Localización de fallas determinadas en código fuente aplicando modelos estadísticos

Jhon Alexander Rojas Isidro

Universidad Nacional de Ingeniería

Asesor: Glen Rodriguez

October 2, 2020

Resumen

En este seminario se presentará un nuevo método para la localización de fallas en código fuente.
El método esta basado en coeficientes de sospechas asignados a cada línea del código fuente a evaluar.
Finalmente se compara con otro tres métodos similares al propuesto.

Objetivos

- 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.

Heurísticas

Son técnicas de solución de problemas que nos permiten obtener pequeñas soluciones con las cuales tomar decisiones de manera muy rápida. Están basadas en conocimientos previos, sentido común o una primera impresión.
Tipos:

- Corazonada educada.
- Sentido común.
- Heurística de disponibilidad.
- Heurística de representatividad.

Coeficiente de sospecha

- Probabilidad asignada a cada parte defectuosa del código fuente.
- Diremos que un método es bueno si puede colocar la parte defectuosa del código fuente con el mayor coeficiente de sospecha.
- De esta manera los desarrolladores solo evaluarán las partes del código con el coeficiente de sospecha más elevado.

Comparación binaria

- Calcular las distancias en definitiva es un procedimiento elemental para el procesamiento de datos.
- Al hablar de otros ámbitos nos referimos por ejemplo a las palabras, cosas o cualquier otro concepto abstracto.
- Estamos interesados en los distintos métodos que tienen como finalidad calcular la distancia entre dos vectores binarios.
- Los valores solo pueden ser 0 y 1.

Comparación binaria

Algunos métodos para calcular la distancia:

- Russel y Rao.
- Concordancia simple.
- Rogers y Tanimoto.
- Kulczynski.

Matriz de cobertura

- Aquella que para un programa en especifico su número de columnas sea igual a la cantidad de lineas del programa más 1 y la cantidad de filas sea igual al número de veces que se ejecuta el mismo.
- Los elementos M[a][b] de la matriz de cobertura tomarán 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 $(+)$ sí la ejecución x del programa fue exitosa, de lo contrario el valor asignado será $(-)$.

Modelos Estadísticos

- Son herramientas usadas con el fin de predecir algún resultado por medio de una ecuación matemática.
- Utiliza variables que serán proporcionadas según el modelo las necesite.
- El modelado estadístico puede predecir una variable dependiente a partir de otras variables que corresponden al entorno base del modelo relacionado.

The DStar Method for Effective Software Fault Localization

Este método tiene su origen en el análisis basado en coeficientes de similitud binaria.
sugiere ubicaciones sospechosas para la localización de fallas.
Utiliza una forma modificada del coeficiente de Kulczynski.

Software Fault Prediction Metrics:A Systematic Literature Review

Este trabajo realiza una revisión, análisis, clasificación de los trabajos relacionados.
Se evaluaron las capacidades de predicción de fallas.

Lenguajes

Par el desarrollo de este seminario se utilizó dos lenguajes de programación:

- Python
- JAVA

Python

Python es un lenguaje de programación interpretado de propósito general.

- Gran cantidad de librerías.
- Sintaxis simple.
- Es de Código Abierto.

Fue utilizado para implementar el algoritmo principal del método utilizado.

JAVA

Java es un lenguaje de programación orientado a objetos.

- Es de Código Abierto.
- Permite la ejecución de un mismo programa en múltiples sistemas operativos.

Fue utilizado para ejecutar los casos de prueba del código fuente.

Defects4J

- Es una base de datos y un marco extensible que proporciona errores reales para permitir estudios reproducibles en la investigación de pruebas de software.
- Contiene 835 errores reales de 5 programas de código abierto del mundo real.
- Cada error va acompañado de un conjunto de pruebas completas que pueden exponer (demostrar) ese error.
- Una vez configurado un programa en Defects4J, se pueden agregar nuevos errores a la base de datos con poco esfuerzo.
- 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.

Definición

En el método que se propone en el presente trabajo, como en métodos similares a este, se utilizará coeficientes de sospecha. El cual se implementó basándonos en diferentes heurísticas con la finalidad de detectar los lugares más probables en donde podemos encontrar la falla en el programa.

Otros trabajos

- Tarantula:

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

- DStar:

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

- Ochiai:

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

Método Propuesto

Nos basamos en heurísticas para generar el coeficiente de sospecha presentado en este seminario.
Las heurísticas que se proponen se detallan a continuación:

Heurísticas

- 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.
- 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.
- 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.

Heurísticas

- 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.
- Se asignará mayor prioridad a la primera y tercera heurística definida.
- Sí no existe ningún fallo en la linea x después de ejecutar n veces el programa, la probabilidad de que el el error se encuentre en la linea x es de 0%.

Método propuesto

La estructura final del coeficiente de sospecha propuesto será:

$$S_{(s)} = \frac{fallo\_ejecutado \cdot (fallo\_ejecutado + exitoso\_no\_ejecutado)}{exitoso\_ejecutado \cdot (fallo\_no\_ejecutado + exitoso\_ejecutado)} \quad (1)$$

Casos de prueba

- Se utilizaron los programas de la base de datos Defects4J que proporciona errores reales.
- De los 835 errores que contiene se usó 331 para el desarrollo del presente seminario
- Los errores proviene de proyectos de código abierto tales como: mockito, commons-lang, commons-math entre otros.
- Cada error va acompañado de un conjunto de pruebas completas que puede demostrar ese error, las cuales se utilizarán con la finalidad de demostrar los objetivos del presente seminario.

Las especificaciones de los programas que usaremos en el presente seminario se detallan en el siguiente cuadro:

Casos de prueba

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

Table: Datos de los programas usados

Aplicación del método

Para poder aplicar el método y comprobar la efectividad del mismo se implementó un algoritmo que determina las principales características necesarias.

Estructura del Algoritmo

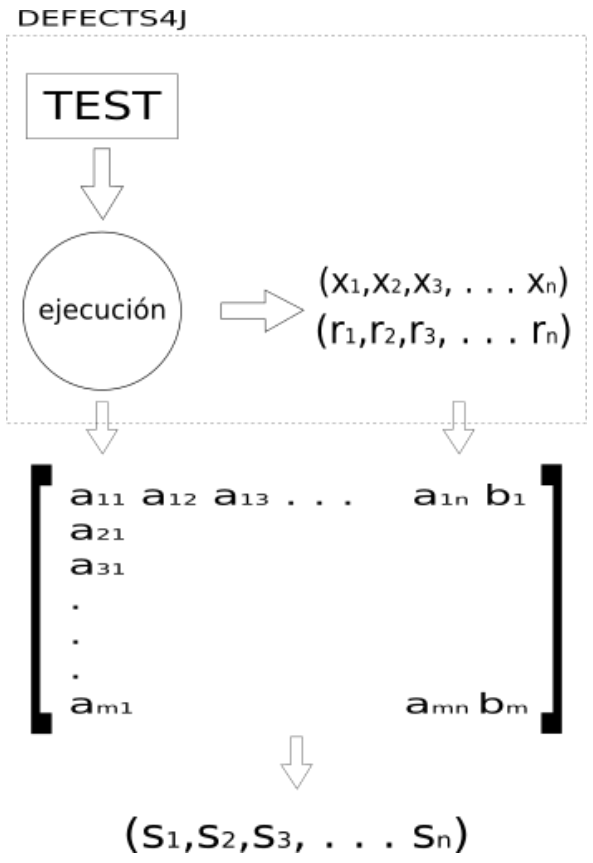

Figure: Estructura del algoritmo

Pseudocódigo algoritmo GetSpecter

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

Resumen

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%

Table: Cantidad de líneas de código a revisar por programa

Closure Compiler 50

El gráfico corresponde al error número 50, del programa Closure Compiler. Para este caso en específico las líneas en las cuales se encuentra el error están comprendidas entre la 435 y 438.

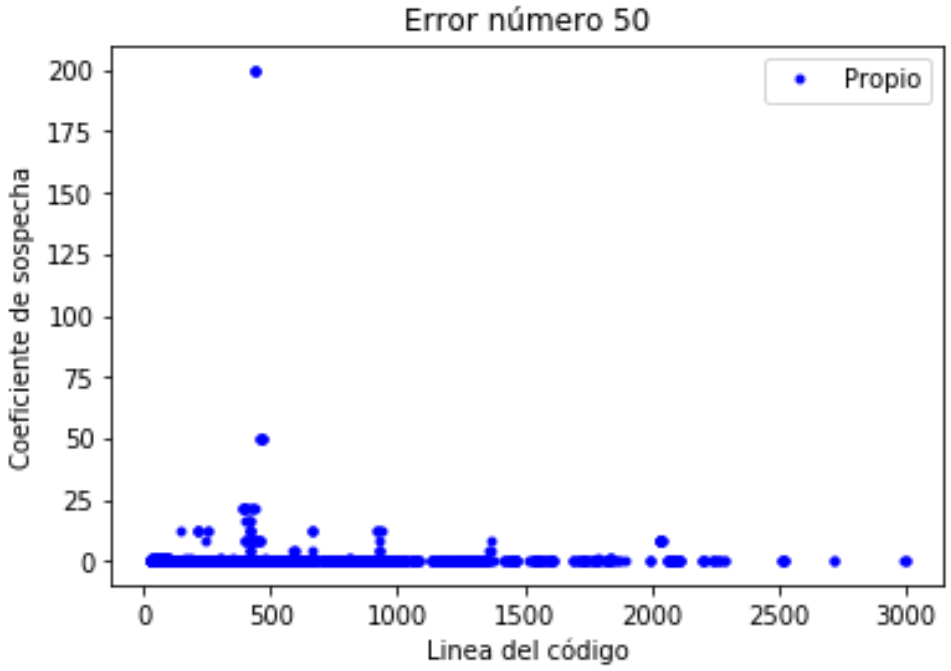

Figure: Coeficientes de sospecha por LOC para el error 50 en el programa Closure Compiler.

Commons Lang 20

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.

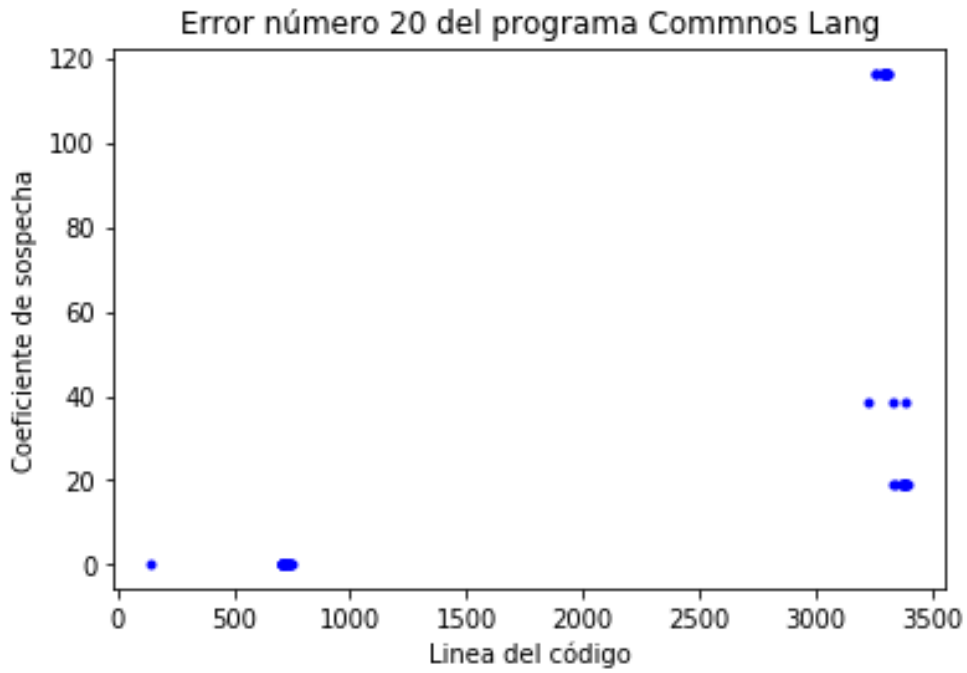

Figure: Coeficientes de sospecha por LOC para el error 20 en el programa Commons Lang.

Joda-Time 16

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.

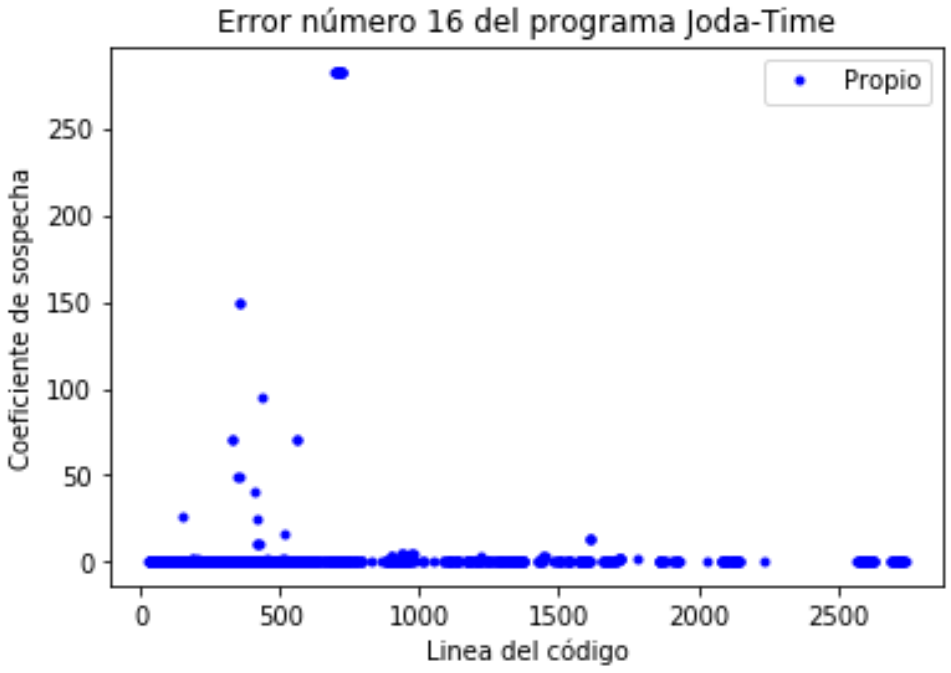

Figure: Coeficientes de sospecha por LOC para el error 16 en el programa Joda-Time.

Resumen

- 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.
- 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.

Cuadro comparativo

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%

Table: Cantidad de líneas de código por revisar por programa y método

Cuadro comparativo

Método	Programa	LOC	LOCr	porcentaje
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%

Table: Cantidad de líneas de código por revisar por programa y método

Gráfico comparativo

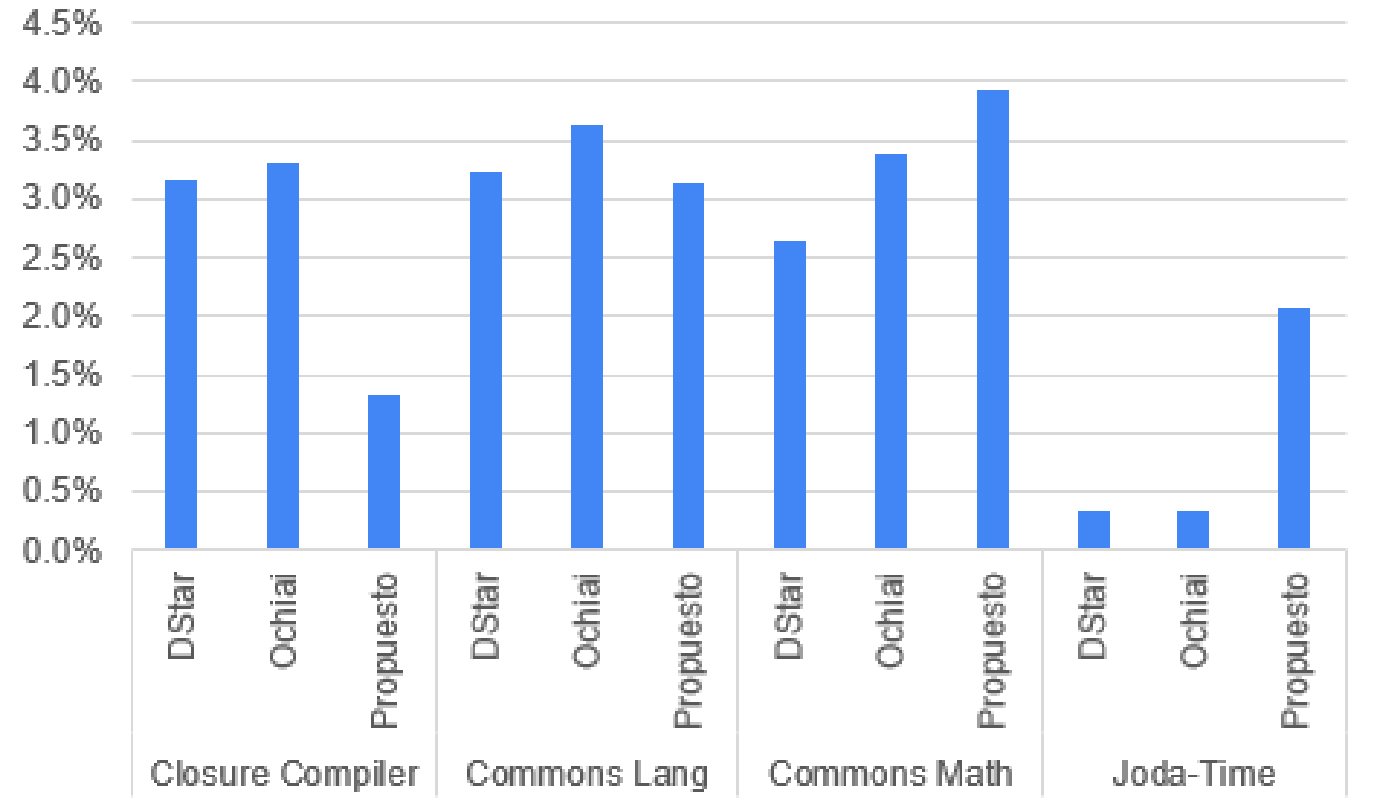

Figure: Porcentaje de LOCr por cada programa y método.

Closure Compiler 1

- Los error se encuentran en el rango de líneas entre 924 - 988.
- Estas líneas tienen los valores mas elevados para los cuatro métodos.
- El coeficiente de sospecha del método propuesto esta muy por encima de los demás valores de las líneas.
- En los demás métodos se observan valores muy cercanos en líneas que no presentan errores.

Closure Compiler 1

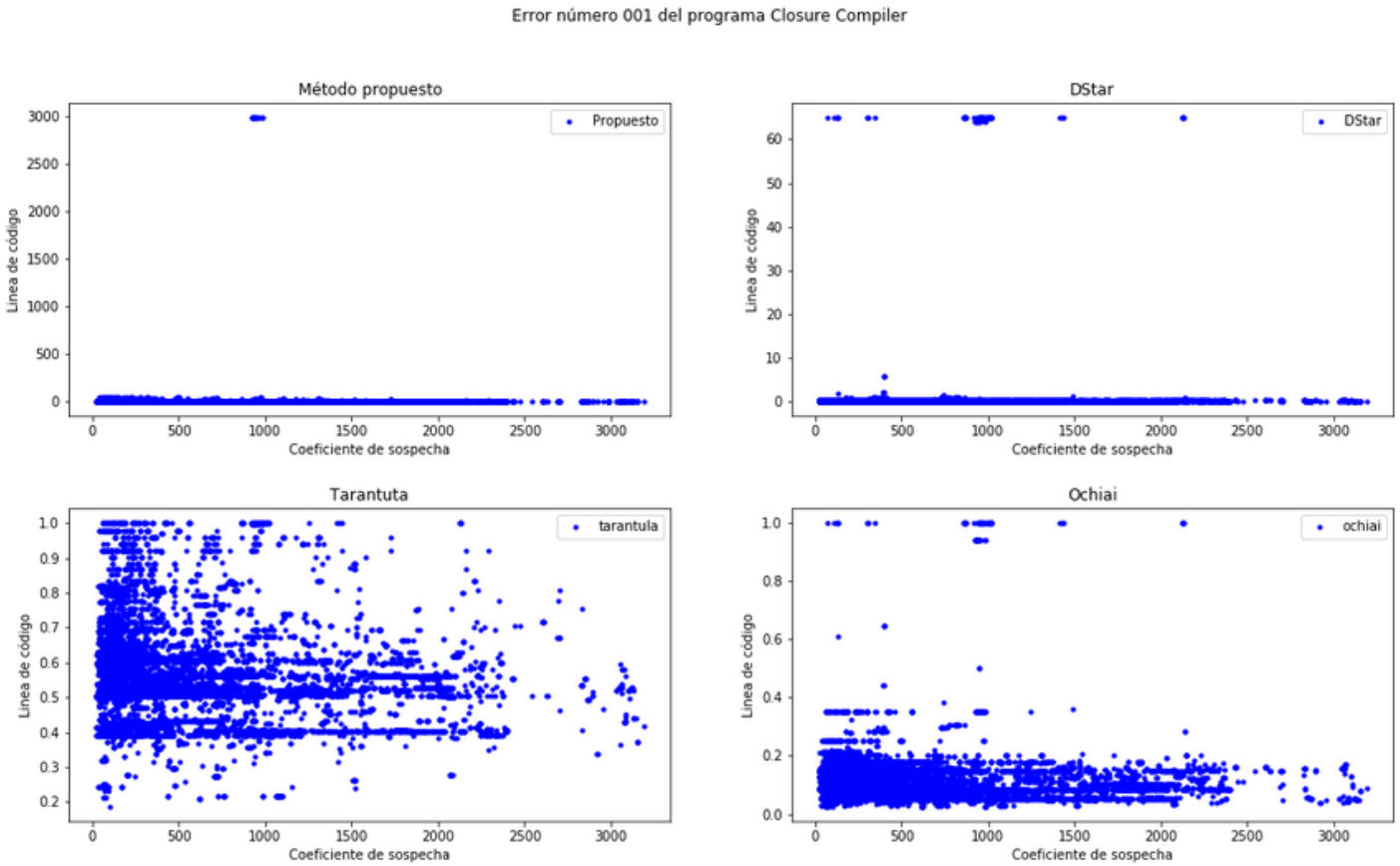

Figure: Coeficientes de sospecha por LOC para el error 001 en el programa Closure Compiler - Error entre las líneas 924 y 988.

Commons Lang 1

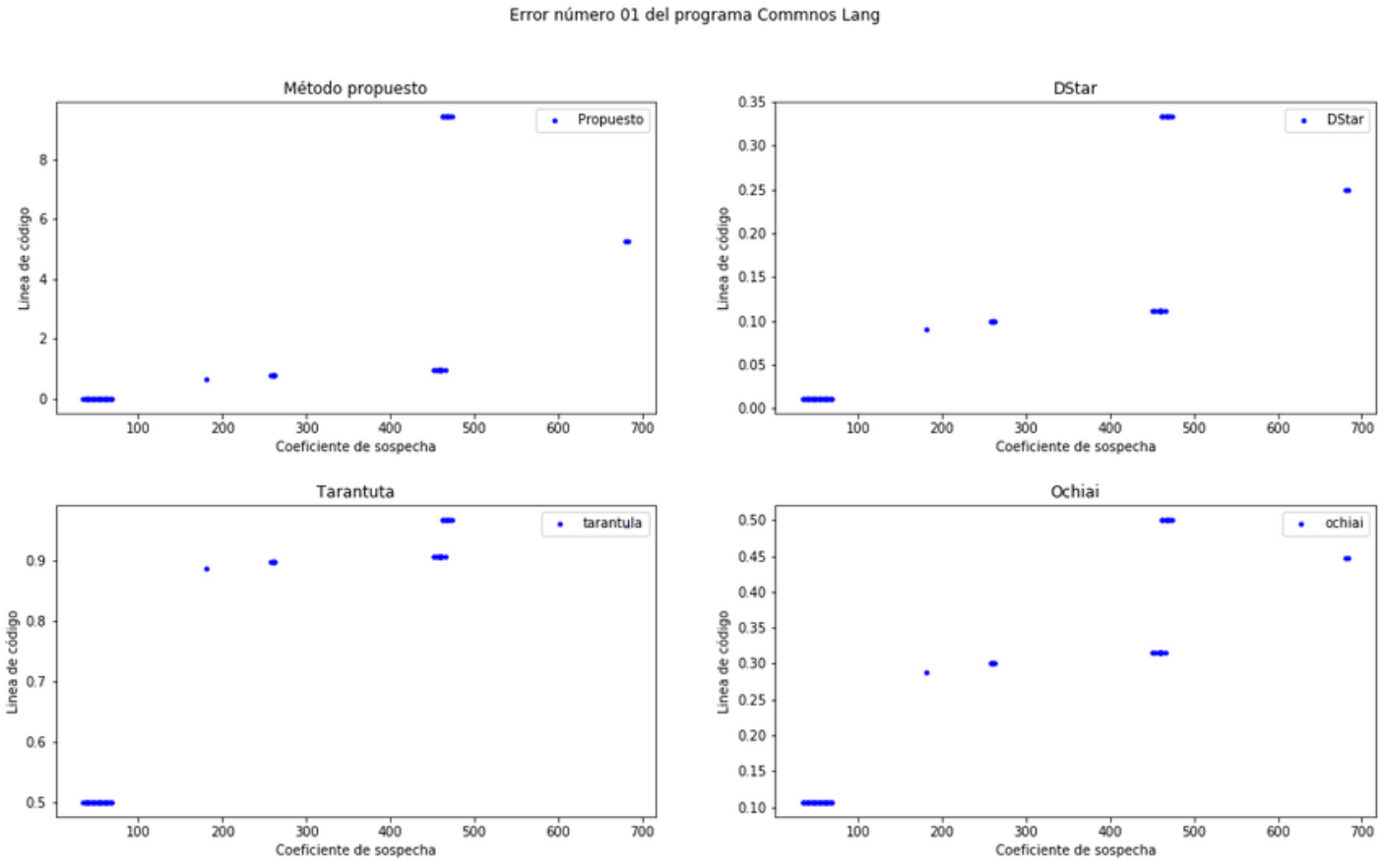

Figure: Coeficientes de sospecha por LOC para el error 01 en el programa Commons Lang - Error entre las líneas 460 y 475.

Commons Math 1

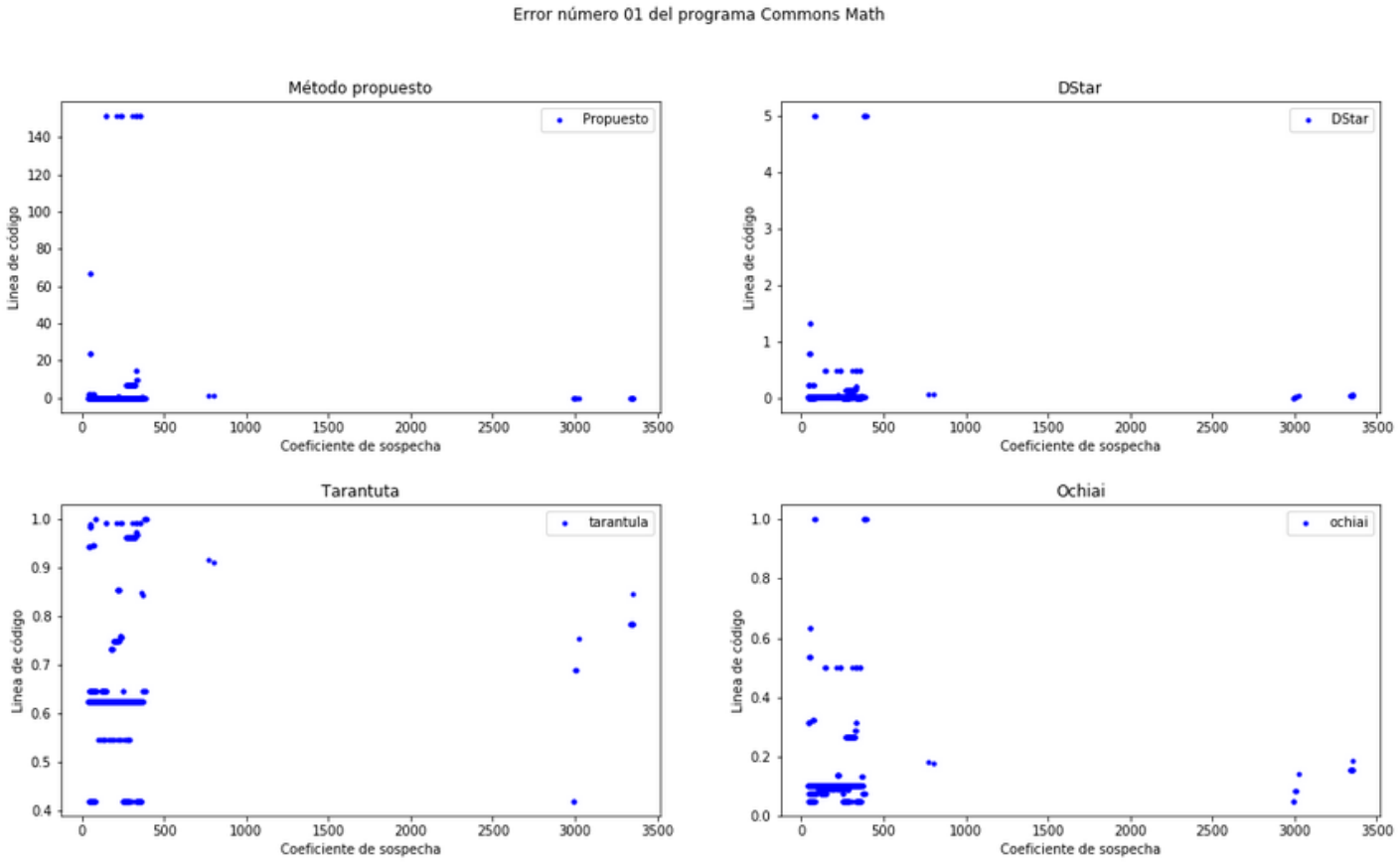

Figure: Coeficientes de sospecha por LOC para el error 001 en el programa Commons Math - Error entre las líneas 380 y 396.

Conclusiones

- Se logró optimizar el método propuesto en el sentido de afinar la localización de fallos.
- Se logró 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.

Conclusiones

- 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*.

Trabajo Futuro

- Programas del mismo tipo.
- Validación con métodos distintos.
- Mejora a la implementación presentada.
- Inclusión de Machine Learning.

Gracias