UNIVERSIDAD NACIONAL DE INGENIERIA

Localización de fallas determinadas en código fuente aplicando modelos estadísticos

por

Jhon Rojas Isidro

Seminario de Tesis II

Asesor: Glen Rodriguez

Facultad de Ciencias

Escuela de Ciencias de la Computación

2 de octubre de 2020

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Capítulo dedicado a exponer las razones y motivos que llevaron a desarrollar el presente seminario; así mismo los objetivos de este trabajo que servirán de lineamientos para su desarrollo; por último una descripción del contenido del presente para un mejor entendimiento de su estructura.

1.1. Motivación

Las heurísticas son técnicas de resolución de problemas, las cuales están basadas en la experiencia, conocimientos previos o simple sentido común y de acuerdo al problema que se desea resolver nos ahorran tiempo. Errores tan básicos como no convertir Kilómetros a millas como en el caso del Mars Climate Observer pueden llevar a causar pérdidas de 125 millones de dólares [15]. En 2017 un informe sobre fallas de software realizado por Tricentis [14] anotó que las fallas de software le costaron a la economía 1.7 billones de dólares en perdidas financieras.

Así con la ayuda de las heurísticas podemos crear métodos que nos ayuden a resolver fácilmente problemas que van desde problemas psicológicos hasta matemáticos

o de ingeniería. Las heurísticas se hacen interesantes por sus amplias aplicaciones reales a la industria actual.

La detección de fallos es uno de los procesos más caros en el desarrollo de software, para tener un ejemplo durante el desarrollo de un proyecto el 20 % está en su desarrollo y 80 % se aplica al mantenimiento [10]. Ian Sommerville estima que el 17 % del esfuerzo de mantenimiento se invierte en localizar y eliminar los posibles defectos de los programas [12].

A partir de aprender y entender la mayoría de las aplicaciones y ventajas de la localización de fallos para las empresas, se generó el interés para realizar el presente trabajo, el cual se acrecentó por el anhelo de generar nuevos y mejores beneficios en modelos aplicables a la industria de la tecnología actual. Entonces la iniciativa de trabajar acerca de detección de fallos en código fuente nace con la misma intención de colaborar en el desarrollo de nuevos modelos ante la creciente necesidad de la industria por automatizar este proceso.

Acerca de la detección de fallas en código fuente, existen múltiples enfoques para este fin, tales como el uso de invariantes o métricas para la predicción de fallos, los cuales cuentan con modelos ya trabajados sobre los cuales se puede hacer mejoras o crear nuevos enfoques con la misma finalidad. Los avances sobre este tema nos permiten optimizar y automatizar el trabajo, ahorrando tanto tiempo como dinero durante el proceso de detección de fallos.

En conclusión se quiere que el lector comprenda los motivos que llevaron al desarrollo del presente trabajo, que si bien no es un estudio muy conocido, genera grandes beneficios para las empresas dedicadas al desarrollo de software.

1.2. Definición del problema

Para un programa, definimos localización de fallos como el problema de determinar si el programa tiene errores y en qué parte del programa se encuentra dicho error, en especial para los casos de prueba que desencadenan dichos errores. El problema de la ubicación de las fallas es una de las más costosas, tediosas y que requieren mucho tiempo durante la depuración de códigos.

La depuración eficaz de código es de vital importancia en la producción de software confiable. Entre los beneficios de la localización de fallas en software tenemos la mejora del proceso de corrección al enfocarse en las ubicaciones detectadas con fallas.

1.3. Objetivos

El objetivo principal de este seminario es localizar fallas en código fuente, en una serie de programas, usando algoritmos basados en heurísticas nuevas propuestas y modelos estadísticos, para finalmente determinar la ubicación de la línea de código que genera el error en el código.

Los objetivos específicos de este trabajo son:

- Optimizar el método propuesto con la finalidad de afinar la localización de fallos.
- Entender los modelos estadísticos óptimos para mejorar la localización de fallas.
- Ajustar el coeficiente de sospecha propuesto de tal manera de obtener mejores resultados.
- Implementar un método de localización de fallos basado en probabilidades.

- Exponer y justificar los resultados obtenidos corroborándolos con los esperados.
- Comparar el funcionamiento del modelo resultante con otros tres que cumplen la misma función.

1.4. Estructura del seminario

Con la finalidad de que el lector entienda la estructura y contenido de este seminario, se presenta un resumen de la finalidad de cada uno de los capítulos del presente seminario de tesis.

- **Introducción**

  Se exponen las razones y motivos que llevaron a desarrollar el presente seminario; así mismo los objetivos de este trabajo que servirán de lineamientos para su desarrollo; por último una descripción del contenido.

- **Estado del Arte**

  Capítulo dedicado a la explicación de los conceptos previos para el desarrollo del seminario. Seguido se expondrán trabajos relacionados a la temática. Así podremos tener una idea general sobre el tema y cual es el punto de inicio sobre el que se empezó el trabajo; de tal manera que podamos notar los diversos aspectos que debemos considerar y mejorar.

- **Tecnologías y herramientas**

  Capítulo dedicado a exponer las herramientas que ayudaron y de qué manera al desarrollo del presente seminario. Seguido de un listado y descripción de las tecnologías y software que se usaron durante el seminario.

- **Método aplicado**

En este capítulo nos dedicaremos a exponer el método final usado basado enheurísticas para la localización de fallas, así como sus beneficios, estructuración y pruebas del método escogido.

- **Resultados**

  En este capítulo nos dedicaremos a exponer los los beneficios o perjuicios del modelo final usado ya detallado en el capítulo 4.

- **Conclusiones y Trabajo Futuro**

  En este capítulo se presentan las conclusiones del presente seminario,obtenidas de acuerdo a los objetivos previamente expuestos.

Capítulo 2

Estado del arte

Capítulo dedicado a la explicación de los conceptos previos necesarios para el desarrollo del seminario. Seguido se expondrán trabajos relacionados a la temática y en qué punto se encuentra el desarrollo del tema de este seminario. Así podremos tener una idea general sobre el tema y cual es el punto de inicio sobre el que se empezó el trabajo; de tal manera que podamos notar los diversos aspectos que debemos considerar y mejorar.

2.1. Trasfondo teórico

Sección dedicada a exponer brevemente los conocimientos primordiales mínimos para el completo entendimiento y comprensión del presente seminario. Conceptos que serán utilizados a medida que se avance con la lectura de los posteriores capítulos.

2.1.1. Heurísticas

Las heurísticas son técnicas de solución de problemas o estrategias para la solución de las mismas; las heurísticas nos permiten obtener pequeñas soluciones las cuales

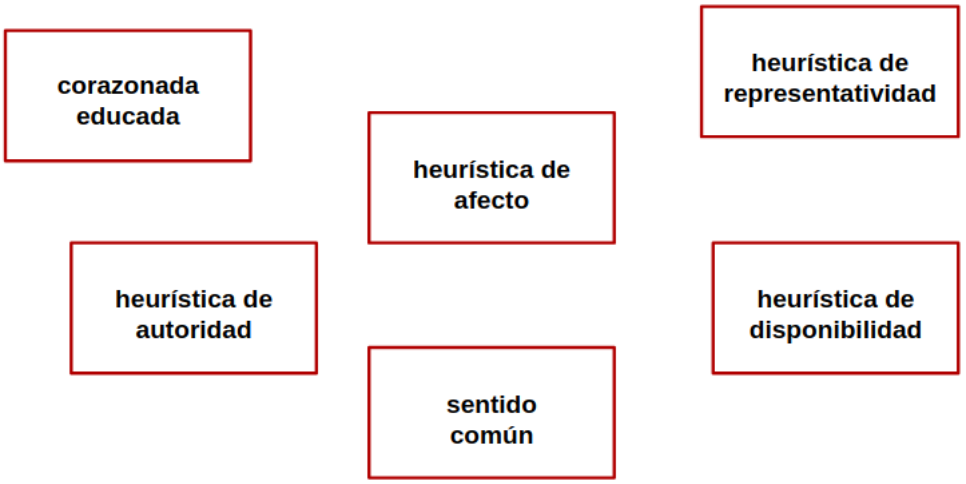

FIGURA 2.1: Tipos de heurísticas
Fuente: Curso Heurísticas y modelos de la Universidad Nacional Autónoma de México

nos permiten tomar decisiones de manera muy rápida incluso cuando hay poca información. Las heurísticas están basadas en conocimientos previos, sentido común o una primera impresión, en las cuales es evidente diferenciar lo que necesitamos de lo que no.

Existen diferentes tipos de heurísticas como se pueden apreciar en la figura 2.1, las cuales mencionamos con más detalle a continuación:

- Corazonada educada: Relacionada con los conocimientos previos que tenemos, damos respuestas rápidas en base a lo que conocemos.
- Sentido común: Sabemos a priori que es lo correcto de lo incorrecto
- Heurística de disponibilidad: Se le conoce así por ser lo primero que viene a nuestra mente.
- heurística de afecto: Relacionada con una rápida impresión que nos da algo
- heurística de representatividad: Tiene que ver con las características de un grupo

2.1.2. Coeficiente de sospecha

Dentro de las técnicas de localización de fallas es muy útil tener una clasificación de las partes defectuosas más probables del código (para nuestro caso, a nivel de línea de código) tal que las partes mencionadas, puedan ser evaluadas posteriormente por los desarrolladores de acuerdo a la probabilidad, a esta probabilidad a partir de ahora la llamaremos coeficiente de sospecha, hasta encontrar la falla. Diremos que un método es bueno si puede colocar la parte defectuosa del código con el mayor coeficiente de sospecha respecto de las demás partes del mismo código. Ahora si a la parte defectuosa del programa que se esta analizando, se le asigna una mayor coeficiente de sospecha (en otras palabras, será la primera en ser analizada), entonces los desarrolladores fueron guiados a la localización de una falla de manera exitosa y automática, sin tener que invertir más esfuerzo que el de ejecutar el método.

Entonces es fácil de entender que mientras mas similar sea el patrón de ejecución de alguna parte del código al patrón de falla de la mayoría de los casos de prueba usados, mas probable es que dicha parte del código sea defectuosa. De la misma manera cuanto mas alejado este el patrón de ejecución de una parte del código del patrón de falla, menos sospechosa parecerá la mencionada parte del código.

2.1.3. Matriz de cobertura

Se define, para fines del presente trabajo, la matriz de cobertura M, como aquella que para un programa en especifico su numero de columnas sea igual a la cantidad de lineas del programa y la cantidad de filas sea igual al número de veces que se ejecuta el mismo código más 1. Los elementos $M[a][b]$ de la matriz de cobertura tomaran los valores de 0 y 1, el valor será 1 cuando la ejecución número a del programa ejecute la linea b del código y tomarán el valor de 0 para el caso contrario. Para un programa de de n lineas de código, el elemento $M[x][n+1]$ tomara los

valores de $(+)$ y $(-)$, donde el valor del elemento será $(+)$ si la ejecución x del mismo fue exitosa de lo contrario el valor asignado será $(-)$

2.1.4. Comparación binaria

Calcular las distancias en definitiva es un procedimiento elemental para el procesamiento de datos. Allá afuera en el mundo real las distancias nos son de mucha utilidad, como por ejemplo para movernos eficientemente, de la misma manera las distancias nos son útiles en muchos otros ámbitos para medir la similitud entre dos entes.

Al hablar de otros ámbitos nos referimos por ejemplo a las palabras, cosas o cualquier otro concepto abstracto. Una vez definido esto podemos hacernos preguntas como ¿Qué tan similares son dos palabras?, pregunta que tendrá múltiples respuestas en base al contexto sobre el cual se formulo.

Ahora, para este seminario en particular, estamos interesados en los distintos métodos que tienen como finalidad calcular la distancia entre dos vectores binarios, los cuales tiene la siguiente forma: $[0, 1, 0, 0, 0, 1, 1]$ y $[0, 0, 1, 0, 0, 1, 0]$.

Si los vectores del ejemplo anterior fueran vectores comunes, los podríamos llevar al plano y calcularíamos la distancia como lo haríamos en un mapa normal pero con n dimensiones; pero debido a que estamos trabajando con vectores binarios cada elemento solo puede ser 0 o 1. Es por este motivo que la solución no es tan simple. Por ejemplo, los vectores binarios pueden servir para codificar si eres hombre (0) o mujer (1); pero siguiendo la misma tendencia deberíamos poder responder a la pregunta ¿Cual es la distancia entre un hombre y una mujer?, lo mismo pasa con palabras, ¿cuál es la distancia entre "hola" y "adiós" no existe un solo método que podamos utilizar para resolver todos los problemas, por esta motivo presentamos la siguiente lista para que podamos entender mejor el concepto:

- Russel y Rao:

"Se trata de una versión binaria del producto interno (punto). Se ofrece una ponderación igual a las coincidencias y a las no coincidencias. Ésta es la medida predeterminada para los datos de similaridad binarios"[5].

- Concordancia simple:

  "Se trata de la razón de coincidencias respecto al número total de valores. Se ofrece una ponderación igual a las coincidencias y a las no coincidencias."[5]

- Jaccard:

  "Se trata de un índice en el que no se toman en cuenta las ausencias conjuntas. Se ofrece una ponderación igual a las coincidencias y a las no coincidencias. Se conoce también como razón de similaridad."[5]

- Rogers y Tanimoto:

  "Se trata de un índice en el que se ofrece una ponderación doble a las no coincidencias."[5]

- Kulczynski: Se trata de la razón de presencias conjuntas sobre todas las no coincidencias. Este índice tiene un límite inferior de 0 y carece de límite superior. No está definido teóricamente cuando no existen no coincidencias; sin embargo, Distancias asigna un valor arbitrario de 9999,999 cuando el valor no está definido o cuando es mayor que esta cantidad.

- Ochiai: ."Este índice es la forma binaria de la medida de similaridad del coseno. Varía entre 0 y 1"[5]

2.1.5. Métricas de software

Uno de los principales objetivos de los desarrolladores de software es producir un sistema, aplicación o producto de alta calidad. Para lograr este objetivo, los desarrolladores de software deben emplear métodos efectivos junto con herramientas modernas dentro del contexto de un proceso maduro de desarrollo del software.

Al mismo tiempo, un buen desarrollador del software y buenos administradores de la ingeniería del software deben medir si la alta calidad se va a llevar a cabo. Las métricas de software pueden emplearse a la valoración cuantitativa de la calidad de algún software, una métrica del software es cualquier medida o conjunto utilizado para conocer o estimar el tamaño de un software o sistema de información. Entre los usos más frecuentes de las métricas del software están el realizar comparaciones costo beneficio y estimaciones de costos en proyectos de software.

2.1.6. Técnicas de generación de pruebas

Tradicionalmente las técnicas de generación de pruebas se han clasificado en: técnicas de caja blanca (o testado estructural), basadas en examinar la estructura de los programas y en establecer criterios para la cubrir determinadas condiciones; y técnicas de caja negra, que generan las pruebas basándose sólo en las especificaciones definidas para los datos de entrada y salida. Las técnicas convencionales usadas para la generación de pruebas de forma automática pueden dividirse entre técnicas al azar, estáticas, dinámicas, metaheurísticas, técnicas de mutación, técnicas específicas orientadas a objetos y técnicas de ágiles.

2.1.6.1. Técnicas al azar

Los datos para las pruebas se generan aleatoriamente de forma que permitan cubrir al máximo los dominios posibles de las variables de entrada. Cuanto mayor es la complejidad del programa o el nivel de cobertura que se desee, más difícil es encontrar las pruebas apropiadas con esta técnica.

2.1.6.2. Técnicas de mutación

Se basan en introducir errores simples (pequeñas desviaciones sintácticas) en programas para intentar cubrir los diferentes programas erróneos que pueden aparecer. Estos cambios vendrán dados por los operadores de mutación.

2.1.6.3. Agile Testing

Las pruebas ágiles se aplican obviamente sobre proyectos de desarrollo ágiles. Las metodologías ágiles surgen como alternativa a las metodologías tradicionales de desarrollo de software, a las que se les achaca ser burocráticas, con mucha documentación y rígidas.

2.1.7. Modelos Estadísticos

Son herramientas usadas con el fin de acercarse a la realidad o predecir algún resultado por medio de una ecuación matemática que utiliza variables que serán proporcionadas según el modelo las necesite. El modelado estadístico puede acercarnos a modelos adecuados de manera rápida, pueden predecir una variable dependiente a partir de otras variables que corresponden al entorno base del modelo relacionado.

2.2. Revisión de la literatura

Sección dedicada al análisis de trabajos relacionados de alguna manera o con el mismo fin al del presente. Con la finalidad de evaluar el estado actual de la tecnología usada, comparar resultados y tener una base para el inicio del y posterior desarrollo del presente seminario.

Habiendo revisado el fundamento teórico relacionado al tema desarrollado, a continuación se revisarán los trabajos previos que influenciaron, inspiraron y encaminaron el presente seminario, detallando brevemente el alcance de los mismos.

2.2.1. The DStar Method for Effective Software Fault Localization

DStar presenta un nuevo enfoque para la detección de fallas, este trabajo fue publicado por W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li en 2014 [16]. Este método tiene su origen en el análisis basado en coeficientes de similitud binaria, este nuevo método sugiere ubicaciones sospechosas para la localización de fallas automáticamente sin requerir ninguna información previa sobre la estructura del programa o la semántica. Proponen el uso de una forma modificada del coeficiente de Kulczynski al cual denominan DStar. El coeficiente de sospecha se construirá a partir de diversas intuiciones con respecto a cómo debe calcularse la sospecha de una declaración, de tal manera que realice cada una de estas intuiciones adecuadamente.

2.2.2. Software Fault Prediction Metrics:A Systematic Literature Review

Trabajo desarrollado por Danijel Radjenović, Marjan Heričko, Richard Torkar, Aleš Živkovič en 2013 [3]. Este trabajo realiza una revisión, análisis, clasificación de los trabajos relacionados a la detección de fallos en software en los cuales el rendimiento del modelo está influenciado por una técnica de modelado y métricas. Se identificaron las métricas más utilizadas y se evaluaron sus capacidades de predicción de fallas para responder a la pregunta de qué métricas son apropiadas para la predicción de fallas.

2.2.3. Using Likely Invariants for Automated Software Fault Localization

Investigación desarrollada por Swarup Kumar Sahoo, John Criswell, Chase Geigle, Vikram Adve en 2013 [13]. Durante el desarrollo del trabajo proponen una técnica de diagnóstico automático para aislar la(s) causa(s) raíz(s) de las fallas de software. Se utilizan probables invariantes de programa, las cuales son generadas automáticamente usando entradas correctas que están cerca de la entrada de activación de fallas, este proceso es importante porque se puede seleccionar un conjunto de ubicaciones de programas candidatos que son posibles causas raíz. Adicionalmente al proceso anterior con la intención de que se recorte el conjunto de causas raíz candidatas, se utiliza la técnica del corte dinámico hacia atrás implementado por software, esto junto a dos nuevas heurísticas de filtrado: filtrado de dependencia y filtrado a través de múltiples entradas con fallas que también están cerca de la entrada con fallas.

2.2.4. A survey on software fault detection based on different prediction approaches

Trabajo publicado por Golnoush Abaei y Ali Selamat en 2014 [1]. El trabajo estudia la predicción de fallas de software basándose en diferentes técnicas de aprendizaje automático, como árboles de decisión, tablas de decisión, bosque aleatorio, red neuronal, Naïve Bayes y clasificadores distintivos de sistemas inmunes artificiales (AIS) asi como el sistema de reconocimiento inmune artificial, CLONALG e Immunos.

Según este estudio, el bosque aleatorio proporciona el mejor rendimiento de predicción para grandes conjuntos de datos y Naïve Bayes es un algoritmo confiable para pequeños conjuntos de datos.

La evaluación del rendimiento se ha realizado en base a tres métricas diferentes, como el área bajo la curva característica de funcionamiento del receptor, la probabilidad de detección y la probabilidad de falsa alarma.

2.2.5. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique

Trabajo desarrollado por James A. Jones and Mary Jean Harrold [6], en el cual se aborda el método de Tarantula para la detección de fallos que tiene su análisis basado en coeficientes de similitud binaria usando para calcular el coeficiente el (los) aprobado (s) (es el número de casos de prueba aprobados que ejecutaron sentencias una o más veces). Del mismo modo, fallido (s) (es el número de casos de prueba fallidos que ejecutaron sentencias una o más veces). totalpassed y totalfailed (son los números totales de casos de prueba que pasan y fallan, respectivamente, en todo el conjunto de pruebas). Y como dato adicional que si alguno de los denominadores evalúa a cero, asignamos cero a esa fracción. La herramienta Tarantula utiliza el modelo de color basado en un espectro de rojo a amarillo a verde. Sin embargo, los tonos resultantes se pueden escalar y cambiar para otros modelos de color. El tono se usa para expresar la probabilidad de que s sea defectuoso o la sospecha. El tono varía de 0 a 1; 0 es el más sospechoso y 1 es el menos sospechoso.

Usando el puntaje de sospecha, clasificamos las entidades de cobertura del programa bajo prueba. El conjunto de entidades que tienen el mayor valor de sospecha es el conjunto de entidades que el programador debe considerar primero al buscar la falla. Si, después de examinar estas declaraciones, no se encuentra la falla, las declaraciones restantes deben examinarse en el orden ordenado de los valores decrecientes de sospecha.

2.2.6. Ask the Mutants: Mutating Faulty Programs for Fault Localization

Este artículo presentado por Seokhyeon Moon; Yunho Kim; Moonzoo Kim; Shin Yoo [18] presenta una nueva técnica de localización de fallas llamada MUSE, técnica de localización de fallas MUtation-baSEd, para superar este problema. MUSE utiliza el análisis de mutaciones para capturar de manera única la relación entre las declaraciones de programas individuales y las fallas observadas. Está libre de la coerción de la clasificación compartida de la estructura de bloques. La prueba básica de mutación se define como la inyección artificial de fallas sintácticas. Sin embargo, nos enfocamos en lo que sucede cuando mutamos un programa que ya está defectuoso y, en particular, la declaración del programa defectuoso. Intuitivamente, dado que un programa defectuoso puede repararse modificando declaraciones defectuosas, mutar (es decir, modificar) las declaraciones defectuosas hará que pasen más casos de prueba fallidos que mutar las declaraciones correctas. Por el contrario, la mutación de las declaraciones correctas hará que fallen más casos de prueba aprobados que la mutación de las declaraciones defectuosas. Esto se debe a que las declaraciones correctas mutantes introducen nuevas declaraciones defectuosas además de las declaraciones defectuosas existentes. Estas dos observaciones forman la base del diseño de nuestra nueva métrica para localización de fallas.

Considere un programa P defectuoso cuya ejecución con algunos casos de prueba da como resultado fallas y proponemos mutar P. Sea mf un mutante de P que muta la declaración defectuosa, y mc uno que muta una declaración correcta. MUSE depende de las siguientes dos conjeturas: Conjetura 1: los casos de prueba que solían fallar en P tienen más probabilidades de pasar en mf que en mc. Conjetura 2: los casos de prueba que solían pasar P son más propensos a fallar en mc que en mf.

2.2.7. FLUCCS: Using Code and Change Metrics to Improve Fault Localization

La investigación desarrollada por Jeongju Sohn y Shin Yoo [17] presenta un método de localización basado en el espectro (SBFL), una técnica de localización que solo se basa en la cobertura y los resultados de aprobación / falla de los casos de prueba ejecutados. Para superar las restricciones de las técnicas basadas únicamente en la cobertura, ampliaron SBFL con código y cambiaron las métricas que se han estudiado en el contexto de la predicción de defectos, como el tamaño, la edad y el abandono del código. Utilizaron los valores de sospecha de las fórmulas SBFL existentes y estas métricas del código fuente como características, aplicaron dos técnicas de aprendizaje para clasificar, programación genética (GP) y máquinas de vectores de soporte de clasificación lineal (SVM). Evaluaron el enfoque con una validación cruzada de diez veces de la localización de fallas a nivel de método, usando 210 fallas del mundo real del repositorio Defects4J. GP con métricas de código fuente adicionales clasifica el método defectuoso en la parte superior para 106 fallas, y dentro de la parte superior para 173 fallas.

2.2.8. Effective Fault Localization using Code Coverage

El trabajo realizado por W. Eric Wong, Yu Qi, Lei Zhao, Kai-Yuan Cai [2] propone un método de localización de fallas basado en la cobertura del código para priorizar códigos sospechosos en términos de su probabilidad de contener errores del programa. El código con un riesgo más alto debe examinarse antes que con un riesgo más bajo, ya que el primero es más sospechoso (es decir, es más probable que contenga errores de programa) que el segundo. El trabajo también plantea una pregunta muy importante: ¿cómo puede cada caso de prueba adicional que ejecuta el programa ayudar a localizar errores del programa? y proponen que con respecto a un fragmento de código, la ayuda introducida por la primera prueba exitosa que

la ejecuta al calcular su probabilidad de contener un error es mayor o igual a la de la segunda prueba exitosa que la ejecuta, que es mayor que o igual al de la tercera prueba exitosa que lo ejecuta, etc. También indican que el método propuesto en base a heurísticas puede reducir efectivamente el dominio de búsqueda para localizar errores del programa.

2.2.9. Evaluating and improving fault localization

Trabajo publicado por Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, Benjamin Keller [8]. Este trabajo se plantea que en investigaciones previas se han determinado qué técnicas de localización de fallas son mejores para encontrar fallas artificiales. Sin embargo, no se sabe qué técnicas de localización de fallas son mejores para encontrar fallas reales. Realizaron un estudio de replicación para evaluar 10 afirmaciones en la literatura que compararon las técnicas de localización de fallas (de las familias basadas en el espectro y en las mutaciones). Utilizamos 2995 fallas artificiales en 6 programas del mundo real. Los resultados respaldan 7 de las afirmaciones anteriores como estadísticamente significativas, pero solo 3 tienen un tamaño de efecto no despreciable. También evaluaron las mismas 10 afirmaciones, utilizando 310 fallas reales de los 6 programas. Todos los resultados anteriores fueron refutados o fueron estadísticamente y prácticamente insignificantes. Se concluye de sus experimentos que las fallas artificiales no son útiles para predecir qué técnicas de localización de fallas funcionan mejor en fallas reales.

2.2.10. Comparación

En la tabla 2.1 podemos observar la comparación entre los distintos métodos antes mencionados:

Método	Generación fallas	Cantidad de código por revisar
D*	Creadas por usuario	18.53 %
Invariantes	Propias del programa	62.5 %
Bosque Aleatorio	Propias del programa	81.14 %
Naïve Bayes	Propias del programa	80.42 %
Tarantula	Creadas por usuario	39.30 %
MUSE	Creadas por usuario	38 %

Cuadro 2.1: Comparación entre métodos

2.2.11. Conclusiones

Notamos que el estudio de la deteccion y localizacion de fallas en software no es un campo nuevo y que presenta diversos enfoques y métodos para un mismo fin. Los cuales nos servirán como bases para desarrollar un método propio para la localización de fallas en código fuente.

Capítulo 3

Tecnologías y herramientas

Capítulo dedicado a exponer las herramientas que ayudaron y de qué manera al desarrollo del presente seminario. Seguido de un listado y descripción de las tecnologías y software que se usaron durante el seminario.

3.1. Hardware

Con el pasar de los años la industria de la innovación tecnológica avanza cada vez mas rápido, gracias a lo cual se a desarrollado hardware con la capacidad de optimizar diversos procesos.

3.1.1. ASUS X555L

El equipo sobre el cual se trabajaron todos los casos de prueba, usada principalmente para calcular los outputs necesarios para generar el vector con los coeficientes de sospecha. Sus especificaciones, mostradas en la Tabla 3.1, son simples y sin ningún rasgo sobresaltante o especial.

Arquitectura de procesador	Intel® Core™ i7-5500U
Frecuencia de procesamiento	2.40GHz
Núcleos de procesamiento	4
Memoria principal	8 GB
Almacenamiento	1 TB HDD
Sistema Operativo	Linux Ubuntu 16.04.1 LTS x86_64

CUADRO 3.1: Especificaciones de ASUS X555L

3.2. Software

3.2.1. Python

Python [11] es un lenguaje interpretado multiuso, y es considerado como una de las mejores opciones para desarrollar programación científica a nivel mundial. El motivo por el cuál se eligió a python fue debido a su versatilidad, por sus capacidades de implementación y por la gran variedad de librerías que nos permitirán acelerar el proceso de implementación y la rápida generación de casos de prueba.

3.2.2. C++

Un programa C ++ es una secuencia de archivos de texto (generalmente archivos de encabezado y fuente) que contienen declaraciones. Se someten a traducción para convertirse en un programa ejecutable, que se ejecuta cuando la implementación de C ++ llama a su función principal.

3.2.3. JupyterLab

JupyterLab [7] es un entorno de desarrollo interactivo basado en la web para portátiles, código y datos Jupyter. JupyterLab es flexible: configure y organice la interfaz de usuario para admitir una amplia gama de flujos de trabajo en ciencia de datos, informática científica y aprendizaje automático. JupyterLab es extensible

y modular: escriba complementos que agreguen nuevos componentes y se integren con los existentes.

3.2.4. JAVA

Java [9] es una tecnología que se usa para el desarrollo de aplicaciones que convierten a la Web en un elemento más interesante y útil. Java no es lo mismo que javascript, que se trata de una tecnología sencilla que se usa para crear páginas web y solamente se ejecuta en el explorador. Java le permite jugar, cargar fotografías, chatear en línea, realizar visitas virtuales y utilizar servicios como, por ejemplo, cursos en línea, servicios bancarios en línea y mapas interactivos.

3.2.5. Defects4J

Defects4J [4] una base de datos y un marco extensible que proporciona errores reales para permitir estudios reproducibles en la investigación de pruebas de software. La versión inicial de Defects4J contiene 357 errores reales de 5 programas de código abierto del mundo real. Cada error real va acompañado de un conjunto de pruebas completo que puede exponer (demostrar) ese error. Defects4J es extensible y se basa en el sistema de control de versiones de cada programa. Una vez que se configura un programa en Defects4J, se pueden agregar nuevos errores a la base de datos con poco o ningún esfuerzo. Defects4J presenta un marco para acceder fácilmente a versiones de programa defectuosas y fijas y a las suites de prueba correspondientes. Este marco también proporciona una interfaz de alto nivel para tareas comunes en la investigación de pruebas de software, lo que facilita la realización y reproducción de estudios empíricos.

Capítulo 4

Método aplicado

En este capítulo nos dedicaremos a exponer el método final usado basado en probabilidades que derivan de heurísticas para la localización de fallas, así como sus beneficios, estructuración y pruebas del método escogido.

4.1. Método propuesto

En el método que se propone en el presente trabajo, como en métodos similares a este, se utilizará un coeficiente llamado de sospecha[1], el cual está basado en coeficientes de comparación binaria[2] las cuales implementaremos basándonos en diferentes heurísticas para detectar los lugares más probables en donde podemos encontrar la falla en el programa.

Adicional a las heurísticas el coeficiente de sospecha será generado a partir de la modificación de los elementos variables que lo componen con la finalidad de optimizar la eficiencia al momento de localizar la falla en el código y asignación de mayor prioridad (peso) a determinadas heurísticas. Estas modificaciones mencionadas son totalmente empíricas y se detallaran más adelante.

[1] Probabilidad de que el error en el código se encuentre en determinada linea, esta probabilidad es asignada a cada linea del código

[2] Distancia entre dos vectores cuyos elementos solo pueden ser 0 o 1

Como mencionamos con anterioridad para generar nuestro coeficiente de sospecha nos basaremos en trabajos previos como los mencionados en la sección 2.2, los cuales tienen coeficientes de sospecha que se generan de la siguiente manera:

Tarantula:

$$S_(s) = \frac{failed(s)/total\_failed}{failed(s)/total\_failed+passed(s)/total\_passed} \tag{4.1}$$

DStar:

$$S_(s) = \frac{failed(s)^*}{passed(s)+(total\_failed-failed(s))} \tag{4.2}$$

Ochiai:

$$S_(s) = \frac{failed(s)}{\sqrt{total\_failed \cdot (failed(s)+passed(s))}} \tag{4.3}$$

Ahora para el método nuevo, el cual se propone en este seminario, nos basaremos en heurísticas para generar un nuevo e innovador coeficiente de sospecha. Las heurísticas que se proponen se detallan a continuación:

1. La probabilidad de que el error se encuentre en una ubicación determinada del programa aumentará si los casos de prueba que ejecutan dicha línea fallan.

2. La probabilidad de que el error se encuentre en una ubicación determinada del programa aumentará si los casos de prueba que no ejecutan dicha línea son exitosos.

3. La probabilidad de que el error se encuentre en una ubicación determinada del programa disminuirá si los casos de prueba que ejecutan dicha línea son exitosos.

4. La probabilidad de que el error se encuentre en una ubicación determinada del programa disminuirá si los casos de prueba que no ejecutan dicha línea fallan.

5. Se asignará mayor prioridad a la primera (1) y tercera (3) heurística definida.

6. Sí no existe ningún fallo en la linea x después de ejecutar n veces el programa, la probabilidad de que el el código se encuentre en la linea x es de 0 %.

La estructura final del coeficiente de sospecha propuesto será:

$$S_(s) = \frac{fallo\_ejecutado \cdot (fallo\_ejecutado + exitoso\_no\_ejecutado)}{exitoso\_ejecutado \cdot (fallo\_no\_ejecutado + exitoso\_ejecutado)} \quad (4.4)$$

Donde:

- $S_(s)$: Coeficiente de sospecha para la linea s
- $fallo\_ejecutado$: Cantidad de veces que la ejecución recorre la linea s y el resultado final es fallido después de todas las ejecuciones.
- $exitoso\_no\_ejecutado$: Cantidad de veces que la ejecución no recorre la linea s y el resultado final es exitoso después de todas las ejecuciones.
- $exitoso\_ejecutado$: Cantidad de veces que la ejecución recorre la linea s y el resultado final es exitoso después de todas las ejecuciones.
- $fallo\_no\_ejecutado$: Cantidad de veces que la ejecución no recorre la linea s y el resultado final es fallido después de todas las ejecuciones.

4.2. Aplicación del método

Una vez que ya tenemos definido el método que se usará para el desarrollo el presente seminario, en esta sección del capítulo se explicará acerca de los casos de prueba que se utilizaron para probar el método escogido, el cual se detallo en la sección anterior. Así mismo se describirá el procedimiento que se utilizó durante la aplicación del método para los distintos casos de prueba.

4.2.1. Casos de prueba

Para este seminario se utilizara como herramienta la base de datos Defects4J, [4] una base de datos y un marco extensible que proporciona errores reales para permitir estudios reproducibles en la investigación de pruebas de software como esta. Esta base contiene 835 errores reales de proyectos de código abierto tales como: mockito, commons-lang, commons-math entre otros. Cada error real va acompañado de un conjunto de pruebas completo que puede demostrar ese error. Defects4J presenta un marco para acceder fácilmente a las versiones de programas defectuosas y reparadas y a las suites de prueba correspondientes a cada una de ellas, los cuales se utilizarán con la finalidad de demostrar los objetivos del presente seminario que se detallaron en la sección 1.3.

Las especificaciones de los programas que usaremos en el presente seminario se encuentran detalladas en el cuadro 4.1.

Programa	#Errores	#Test	Líneas de Código
Commons Lang	65	2245	22K
Joda-Time	27	4130	28K
Commons Math	106	3602	85K
Closure Compiler	133	7927	90K

Cuadro 4.1: Datos de los programas usados

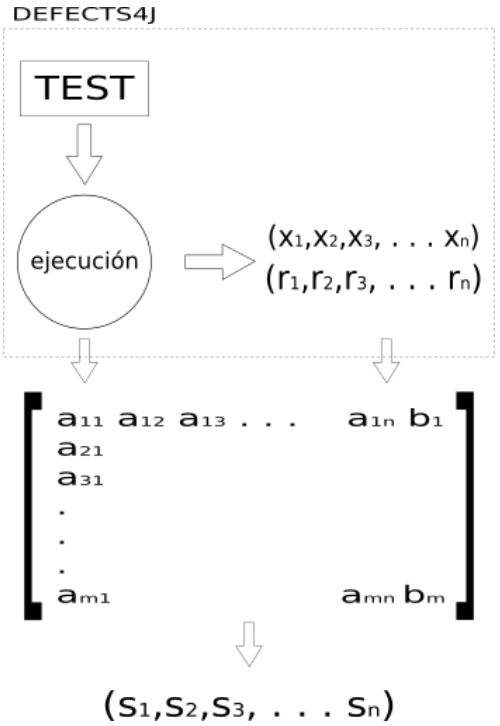

FIGURA 4.1: Estructura del algoritmo
Fuente: creación propia

4.2.2. Estructura del algoritmo

Para poder aplicar el método y comprobar la efectividad del mismo se implementó un algoritmo que determina las principales características necesarias para su cálculo. Para la mejor comprensión del algoritmo implementado y el proceso se elaboró el diagrama que se aprecia en la figura 4.1.

Según podemos observar el diagrama de la figura 4.1, obtendremos los casos de prueba a partir de la base de datos Defects4J 3.2.5, detallados en la sección 4.2.1, a partir de estos y usándolos como inputs dentro del mismo Defects4J se ejecutarán los programas sobre los cuales estamos localizando los errores; como outputs de la ejecución previa de los programas usados para este estudio, tendremos dos listas con datos que servirán para obtener determinar si el programa fue exitoso o no, el primero de estos outputs será una lista con los outputs reales (r_n) por cada ejecución de acuerdo a cada uno de los casos de prueba del input, la otra lista tiene los datos esperados después de cada ejecución (x_n). Adicional el último

output será la matriz de cobertura $A[m][n]$ 2.1.3, en la cual la cantidad de filas (m) será el número de casos de prueba que usemos como inputs y cada columna representa el número de líneas del programa sobre el cual estamos localizando el error; el valor de los elementos de la matriz A se determina a partir de la ejecución del programa y a los dos outputs mencionados con anterioridad, la descripción se narra con mayor detalle en la sección 2.1.3.

Como último paso, ya con todo lo necesario para hacer los cálculos, será aplicar el método descrito en la sección 4.1 y algunos otros métodos como los expuestos en la misma sección, para así poder obtener los coeficientes de sospecha por cada línea de del programa y así poder determinar la eficiencia del método propuesto.

Para resumir el procedimiento del método propuesto, se muestran los algoritmos 1 y 2. De los cuales se definen sus variables en la tabla 4.2.

P	Programa con errores
T	Casos de Prueba
O	Salidas esperadas sin errores
A	Matriz de cobertura
b	Comparación entre output real y esperado
$longitud()$	retorna la cantidad de elementos
$real$	Salida real de la ejecución
S	coeficientes de sospecha del método propuesto
$failed$	veces que el programa fallo al recorrer determinada linea
$totalpassed$	cantidad de veces que falla el programa
$passed$	veces que el programa fue exitoso al recorrer determinada linea
$totalpassed$	cantidad de veces que se ejecuta exitosamente el programa

CUADRO 4.2: Variables y su significado para el algoritmo 1, 2

El algoritmo 1 tiene como entradas el código P sobre el cual se buscará errores, los casos de prueba para ejecutar P y las salidas esperadas O al ejecutar el programa si no tuviera errores. Y nos retorna como output la matriz de cobertura A en la cual las filas serán la cantidad de casos de P y las columnas la cantidad de lineas

del P.

Algoritmo 1: GetSpecter

Entrada: Programa con errores P

Entrada: Casos de prueba T

Entrada: Salidas esperadas O

Salida : Matriz de cobertura A

1 **para** $i \leftarrow 1$ *hasta* $longitud(T)$ **hacer**

2 $\quad real, n_i \leftarrow P(T_i);$

3 $\quad b_i = Compare(real, O_i)$

4 $\quad A[i] \leftarrow n_i, b_i$

Finalmente obtendremos el coeficiente sospecha por cada linea de de P con el algoritmo 2, el cual utiliza como inputs las salidas del algoritmo 1. Y devuelve

como output los coeficientes de sospecha calculados con el método propio.

Algoritmo 2: GenerateSuspicionCoefficient

Entrada: Matriz de cobertura A

Salida : Vector con los coeficientes de sospecha S

```
1  n_elements = longitud(A[0])
2  para line in A hacer
3      coverage[i] ← line[: −1]
4      sign[i] ← line[−1]
5      para j in range (longitud(coverage[i])) hacer
6          si coverage[i][j] ← 1 entonces
7              covered_elements[i].append(j)
8      i + +
9  totalfailed = 0
10 failed = []
11 totalpassed = 0
12 passed = []
13 para i in longitud(sign[0]) hacer
14     si sign[i] == '-' entonces
15         totalfailed + = 1
16         para element_number in covered_elements[i] hacer
17             failed[element_number]+ = 1
18     en otro caso
19         totalpassed + = 1
20         para element_number in covered_elements[i] hacer
21             passed[element_number] + = 1
22 para element in range(n_elements) hacer
23     S = (failed[element] · (failed[element] + totalpassed −
       passed[element]))/(passed[element] · (totalfailed − failed[element] +
       passed[element]))
24 Retornar S
```

Capítulo 5

Resultados

En este capítulo nos dedicaremos a exponer los los beneficios o perjuicios del modelo final usado ya detallado en el capítulo 4.

5.1. Resultados del método propuesto

Para esta primera sección del capítulo, se detallarán los resultados obtenidos al aplicar el modelo descrito en el capítulo 4,sobre los programas y casos de prueba mencionados en la sección 4.2.1 de la base de datos Defects4J.

Programa	#Errores	#Test	LOC	LOCr	porcentaje
Commons Lang	65	2245	22K	691	3.14 %
Joda-Time	27	4130	28K	579	2.07 %
Commons Math	106	3602	85K	3344	3.93 %
Closure Compiler	133	7927	90K	1188	1.32 %

Cuadro 5.1: Cantidad de líneas de código por revisar por programa

El método propuesto en el presente seminario logra detectar todas las fallas presentes en cada uno de los programas usados en la evaluación; sin embargo al momento de determinar que tan eficiente es el método también se tiene que considerar la cantidad de líneas de código se deben revisar ($LOCr$) debido a que al momento

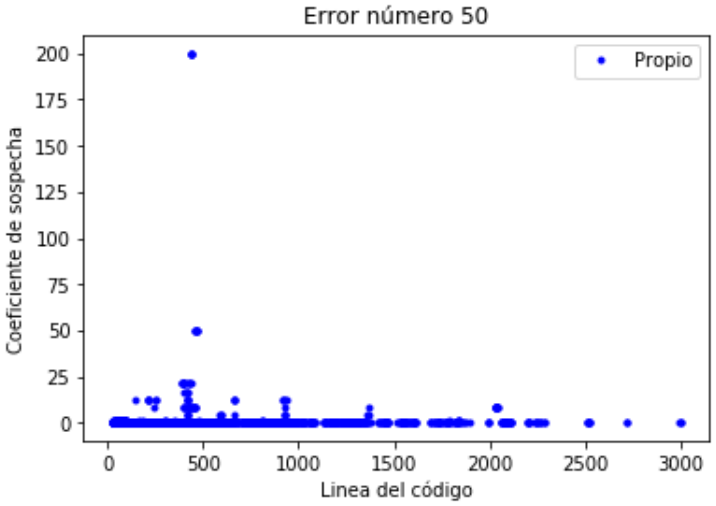

FIGURA 5.1: Coeficientes de sospecha por LOC para el error 50 en el programa Closure Compiler.

de asignar el coeficiente de sospecha a una línea determinada se podrían asignar valores cercanos a los coeficientes de las líneas en las que se encuentra el error a otras que no presentan ningún error generando falsos positivos. En el cuadro 5.1 se puede apreciar la *LOCr* por cada y método y el porcentaje que este representa del total del código. Con estos datos podemos afirmar que el método es confiable y eficiente porque además de detecta el error el porcentaje de líneas por revisar se encuentra por debajo del 4 %.

En las gráficas 5.1, 5.2, 5.3 5.4 se exponen un ejemplo por cada programa usado para este trabajo. En cada gráfico se muestra el coeficiente de sospecha asignado a cada linea de código por el que recorre la ejecución de los test en cada uno de los programas que se uso como ejemplo para probar el método; gracias a los gráficos podemos, en cada gráfico, notar de manera visual que los coeficientes de sospecha para las líneas de código en las que se encuentra el error del programa son mas altas que en las demás líneas ejecutadas.

Gracias al gráfico 5.1 que corresponde al error número 20, etiquetas asignadas por Defects4J, del programa Closure Compiler podemos apreciar los coeficientes de sospecha asignados a cada LOC después de la ejecución de los test que evidencian

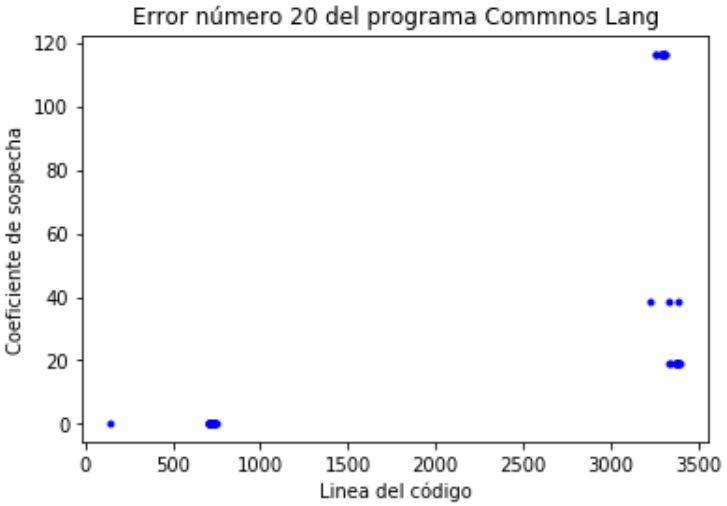

FIGURA 5.2: Coeficientes de sospecha por LOC para el error 20 en el programa Commons Lang.

este error. Para este caso en específico las líneas en las cuales se encuentra el error están comprendidas entre la 435 y 438.

En la figura 5.2, del error 20 del programa Commons Lang, que presenta error en el rango de líneas que se encuentra entre 3293 - 3308, podemos resaltar que el coeficiente de sospecha para las linea mencionadas tienen los valores mas elevados, por lo cual el método cumplió con éxito la detección. Adicionalmente se debe mencionar que los coeficientes de sospecha de las demás líneas se encuentran muy por debajo del valor de las líneas que si presentar error, lo cual representa menos lineas de código a ser evaluadas.

En la figura 5.3, que corresponde a el error 56 del programa *Commons Math*, que presenta error entre las línea 75 y 87, es bastante notorio que el coeficiente de sospecha para el rango mencionado tiene los valores mas elevados, siendo de esta manera el método cumplió con éxito la detección.

En la figura 5.4, del programa *Joda-Time* y error número 16, que presenta error entre las líneas 700 y 723, se puede apreciar rápidamente que el coeficiente de sospecha al rededor de las líneas en mención tienen el valor mas elevado, por lo cual será inequívocamente el rango de líneas en la cual se encuentra el error y no se

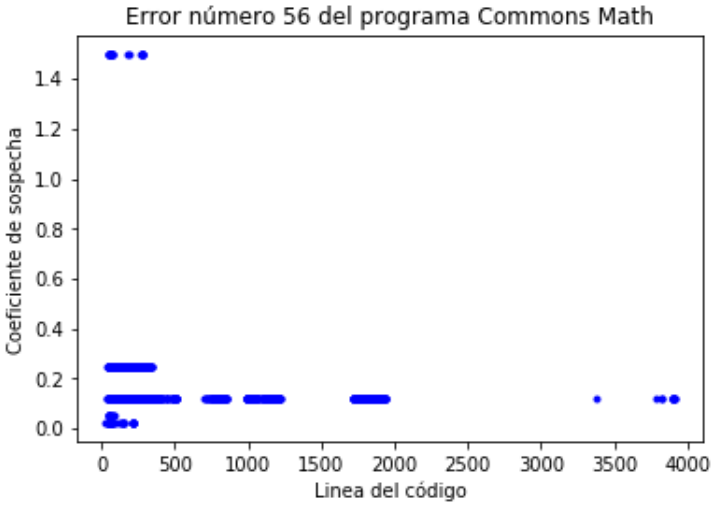

FIGURA 5.3: Coeficientes de sospecha por LOC para el error 56 en el programa Commons Math.

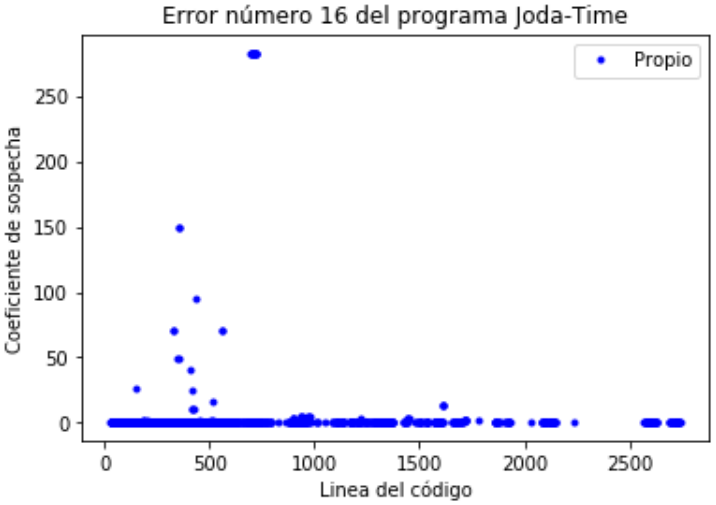

FIGURA 5.4: Coeficientes de sospecha por LOC para el error 16 en el programa Joda-Time.

deberá buscar en otras líneas debido a la gran diferencia de valores con las demás lineas.

5.2. Comparación con otros métodos

En este apartado expondremos los resultados obtenidos por el método propuesto en este seminario y por otros 3 métodos similares. Los casos de prueba usados

para este apartado serán los mismo usados en la sección anterior que y fueron detallados en la sección 4.2.1.

Método	Programa	LOC	LOCr	porcentaje
Propuesto	Closure Compiler	90K	1188	1.32 %
Propuesto	Commons Lang	22K	691	3.14 %
Propuesto	Commons Math	85K	3344	3.93 %
Propuesto	Joda-Time	28K	579	2.07 %
DStar	Closure Compiler	90K	2832	3.15 %
DStar	Commons Lang	22K	709	3.22 %
DStar	Commons Math	85K	2254	2.65 %
DStar	Joda-Time	28K	98	0.35 %
Ochiai	Closure Compiler	90K	2981	3.31 %
Ochiai	Commons Lang	22K	800	3.63 %
Ochiai	Commons Math	85K	2869	3.38 %
Ochiai	Joda-Time	28K	98	0.35 %
Tarantula	Closure Compiler	90K	66153	73.84 %
Tarantula	Commons Lang	22K	2116	9.62 %
Tarantula	Commons Math	85K	10188	11.99 %
Tarantula	Joda-Time	28K	5773	20.62 %

CUADRO 5.2: Cantidad de líneas de código por revisar por programa y método

Igual que en la tabla 5.1, la tabla 5.2 nos muestra la cantidad de líneas de código por revisar por cada programa, sobre el cual se evaluó el método propuesto y los tres métodos con los cuales se comparará el propuesto en el presente seminario. Se debe mencionar que todos los modelos usados para comparar el propuesto logran asignarle el coeficiente de sospecha más alto a las líneas en las cuales se localiza el error; sin embargo para algunos casos el valor asignado a las líneas en mención es muy cercano a otras líneas en las cuales no hay ninguna falla, esto nos dará como resultado muchas mas líneas a evaluar antes de localizar la ubicación del error en el programa.

Por ejemplo para el programa *Closure Compiler* en total se tendrán que evaluar 1188,2832,2981,66153 para los métodos Propuesto, DStar, Ochiai y Tarantula respectivamente. Para este programa en particular podemos decir que el método propuesto fue mucho mas eficiente dado que localizo los errores de manera correcta

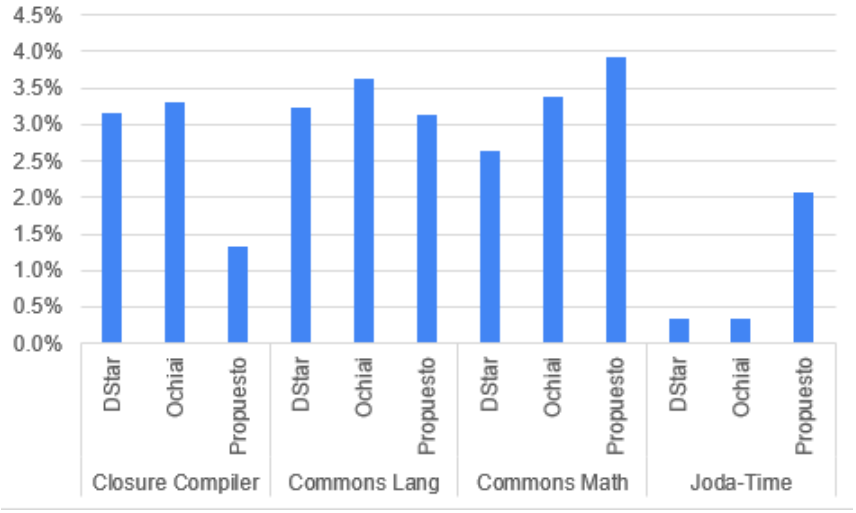

FIGURA 5.5: Porcentaje de LOCr por cada programa y método.

y además las líneas de código totales a evaluar para localizar las fallas son menos de la mitad que en los otros tres métodos.

En la gráfica de barras 5.5 se puede apreciar mejor la comparación de cada método por programa del porcentaje de LOC por revisar del total. Para los programas *Closure Compiler* y *Commons Lang* el método propuesto cuanta con menos líneas de código por revisar lo cual lo convierte en el mejor método para la localización de fallas para los programas en mención; sin embargo para los programas *Commons Math* y *Joda-Time* en método propuesto admite más LOC por revisar antes de localizar las fallas siendo así no óptimo para estos últimos.

En las gráficas 5.6, 5.7, 5.8 5.9 se exponen un ejemplo por cada programa y método usado para este trabajo. En cada gráfico se muestra el coeficiente de sospecha asignado a cada linea de código por el que recorre la ejecución de los test en cada uno de los programas que se uso como ejemplo para probar el método.

En la figura 5.6, que corresponde al error 001 del programa *Closure Compiler*, que presenta error en el rango de líneas que se encuentra entre 924 - 988, podemos resaltar que efectivamente las líneas mencionadas tienen los valores mas elevados para los cuatro métodos. Adicional a lo mencionado de este gráfico rescatamos que el coeficiente de sospecha del método propuesto esta muy por encima de los

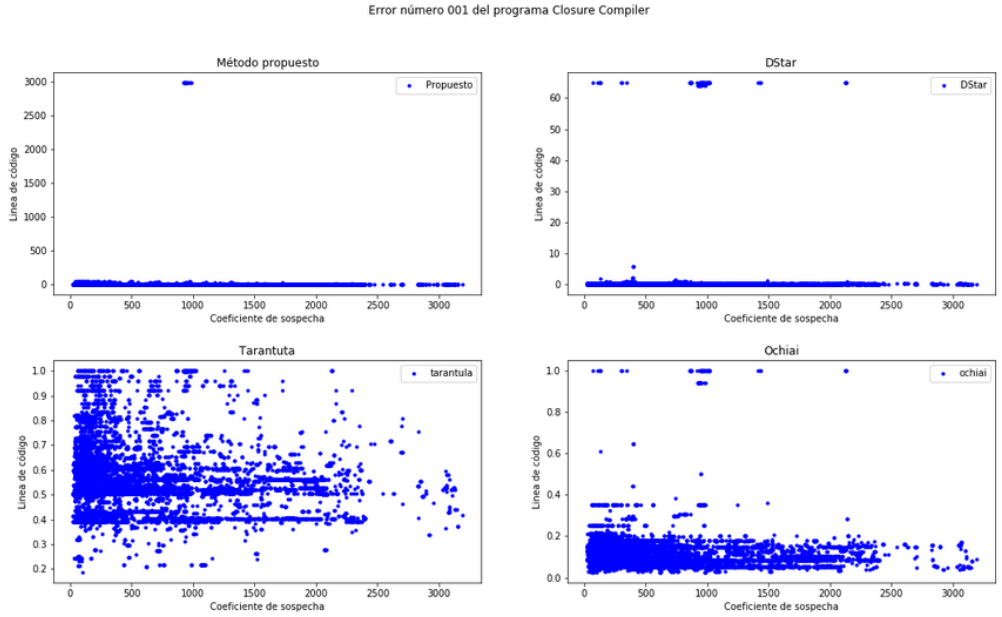

FIGURA 5.6: Coeficientes de sospecha por LOC para el error 001 en el programa Closure Compiler.

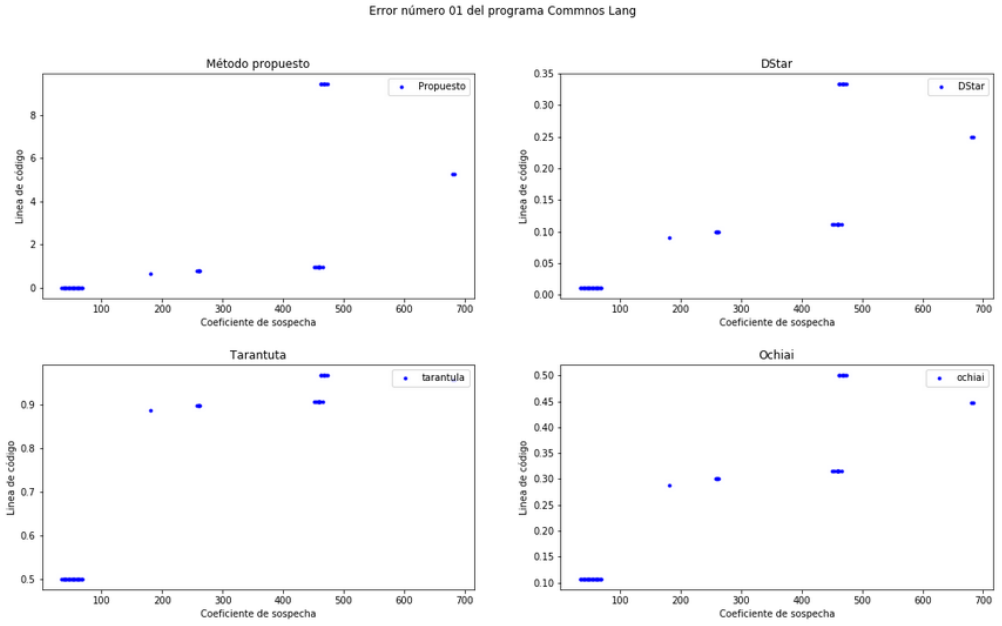

FIGURA 5.7: Coeficientes de sospecha por LOC para el error 01 en el programa Commons Lang.

demás valores de las líneas, lo cual no se cumple en los demás métodos en los que observamos valores muy cercanos en líneas que no presentan errores sobre todo para el método *Tarantula*, lo que origina que se tengan que revisar muchas mas líneas antes de localizar la falla.

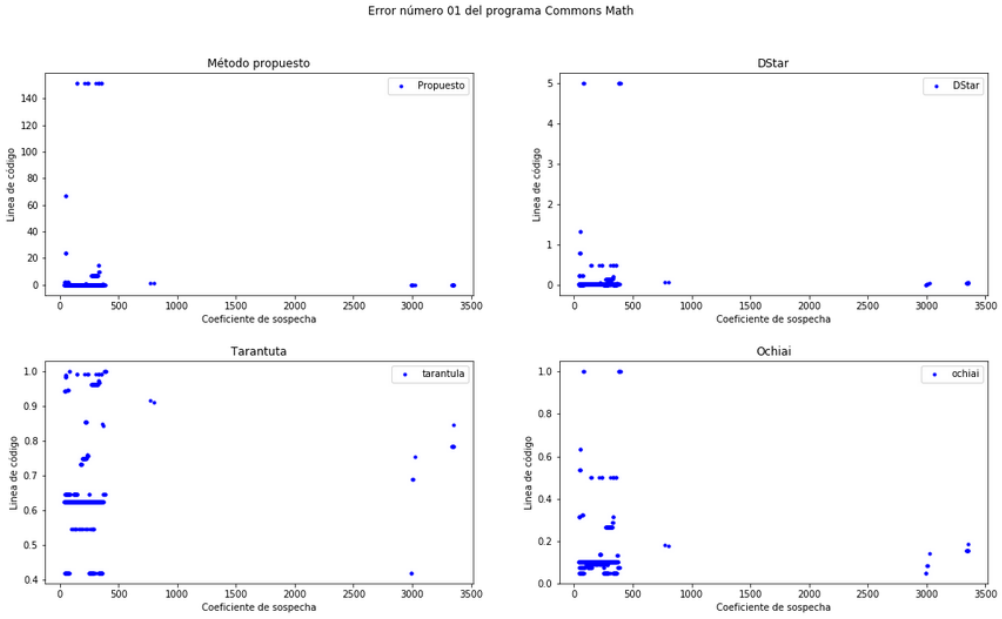

FIGURA 5.8: Coeficientes de sospecha por LOC para el error 001 en el programa Commons Math.

En la figura 5.7, del programa *Commons Lang* y error 01, que presenta error entre las línea 460 y 475, es bastante notorio que el coeficiente de sospecha para dichas líneas tienen los valores más elevado. Como en el caso anterior el coeficiente de sospecha de las líneas en las que se encuentra el error es mucho mayor que el resto de líneas para el método propuesto, lo cual no se cumple para los demás métodos.

En la figura 5.8, del programa *Commons Math* y error 001, que presenta error en el rango de líneas del 380 al 396, se puede apreciar rápidamente que el coeficiente de sospecha para estas línea tiene los valores más elevado. Para este programa podemos notar del gráfico que el método propuesto tiene más LOC por revisar en comparación con los métodos DStar y Ochiai pero menos que Tarantula.

En la figura 5.9, del error 01 del programa *Joda-Time*, que presenta error entre las líneas 215 y 230, podemos resaltar que el coeficiente de sospecha para las lineas antes mencionadas tienen los valores elevados para todos los métodos. Para este caso como en el anterior el método propuesto tiene más LOC por revisar en comparación con los métodos DStar y Ochiai pero menos que Tarantula.

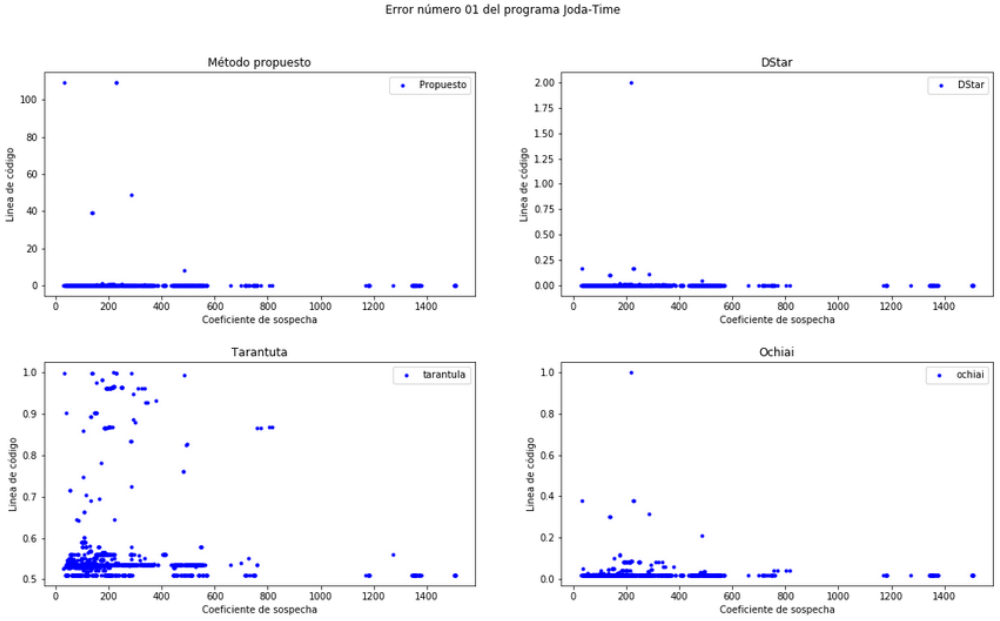

FIGURA 5.9: Coeficientes de sospecha por LOC para el error 01 en el programa Joda-Time.

Capítulo 6

Conclusiones y Trabajo Futuro

En este capítulo se presentan las conclusiones del presente seminario, obtenidas de acuerdo a los objetivos previamente expuestos.

6.1. Conclusiones

El presente trabajo nos permitió alcanzar los objetivos que se plantearon en la sección 1.3, estas conclusiones se detallan a continuación:

- Se logró optimizar el método propuesto en el sentido de afinar la localización de fallos.
- Se logro comprender un modelo estadístico con el cual el método propuesto es mas óptimo en la localización de fallos en código fuente.
- Se ajustó el cálculo del coeficiente de sospecha y se obtuvieron mejores resultados para 2 programas.
- Se logro implementar exitosamente el método propuesto para la localización de fallos en código fuente basado en probabilidades.

- Se expuso y justifico los resultados que se obtuvieron durante el desarrollo del presente seminario en los capítulos anteriores.
- Junto con los resultados del método propuesto para el desarrollo del presente seminario, también se comparó su efectividad con otros tres métodos que cumpla la misma función: *DStar*,*Ochiai*,*Tarantula*.

6.2. Trabajo Futuro

En esta sección final se presentan propuestas para una mejora futura del modelo propuesto.

Programas del mismo tipo

Especializar el método para programas del mismo tipo de tal manera que este se sea el mas eficiente para un tipo especial de programa que tengan estructuras comunes, como pueden ser: motor de Juegos,librerías de redes, robótica entre otros.

Validación con métodos distintos

Así como en este seminario se logró hacer una comparación entre el método propuesto y otros métodos similares, como trabajo a futuro quedaría la medición del método propuesto con otros métodos que no se basen estadísticas tales como métodos de localización basado en invariantes por ejemplo.

Mejora a la implementación presentada

Lograr mejorar la implementación del algoritmo presentado en este seminario. Esto nos permitirá obtener soluciones en menor tiempo y más eficientes al momento de ejecutar el algoritmo para diferentes tipos de programas.

Inclusión de machine learning

Junto al método presentado en este seminario se presenta la posibilidad de usar machine learning con la finalidad de automatizar el proceso de localización de errores en versiones posteriores a los programas sobre los cuales se aplica el método o en programas similares.

Bibliografía

[1] Abaei, G. . S. [DOI: https://doi.org/10.1007/s40595-013-0008-z], 'Using likely invariants for automated software fault localization.', *A. Vietnam J Comput Sci (2014) 1: 79* .

[2] Cai, W. E. W. . Y. Q. . L. Z. . K.-Y. [DOI: 10.1109/COMPSAC.2007.109], 'Effective fault localization using code coverage.', *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)* .

[3] D. Radjenović, M. Heričko, R. T. A. [DOI: http://dx.doi.org/10.1016/j.infsof.2013.02.009], 'Software fault prediction metrics:a systematic literature review.', *Information and Software Technology (2013)* .

[4] Ernst, R. J. D. J. . M. D. [n.d.], 'Defects4j: a database of existing faults to enable controlled testing studies for java programs', Accedido en 11-07-2020 a https://dl.acm.org/doi/abs/10.1145/2610384.2628055?download=true.

[5] IBM [2012], 'Distancias: Medidas de similaridad para datos binarios', https://www.ibm.com/. Accedido en 02-12-2019.

[6] Jones, J. A. and Harrold, M. J. [DOI: 10.1145/1101908.1101949], 'Empirical evaluation of the tarantula automatic fault-localization technique.', *ASE '05 Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, 2005)* .

[7] jupyter [n.d.], 'Jupyterlab 1.0: Jupyter's next-generation notebook interface', Accedido en 02-12-2019 a https://jupyter.org/.

[8] Keller, S. P. . J. C. . R. J. . G. F. . R. A. . M. D. E. . D. P. . B. [10.1109/ICSE.2017.62], 'Evaluating and improving fault localization.', *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* .

[9] ORACLE [n.d.], '¿qué es java?', Accedido en 11-07-2020 a https://www.java.com/es/about/whatis$_java.jsp$.

[10] Pfleeger, S. L., y. A. J. M. . [n.d.], 'Software engineering: Theory and practice (4.a ed.). prentice hall'.

[11] Python [n.d.], 'Python', Accedido en 02-12-2019 a https://www.python.org/about/.

[12] Sommerville, I. . [n.d.], 'Software engineering (8.a ed.). addison wesley.'.

[13] Swarup Kumar Sahoo, John Criswell, C. G. V. A. [DOI: 10.1145/2451116.2451131], 'Using likely invariants for automated software fault localization.', *ASPLOS 2013 - 18th International Conference on Architectural Support for Programming Languages and Operating Systems* .

[14] TRICENTIS [n.d.], 'Software fail watch 5th edition.', Accedido en 01-10-2013 a https://www.tricentis.com/wp-content/uploads/2019/01/Software-Fails-Watch-5th-edition.pdf.

[15] VALENZUELA, J. [n.d.], 'La mars climate se estrelló en marte porque la nasa no tradujo kilómetros a millas.', Accedido en 02-12-2019 a https://elpais.com/diario/1999/10/02/sociedad/938815207_{8}50215.*html*.

[16] W. Eric Wong, Vidroha Debroy, R. G. and Li., Y. [DOI: 10.1109/TR.2013.2285319], 'The dstar method for effective software fault localization.', *IEEE Transactions on Reliability (Volume: 63 , Issue: 1 , March 2014)* .

[17] Yoo, J. S. . S. [DOI: 10.1145/3092703.3092717], 'Fluccs: Using code and change metrics to improve fault localization.', *the 26th ACM SIGSOFT International Symposium* .

[18] Yoo, S. M. . Y. K. . M. K. . S. [DOI: 10.1109/ICST.2014.28], 'Ask the mutants: Mutating faulty programs for fault localization.', *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation* .