UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Servicio de predicción automática de fallos potenciales debido a nuevos commits en repositorios de código

Diego Fermín Sanz Alonso

Febrero de 2019

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Servicio de predicción automática de fallos potenciales debido a nuevos commits en repositorios de código

Autor: Diego Fermín Sanz Alonso

Directores: Pablo Bermejo López

Luis de la Ossa Jiménez

Febrero de 2019

Resumen

Resumen

Agradecimientos

Agradecimientos

Índice de contenido

Índice de figuras

No se encuentran elementos de tabla de ilustraciones.

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 o BitBucket. 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, se pretende 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 se creará un servicio potencialmente desplegable en el que el usuario introduzca información del commit que va a realizar sobre su repositorio compartido, 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.

Durante el desarrollo de todo el trabajo se buscará cumplir con la Primera Vía de la filosofía DevOps [1], la cual busca agilizar al máximo posible el flujo de trabajo para así entregar valor al cliente lo antes posible. Para ello, algunos elementos como el control de versiones o la Integración Continua son esenciales, por lo que se incluirán en este trabajo.

1.2 MÉTODO Y FASES DE TRABAJO

Las fases de trabajo pueden resumir en 3 fases principales, que son:

1. Adquisición de los datos.
2. Creación de un modelo a partir de la información extraída.
3. Creación de un servicio a partir del modelo creado en la fase anterior.

El desarrollo del proyecto seguirá la metodología de desarrollo Scrum [4], siguiendo recomendaciones para su adaptación de [5]. 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, 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. 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.

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 referencia en [5].
- Posible despliegue automático una vez creado el modelo.

1.3 ESTRUCTURA DE LA MEMORIA

Este trabajo se dividirá en un total de 5 capítulos, incluyendo este capítulo introductorio.

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 modelo de predicción. En este capítulo explicaremos cómo hemos diseñado y optimizado diferentes posibles modelos de predicción, y mostraremos diferentes métricas sobre el modelo que finalmente se ha seleccionado para incluir en el servicio desplegado.

El cuarto capítulo tratará el tema del despliegue, en el que explicaremos cómo hemos creado el servicio y qué métodos se han llevado a cabo para conseguir un despliegue automático de la aplicación 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

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 [8] 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 que 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 [9] 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 [10].

Sin embargo, necesitamos también información relativa a cuándo un commit ha provocado un fallo o no. Existen otros artículos que analizan la categorización de los issues de GitHub a partir de las etiquetas asociadas en los distintos proyectos [11]. No obstante, en [11] 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, podríamos 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 [12], que recopila el resultado de las pruebas realizadas por Travis-CI [13], 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 [14], 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 [15].

Entrando un poco más en profundidad en [14] [15], el proceso que se ha 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 [16]. 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, se creó en [15] una base de datos pública con versiones de un conjunto de proytectos 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 estos datos, 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.

CAPÍTULO 3

MODELO DE PREDICCIÓN

CAPÍTULO 4

DESPLIEGUE DE LA APLICACIÓN

CAPÍTULO 5

CONCLUSIONES Y PROPUESTAS

5.1 CONCLUSIONES

5.2 TRABAJO FUTURO

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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].

Anexos

A.1. Anexo 1: Metodología de trabajo