



Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2020/2021

Trabajo de Fin de Máster

**Análisis, mejora del código y pruebas de la
librería CF4J: Collaborative Filtering for Java**

Autor: Dionisio Cortés Fernández
Tutor: Francisco Gortázar Bellas



Contenido

Objetivos	3
Introducción	4
Estado inicial	5
GitHub Actions	6
Mutation testing	8
Mejoras en el código	12
Implicaciones de las métricas de calidad	13
Pruebas de carga	15
Creando la aplicación	15
Smoke tests	15
JMETER	16
Pruebas de carga en el cloud	19
Desplegando con helm	23
Usando últimas versiones	24
Conclusiones y trabajo futuro	25
Bibliografía	26

Para mis ojos, gracias por darme tregua y dejarme escribir esto, os cuidaré lo que esté en
mi mano.

Para mi mente, no dejaremos de sufrir, cuál será nuestra siguiente aventura y el siguiente
elefante rosa.

A los que me rodean, gracias por la paciencia.

A mi sobrina, tu tío debería enseñarte cosas, pero aprende de ti más de lo que aun te
enseña.

Objetivos

El presente trabajo tiene como objetivo usar herramientas y técnicas vistas en el máster para mejorar un proyecto existente, intentando acercarse lo más posible a lo que nos pasaría en el mundo laboral que es más probable que tengamos que enfrentarnos a código heredado que empezar un proyecto desde cero. En este trabajo no contamos con un entorno laboral, pero si real al ser una librería activa y pública, que sería otro ejemplo del mismo caso, si usamos un software que es libre y vemos que lo podemos mejorar o ampliarlo es mejor colaborar que hacer un fork o empezar algo similar desde cero.

Las herramientas y técnicas que se van a usar son:

Github actions, que usaremos para montar un entorno de integración continua, lo que nos permitirá poder analizar la librería cada vez que se suba código nuevo y pasar todos los tests de forma automática para asegurarnos que el software está siempre funcionando y detectar los errores lo antes posible en caso de que los hubiera.

sonarQube analizará la calidad cada vez que se suba código, lo que nos va a permitir ver la evolución del proyecto a medida que este evoluciona, lo que nos va a permitir tomar medidas correctoras de forma temprana.

Mutation testing nos va a permitir conocer el estado de los tests, si son confiables o no, permitiendo mejorarlos para que los tests que tengamos sean de calidad y realmente prueben lo que deberían de probar.

Pruebas de carga que nos va a permitir medir el comportamiento de los diversos algoritmos para conocerlos mejor y detectar si nos enfrentamos a alguna regresión si se implementan los algoritmos de otra manera. Estas pruebas de carga requieren de la creación de un wrapper rest que expone unos endpoints para hacer las llamadas. Estas pruebas de carga se completarán montando un entorno cloud en google cloud platform con kubernetes para acercarlo un poco más a lo que sería un entorno productivo.

Introducción

[CF4J: Collaborative Filtering for Java](#) es una librería de Filtrado Colaborativo escrita en Java para realizar experimentos de investigación sobre Sistemas de Recomendación. El TFM va a girar en torno a la mejora de esta librería.

La decisión de hacer este TFM sobre una librería existente de sistemas de recomendación está basado en intentar dar una continuidad a lo que fueron mi TFC y TFG, los dos ligados a los sistemas de recomendación, pero esta vez enfocado a lo estudiado en este máster referente a buenas prácticas y mejora del código siendo un aliciente el poder aportar algo a la comunidad.

Un reto importante es el trabajar sobre un proyecto existente, que es lo que suele pasar en el mundo empresarial donde normalmente hay más mantenimiento y evolución de proyectos existentes que proyectos que empiezan desde cero.

Dentro de este máster, se han visto herramientas como github actions que se usarán para tener un pipeline que nos de una integración continua y ver la calidad del código cada vez que se pone código nuevo y detectar cualquier fallo cuanto antes.

A lo largo del máster se han estudiado técnicas para el análisis de código con herramientas como sonarQube dentro del marco de la integración y entrega continua, se usarán estas técnicas para analizar el código de esta librería.

La primera parte del máster estuvo dedicada a la arquitectura, diseño y calidad de software. Gracias a este aprendizaje, vamos a mejorar la calidad del software siempre que sea posible usando las métricas que salgan del pipeline con sonarQube.

Un módulo muy importante fue el módulo de testing que aquí aplicaremos usando la técnica de mutation testing para ver el estado de los tests del proyecto.

En el máster también se han visto técnicas de load testing en pruebas de servicios de internet y se usarán en este proyecto para ver qué rendimiento que tienen los algoritmos y ver lo importante que es para ver lo que es capaz de soportar la aplicación. Al ser una librería, para realizar estas pruebas, se requiere del desarrollo de una aplicación que será un api rest para usar los diferentes algoritmos.

Por último, al ser un fork de una librería viva, se espera que se hagan cambios que creen conflictos por lo que se hará uso de git de tal forma que se saque provecho al módulo de git visto en el máster.

Estado inicial

El proyecto elegido, [cf4j](#) es una librería que implementa diversos algoritmos de sistemas de recomendación y que es muy útil para gente que quiere adentrarse en ese mundo y una referencia en el mundo académico.

Algo que suele pasar en el mundo del software es que se tiene una tendencia a reinventar la rueda. ¿Por qué mejor no colaborar y ver si podemos mejorar el software existente aprovechando que el código es público? Seguramente tengamos una visión algo distinta de cómo deberían ser las cosas y algunas decisiones no nos gusten, pero casi siempre lo que pasa es que las decisiones no son mejores ni peores, sino soluciones de compromiso.

Otra cosa que suele ser frecuente, es la preferencia por empezar proyectos. Siempre se piensa que el proyecto nuevo va a ser el proyecto definitivo donde vamos a aplicar todo lo que sabemos y acabará con una calidad excelente. Esto tiene algunos matices como son:

- Estamos en continuo aprendizaje y siempre saldrá algo que mejorar
- En el mundo empresarial esto es poco viable, porque se invierte mucho dinero en software que no podemos ir tirando cada vez que queramos empezar de nuevo
- Perdemos el aprendizaje de enfrentarnos a cosas que se hicieron en un contexto determinado.

GitHub Actions

El proyecto tiene Travis CI ¹ como sistema de integración continua. Una vez que ya se tiene un sistema de integración continua es complicado y costoso cambiarlo, por lo que es un objetivo complicado que el proyecto migre.

Estando en github, tiene sentido usar las github actions para que todo quede en el mismo sitio y no depender de sistemas externos. Además, existe cierta polémica con Travis CI ya que expusieron datos (Vulnerabilidad en Travis CI expone credenciales secretas ²).

Lo que se quiere hacer en este proyecto es algo parecido a lo que haríamos con un proyecto legacy. Lo primero que hay que hacer es evaluar el estado.

Esta evaluación la vamos a hacer usando github actions y SonarQube ³.

SonarQube es una herramienta (en este proyecto se usa la versión SaaS) que nos va a ayudar a detectar ciertos problemas idealmente antes de que el software llegue a los usuarios.

En términos de calidad de software, aseguramiento de la calidad, testing y todas las técnicas para hacer un software de calidad y robusto lo mejor es empezar a inspeccionar la calidad lo antes posible. Por eso, lo primero es montar un sistema de integración continua que nos ayude a pasar los tests de forma automática y herramientas que nos den unas heurísticas de cómo de bueno es nuestro código.

¿Por qué esto es tan importante? Los errores son mucho más fáciles de detectar y de solucionar al inicio de un proyecto que cuando ya está implantado y cuantos más errores detectemos antes de que llegue al usuario mejor.

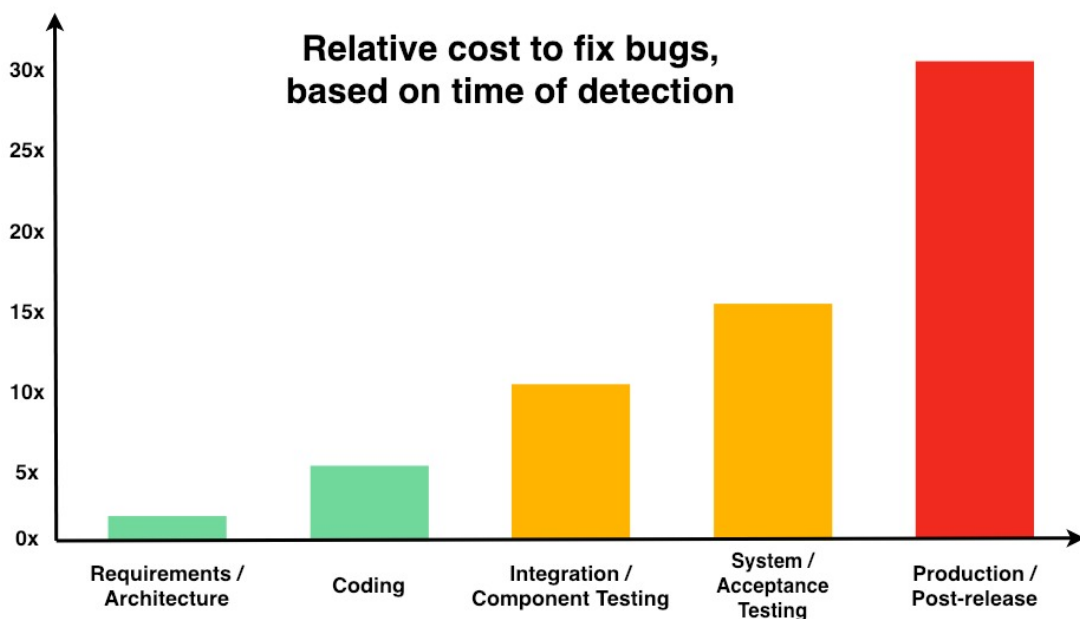


Figura 1: [Coste de arreglar bugs dependiendo su detección](#)

En la práctica, y a pesar de la mayor concienciación sobre la importancia de la calidad de software, seguimos dejando la inspección de calidad para el final. Esto nos pasa también

con la seguridad, que la postergamos hasta que por desgracia sufrimos algún ataque y ya es demasiado tarde, bien por daños económicos o pérdida de reputación. La [action](#) de github está disponible en el repositorio.

La action tiene las siguientes características:

- Se ejecuta con cada push (se está usando trunk base development ⁴)
- Se ejecuta con los pull request, por si quieren adoptarlo, que cada pull request sea inspeccionada para ver si se cumplen todos los tests.
- Ejecuta sonar y todos los tests.
- Hace uso de caches del directorio .m2 y sonar para hacer las builds más rápidas.

También hay que hacer cambios en el pom.xml para añadir el plugin de sonar y jacoco para la cobertura, el [pom](#) completo se puede encontrar en el repositorio.

Después de toda la configuración, estos son los datos que nos arroja Sonarqube.

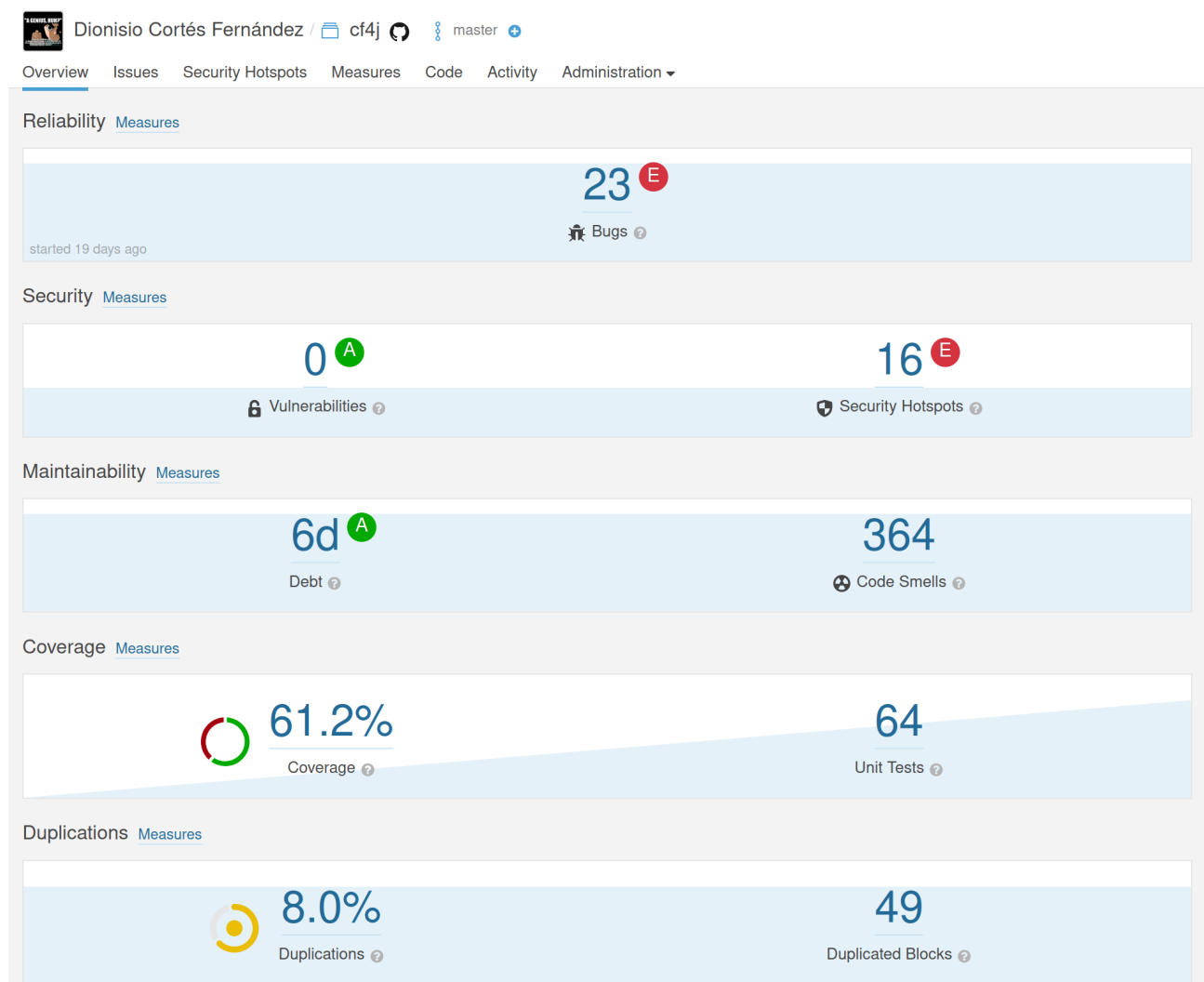


Figura 2: Imagen inicial de sonar

El estado del proyecto no es malo, pero por ejemplo tenemos un 61.2% de cobertura, dentro de ese 61.2% ¿Cómo sabemos que esos tests son confiables?

Mutation testing

Vamos a aplicar mutation testing para ver que los tests son confiables.

El concepto Mutation testing fue acuñado por primera vez en 1978, por Richard J. Lipton, R. A. DeMillo y F.G. Sayward en el paper Hints on Test Data Selection: Help for the Practicing Programmer ⁵.

Mutation testing consiste en introducir pequeñas modificaciones en el código (bytecode en el caso de java), a las que se denominan mutantes.

- Si las pruebas pasan al ejecutarse sobre el mutante, el mutante sobrevive.
- Si las pruebas no pasan al ejecutarse sobre el mutante, el mutante muere.

El objetivo es que todos los mutantes mueran ya que si al cambiar el código el test sigue pasando, es que ese test no es muy relevante.

Este concepto se basa en dos hipótesis:

- Hipótesis del programador competente: La mayoría de los errores introducidos son pequeños errores sintácticos.
- Hipótesis del efecto de acoplamiento: Pequeños fallos acoplados pueden dar lugar a otros problemas mayores.

Sobre la hipótesis del programador competente hay literatura reciente ya que es un tema sobre el que sigue investigando por ejemplo en el artículo: A new perspective on the competent programmer hypothesis through the reproduction of bugs with repeated mutations ⁶. Si bien están de acuerdo, piensan que tal vez falten operadores para reproducir bugs.

También se puede leer como grandes corporaciones como Facebook hacen uso de estas técnicas aplicando machine learning, por lo que vemos que es un tema de bastante actualidad.

What It Would Take to Use Mutation Testing in Industry—A Study at Facebook ⁷.

Cada vez tiene más sentido usar e investigar sobre mutation testing por dos razones esenciales:

- Cada vez tenemos más capacidad de cómputo.
- Cada vez tenemos más datos para aplicar técnicas como machine learning y hacerlo más eficiente y eficaz.

Para aplicar los tests de mutación vamos a usar PIT ⁸ que parece la librería más madura, mantenida y sobre la que se están implementando nuevas técnicas como extreme mutation testing de la mano de proyectos europeos como STAMP ⁹.

Si miramos la cobertura y las mutaciones, salvo `es.upm.etsisi.cf4j.qualityMeasure` (es una clase abstracta y no tiene sentido evaluarla) y util, el resto están bastante bien.

La columna line coverage nos da la métrica clásica de que líneas ejercita, la barra verde nos dice cuántas líneas se ejercitan y en rojo las que no.

La columna mutation coverage nos dice cuantos mutantes mueren y cuantos sobreviven, los verdes son los que mueren y los rojos los que sobreviven.

Si observamos vemos que tenemos un 66% de cobertura de líneas (que difiere de lo que nos dice sonarqube) dentro de esas líneas cubiertas, el 61 están bien cubiertas por los casos de prueba lo que nos da una fortaleza del 83%

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
77	66% <div><div>1940/2943</div></div>	61% <div><div>1412/2312</div></div>	83% <div><div>1412/1711</div></div>

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
es.upm.etsisi.cf4j.data	10	75% <div><div>307/411</div></div>	46% <div><div>80/175</div></div>	70% <div><div>80/115</div></div>
es.upm.etsisi.cf4j.data.types	3	80% <div><div>37/46</div></div>	80% <div><div>28/35</div></div>	90% <div><div>28/31</div></div>
es.upm.etsisi.cf4j.qualityMeasure	1	14% <div><div>3/22</div></div>	0% <div><div>0/7</div></div>	0% <div><div>0/0</div></div>
es.upm.etsisi.cf4j.qualityMeasure.prediction	8	96% <div><div>79/82</div></div>	95% <div><div>75/79</div></div>	96% <div><div>75/78</div></div>
es.upm.etsisi.cf4j.qualityMeasure.recommendation	7	86% <div><div>128/148</div></div>	97% <div><div>84/87</div></div>	97% <div><div>84/87</div></div>
es.upm.etsisi.cf4j.recommender	1	100% <div><div>12/12</div></div>	100% <div><div>4/4</div></div>	100% <div><div>4/4</div></div>
es.upm.etsisi.cf4j.recommender.knn	2	86% <div><div>118/137</div></div>	84% <div><div>48/57</div></div>	91% <div><div>48/53</div></div>
es.upm.etsisi.cf4j.recommender.knn.itemSimilarityMetric	11	96% <div><div>253/263</div></div>	79% <div><div>271/341</div></div>	85% <div><div>271/317</div></div>
es.upm.etsisi.cf4j.recommender.knn.userSimilarityMetric	12	95% <div><div>276/290</div></div>	82% <div><div>306/374</div></div>	88% <div><div>306/349</div></div>
es.upm.etsisi.cf4j.recommender.matrixFactorization	8	86% <div><div>626/730</div></div>	70% <div><div>465/663</div></div>	75% <div><div>465/616</div></div>
es.upm.etsisi.cf4j.recommender.neural	1	90% <div><div>55/61</div></div>	63% <div><div>15/24</div></div>	71% <div><div>15/21</div></div>
es.upm.etsisi.cf4j.util	3	39% <div><div>22/57</div></div>	37% <div><div>27/73</div></div>	100% <div><div>27/27</div></div>
es.upm.etsisi.cf4j.util.optimization	2	0% <div><div>0/143</div></div>	0% <div><div>0/59</div></div>	0% <div><div>0/0</div></div>
es.upm.etsisi.cf4j.util.plot	7	0% <div><div>0/514</div></div>	0% <div><div>0/320</div></div>	0% <div><div>0/0</div></div>
es.upm.etsisi.cf4j.util.process	1	89% <div><div>24/27</div></div>	64% <div><div>9/14</div></div>	69% <div><div>9/13</div></div>

Report generated by [PIT](#) 1.6.8

Figura 3: Imagen de la ejecución de PIT con mutation testing

Si miramos el fichero DataModel.java, vemos que tenemos mutantes que sobreviven en rojo fuerte y con explicación, líneas cubiertas por los tests en verde y líneas en rojo más claro que no están cubiertas.

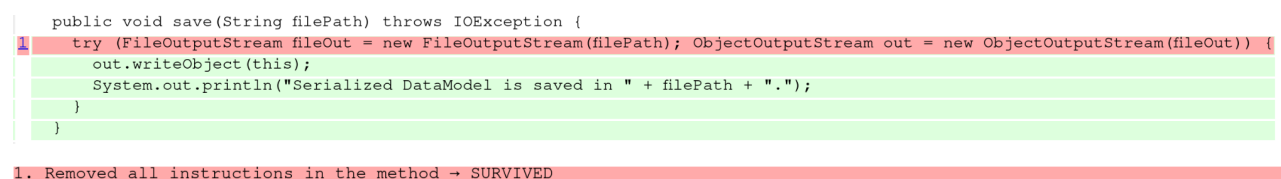


Figura 4: Imagen de la ejecución de PIT con un mutante vivo con su explicación y líneas con cobertura



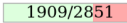
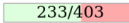
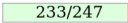
Figura 5: Imagen de la ejecución de PIT con líneas sin cubrir

Al ser código heredado, podemos usar una técnica menos agresiva e ir mejorando poco a poco. Esta técnica se conoce como extreme mutation testing. Este tipo de mutation testing fue propuesto en el paper Will My Tests Tell Me If I Break This Code? ¹⁰.

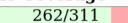
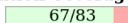

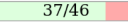
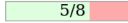
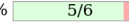
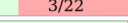

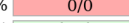
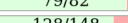
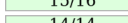
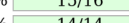
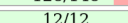
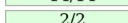
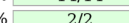
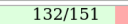
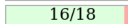
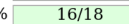
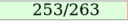
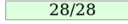
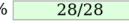
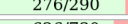
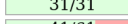
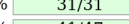
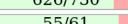
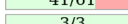
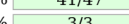
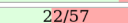
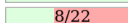
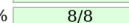
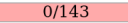
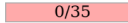
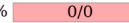


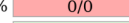
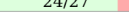
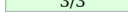
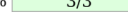






Las ventajas que ofrece es que muta no a nivel de operadores, sino a nivel de método, lo que nos hace tener muchos menos mutantes dándonos una mayor rapidez y un buen punto de partida para saber dónde tenemos que poner el foco en mejorar.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
74	67% 	58% 	94% 

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
es.upm.etsisi.cf4j.data	7	84% 	81% 	94% 
es.upm.etsisi.cf4j.data.types	3	80% 	63% 	83% 
es.upm.etsisi.cf4j.qualityMeasure	1	14% 	0% 	0% 
es.upm.etsisi.cf4j.qualityMeasure.prediction	8	96% 	94% 	94% 
es.upm.etsisi.cf4j.qualityMeasure.recommendation	7	86% 	100% 	100% 
es.upm.etsisi.cf4j.recommender	1	100% 	100% 	100% 
es.upm.etsisi.cf4j.recommender.knn	2	87% 	89% 	89% 
es.upm.etsisi.cf4j.recommender.knn.itemSimilarityMetric	11	96% 	100% 	100% 
es.upm.etsisi.cf4j.recommender.knn.userSimilarityMetric	12	95% 	100% 	100% 
es.upm.etsisi.cf4j.recommender.matrixFactorization	8	86% 	67% 	87% 
es.upm.etsisi.cf4j.recommender.neural	1	90% 	100% 	100% 
es.upm.etsisi.cf4j.util	3	39% 	36% 	100% 
es.upm.etsisi.cf4j.util.optimization	2	0% 	0% 	0% 
es.upm.etsisi.cf4j.util.plot	7	0% 	0% 	0% 
es.upm.etsisi.cf4j.util.process	1	89% 	100% 	100% 

Report generated by PIT 1.6.8

Figura 6: Imagen de la ejecución de PIT con extreme mutation testing

Se puede observar que no es tanta la diferencia entre los dos análisis, pero observamos que depende de la implementación del motor de mutación, cosas que deberían ser ajenas a mutation testing cambian, como por el ejemplo la cobertura de líneas.

	Mutation testing	Extreme mutation testing
Line coverage	66%	67%
Mutation coverage	61%	58%
Test strength	83%	94%

Tabla 1: Comparación de análisis entre técnicas de mutation testng

Lo que si se observa es una mejora sustancial en el tiempo de ejecución al tener menos mutaciones y por ello menos tests que ejecutar.

Timings	Mutation testing	Extreme mutation testing
pre-scan for mutations	< 1 second	< 1 second
scan classpath	< 1 second	< 1 second
coverage and dependency analysis	2 seconds	2 seconds
build mutation tests	< 1 second	< 1 second
run mutation analysis	2 minutes and 53 seconds	1 minutes and 8 seconds

Total	2 minutes and 57 seconds	1 minutes and 11 seconds
Ran tests	2275	273
Total time	06:02 min	02:32 min

Tabla 2: Comparación de tiempos entre técnicas de mutation testing

Antes de realizar ninguna modificación sobre el código fuente, bien sea para introducir nuevas características o arreglar algún defecto, vamos a mejorar esos tests para asegurarnos que futuros cambios en el código, van a estar cubiertos por unos tests algo más confiables. Con cambios en los tests hemos mejorado la cobertura de mutación del 58% al 63% y el test strength del 94% al 95%.

Dentro de la clase util, encontramos métodos sin usar que inicialmente borramos, ya que si no se usan, no tiene sentido tenerlos. Posteriormente, tras traernos cambios del repositorio, se han vuelto a poner debido a que se ha visto que si se usaban y rompían la build. Inicialmente también se ha visto que algunas de las clases con menos cobertura no se está usando, como por ejemplo ManualDataSet, pero al no ser una utilidad, ese tipo de clases no se han intentado quitar al ser core de la librería.

Estas mejoras se han plasmado en una [pull request](#).

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
74	68% 1924/2827	63% 249/393	95% 249/261

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
es.upm.etsisi.cf4j.data	7	86% 266/311	81% 67/83	94% 67/71
es.upm.etsisi.cf4j.data.types	3	89% 41/46	63% 5/8	83% 5/6
es.upm.etsisi.cf4j.qualityMeasure	1	14% 3/22	0% 0/7	0% 0/0
es.upm.etsisi.cf4j.qualityMeasure.prediction	8	96% 79/82	94% 15/16	94% 15/16
es.upm.etsisi.cf4j.qualityMeasure.recommendation	7	86% 128/148	100% 14/14	100% 14/14
es.upm.etsisi.cf4j.recommender	1	100% 12/12	100% 2/2	100% 2/2
es.upm.etsisi.cf4j.recommender.knn	2	87% 132/151	89% 16/18	89% 16/18
es.upm.etsisi.cf4j.recommender.knn.itemSimilarityMetric	11	96% 253/263	100% 28/28	100% 28/28
es.upm.etsisi.cf4j.recommender.knn.userSimilarityMetric	12	95% 276/290	100% 31/31	100% 31/31
es.upm.etsisi.cf4j.recommender.matrixFactorization	8	87% 633/730	93% 57/61	93% 57/61
es.upm.etsisi.cf4j.recommender.neural	1	90% 55/61	100% 3/3	100% 3/3
es.upm.etsisi.cf4j.util	3	67% 22/33	67% 8/12	100% 8/8
es.upm.etsisi.cf4j.util.optimization	2	0% 0/143	0% 0/35	0% 0/0
es.upm.etsisi.cf4j.util.plot	7	0% 0/508	0% 0/72	0% 0/0
es.upm.etsisi.cf4j.util.process	1	89% 24/27	100% 3/3	100% 3/3

Report generated by PIT 1.6.8

Figura 7: Imagen de la ejecución de PIT con mutation testing tras la mejora

Mejoras en el código

Después de mejorar la batería de tests, estamos en disposición de intentar mejorar la calidad del código.

Las mejoras en el código, no resultan fáciles cuando no tienes un contexto del dominio y no eres conocedor del código ya que puede llegar a faltar semántica. Lo que sí se puede hacer son mejoras a nivel de lenguaje, por ejemplo, usando try with resources para los ficheros que se abren. También se han controlado potenciales divisiones por cero, conversiones de tipos e interrumpir el thread si se da una excepción que lo interrumpe.

SonarQube nos avisaba de alertas de seguridad. Estas alertas se han revisado y son producidas por usar el método random que no es seguro porque se puede reproducir, pero es algo que en este proyecto es deseable para poder reproducir los cálculos de los algoritmos. Por lo que estas alertas se han marcado como revisadas sin hacer nada con ellas.

Estas mejoras se han plasmado en un [pull request](#).

Implicaciones de las métricas de calidad

Al estar trabajando en una librería viva, tenemos que estar pendientes de los cambios que se produzcan en la misma. Para poder hacer contribuciones al repositorio, lo primero que necesitamos es un fork para poder trabajar y luego hacer un pull request para que puedan integrar los cambios.

Una vez que teníamos cambios, teníamos que ir sincronizando nuestro fork con sus cambios para estar actualizados. A este respecto nos hemos encontrado que a medida que íbamos trabajando, la rama ha ido divergiendo de la original.

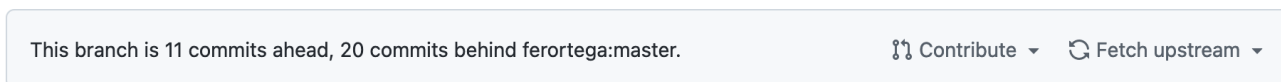


Figura 8: Imagen de la divergencia

Por lo que poco a poco hemos ido trayéndonos los cambios de esa rama a la nuestra.

Después de resolver todos los conflictos, estamos listos para seguir haciendo más cambios con todos los suyos ya integrados.



Figura 9: Imagen tras sincronizar el repositorio

A medida que se ha ido sincronizando, el código que se añade, no cumple con las condiciones de calidad, ya que en su extremo no se tiene una herramienta como sonarQube. Se debería tener sonarQube en el origen para que la calidad no disminuyera. Podemos ver cómo al añadir código en un merge, el código tenía menos del 80% de cobertura y hay más de un 3% de código duplicado

A lo largo de la realización del TFM sonarqube ha cambiado, por eso las capturas son diferentes.

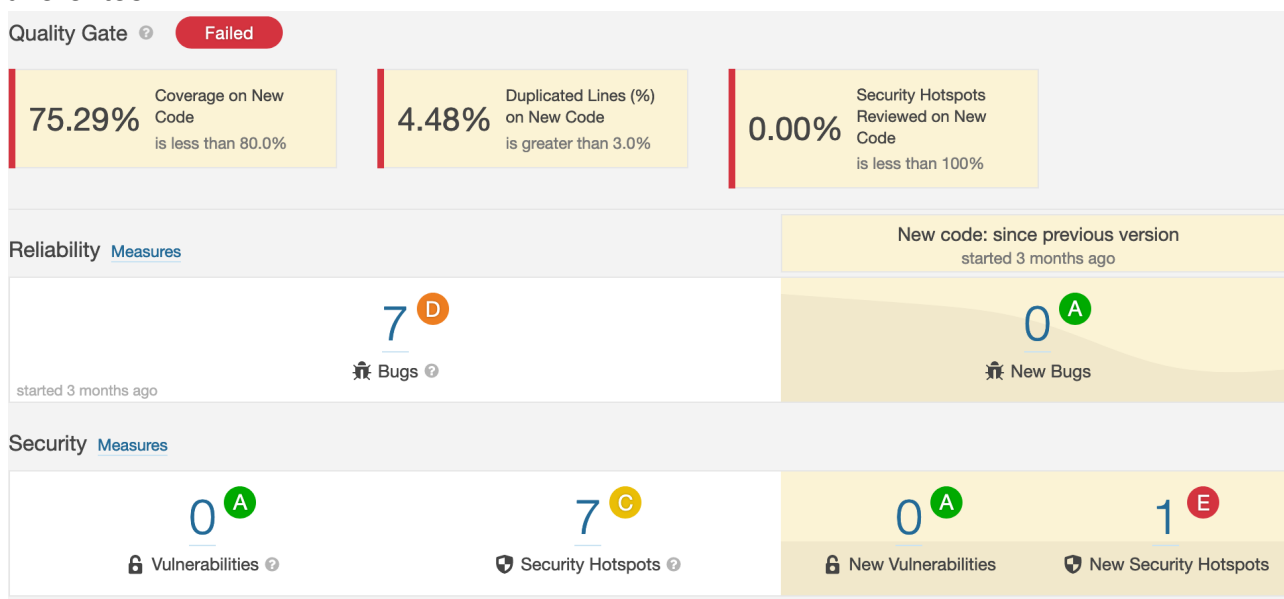


Figura 10: Imagen de sonarqube tras un merge

La evolución de la librería se ha realizado de forma paulatina en lo que se han denominado iteraciones (I) y las sincronizaciones del repositorio en (M)

Primero vemos que se aumentó la cobertura del 61.2 al 62.1 con la mejora de los tests usando mutation testing. En la primera sincronización se ve que aumenta la cobertura y luego se mantiene, pero sin las medidas adecuadas, en la última sincronización baja al 59.7 que es menor que el estado inicial.

También se puede observar cómo se han ido reduciendo los bugs de 23 a 5 y luego se han introducido dos nuevos.

Metric	I1	I2	I3	I4	M1	M2	M3
code_smells	364	359	358	360	344	344	360
bugs	23	21	19	5	7	7	7
ncloc	5850	5814	5808	5817	6223	6223	6669
uncovered_lines	1288	1252	1249	1250	1285	1285	1514
duplicated_lines	902	902	902	935	943	943	1125
duplicated_lines_density	8.0	8.1	8.1	8.4	7.9	7.9	8.7
violations	387	380	377	365	351	351	367
coverage	61.2	62.1	62.1	62.1	63.5	63.5	59.7
lines_to_cover	3243	3220	3212	3217	3422	3422	3651
vulnerabilities	0	0	0	0	0	0	0
security_hotspots	16	16	16	5	7	5	7

Tabla 3: Evolución de la librería a lo largo del tiempo

Como se puede observar después de la corrección, si no se tiene cuidado, ligeramente se puede volver a empeorar, por lo que es importante tener sistemas que nos vayan diciendo cómo vamos para siempre añadir calidad.

Pruebas de carga

Uno de los objetivos de este TFM es usar la librería y realizar distintas pruebas de carga a los diferentes algoritmos. Estas pruebas nos van a ayudar a conseguir tener un mayor conocimiento de la aplicación y la librería además de ayudarnos a detectar posibles regresiones. Estas pruebas normalmente son olvidadas y son muy importantes ya que nos ayudan a identificar puntos débiles en el sistema y cómo se podría degradar el servicio a medida que usuarios concurrentes lo usan.

Para realizar las pruebas de carga, al ser una librería, tenemos que crear una aplicación. Esta aplicación es un API rest a la que se llama y da los resultados de los algoritmos. Estos resultados siempre se dan sobre la base de datos de MovieLens100K ya que la librería está pensada mayormente para hacer comparaciones entre algoritmos, que es lo más interesante a nivel académico, nosotros vamos a darle otra óptica y ver que rendimiento tienen.

Creando la aplicación

Antes de empezar a crear la aplicación para poder usar la librería, es conveniente tener un sistema de integración continua que nos ayude a pasar los tests que tengamos para esa aplicación. También, este sistema de integración continua nos va a crear una imagen docker cada vez que se haga un push a master, que nos va a ayudar a tener la aplicación lista para ser usada en cualquier sitio del que se disponga de docker, pero abstrayéndonos del resto de herramientas necesarias como la jvm, haciéndolo más fácil de utilizar.

El [workflow](#) que nos da esta integración continua se puede consultar en el repositorio.

Para usar la librería se ha desarrollado una aplicación springboot que hace uso de la librería exponiendo una api rest con dos endpoints:

- /matrixFactorization.
- /knncomparison.

Smoke tests

Se han realizado smoke tests para asegurar que todo funciona correctamente y no se dejan introducir parámetros incorrectos, que es la parte que nos implica.

Con estos tests no buscamos probar de forma exhaustiva la aplicación, que es un wrapper rest de la librería para hacer las pruebas de carga pero queremos comprobar que con cada versión nueva de la librería, los algoritmos funcionan.

A modo de smoke test ¹¹ se hace una pasada rápida por todos los algoritmos (sin esperar que los resultados sean correctos) pero verificando que entre versión y versión no se introduce algún tipo de fallo.

JMETER

Tal vez, una de las pruebas menos realizadas, son las pruebas de carga. Cada vez, somos más conscientes de que los tests unitarios y de integración son importantes, ya que nos permiten de alguna forma, tener controlado que nuestro código, en caso de cambio, falle si ese cambio no es correcto. Pero lo que nunca se suele hacer es probar cómo se comporta un sistema bajo carga. Imaginemos que queremos usar la librería sobre la que se está realizando en TFM u otra, ¿cómo medimos cuantas peticiones podemos aguantar? ¿Como podemos tener controlado que ante un cambio de versión de la librería, se sigue comportando igual? ¿Nos interesa limitar los algoritmos a los que menos consumen, aunque den peores resultados por tener menos coste de infraestructura?

En el máster hemos realizado prácticas con artillery, para usar otra herramienta vamos a ver JMeter ¹² cuyo fichero de [configuración](#) se puede consultar en el repositorio.

JMeter es una herramienta de pruebas de carga con las siguientes características:

- Soporta diferentes tipos de SUT
 - Web - HTTP, HTTPS (Java, NodeJS, PHP, ASP.NET, ...)
 - SOAP / REST Webservices
 - FTP
 - Database via JDBC
 - LDAP
 - Message-oriented middleware (MOM) via JMS
 - Otros: Native commands or shell scripts, TCP, Java Objects, ...
- Licencia Apache 2
- Soporta pruebas distribuidas

El primer elemento a añadir es un ThreadGroup o (grupo de hilos) que es el elemento que controla el número de hilos usado por JMeter para ejecutar el test.

El ThreadGroup nos permite controlar:

- El número de hilos a utilizar que en nuestro caso lo hemos puesto a 10
- El tiempo de ramp-up que en nuestro caso lo hemos puesto a 1
- El número de veces que se debe ejecutar el test que en nuestro caso lo hemos puesto a 10

A nivel de Muestreadores (Samplers), se han usado muestreadores HTTP para hacer las llamadas REST a los endpoints que hemos creado en el apartado anterior. Estos samplers indican a JMeter que realice una determinada petición y que espere la respuesta siguiendo el orden en que aparecen dentro del Thread Group.

Por último para recoger los resultados se han puesto dos listeners los listeners nos permiten guardar en disco los datos recibidos de cada petición. Se han usado dos: View Results Tree para ver cada una de las peticiones y luego un Summary Report que nos permite ver un resumen de lo que ha pasado y tomar decisiones de una forma más ágil.

En nuestro caso concreto vemos que el algoritmo con más rendimiento es spearmank y el que menos singularities, estando todos los algoritmos con tiempos similares estando entre los 10 y 15 segundos por lo que dependiendo de nuestras necesidades y requisitos tenemos la información necesaria para tomar decisiones.

Algorithm	Samples	Average	Min	Max	Std. Dev.	Throughput
ADJUSTEDCOSINE	100	12531	9646	13724	738.04	79.121
CORRELATION	100	12093	9762	14009	501.03	82.002
COSINE	100	13255	11481	14140	474.63	74.918
JACCARD	100	13069	11252	14495	722.55	75.850
JMSD	100	13332	11774	14289	497.23	74.480
MSD	100	13342	10753	16111	594.22	74.482
SINGULARITIES	100	14957	13708	16084	425.27	66.508
SPEARMANRANK	100	11164	9554	12277	441.06	88.812

Tabla 4: Resumen de los tiempos de los algoritmos

Throughput

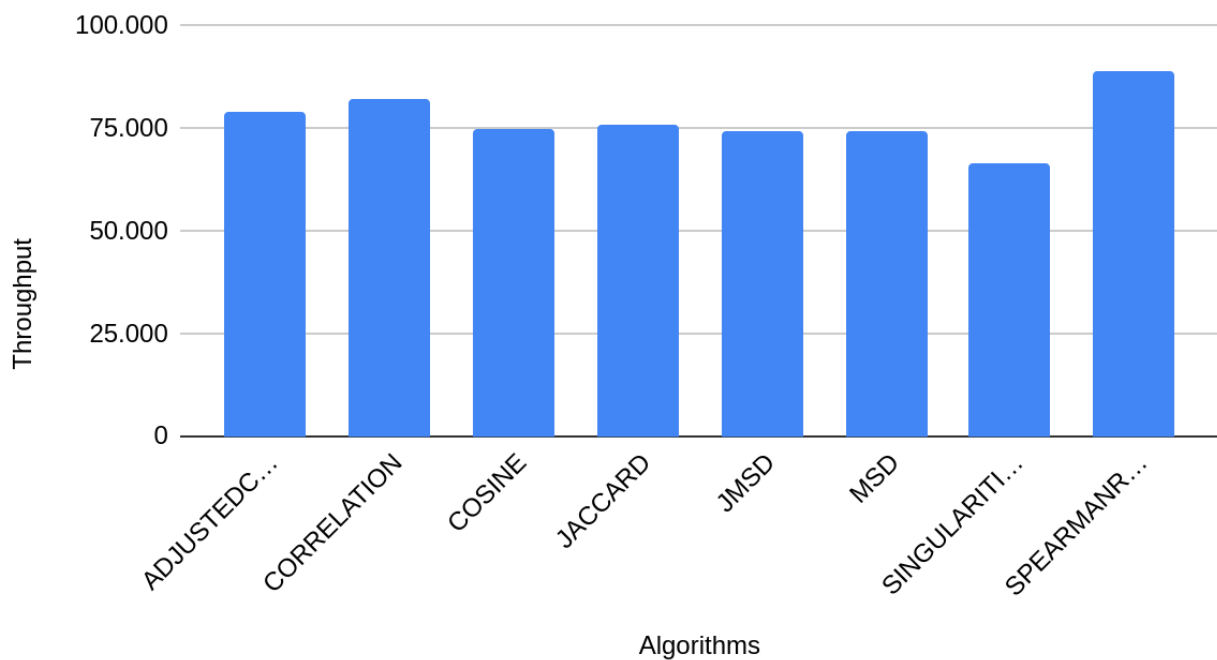


Figura 12: Imagen del rendimiento de los algoritmos

Si vemos la media

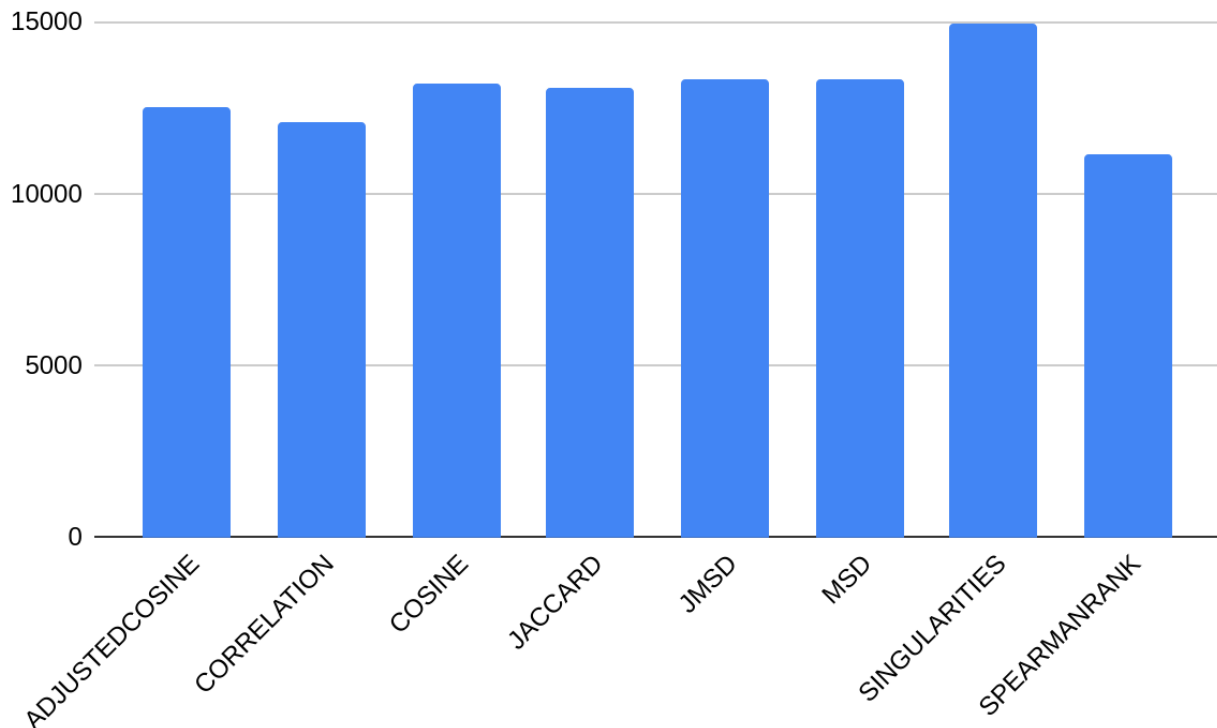


Figura 13: Imagen de las respuestas medias de los algoritmos

Teniendo la oportunidad de hacer esto de forma sencilla, automática si se quiere, es muy fácil ver si con sucesivas versiones de la librería se mejora el rendimiento o no, lo que nos permite ver posibles regresiones que surjan en sucesivas versiones de la librería. También es fácil tener una mejor previsión de los requisitos que hay que tener dependiendo del algoritmo, incluso tomar decisiones del tipo, si hay poca gente en el sistema usa el algoritmo a que da mejores resultados aunque consuma más, y si hay más carga, usa el algoritmo b que da resultados aceptables. Todo al final acaba siendo un trade-off.

Pruebas de carga en el cloud

Las pruebas locales son un punto de partida muy interesante para ver cómo se comportan los algoritmos, pero no es suficiente. La tendencia a día de hoy es que las aplicaciones se ejecuten en un entorno cloud por las ventajas que nos ofrecen estos entornos en cuanto a costes de mantenimiento, escalabilidad y servicios que nos ofrecen. Estas pruebas nos van a llevar a poder tomar decisiones de dimensionado de la infraestructura de una forma más real y a tomar mejores decisiones.

Durante el máster se ha visto aws como entorno cloud. Aquí se va a usar google cloud para experimentar otro entorno cloud.

Buscando literatura de cómo realizar pruebas de carga en entornos cloud, google nos propone una solución con kubernetes y locust ^{13 17}.

Aprovechando lo visto con kubernetes, se ha montado un entorno en google cloud que consiste en un cluster de kubernetes y se ha creado una imagen con locust para hacer pruebas de carga, de esta forma se usa algo distinto a jmeter y aws.

Las principales ventajas de Locust es la posibilidad de tener un maestro con muchos esclavos y que está más cerca al código.

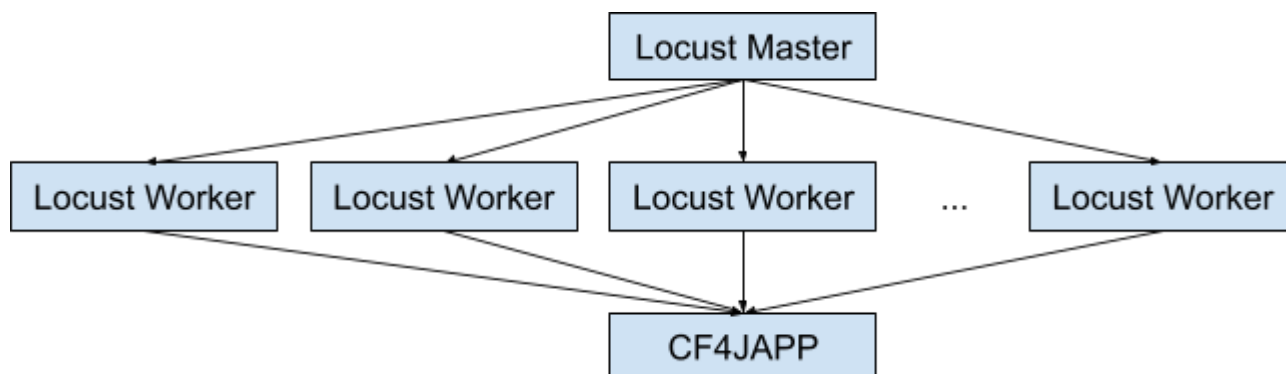


Figura 14: Imagen del despliegue de locust en el cluster

Los ficheros de [kubernetes](#) y la [imagen de locust](#) se encuentran en el repositorio.

El cluster de kubernetes es el siguiente:

DESCRIPCIÓN GENERAL

OPTIMIZACIÓN DE COSTOS

VISTA PREVIA

Filtro

Ingresar el nombre o el valor de la propiedad

<input type="checkbox"/>	Estado	Nombre ↑	Ubicación	Cantidad de nodos	CPU virtuales totales	Memoria total
<input type="checkbox"/>	✓	gke-load-test	us-central1-b	4	8	16 GB

Figura 15: Imagen del cluster de kubernetes en google cloud

Las cargas de trabajo son unidades de procesamiento implementables que se pueden crear y administrar en un clúster.

DESCRIPCIÓN GENERAL

OPTIMIZACIÓN DE COSTOS

VISTA PREVIA

Filtro

Es objeto de sistema : Falso

Filtrar cargas de trabajo

<input type="checkbox"/>	Nombre ↑	Estado	Tipo	Pods	Espacio de nombres	Clúster
<input type="checkbox"/>	cf4j-app	OK	Deployment	1/1	default	gke-load-test
<input type="checkbox"/>	locust-master	OK	Deployment	1/1	default	gke-load-test
<input type="checkbox"/>	locust-worker	OK	Deployment	5/5	default	gke-load-test

Figura 16: Imagen de las cargas de trabajo en el cluster de kubernetes

Como podemos ver, en el cloud los resultados son similares en cuanto a tiempos (máquinas e2-medium de google cloud), pero aquí vemos que hay peticiones que fallan.

Algorithms	Samples	Failures	Average
ADJUSTEDCOSINE	719	643	14324
CORRELATION	719	649	12596
COSINE	719	649	13995
JACCARD	719	649	13953
JMSD	719	649	14123
MSD	719	649	14331
SINGULARITIES	719	649	13830
SPEARMANRANK	719	649	10903

Tabla 5: Resumen de los tiempos de los algoritmos en el cloud

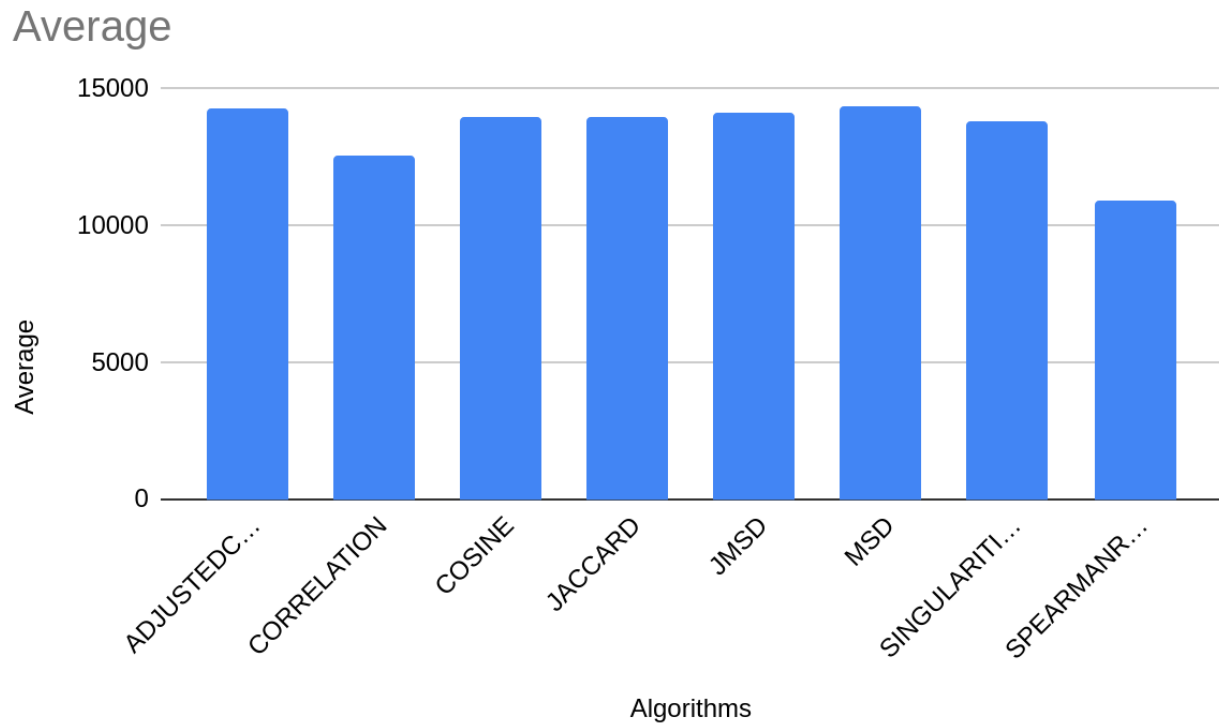


Figura 17: Imagen de la carga media en el cluster de kubernetes

La ventaja de las pruebas en el cluster es que nos permiten afinar cómo va a ser un entorno real, podemos manejar mejor como va a escalar la aplicación, ver cómo podemos bajar tiempo poniendo mas maquinas o cualquier situación que se nos de en un entorno real.

Si comparamos los resultados locales con los obtenidos en el cloud:

Algorithms	Local average	Cloud average
ADJUSTEDCOSINE	12531	14324
CORRELATION	12093	12596
COSINE	13255	13995
JACCARD	13069	13953
JMSD	13332	14123
MSD	13342	14331
SINGULARITIES	14957	13830
SPEARMANRANK	11164	10903

Tabla 6: Resumen de los tiempos de los algoritmos en el cloud frente a local

Local average y Cloud average

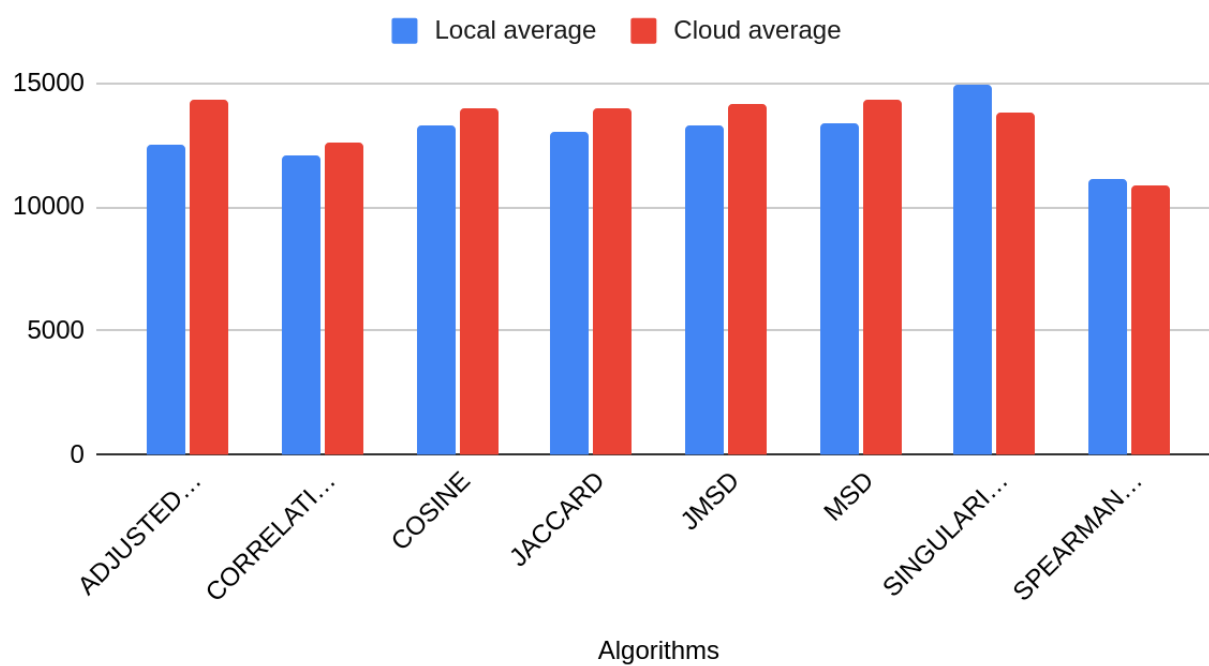


Figura 18: Imagen de la carga media en el cluster de kubernetes frente a local

Desplegando con helm

Una vez tenemos todo en el cloud, un objetivo interesante es poder hacer despliegues reproducibles.

Helm ¹⁸ es muy interesante por varios motivos:

- Nos permite instalar de forma sencilla software de terceros como podría ser una base de datos. En nuestro caso no usamos software de terceros. Este software de terceros es fácilmente personalizable a nuestras necesidades.
- Tener una forma de hacer instalaciones reproducibles, ya que helm se encarga de hacer toda la instalación y no tendríamos que ir uno por uno aplicando todos los ficheros de kubernetes.
- Actualizaciones más sencillas y fáciles de usar ya que los charts de helm se pueden versionar y con un simple comando se puede realizar la instalación.
- Rollbacks más sencillos, si algo falla es sencillo volver a la versión anterior.
- En el caso de cambiar de proveedor cloud es más sencillo al tenerlo todo centralizado y poder hacer el despliegue con una sola línea de comando.

En una aplicación con pocos componentes como esta, la versión con ficheros planos de kubernetes necesita tres comandos ejecutados en orden, mientras que la versión con helm solo necesitamos uno.

Partiendo de la base de los ficheros kubernetes se ha desarrollado un chart de helm para hacer los despliegues de forma rápida y centralizada.

Cada release de la aplicación genera un secreto en kubernetes para almacenar datos propios de la release.

<input type="checkbox"/>	sh.helm.release.v1.v1.v1	 Secret	default	gke-load-test
<input type="checkbox"/>	sh.helm.release.v1.v1.v2	 Secret	default	gke-load-test

Figura 19: Secretos de helm con cada release

El chart de helm está disponible en el [repositorio](#).

Usando últimas versiones

Uno de los alicientes del máster es haber visto las últimas versiones de las librerías, frameworks, herramientas más usadas.

Hace relativamente poco, se ha lanzado la versión 17 de java y la versión 2.6.1 de springboot.

A la hora de preparar las llamadas, se ha preparado una pequeña factoría para que en base a los algoritmos que se manden vía query param, se obtenga el objeto concreto que se quiere. La versión 17 de java nos trae una novedad interesante: [Pattern Matching for Switch](#). Esta novedad me ha parecido interesante porque nos permite escribir de forma muy concisa (casi como usando un mapa) lo que tenemos que devolver en base a lo que nos llega. Esto ya lo incorporan lenguajes como kotlin.

Conclusiones y trabajo futuro

Durante este trabajo se ha visto que no es tan fácil colaborar en un proyecto que hacerlo desde cero, ya que hay que entender lo que ya hay y mientras estás desarrollando tienes que sincronizar tus cambios con los del proyecto.

A medida que se iba sincronizando se veía que al no tener sonar en el otro extremo, las métricas de sonar no se iban cumpliendo, es importante que todo el mundo esté alineado para que la calidad se mantenga. También es importante que esto sea automático para que no suponga una carga extra para el equipo.

Las técnicas de mutation testing son costosas, con técnicas como extreme mutation testing se reducen los tiempos, pero son costosos para meterse en el pipeline que se ejecuta con cada push. Se podría meter como parte de una build de nightly para ir poco a poco mirando la calidad de los tests con métricas distintas a las líneas ejercitadas.

Los tests de carga en local nos pueden dar una idea de cómo se comporta el sistema, pero hasta que no lo ponemos en un entorno real, no sabemos qué es lo que realmente va a pasar ya que entran en juego cosas como latencias de red, máquinas sobre las que se va a ejecutar todo, etc. Estas pruebas también nos permiten ver regresiones en la librería antes de que llegue a un entorno de producción. Hacer las pruebas de carga de forma distribuida nos da una mejor perspectiva de cómo se comporta el sistema ya que simula mejor usuarios reales repartidos en distintas localizaciones.

Este trabajo ha sido más en amplitud que en profundidad, por lo que muy diferentes temas son ampliables. Como trabajo futuro se podría:

- Profundizar en qué operadores son necesarios para mejorar la hipótesis del programador competente.
- Analizar código de github y usar técnicas de machine learning para hacer tests de mutación automáticos en base a ese aprendizaje.
- Profundizar en cómo se podría mejorar la librería desde un punto de vista de la arquitectura del software.
- Migrar travisCI a github actions.
- Automatizar la creación del entorno.
- Poner sonar en el repositorio.

Bibliografía

1. "Travis CI - Test and Deploy Your Code with Confidence." <https://travis-ci.org/>
2. "Vulnerabilidad en Travis CI expone credenciales secretas - Una al Día." 14 sept. 2021
<https://unaaldia.hispasec.com/2021/09/vulnerabilidad-en-travis-ci-expone-credenciales-secretas.htm>
3. "SonarQube: Code Quality and Code Security." <https://www.sonarqube.org>
4. "Trunk Based Development." <https://trunkbaseddevelopment.com>
5. "Hints on Test Data Selection: Help for the Practicing Programmer."
<http://ieeexplore.ieee.org/document/1646911>
6. "A new perspective on the competent programmer hypothesis ... - arXiv."
<https://arxiv.org/abs/2104.02517>
7. "[2010.13464] What It Would Take to Use Mutation Testing in Industry." 26 oct. 2020,
<https://arxiv.org/abs/2010.13464>
8. "PIT Mutation Testing." <https://pitest.org>
9. "STAMP project." <https://www.stamp-project.eu>
10. "Hints on Test Data Selection: Help for the Practicing Programmer."
<http://ieeexplore.ieee.org/document/1646911>
11. "A new perspective on the competent programmer hypothesis ... - arXiv."
<https://arxiv.org/abs/2104.02517>
12. "[2010.13464] What It Would Take to Use Mutation Testing in Industry." 26 oct. 2020,
<https://arxiv.org/abs/2010.13464>
13. "Will my tests tell me if I break this code?." 14 may. 2016,
<https://dl.acm.org/doi/10.1145/2896941.2896944>
14. "Smoke test: Detecta lo más pronto posible si los elementos críticos" 27 jun. 2014,
<https://www.javiergarzas.com/2014/06/smoke-test-en-menos-de-10-min.html>
15. "Apache JMeter - Apache JMeter™." <https://jmeter.apache.org>
16. "Distributed load testing using Google Kubernetes Engine."
<https://cloud.google.com/architecture/distributed-load-testing-using-gke>
17. "Locust - A modern load testing framework" <https://locust.io>
18. "Helm" <https://helm.sh>
19. Material de clase