

El tercer principio dicta que en una metodología ágil en un proyecto de ciencia de datos es más recomendable prototipar experimentos que implementar tareas. Es muy común que algunas tareas de este tipo de proyectos tengan como artefactos ablas, gráficas o informes, en lugar de una funcionalidad nueva en un servicio o el arreglo de un fallo. Por lo tanto, se deberían considerar como experimentos más que como tareas.

En este proyecto, algunas de las *issues* creadas en nuestro Backlog tienen este carácter, por lo que no se resuelven con un nuevo bloque de código, sino como un resumen de las

pruebas realizadas. Estos cambios quedan documentados en cuadernos Jupyter, pero no forman parte del servicio final.

4.1.4 Integrar la opinión de los datos

En el desarrollo de aplicaciones tradicional suelen coexistir tres perspectivas: la de los clientes, la de los desarrolladores y la del negocio. Sin embargo, en proyectos de análisis debería añadirse una nueva perspectiva: la de los datos. Ya que gracias a los datos desarrollamos el servicio, es conveniente tenerlos en cuenta durante todas las fases del proyecto, desde el diseño hasta la fase final.

En este trabajo, los datos han tenido un papel vital durante su desarrollo. El comienzo del trabajo consistió en la recogida de estos, para poder ver la información que nos podrían dar. De hecho, debido a la naturaleza de los datos que teníamos, se optó por crear un servicio descriptivo en lugar de utilizar un modelo de predicción, ya que con la información que proprocionaban no era posible crear un modelo que diese la información deseada.

4.1.5 Subir y bajar la pirámide del valor de los datos

En la Imagen 4.1 se muestra la pirámide del valor de los datos. En esta pirámide se expresa el aumento del valor generado a medida que refinamos los datos. El primer nivel consiste en la recolección de los datos, para proceder posteriormente a la creación de tablas, de informes, de predicciones y de la fase final, acciones, que es en la que se producen resultados verdaderamente tangibles para los usuarios de nuestro servicio.

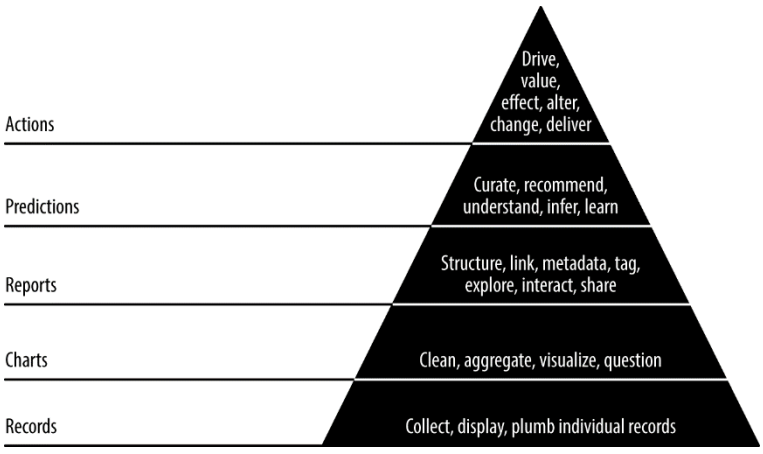

Imagen 4.1: Data-value pyramid

El quinto principio dicta que estos pasos no tienen por qué ser siempre ascendentes. Esta pirámide se debe tener en cuenta, pero no debe considerarse una regla. Por lo tanto, es posible tanto saltarse algún paso si no es necesario, como volver a un paso anterior si es

conveniente. En nuestro caso no ha sido necesario esta vuelta atrás en las etapas, pero de haber sido necesario se podría haber hecho sin problema.

4.1.6 Descrubir y seguir el camino crítico hacia el producto

El sexto principio dice que, para maximizar la probabilidad de éxito, se deben estudiar bien los datos disponibles mediante la experimentación, para posteriormente realizar la mejor implementación posible para la consecución de nuestro objetivo.

En este trabajo el objetivo principal era ayudar a los desarrolladores a detectar fallos en una etapa temprana. Sin embargo, mediante la experimentación se llegó a la conclusión de que el modelo predictivo no era la mejor solución, por lo que se encontró otro camino, conseguir nuestro objetivo mediante la vía descriptiva. Una vez encontrado el camino para cubrir nuestro problema, se siguió explotando este camino hasta tener el servicio final.

4.1.7 Explicar el proceso completo, no el estado final

Este principio, que está muy relacionado con el segundo, nos dice que debemos documentar el proceso completo en lugar de mostrar exclusivamente el resultado final. En este tipo de proyectos puede pasar que en algún sprint no haya un incremento que poder mostrar, por lo que podríamos caer en el riesgo de no ser ágiles. Por ello, se deben mostrar los avances, aunque sean experimentos fallidos o pasos intermedios antes de unos resultados presentables.

Durante el desarrollo del trabajo se han documentado todos los procesos relativos a la ciencia de datos elaborados, de manera que, en la revisión de los sprints, aunque no se haya conseguido un modelo final o un avance en el servicio, se ha documentado igualmente el trabajo realizado.

4.2 DOCUMENTACIÓN DEL PROYECTO

Teniendo en cuenta los principios básicos de Scrum y la adaptación de [5] explicada en la sección anterior, se mostrará el resumen de la documentación recogida durante el desarrollo del proyecto. Se han realizado un total de 19 sprints, generalmente quincenales aunque en ocasiones se ha cambiado para adaptarnos a las necesidades del proyecto.

En la imagen Imagen 4.2 se puede ver el ritmo del proyecto a lo largo del tiempo durante los últimos 6 meses, aunque el proyecto comenzó en julio. En la gráfica aparece un círculo de color morado cada vez que se cierra una tarea, mostrando de manera acumulada los puntos de historia realizados a lo largo del tiempo. Además, en amarillo se aprecia una

línea que estima el tiempo estimado del final del proyecto. (CAMBIARLO CUANDO ESTÉ ACABADO)

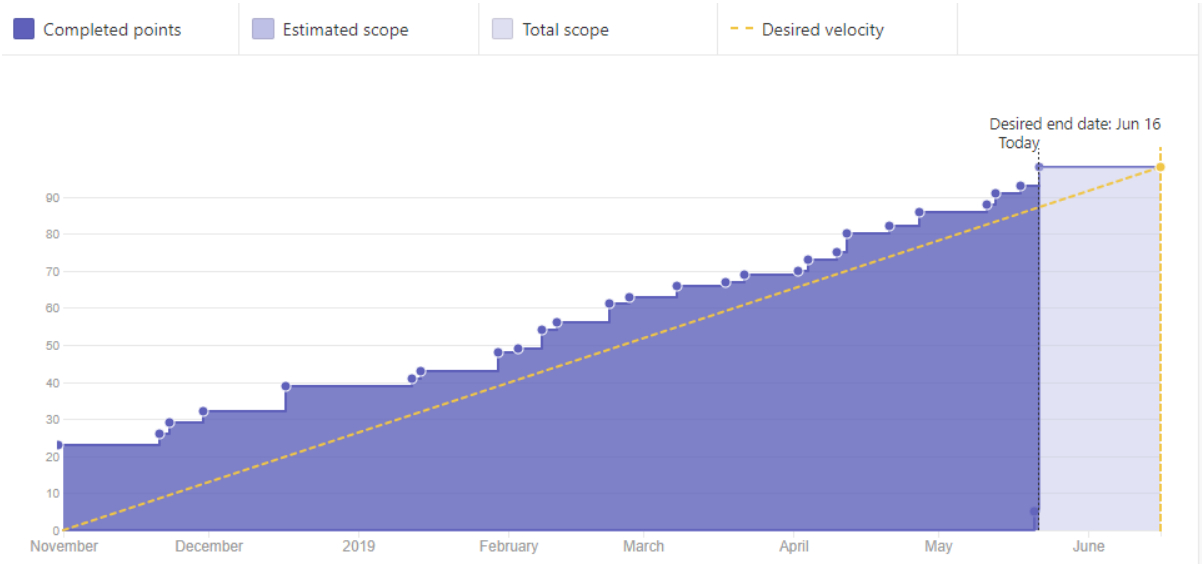

Imagen 4.2: Burnup Chart del proyecto

En la Imagen 4.3 se puede apreciar el Control Chart del proyecto. En este gráfico se muestra el tiempo medio de desarrollo de todas las tareas (la línea azul, que tiene un valor de 12,6), así como el valor medio de las 4 últimas tareas realizadas en cada momento (línea verde). Como se puede observar, a medida que se avanza en el proyecto disminuye este tiempo de desarrollo, ya que se tiene un dominio mayor del proyecto global, y además en la parte final del proyecto hay varias tareas relacionadas con documentación, que suele tener un tiempo menor que las tareas de desarrollo.

Además, aparece el tiempo que se ha tardado en cada tarea a lo largo del tiempo. Cada punto de la gráfica de Imagen 4.3 es una tarea individual, de manera que aquellas que han necesitado más tiempo para desarrollarse se encuentran en la parte superior del gráfico. Se puede observar un valor mucho más alto de lo común, representado como una flecha, denominado valor atípico o outlier. En este caso, la tarea es la #31: Preparar código para crear el servicio. En esta tarea se estuvo más tiempo del esperado debido a conflictos del programa encargado de analizar el código estático, llamado SourceMeter. Al lanzarlo en un entorno local se ejecutaba sin problemas, pero a la hora de integrarlo en el servicio para utilizarlo en la nube se produjeron varios fallos que alargaron el tiempo necesario para acabar esta tarea. El tiempo de esta tarea es de 44 días, más de 3 veces la media total.

Existen otras dos tareas que se encuentran en la parte superior del gráfico, aunque no son valores tan extremo como el caso mostrado antes. La primera de las dos tareas es la #32: Estudiar DeployHub, ya que a la hora de estudiar esta plataforma de despliegue se observó

que había aspectos de la documentación desactualizados o con enlaces erróneos, y a la hora de desplegar era bastante complejo. Por ello, se optó por desplegar con Heroku en lugar de con este servicio. La otra tarea es la #49: Estructura Capítulo 2 - Estado del arte. En esta tarea se tardó más de lo esperado porque se realizó de manera simultánea con el intento del modelo de predicción que se explica en la sección 3.3.

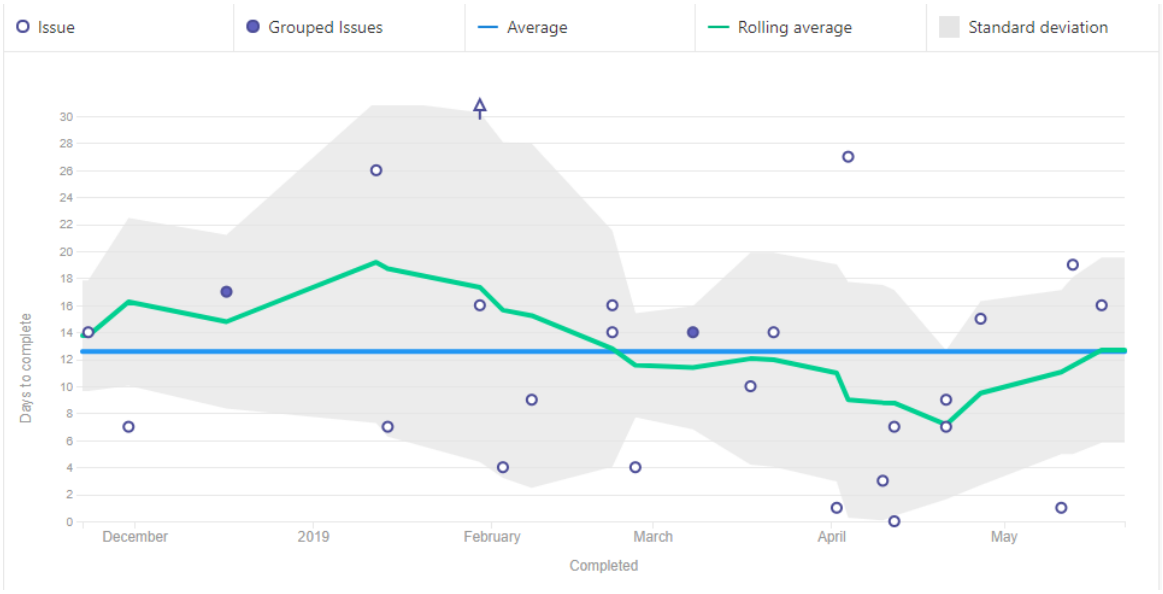

Imagen 4.3: Control Chart del proyecto

Tras estos esquemas, que muestran el desarrollo del proyecto desde el punto de vista más general, se va a mostrar la documentación individual de cada Sprint, describiendo lo que se ha hablado en el Sprint Planning y en la Sprint Review.

4.2.1 Sprint 1: Asentar conceptos Dev-Ops

En este primer Sprint, que comenzó el 6 de Julio de 2018 y acabó el 20 de Julio, se comenzó a estudiar qué se deseaba desarrollar a lo largo del proyecto. Se decidió que se iba a seguir una metodología ágil y que iba a seguirse la filosofía Dev-Ops.

En el sprint planning se comentaron las tecnologías que se iban a utilizar durante el desarrollo del proyecto, para poder estudiarlas, aunque todavía no se hubiera decidido el servicio que se iba a crear. Se comentaron cuatro áreas de estudio: los conceptos básicos de la filosofía Dev-Ops; Travis-CI, el servicio usada para la Integración Continua; Tensorflow, para poder crear redes neuronales; y ZenHub, el complemento de GitHub para la gestión ágil del proyecto.

Durante el desarrollo de este Sprint, se vieron las capacidades de Travis-CI, realizando una prueba de concepto en la que se ejecutaba una prueba automáticamente tras subir un cambio a GitHub. Además, se vieron los principios básicos de Dev-Ops, centrándonos sobre todo en la Primera Vía, que se basa en proporcionar valor al cliente lo antes posible. Además, se estudiaron ligeramente Tensorflow y ZenHub, pero no se acabaron de ver todas sus funcionalidades.

En la sprint review se comentó que no hubo tiempo suficiente para estudiar Tensorflow y ZenHub, por lo que esas tareas no se cerraron en este Sprint. En cuanto a la retrospectiva, al ser el primer sprint no se comentó nada destacado.

4.2.2 Sprint 2: Acabar de estudiar tecnologías

El segundo sprint es el más largo de nuestro trabajo, ya que comenzó el 20 de julio y terminó el 7 de septiembre, aunque hay que tener en cuenta que hay un período de vacaciones durante esta etapa. En este sprint se continúan asentando conceptos y tecnologías que se van a utilizar en el proyecto.

En el sprint planning se decidió continuar con el estudio de ZenHub, ya que al ser la herramienta que se iba a utilizar para la gestión del proyecto, era prioritario saber todas sus capacidades. Además, se decidió el lenguaje de programación que se iba a utilizar, Python, y la forma en la que íbamos a ofrecer el servicio, mediante una aplicación web. Por lo tanto, se decisió un framework web para Python, Django.

Durante el desarrollo del Sprint, se acabó de estudiar ZenHub, creando el repositorio en GitHub del proyecto y creando las tareas pensadas hasta ese momento. Además, se organizaron las tareas ya creadas en el dashboard online que nos proporciona ZenHub para que tanto los tutores como el alumno pudieran visualizar el trabajo que se estaba realizando a lo largo del tiempo. Además, se realizaron varios tutoriales [24] [25] de Django para entender este framework de desarrollo web.

En el sprint review de este sprint se repasaron los avances con ZenHub y Django, de manera que ya quedaron entendidos todos los conceptos relativos al desarrollo ágil y a las herramientas iniciales que se iban a utilizar. En la retrospectiva se recalcó la importancia de mantener actualizado el dashboard online de Zenhub para poder ver en cada momento el trabajo que se está realizando.

4.2.3 Sprint 3: Elección del problema a tratar

En este sprint, que comenzó el 7 de septiembre y finalizó el 25, se realizó el problema a tratar a lo largo del resto del proyecto.

En el sprint planning los tutores propusieron el objetivo que se buscaba: crear un servicio que advirtiese al desarrollado de posibles fallos antes de comenzar las pruebas automáticas que se realizan durante el proceso de Integración Continua. Este tema se aceptó y se dieron las dos primeras páginas a estudiar: TravisTorrent, que guardaba resultados de pruebas automáticas en Travis-CI, y GitHubTorrent, que almacenaba información de todos los repositorios públicos de GitHub. El objetivo de estas dos páginas era poder cruzar sus datos y realizar un modelo predictivo a partir de los mismos.

Durante el desarrollo de este sprint se leyó el artículo [3], que explicaba los eventos inusuales que los desarrolladores consideraban más útiles conocer. Además, se descargó toda la información disponible en TravisTorrent, y se intentó hacer lo mismo con GitHubTorrent pero tenía un volumen de información demasiado grande para poder tratarlo de manera convencional.

En el sprint review se comentó este problema, por lo que se desplazó esta tarea hasta el sprint siguiente. Además, se puso en común las conclusiones extraídas en [3] y se explicó la estructura de los datos extraídos de TravisTorrent.

4.2.4 Sprint 4: Resolver problemas con GitHubTorrent

En este sprint, que empieza el 25 de septiembre y acaba el 9 de octubre, se intentan descargar los datos de GitHubTorrent, ya que el gran tamaño de este conjunto de datos lo dificulta.

En el sprint planning se decide que en este sprint hay que centrarse exclusivamente en la descarga de los datos, aunque por dos vías: o bien se consiguen descargar los datos completos, o bien se busca alguna alternativa en la que tengamos los datos en una versión más reducida.

Durante este sprint se intentan descargar los datos, así como buscar un subconjunto de estos de forma fallida. Para descargar los datos se observó que en la página de GitHubTorrent había una altenativa a la descarga directa de toda la información, ya que se podían realizar consultas por ssh. Para ello era necesario pedir una clave con la que acceder al servidor y poder coger datos, por lo que se envió la petición, pero no llegó durante este sprint. Además, como sí que teníamos la estructura se comenzó a ver cómo cruzar estos datos con los de TravisTorrent.

En el sprint review se comentó cómo se podrían cruzar ambas bases de datos, y se explicó que hasta que no llegase el permiso para entrar al ssh no se podían descargar los datos, por lo que el avance en este sprint fue menor de lo esperado. En cuanto a la

retrospectiva, se menciona que. para no ralentizar el desarrollo del proyecto, además de las tareas propias del estudio de los datos era conveniente comenzar la documentación.

4.2.5 Sprint 5: Continuar con GitHubTorrent y hacer anteproyecto

En el sprint 5, que abarcó del 9 de octubre al 31 de ese mismo mes, se prosiguió con la descarga de los datos que proporcionaba GitHubTorrent, y se comenzó con la documentación escrita del trabajo.

En el sprint planning se decidió que había que terminar el anteproyecto para este sprint, ya que los plazos de este nos obligaban, y que había que continuar con la descarga de los datos.

A lo largo de este Sprint se desarrolló el anteproyecto, y el documento se validó satisfactoriamente. Además, se prosiguió con la descarga de los datos, pudiendo extraer la información de un subconjunto de 5 proyectos, aunque el tiempo y el procesamiento que se necesitó para esto era muy elevado, ya que la cantidad de datos que manejaba esta página era excesiva.

En el sprint review se comentaron los puntos a mejorar que se enviaron en la revisión del anteproyecto, para tenerlos en cuenta a la hora de realizar la memoria final. En cuanto a los datos de los 5 proyectos, como se extrajeron en una fase muy tardía del sprint se dejó el análisis de los datos para el siguiente sprint.

4.2.6 Sprint 6: Investigación de varios artículos

En el sprint 6, que empezó el 31 de octubre y terminó el 16 de noviembre, se realizó una búsqueda en el estado del arte del campo que estábamos estudiando, para poder ver alguna otra alternativa, ya que extraer datos de una base de datos tan grande resultaba complicado.

En el sprint planning se enunmeraron una serie de artículos de investigación realtivos al análisis de cambios de código y de detección de fallos. Se decidió que tanto los tutores como el alumno iban a tratar de recopilar la información más importante para ponerla en común en el sprint review.

Durante el sprint se estudiaron varios de estos artículos. Durante la sprint review se comentaron aspectos destacables de cada proyecto, aunque se encontraron 2 que encajaban en nuestro problema mucho mejor de los demás [19] [20].

4.2.7 Sprint 7: Artículos de predicción de bugs y descarga de su base de datos

En el sprint 7, que tuvo lugar entre el 16 de noviembre y el 30, se decidió estudiar y comprender los artículo relativos a la caracterización de los fallos de código en repositorios compartidos [19], así como el artículo en el que mostraban una base de datos de bugs pública [20].

En el sprint planing se decidió que este conjunto de datos podría proporcionar una información más útil que la encontrada en TravisTorrent y GitHubTorrent, por lo que se decide estudiar ambos artículos y descargar la información.

Durante el desarrollo del sprint se consiguen descargar estos datos de manera satisfactoria, y mucho más rápido que la base de datos de GitHubTorrent ya que el volumen de datos que manejaban no era tan alto. También se leyeron los artículos correspondientes.

En el sprint review se explica la estructura de los datos y se dan por buenos, de manera que en los próximos sprints se decide utilizar este conjunto de datos y no los extraídos antes.

4.2.8 Sprint 8: Generar un primer modelo y extraer características de un commit

En el sprint 8, que va desde el 30 de noviembre hasta el 14 de diciembre, se decide bifurcar el trabajo en dos áreas: la creación de un modelo de predicción y el avance en el desarrollo del servicio.

En el sprint planning se decide que, una vez encontrada la base de datos de [20], se puede intentar crear un modelo predictivo a partir del mismo. Además, para poder tratar casos futuros, también es necesario poder extraer las características de un commit con SourceMeter, la herramienta utilizada en [20], para que las características sean las mismas tanto en el modelo como en los commits estudiados.

En el desarrollo del sprint, relacionado con la creación del modelo se optó por un pequeño estudio de distintos modelos para un subconjunto balanceado de los datos. Tras esto se obtiene un modelo RandomForest que conseguía sobre el propio subconjunto de datos unos resultados en la validación cruzada de un 70% de acierto. Además, se estudia como utilizar SourceMeter para extraer las características de una clase modificada en un commit, y se consigue extraer la misma información que la recogida en la base de datos.

En el sprint review, se explica el modelo de predicción creado y cómo se han extraído las características a partir de un commit. El modelo, al ser una primera versión, se da como bueno a falta de poder encontrar otro con unos resultados mejores.

4.2.9 Sprint 9: Estudiar entornos para desplegar: DeployHub y Heroku

El sprint 9 duró un mes, del 14 de diciembre de 2018 al 14 de enero de 2019, ya que había festividades que provocaron que se alargara la duración de este sprint. En este mes nos centramos en estudiar plataformas de despligue, para poder crear el servicio en la nube.

Durante el sprint planning se decidió comenzar a estudiar DeployHub, una herramienta de despliegue. Se decidió que en este sprint se estudiara esta herramienta y se intentara desplegar el servicio aquí.

Durante el desarrollo del sprint se comenzó a estudiar esta herramienta, pero a medida que se iban buscando las funcionalidades de la misma se comenzó a ver que había documentación muy poco actualizada o directamente inexistente, y su funcionamiento no era sencillo. Por ello, a mitad del sprint se propuso Heroku en lugar de DeployHub para este despliegue. Heroku, a diferencia del anterior, estaba mucho mejor documentado por lo que se pudo ver su funcionalidad de manera más sencilla. Por lo tanto, se comenzaron a hacer pruebas sobre Heroku, aunque el servicio no quedó completamente desplegado por conflictos a la hora de ejecutar SpurceMeter en esta plataforma.

En el sprint review se explicó la elección de Heroku como herramienta de despliegue y se contó el problema con la ejecución de SourceMeter en ella. Tanto tutores como alumno quedaron en intentar buscar una solución para este problema en el siguiente sprint.

4.2.10 Sprint 10: corregir problemas SourceMeter

Como se comentó en la sprint review anterior, en este sprint, que empieza el 14 de enero de 2019 y acaba el 30 de enero, nos centraremos en subsanar los problemas relativos a la ejecución de SourceMeter en Heroku. En el sprint planning solamente se decide continuar con este problema.

Durante el desarrollo del sprint se intentó arreglar el problema. Tras muchos intentos fallidos de despliegue, de la modificación del valor de las variables de entorno y de arreglar problemas derivados del desarrollo en Windows (ya que Heroku trabaja sobre Linux y había discrepancias), con la ayuda de los tutores se consiguió solucionar el problema.

En el sprint review se revisó la solución planteada, y al observar que el problema quedaba arreglado se pudo cerrar la tarea más complicada del proyecto.

4.2.11 Sprint 11: Mejora del servicio. Eliminar archivos al acabar de procesar y crear workers que trabajen en segundo plano

Una vez arreglados los fallos presentes en el despliegue, en este sprint, que empieza el 30 de enero y finaliza el 8 de febrero, se pulen detalles del servicio. En el sprint planning se decide que es necesario tanto gestionar el borrado de ficheros cuando dejan de ser necesarios, como incluir el uso de un worker que realice las tareas más pesadas en un segundo plano.

En el desarrollo del sprint se elaboran los dos puntos mencionados antes, logrando una versión estable del proyecto. Por lo tanto, en el sprint review se muestra el resultado del servicio.

4.2.12 Sprint 12: Creación de un modelo de predicción con mejores resultados

En el sprint 12, que abarca del 8 al 22 de febrero, nos centramos en la mejora del modelo de predicción, puesto que ya hemos conseguido una versión del servicio que está desplegada en la nube.

En el sprint planning se decide realizar un análisis de los datos para poder crear un modelo de predicción que proporcione mejores resultados que el actual, así como el inicio de la estructura de la memoria final.

Durante el desarrollo de este sprint se desarrolla un cuaderno Jupyter con el análisis de las variables que proporciona SourceMeter para cada una de las clases de un proyecto. En este análisis, que se explica más en detalle en el capítulo 5, se observa la distribución de las distintas características para realizar un primer preprocesamiento antes de la creación de modelos de predicción.

Tras este análisis, se intenta crear un modelo que proporcione unos buenos resultados, pero no se consigue. En el sprint 8, en el que se crea un primer modelo básico, los resultados se mostraron en base al subconjunto balanceados de datos utilizados. Sin embargo, al utilizar los datos completos los resultados arrojados son bastante peores.

En el sprint review se comentan las conclusiones del análisis realizado, y se explica que los modelos creados no proporcionan unos resultados tan positivos como los expuestos en [20].

4.2.13 Sprint 13: Creación de modelos interpretables

El sprint 13 empieza el 22 de febrero y finaliza el 8 de marzo. En el sprint planning se decidió intentar crear un modelo de predicción interpretable. Ya que los resultados de los otros modelos no eran todo lo bueno que se deseaba, se intenta al menos que la solución que aporten sea interpretable de cara al usuario.

Durante el desarrollo de este sprint se intentan crear modelos a partir de dos vías: árboles de deicisón y regresión logística con regularización. Sin embargo, en ninguno de los dos casos se consiguen unos resultados buenos. Además, se comienza con el capítulo introductorio de la memoria. En el sprint review se vuelven a mostrar estos resultados negativos y se revisa la introducción que se había escrito.

4.2.14 Sprint 14: Mejorar resultados del modelo y comenzar estado del arte

El sprint 14 transcurre entre el 8 y el 29 de marzo, y en él intentamos tanto continuar con nuestra creación de modelos válidos como escribir nuevas secciones en la memoria final.

En el sprint planning se proponen diversas vías para poder conseguir un modelo de predicción aceptable: utilizar una máquina vector soporte (SVM), y modificar el conjunto de entrenamiento replicando casos de la clase más escasa (las clases con bugs) y eliminando casos de la otra clase (elementos sin errores).

Sin embargo, durante el desarrollo de este sprint tampoco se consiguen buenos resultados. En el sprint review se comentan los intentos realizados durante este sprint, pero los resultados siguen sin ser tan buenos como se esperaban. Se opta, por tanto, por buscar otra alternativa.

4.2.15 Sprint 15: Modificar el servicio para que haga un análisis descriptivo

En este sprint, que empieza el 29 de marzo y finaliza el 5 de abril, se cambia el punto de vista del servicio. En el sprint planning se elige una alternativa a la creación del modelo predictivo: hacer el modelo descriptivo, de manera que se muestre al usuario información de las clases modificadas, pero sin realizar una predicción de la presencia de fallos.

Durante este sprint se buscan las 10 variables más correlacionadas con la presencia de errores. Este análisis se explica más en profundidad en la sección 3.2. En el sprint review se muestran las 10 variables más correlacionada sy se explica cómo ha cambiado el servicio para hacerlo descriptivo.

4.2.16 Sprint 16: Conseguir que el usuario nos proporcione su repositorio local

En el sprint 16, que abarca del 5 al 29 de abril, se decide refinar el servicio una vez conseguido que sea descriptivo. En el sprint planning se decide que es conveniente que, en lugar de indicar el usuario un enlace a un repositorio nos proporcione su repositorio local para poder analizar directamente los archivos modificados en su entorno local.

Durante este sprint se realiza esta tarea, modificando la lógica del servicio para que consiga recibir el repositorio comprimido que envíe el usuario, extraiga los cambios que ha hecho en su entorno local y realice un análisis exclusivamente de los archivos modificados en estos cambios.

En el sprint review se muestra el servicio funcionando correctamente en el entorno local, aunque al desplegarlo en la nube se producían algunos problemas, como por ejemplo que Heroku no permite volúmenes compartidos entre contenedores, y estábamos utilizando esa funcionalidad en nuestro entorno local.

4.2.17 Sprint 17: Crear flujo de CI/CD

Enel sprint 17 se finaliza el desarrollo del servicio, creando el flujo de Integración y Despliegue Continuo de manera que cualquier cambio de código futuro pueda desplegarse automáticamente. Este sprint empieza el 29 de abril y finaliza el 17 de mayo.

En el sprint planning se decide utilizar una alternativa a los volúmenes compartidos entre contenedores, ya que Heroku no permite esta característica, así como acabar de implementar el flujo de CI/CD.

Durante el desarrollo del sprint, se estudian varias vías para transferir archivos entre web y worker para eliminar el volumen compartido. Finalmente se opta por Amazon S3, ya que tiene una capa gratuita que nos permite almacenar hasta 5GB de manera gratuita y la transferencia de archivos mediante su API es bastante sencilla [26]. Por otra parte, con la ayuda de los tutores se crea el flujo de CI/CD, de manera que cualquier nuevo cambio de código podría desplegarse de manera automática siempre que se pasen todas las pruebas de manera satisfactoria.

En el sprint review se repasa el servicio completo y se decide que, una vez creado el flujo de CI/CD, el desarrollo del servicio queda finalizado y ya solo quedaría avanzar con la documentación escrita.

4.2.18 Sprint 18: Elaborar lo que falta de la memoria

En el sprint 18, que va del 17 de mayo hasta el 31 de junio, únicamente se avanza en la memoria, añadiendo los puntos que faltaban y corrigiendo aquellos que los tutores van indicando durante sus revisiones.

En el sprint planning se repasa la estructura final, indicando qué partes faltan y cuáles se deben corregir. Durante todo este sprint se van añadiendo y arreglando los puntos indicados por los tutores. (FALTA ESTE SPRINT REVIEW)

4.2.19 Sprint 19: Ultimar detalles de la memoria

El último sprint del proyecto comienza el 1 de junio y finaliza el 21, una semana antes de la fecha límite para el depósito del TFM. (SPRINT PLANNING, DESARROLLO Y REVIEW POR DETERMINAR)

4.2.20 Resumen

Como conclusión de esta parte, en la Imagen 4.4 se aprecia el diagrama de Gantt correspondiente a este trabajo. Al ser una metodología ágil, se ha considerado el desarrollo como un todo, aunque se ha discriminado entre el desarrollo del propio servicio y la documentación. También se ha separado la parte previa del estudio de conceptos básicos, ya que es necesario para la creación del servicio, pero no forma parte de ello. Además, se han destacado algunos de los hitos del proyecto, como la entrega del anteproyecto y de la memoria, o la creación del flujo de trabajo de Integración y Despliegue Continuo.

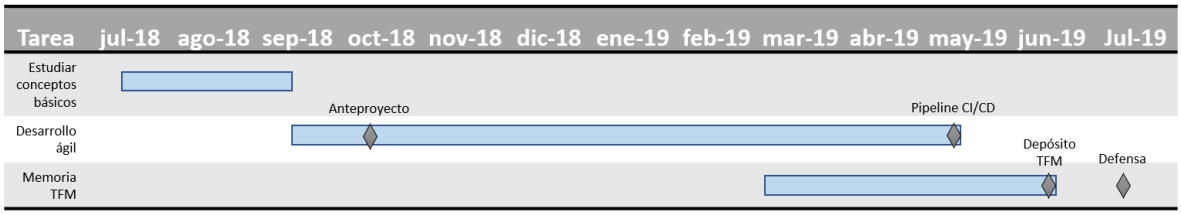

Imagen 4.4: Digrama de Gantt del proyecto

CAPÍTULO 5 DESPLIEGUE CONTINUO

En este capítulo se entrará más en detalle sobre cómo se ha implementado el servicio y la estructura utilizada para su despliegue, enumerando y explicando cada una de las tecnologías utilizadas durante la evolución de este servicio.

En la Imagen 5.1 se muestra el *pipeline* o flujo de trabajo adoptado durante este trabajo. En primer lugar, se realizan los cambios convenientes en el entorno de trabajo del usuario. A continuación, cuando se tiene un cambio que se quiere llevar al despliegue final, se suben los cambios al repositorio remoto, en nuestro caso GitHub. También se ha utilizado ZenHub[3], una herramienta que complementa a GitHub para la gestión ágil del proyecto.

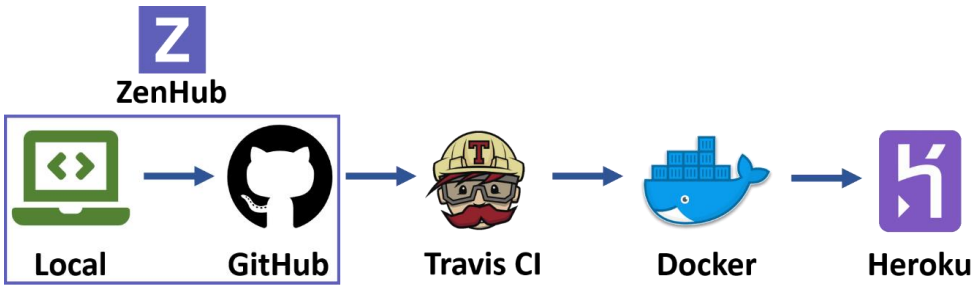

Imagen 5.1: Pipeline de desarrollo

Tras subirse a GitHub, automáticamente Travis CI [18], que es la herramienta de Integración Continua que hemos utilizado, ejecutará el plan de pruebas que hayamos creado para comprobar que los desarrollos realizados no han provocado variaciones en el comportamiento usual del servicio. Si las pruebas no se pasan, se avisaría al usuario y se procedería a arreglar los fallos. Si las pruebas son correctas, se despliega automáticamente. Para ello, se crean una serie de contenedores Docker[4] con funciones diferenciadas, y estos contenedores se despliegan con Heroku[5], dejando el servicio completamente funcional.

[3] https://www.zenhub.com/
[4] https://www.docker.com/
[5] https://www.heroku.com/

5.1 Desarrollo

El proyecto se ha desarrollado con el lenguaje de programación Python, utilizando un framework denominado Django[6]. Django es un framework de alto nivel de Python para desarrollo web que trata de conseguir un desarrollo rápido y un diseño limpio. Este framework se basa en una arquitectura software conocida como modelo-vista-controlador [27], o simplemente MVC, que trata de separar el desarrollo en tres capas diferenciadas: el modelo, que define estructura de los datos; el controlador, que gestiona las peticiones del usuario; y la vista, que maneja la presentación del contenido.

La parte del controlador es la encargada de gestionar los distintos tipos de llamada que realiza el usuario. En este caso, solo se ha proporcionado una posible URL, a la cual el usuario accede e introduce el repositorio que desea analizar.

En la parte de la vista se crean las distintas plantillas que se utilizan para mostrar la información. Aquí se han tenido que crear tres diferentes: una para permitir al usuario introducir su repositorio; otro para indicarle que la petición se está procesando y debe esperar; y una última para mostrar los resultados del análisis una vez acabado.

Por último, en cuanto al modelo de datos, no se ha creado una estructura específica para este servicio. En esta capa se suelen definir las entidades que forman la base de datos si se necesitan almacenar los datos a largo plazo. Sin embargo, ya que en nuestro caso únicamente queremos ofrecer los resultados cuando se procesan, pero no guardarlos durante más tiempo, tan solo habrá que guardar la información temporalmente y mostrar al usuario los resultados cuando estén calculados.

Todo el código ha sido gestionado utilizando GitHub para el control de versiones. Además, para facilitar la gestión ágil del proyecto se ha utilizado ZenHub[7], lo que nos permite manejar y visualizar qué incidencias estamos tratando en cada momento, como se puede ver en el ejemplo de la Imagen 5.2.

5.2 Integración Continua

Cada vez que se hace un nuevo incremento de nuestro servicio, se suben los cambios a GitHub. Al subirlo, y gracias a la integración que Travis CI tiene con GitHub, el proyecto pasa una serie de pruebas automáticas para asegurar que el comportamiento sigue siendo el esperado en todos los componentes de nuestro servicio. Si las pruebas que hemos definido

[6] https://www.djangoproject.com
[7] https://www.zenhub.com

fallan, se avisa al equipo y el despliegue se anula, con el objetivo de que arreglen los defectos encontrados. Si son correctas, como se ve en el ejemplo de la Imagen 5.3, comienza el proceso de Despliegue Continuo.

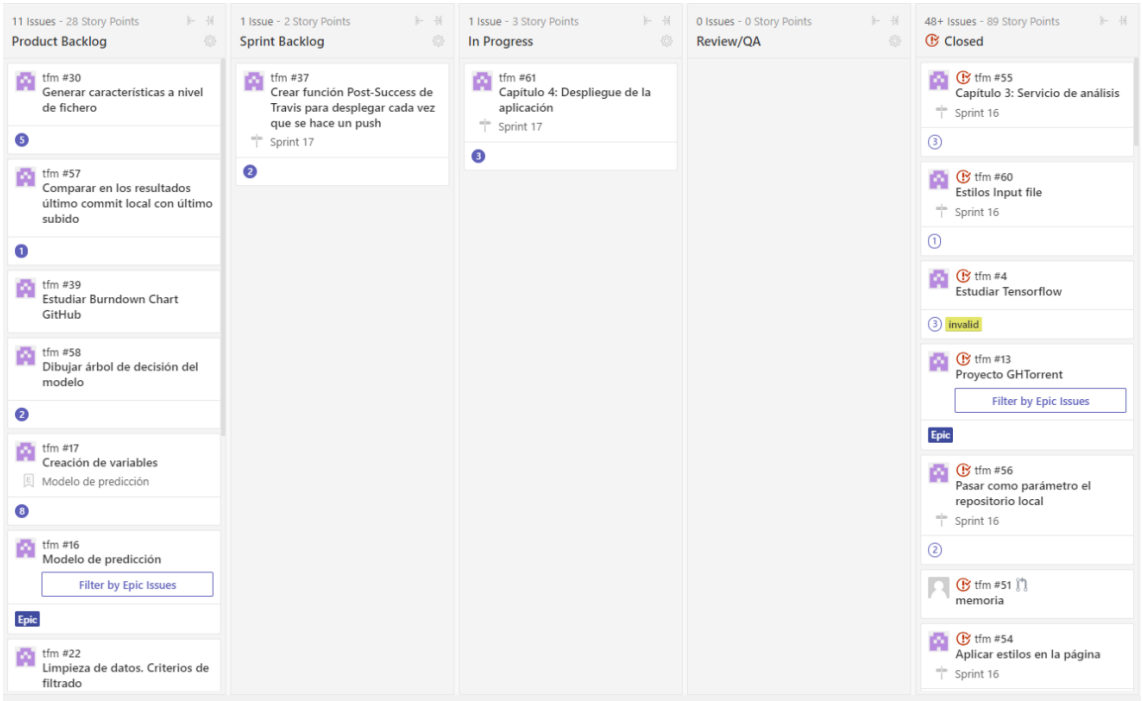

Imagen 5.2: Visualización de incidencias en ZenHub

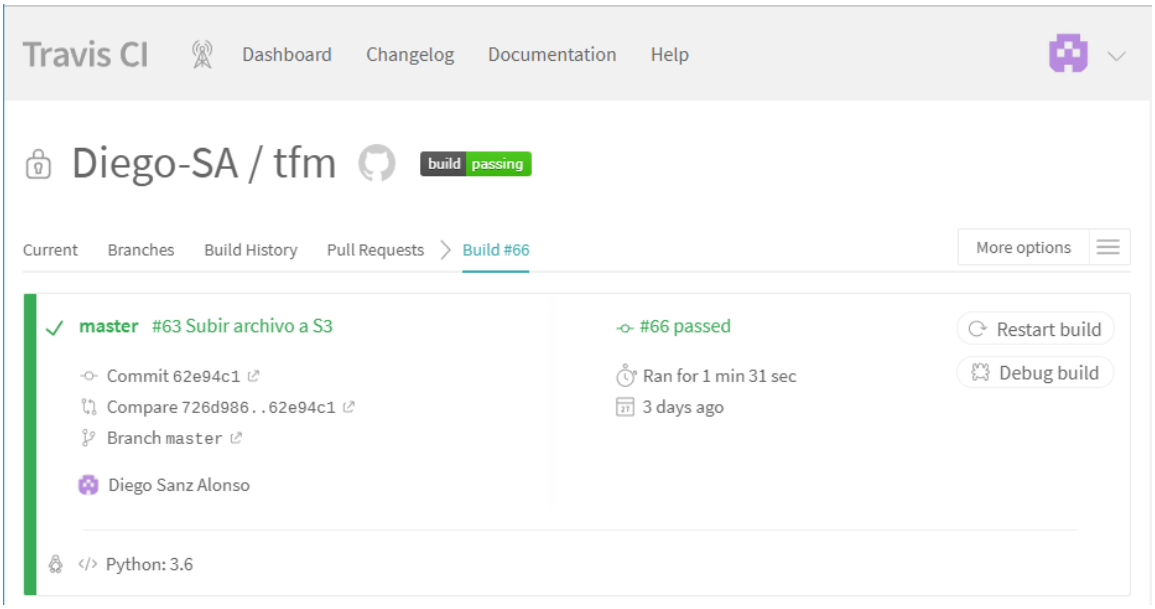

Imagen 5.3: Ejemplo Build en Travis CI

Para realizar el proceso de Despliegue Continuo se ha definido en la configuración de Travis que, tras pasar correctamente las pruebas, comience la creación de los contenedores y el posterior despliegue. La aplicación está desplegada en la nube con Heroku[8]. Esta plataforma permite ejecutar aplicaciones en múltiples lenguajes de programación y con una escalabilidad muy rápida. En nuestro caso, desplegaremos en Heroku utilizando Docker Compose, una herramienta para orquestar un conjunto de contenedores Docker de manera simultánea [28]. En total, la aplicación está formada por 3 contenedores Docker, que se organizan como muestra la Imagen 5.4.

El primero de ellos, al que se ha denominado *Web*, es el encargado de las peticiones web, de manera que gestionará todas las llamadas que los usuarios realicen. Este contenedor será el responsable de recibir el repositorio que se va a analizar y guardarlo en Amazon S3[9], el servicio de almacenamiento de Amazon, además de mostrar los resultados finales.

El segundo contenedor, denominado *Worker*, hará las funciones más pesadas, de forma que se ejecuten en segundo plano y el usuario no tenga grandes tiempos de espera. La función principal de este contenedor es la de procesar los repositorios que los usuarios proporcionan para realizar el análisis explicado en el capítulo anterior. Este contenedor descarga los repositorios almacenados en S3 por el contenedor anterior.

El tercer y último contenedor es el encargado de comunicar los dos anteriores, de manera que cuando un usuario realiza una petición para analizar su repositorio, se almacene mediante el primer contenedor, y este mande un mensaje al segundo para que realice el análisis. Para este contenedor se ha utilizado una librería de Python denominada Redis Queue (RQ) [29].

En la Imagen 5.4 se puede ver un esquema de los contenedores que conforman nuestro servicio y las interacciones entre ellos. Como se aprecia, los contenedores Web y Worker se basan en nuestra aplicación de desarrollada en Django, y la Web se encarga de subir información a Amazon S3 mientras que el Worker solo la descarga. Por su parte, el tercer contenedor parte de una imagen de Redis, siendo la encargada de recibir las peticiones de los otros dos contenedores para informar cuándo debe empezar un proceso o cándo ha terminado.

Hay dos razones por la que se utiliza un servicio externo como Amazon S3 de forma temporal para el envío de información entre Web y Worker. En primer lugar, el sistema de archivos de Heroku es efímero[10] [30], por lo que si se intenta procesar un repositorio y antes

[8] https://www.heroku.com/
[9] https://aws.amazon.com/es/s3/
[10] Cada vez que se reinicia el servicio, los datos almacenados se eliminan

de acabar se para o se reinicia alguna de las instancias, el proceso fallaría puesto que el repositorio ya no estaría almacenado. Además, en el despliegue con Heroku no se permite el uso de volúmenes de datos compartidos entre contenedores, por lo que necesitamos una herramienta que actúe de intermediaria entre ellos.

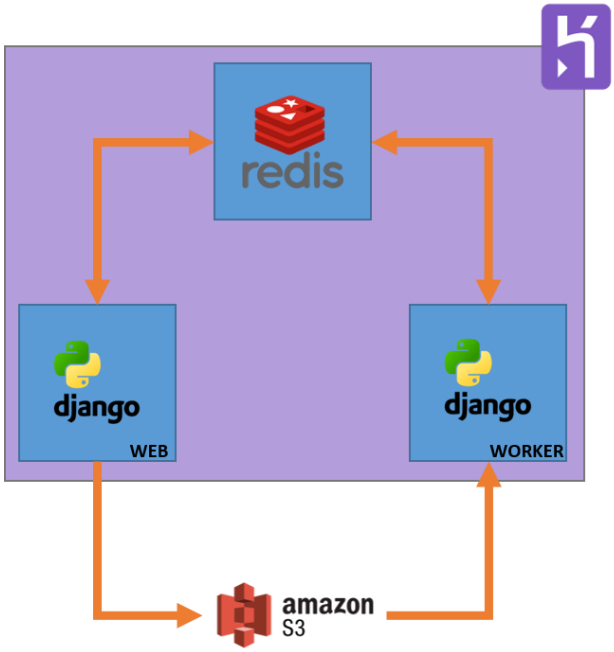

Imagen 5.4: Diagrama de Despliegue

Para desplegar el servicio debemos, en primer lugar, crear la configuración de los contenedores Docker de la web y el worker. Dicha configuración puede consultarse en el Anexo A.1. A continuación, es necesario definir la configuración de Docker Compose, indicando en cada uno de los tres contenedores su origen, que para los dos primeros partirá de las imágenes creadas en los archivos Docker mencionados antes, y para el contenedor de Redis se usará una imagen predeterminada, como se aprecia en la imagen anterior. Esta configuración puede verse en el Anexo A.2. Una vez hechos estos archivos de configuración, Travis CI es capaz de desplegarlo automáticamente en Heroku, de manera que el servicio se quede completamente operativo con cada incremento de código realizado. La configuración establecida en Travis se encuentra en el Anexo A.3.

En la Imagen 5.5 se puede apreciar el resultado de nuestro pipeline de CI/CD. En primer lugar, y tras cargar una serie de variables de entorno definidas en Travis, se ejecutan las pruebas. Si estas son correctas, como en este caso, comienza el despliegue. Para ello, en primer lugar, se instalan las dependencias necesarias para nuestro proyecto. A continuación, se prepara el despliegue mediante la creación del Docker Compose explicado antes. Finalmente, con Heroku, se despliega el resultado final para que el usuario final pueda acceder a nuestro servicio vía web como se muestra en la Imagen 3.4.

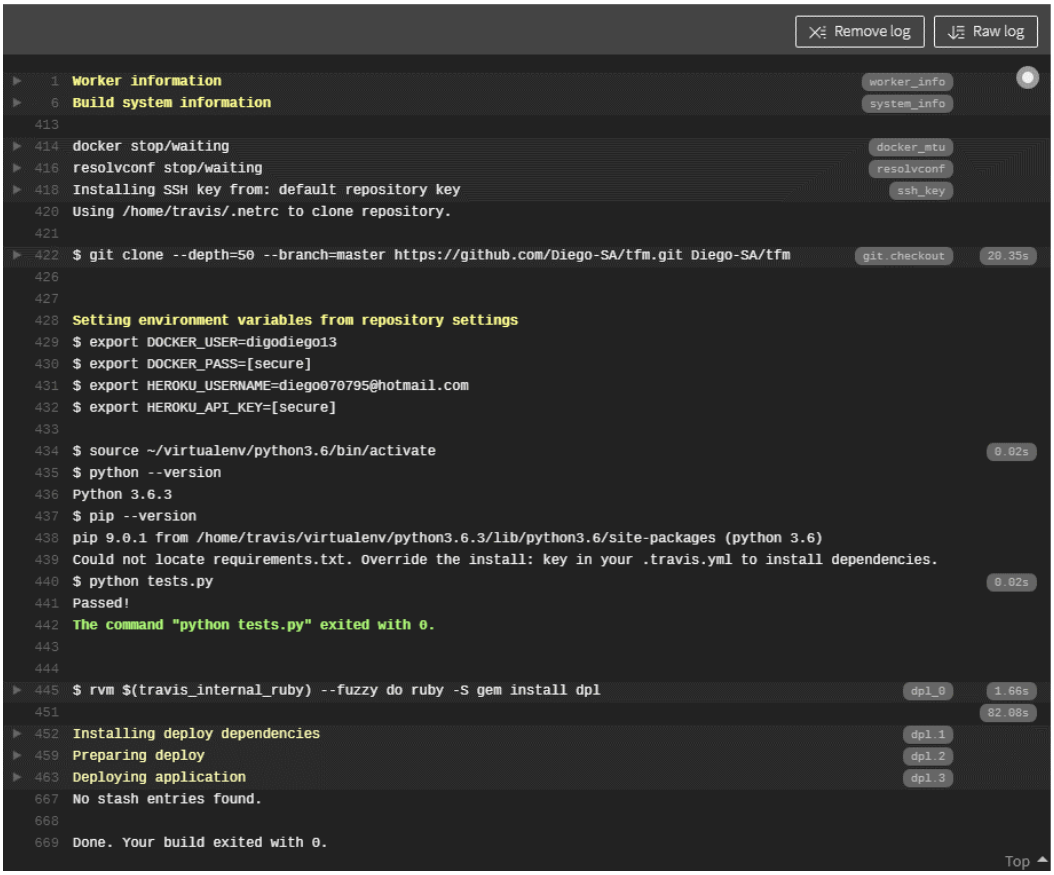

Imagen 5.5: Resultado de los scripts de CI/CD en Travis

CAPÍTULO 6
ANÁLISIS DE DATOS

En este capítulo se hará un resumen del estudio realizado con los datos de la base de datos de bugs disponible gracias a [20]. Se explicará el análisis exploratorio realizado para estudiar tanto la variable que queremos predecir, también llamada variable clase, que en este caso es el número de bugs; como las variables predictoras, que son las distintas métricas que se extraen de cada clase del proyecto. Además, se enumerarán las variables escogidas a mostrar al usuario, ya que como se comentó en el capítulo 3 el servicio finalmente sigue un método descriptivo.

En la base de datos de [20] disponemos de un total de 182.671 registros. Cada uno de ellos hace referencia a una clase de un proyecto Java, y en todos ellos vienen recogidas distintas métricas de código estático (un total de 105) y el número de bugs que presentan. El análisis de datos completo se puede encontrar en el repositorio del proyecto, aunque aquí se comentarán los puntos principales del estudio.

6.1 VARIABLE CLASE

La primera variable que se debe analizar es la más importante. En este caso es el número de errores que existen en una clase. El objetivo del servicio consiste en ayudar a los desarrolladores a encontrar fallos en sus cambios de código, por lo que todo el análisis debe centrarse en este punto.

En la Imagen 6.1 se muestra una gráfica con la distribución de los números de bugs presentes en cada una de las clases disponibles en la base de datos. Se puede apreciar como casi todos los valores son igual a 0. Esto provoca que no se vea con precisión el valor en el resto de los valores. Para subsanar esto, en la Imagen 6.2 se ha cambiado el eje y a una escala logarítmica. En esta escala se puede observar que en las clases que tienen bugs, predomina un número reducido de errores.

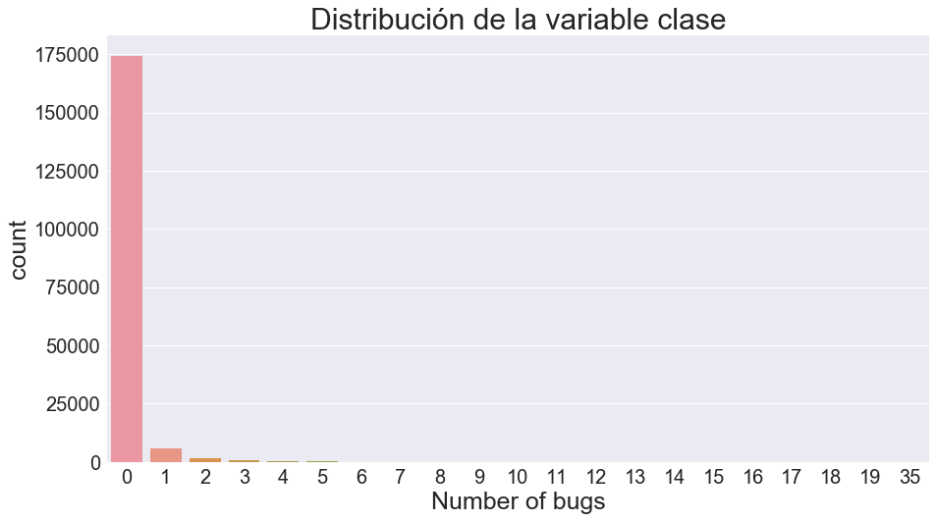

Imagen 6.1: Distribución de la variable "Número de bugs"

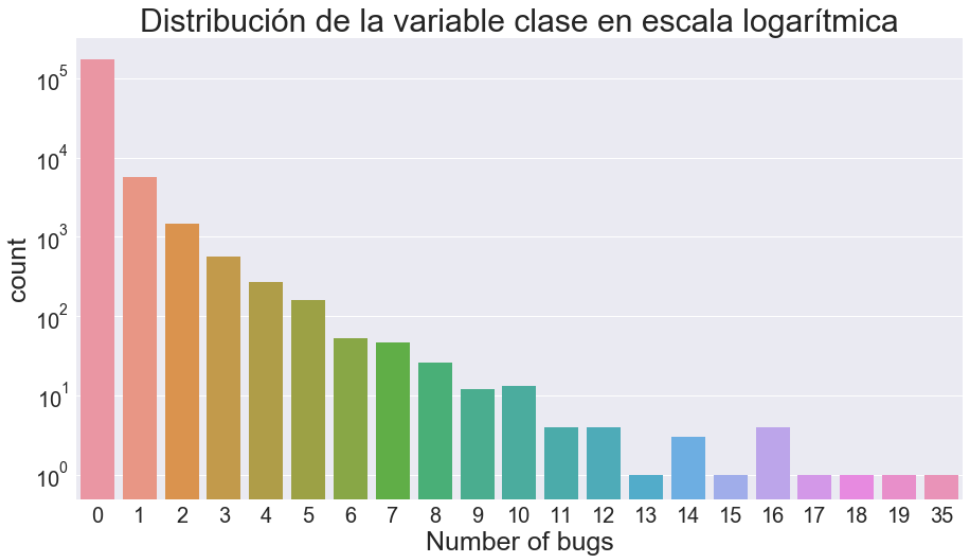

Imagen 6.2: Distribución de la variable "Número de bugs" en escala logarítmica

Sin embargo, no necesitamos un valor exacto del número de errores en cada clase. El objetivo del servicio es ayudar al desarrollador a que detecte que sus cambios presentan fallos o no, pero determinar el número es una tarea más compleja. Además, de cara a la creación de un posible modelo de predicción, es conveniente hacer una partición binaria de esta variable, discriminando entre clases que no contienen bugs, y clases que sí lo tienen.

Si realizamos esta partición binaria, nos encontramos con un total de 8.275 clases con bugs, frente a las 174.396 clases que no tienen errores conocidos. Esto supone tan solo el 4,53% de los datos, por lo que nos encontramos ante un conjunto de datos claramente

desbalanceado. Si se quiere crear un modelo de predicción es conveniente tener este factor en cuenta.

6.2 VARIABLES PREDICTORAS

Tras estudiar la variable clase y binarizarla, se puede comenzar el análisis del resto de características. Hay un total de 106 variables, representando cada una de ellas una métrica de código estático. Debido a este número tan elevado de características, se procederá a intentar eliminar variables que no muestren información relevante.

Esta ausencia de información puede deberse a dos factores principales. En primer lugar, puede existir una varianza muy baja entre los valores de una de las características, por lo que no aportará valor al estudio. Aquí hay que tener en cuenta que, al tener una base de datos desbalanceada, una varianza no muy alta no tiene por qué suponer que la métrica que se intenta estudiar no sea útil. Por ello, únicamente se eliminarán del estudio aquellas variables que tengan prácticamente todos sus valores iguales, ya que solo aportarían ruido en nuestro análisis.

El segundo punto a tener en cuenta es la correlación. Existen métricas que pueden aportar una información muy similar a otra. Por poner un ejemplo más específico, una de las variables disponibles es el número de líneas de código de una clase. Esta variable podría ser muy similar al número de líneas lógicas de la clase, que solo variaría si existiesen dos instrucciones en la misma línea física, algo no muy común. Sin embargo, es necesario calcular la correlación para asegurar la similaridad de información de estas clases. En este caso, la correlación es del 98.4%, por lo que se puede eliminar una de las dos sin apenas perder información.

Se ha realizado este análisis en todas las características, agrupándolas en un total de 10 categorías, y estudiando la relación únicamente entre las características del mismo grupo. A continuación, se procederá a explicar cada uno de estos grupos.

6.2.1 Posición de la clase

En este primer grupo tenemos un total de 4 características, que indican la posición de la clase dentro del fichero global. Esta posición viene indicada a nivel de línea y de columna, guardando tanto el inicio como el final de ambos.

6.2.2 Métricas relacionadas con código clonado

En el siguiente bloque se encuentran las características relacionadas con código clonado. En este grupo encontramos hasta 8 métricas diferentes, referidas al número de líneas clonadas, al porcentaje de líneas clonadas sobre el total, al número de clases clones, etc. Sin embargo, de esas 8 se han eliminado 3 por tener una correlación muy alta con otra métrica, por lo que al mostrar prácticamente la misma información podemos eliminar variables innecesarias.

6.2.3 Métricas de cohesión

En este grupo solo hay una métrica, denominada índice de cohesión de métodos o *Lack of cohesion in Methods* (LCOM5), que indica el uso que hacen los métodos de los atributos en una clase. Esta métrica será 0 cuando no tenga métodos; 1 cuando tiene el valor ideal, que se produce cuando todos los métodos de una clase están relacionados tanto de forma directa como indirecta; o mayor de 1, cuando la clase probablemente pueda ser descompuesta en 2 o más clases [31]. Por lo general valores mayores de 1 no serían recomendables, salvando excepciones como clases de utilidades. En la Imagen 6.3 se observa la distribución de esta métrica (se han eliminado valores mayores de 10, que eran residuales, para favorecer la correcta visualización del gráfico).

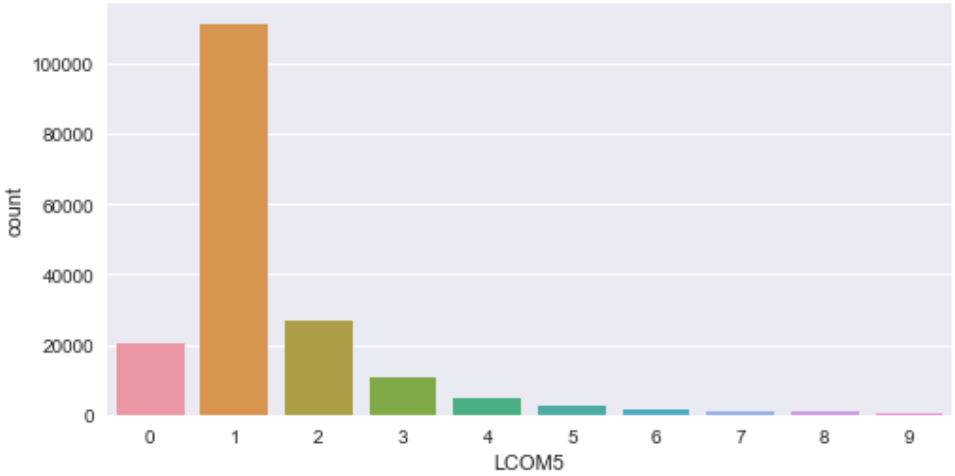

Imagen 6.3: Distribución de la métrica LCOM5

6.2.4 Métricas de complejidad

En el siguiente grupo se encuentran las métricas relacionadas con la complejidad de la clase: grado de anidamiento, número de caminos independientes de una clase, etc.

6.2.5 Métricas de acoplamiento

El siguiente grupo es el de las métricas de acoplamiento, que engloba las características que tienen que ver con el uso de unas clases sobre otras.

6.2.6 Métricas de documentación

En este grupo se encuentran hasta un total de 8 características relacionadas con la cantidad de documentación presente en un proyecto. Sin embargo, en este grupo se pueden eliminar 2 de las características al tener una correlación muy alta con respecto a otras de esta categoría.

6.2.7 Métricas de herencia

En el siguiente grupo tenemos métricas relacionadas con herencia, como la profundidad del árbol de herencia (tamaño del camino de la clase al ancestro más lejano en su árbol de herencia), número de clases padres, hijas, etc.

6.2.8 Métricas de tamaño

En esta octava categoría se encuentran métricas relacionadas con el tamaño de la clase, en términos de líneas de código, atributos, métodos, etc. Este grupo es de los más numerosos ya que se hacen distinción entre elementos locales de la clase y globales, públicos y privados, etc. Esta categoría tiene hasta 20 características, aunque se pueden eliminar 2 debido a su correlación tan fuerte con otras características de este grupo.

6.2.9 Warnings

En este grupo se indican el número de advertencias que se detectan en la clase, distinguiendo entre bloqueantes, críticas, de importancia alta, de importancia baja e informativas.

6.2.10 Cumplimiento de reglas

En este último grupo se guardan distintas características relativas a la violación de reglas de buenas prácticas desde diferentes puntos de vista (de tamaño de la clase, del uso y colocación de llaves, de clonado de código…). Hay un total de 37 variables en esta categoría, aunque se han podido eliminar hasta 21 debido a su baja varianza, ya que en muchos casos incluso no existían valores distintos de 0.

6.2.11 Conclusiones

Siguiendo el primer criterio se han eliminado 14 variables, y teniendo en cuenta el segundo se han eliminado 21. Esto provoca que el conjunto final de variables descienda a 76, reduciendo en un 33% el número de características de nuestra base de datos.

Esta limpieza de los datos permitiría una creación de un modelo de predicción con mucho menos ruido que el conjunto de datos original. Sin embargo, ya que en este proyecto nos hemos centrado en el carácter descriptivo, el número de métricas sigue siendo excesivo, por lo que hay que hacer una selección entre las variables restantes.

6.3 SELECCIÓN DE VARIABLES

Para realizar esta selección se ha calculado la correlación entre cada una de las variables predictoras con la variable clase, y se han ordenado en función de este valor. Posteriormente, se han analizado las variables en orden descendente en función de esta correlación, eliminando variable que, aunque no tuvieran una correlación muy alta entre ellas, mostraban una información similar. Este tipo de variables no ayudarán al usuario, ya que si ponemos variables muy parecidas el valor de la información será escasa.

Por ejemplo, una de las variables con una de las correlaciones más alta con la variable clase es el grado de anidamiento. Existen dos métricas que muestran esta información, aunque con un matiz diferencial. Una de ellas cuenta sentencias consecutivas de if-else-if como una sola instrucción, mientras que la otra las cuenta como dos. Estas variables no están tan correlacionadas como podría parecer a priori (aunque tiene un valor alto, un 81,59%), por lo que se han mantenido ambas en el estudio. A pesar de ello, de cara al usuario final es indiferente mostrar una u otra, por lo que la otra se excluirá de los resultados.

Con todo, se han seleccionado un total de 10 métricas que se mostrarán al usuario. Estas métricas son las siguientes:

- *Coupling betweeen Object classes*, número de usos de otras clases. Clases con este valor muy alto serán muy dependientes de otros módulos, y por tanto más difíciles de testear y utilizar, además de muy sensibles a cambios.
- *Nesting Level Else-If*, grado de anidamiento máximo de una clase, contando bloques de tipo if-else-if como un solo nivel.
- *Response set for Class*, combinación de número de métodos locales y métodos llamados de otras clases.
- *Complexity Metric Rules*, violaciones en las buenas prácticas relativas a métricas de complejidad.

- *Weighted Methods per Class*, número de caminos independientes de una clase. Se calcula como la suma de la complejidad coclomática de los métodos locales y bloques de inicialización.
- *Documentation Metric Rules*, violaciones de buenas prácticas relativas a la cantidad de comentarios y documentación.
- *Coupling Metric Rules*, violaciones en las buenas prácticas relativas al acoplamiento de las clases.
- *Total Number of Local Attributes*, número de atributos locales de cada clase.
- *Warning Info*, advertencias de tipo *WarningInfo* en cada clase.
- *Size Metric Rules*, violaciones en las buenas prácticas relativas al tamaño de las clases.

CAPÍTULO 7
CONCLUSIONES Y PROPUESTAS

7.1 CONCLUSIONES

El objetivo principal del trabajo consistía en poder advertir al programador de posibles fallos en su código antes de subirlo a su repositorio compartido online, con el fin de reducir el riesgo de que se interrumpa el flujo de trabajo de todo el equipo. Tras el desarrollo del proyecto, se ha conseguido crear un servicio descriptivo que muestra al usuario un subconjunto de las métricas más importantes desde el punto de vista de la detección de fallos.

No solo se ha implementado el servicio, sino que además se ha conseguido desplegar en la nube, de manera que cualquier usuario puede utilizar nuestro servicio vía web. Además, se ha creado un flujo de trabajo que incluye técnicas de Integración Continua y Despliegue Continuo, por lo que cada vez que se desarrolle un incremento nuevo de código se pasará un conjunto de pruebas automáticamente y, si no hay ningún fallo, el servicio se despliega automáticamente con los nuevos cambios, reduciendo así el tiempo entre el desarrollo de una nueva funcionalidad o un arreglo de algún fallo y la llegada de ese cambio al usuario final.

Además del servicio descriptivo, se ha intentado crear un modelo de predicción que nos diga automáticamente si las clases modificadas por un desarrollador tenían una alta probabilidad de fallo o no. Sin embargo, los resultados en este aspecto no son tan positivos como se esperaba, por lo que se ha mantenido como una versión en fase Beta que el usuario puede consultar, pero lo que impera será el carácter descriptivo del servicio.

7.2 TRABAJO FUTURO

Ya que el servicio se ha logrado desplegar y se ha creado el flujo de CI/CD, incluir nuevas funcionalidades debería ser una tarea sencilla. Si no se modifica la arquitectura, solo habría que realizar los cambios de código pertinentes, sin tener que preocuparnos de la parte operacional o de sistemas.

Uno de los puntos en los que se puede avanzar en este trabajo es la reducción de restricciones en los proyectos válidos para el estudio. Actualmente solo se tiene la capacidad de analizar proyectos con lenguaje de programación Java que estén alojados en GitHub. Se podrían incluir nuevos lenguajes de programación, teniendo en cuenta que las métricas pueden ser diferentes y habría que hacer un análisis exhaustivo en cada caso. Además, se podría permitir el estudio de repositorios en otras plataformas, como Bitbucket o GitLab[11]. En este caso habría que estudiar cómo proporcionar el repositorio al servicio, así como identificar los archivos que se han modificado localmente, pero la parte frontal del servicio no debería sufrir modificaciones en este aspecto.

Otro de los aspectos que se podrían tratar consiste en crear un modelo de predicción con unos buenos resultados. Para esto sería necesario buscar otra base de datos, ya que con la que se ha obtenido en [20] no se ha podido crear un modelo con un gran acierto. Una de las vías en las que se podría investigar consiste en buscar variables distintas a métricas estáticas de la clase. Por ejemplo, se podrían buscar características a nivel de fichero, si el proyecto contiene varias clases en un solo fichero, ya que es de esperar que, si eso ocurre, estas clases estén relacionadas entre sí y fallos en una de ellas puedan repercutir en las demás.

Otro tipo de características que podrían dar mucho valor a la hora de la creación de un modelo predictivo son aquellas relativas a información histórica de la clase. La mayoría de las plataformas de gestión de repositorio tienen control de versiones, por lo que a través del mismo podríamos sacar distinta información, como el número de veces que se ha cambiado una clase o el número de desarrolladores diferentes que han hecho algún tipo de cambio sobre la misma.

[11] https://about.gitlab.com/

BIBLIOGRAFÍA

[1] G. Kim, J. Humble y P. Debois, The DevOps Handbook, IT Revolutions, 2016.

[2] Verifysoft, «Halstead metrics,» 10 Agosto 2017. [En línea]. Available: https://www.verifysoft.com/en_halstead_metrics.html.

[3] C. Treude, L. Leite y M. Aniche, «Unusual events in GitHub repositories,» *The Journal of Systems & Software*, nº 142, pp. 237-247, 2018.

[4] «Guía de Scrum,» Noviembre 2017. [En línea]. Available: www.scrum.org.

[5] R. Jurney, Agile Data Science 2.0, O'Reilly Media, 2017.

[6] D. Petrov, «Data Version Control: iterative machine learning,» FullStackML, Mayo 2017. [En línea]. Available: https://www.kdnuggets.com/2017/05/data-version-control-iterative-machine-learning.html.

[7] T. Volk, «Seven steps to move a DevOps team into the ML and AI world,» DevOpsAgenda, Abril 2018. [En línea]. Available: https://devopsagenda.techtarget.com/opinion/Seven-steps-to-move-a-DevOps-team-into-the-ML-and-AI-world.

[8] Atlassian, «What is version control,» [En línea]. Available: https://www.atlassian.com/git/tutorials/what-is-version-control. [Último acceso: 07 Abril 2019].

[9] Smartsheet, «Software Version Control,» [En línea]. Available: https://www.smartsheet.com/software-version-control. [Último acceso: 07 Abril 2019].

[10] Apache Software Fundation, «Apache Subversion,» [En línea]. Available: https://subversion.apache.org/. [Último acceso: 2019 Abril 04].

[11] Git, «Git,» [En línea]. Available: https://git-scm.com/. [Último acceso: 2019 Abril 07].

[12] Atlassian, «Continuous Integration vs Delivery vs Deployment,» Atlassian, [En línea]. Available: https://es.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. [Último acceso: 07 Abril 2019].

[13] Y. Weicheng, B. Shen y B. Xu, «MiningGitHub: Why Commit Stops,» *Asia-Pacific Software Engineering Conference*, nº 20, pp. 165-169, 2013.

[14] G. Gousios, «The GHTorrent Dataset and Tool Suite,» de *Proceedings of the 10th Working Conference on Mining Software Repositories*, San Francisco, CA, USA, 2013, pp. 233-236.

[15] G. Georgios, «The GHtorrent Project,» 2013. [En línea]. Available: http://ghtorrent.org/. [Último acceso: 13 Marzo 2019].

[16] J. Cabot, J. L. Cánovas Izquierdo, V. Cosentino y B. Rolandi, «Exploring the Use of Labels to Categorize Issues in Open-Source Software projects,» *International Conference on Software Analysis, Evolution and Reenginering*, nº 22, 2015.

[17] M. Beller, G. Gousios y A. Zaidman, «TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration,» *Proceedings of the 14th working conference on mining software repositories*, 2017.

[18] Travis CI, «Travis CI,» 2011. [En línea]. Available: https://travis-ci.org/. [Último acceso: 13 Marzo 2019].

[19] P. Gyimesi, G. Gyimesi y Z. Tóth, «Characterization of Source Code Defects by Data Mining Conducted on GitHub,» *Computational Science and Its Applications - ICCSA*, vol. 9159, pp. 47-62, 2015.

[20] Z. Tóth, P. Gyimesi y R. Ferenc, «A Public Bug Database of GitHub Projects and its Application in Bug Prediction,» *Computational Science and Its Applications - ICCSA*, vol. 9789, pp. 625-638, 2016.

[21] FrontendArt, «SouceMeter - Free-to-use, Advanced Source Code Analysis Suite,» 2016. [En línea]. Available: https://www.sourcemeter.com/resources/java/. [Último acceso: 03 Abril 2019].

[22] S. Tuli, «Learn How to Set UP a CI/CD Pipeline From Scratch,» 10 Agosto 2018. [En línea]. Available: https://dzone.com/articles/learn-how-to-setup-a-cicd-pipeline-from-scratch. [Último acceso: 18 Abril 2019].

[23] XGBoost, «XGBoost Documentation,» [En línea]. Available: https://xgboost.readthedocs.io/en/latest/. [Último acceso: 19 Abril 2019].

[24] Django, «Escribiendo su primera aplicación en Django, parte 1,» [En línea]. Available: https://docs.djangoproject.com/es/2.0/intro/tutorial01/. [Último acceso: 2018 Agosto 1].

[25] Djangogirls, «Tu primer proyecto en Django,» [En línea]. Available: https://tutorial.djangogirls.org/es/django_start_project/. [Último acceso: 2018 Agosto 20].

[26] Amazon, «Operation on Objects - Amazon Simple Storage Service,» 1 Marzo 2006. [En línea]. Available: https://docs.aws.amazon.com/es_es/AmazonS3/latest/API/RESTObjectOps.html. [Último acceso: 2019 Mayo 1].

[27] Django, «FAQ: General | Documentación Django,» [En línea]. Available: https://docs.djangoproject.com/es/2.2/faq/general/. [Último acceso: 17 Mayo 2019].

[28] Heroku Dev Center, «Local Development with Docker Compose,» 05 Febrero 2019. [En línea]. Available: https://devcenter.heroku.com/articles/local-development-with-docker-compose. [Último acceso: 02 Mayo 2019].

[29] V. Driessen. [En línea]. Available: https://python-rq.org/. [Último acceso: 2019 Mayo 02].

[30] Heroku Dev Center, «Dynos and the Dyno Manager,» 2019 Enero 22. [En línea]. Available: https://devcenter.heroku.com/articles/dynos#ephemeral-filesystem. [Último acceso: 17 Mayo 2019].

[31] M. A. Rodríguez, «Sonar y Total Quality: Midiendo la calidad total de nuestros proyectos,» [En línea]. Available: https://www.adictosaltrabajo.com/2012/12/12/sonar-total-quality/. [Último acceso: 2019 Mayo 25].

Anexos

A.1. Dockerfiles para la construcción de los contenedores

Dockerfile Web Container

```
1  FROM  paberlo/alpine-scikit-django-jdk8
2
3  # variables para dockerizacion
4  ENV HOME=/home
5  ENV PROYECTO=tfm_diego
6
7  #copiar codigo
8  COPY webapp/ $HOME/$PROYECTO
9
10 # cambiar punto de entrada al directorio del proyecto
11 WORKDIR $HOME/$PROYECTO
12
13 CMD python3 manage.py runserver 0.0.0.0:$PORT
14 FROM  paberlo/alpine-scikit-django-jdk8
```

Dockerfile Worker Container

```
1  FROM  paberlo/alpine-scikit-django-jdk8
2
3  # variables para dockerizacion
4  ENV HOME=/home
5  ENV PROYECTO=tfm_diego
6
7  #copiar codigo
8  COPY webapp/ $HOME/$PROYECTO
9
10 # cambiar punto de entrada al directorio del proyecto
11 WORKDIR $HOME/$PROYECTO
12
13 CMD python3 worker.py
14 FROM  paberlo/alpine-scikit-django-jdk8
```

A.2. Contenido del Docker Compose

Docker Compose

```
1  version: '3.6'
2      services:
3    web:
4      image: tfm-diego-web:latest
5      command: python3 -u manage.py runserver 0.0.0.0:8000
6      depends_on:
7        - redis
8      environment:
9        - REDIS_HOST=redis
10       - PYTHONUNBUFFERED=0
11       - LOCAL=true
12     ports:
13       - "8000:8000"
14   worker:
15     image: tfm-diego-worker
16     command: python3 -u worker.py
17     depends_on:
18       - redis
19     environment:
20       - PYTHONUNBUFFERED=0
21       - LOCAL=true
22   redis:
23     image: redis:4.0.14-alpine
```

A.3. Archivo de configuración de Travis

Travis CI Configuration

```
1  language: python
2       python:
3  - '3.6'
4  script:
5  - python tests.py
6  deploy:
7    provider: script
8    #https://docs.travis-ci.com/user/deployment/script
9    script: bash $TRAVIS_BUILD_DIR/travis_deploy.sh
10   on:
11     branch: master
```