



**UNIVERSIDAD AUTÓNOMA METROPOLITANA**
**DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA**
**UNIDAD IZTAPALAPA**
**POSGRADO EN CIENCIAS Y TECNOLOGÍAS DE LA INFORMACIÓN**

# IDENTIFICACIÓN DE POTENCIALES HOTSPOTS EN EL SOFTWARE USANDO MÉTODOS DE CLASIFICACIÓN

TESIS QUE PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS (CIENCIAS Y TECNOLOGÍAS DE LA INFORMACIÓN)

PRESENTA:
**HUMBERTO REYES SAN PEDRO**

ASESORES:
**DR. HUMBERTO CERVANTES MACEDA**
**DR. ABEL GARCÍA NÁJERA**

SINODALES:
**DR. EDUARDO FILEMÓN VÁZQUEZ SANTACRUZ**
**DR. ABEL GARCÍA NÁJERA**
**MTRO. VICTOR HUGO ESCANDÓN BAILÓN**

Iztapalapa, Ciudad de México a 18 de octubre de 2023.

# AGRADECIMIENTOS

A la Universidad Autónoma Metropolita en su unidad Iztapalapa que nuevamente me abrió sus puertas para dotarme de conocimientos actuales.

Al posgrado de Ciencias y Tecnologías de la Información de la división de Ciencias Básicas e Ingeniería por darme la oportunidad de participar en el posgrado de maestría.

A todos mis profesores del posgrado por todos sus consejos, experiencias, conocimiento y exigencias que tuvieron en la impartición de cada una de sus clases.

A mis asesores por sus consejos para enriquecer la investigación, por su exigencia al momento de documentar la tesis, por su paciencia del tiempo que me llevo terminar la documentación de la tesis y sobre todo por las buenas personas que son y que fueron empáticos en el periodo de pandemia que nos tocó vivir, difícil, por cierto.

A mis sinodales por el tiempo que dedicaron para revisar mi tesis, realizar sus observaciones para clarificar lo que trataba de transmitir y de igual manera, por las buenas personas que son.

Finalmente, a mi amada familia. A mi esposa Rocio, a mi hija Eréndira, a mi hijo Aníbal y a mi hija Iraida quienes fueron mi fuente de motivación para cumplir con este ciclo en mi vida y por lo cual estaré siempre agradecido con ellos.

# CONTENIDO

# Resumen

El desarrollo continuo del software es una realidad en estos tiempos. Los cambios en el software pueden deberse por el desarrollo de nuevas funcionalidades, a la corrección de defectos o a la reparación de algunas decisiones de diseño o de desarrollo, convenientes a corto plazo al dejarse de hacer, pero que generarán deterioro de algunos componentes del sistema. El resultado de estas decisiones se le conoce como la deuda técnica del sistema. En esta tesis estamos interesados en identificar fuentes de deuda técnica que generan deterioro en los componentes.

La identificación de fuentes de deuda técnica es una actividad fundamental en el desarrollo de software, para lo cual proponemos utilizar modelos de software que permitan identificar de manera eficiente los módulos propensos al cambio o propensos al error, a los que llamamos *hotspots*, y con ello ayudar a los equipos de desarrollo y tomadores de decisiones a guiar de mejor manera los recursos a invertir para pagar la deuda técnica, por ejemplo, mediante refactorización.

Este estudio examina la efectividad de algoritmos de clasificación, de métricas de software y del análisis de los valores de las métricas, con el fin de encontrar una combinación de métricas que mejor nos ayude a identificar *hotspots*.

Se usaron once aplicaciones de código abierto orientado a objetos para llevar a cabo nuestro estudio y validación de resultados. Parte de las aplicaciones se usaron como datos de entrenamiento y el resto como datos de prueba para los algoritmos, con lo que tuvimos la posibilidad de realizar validaciones de los resultados de nuestra clasificación.

Se obtuvieron resultados que permiten sugerir un método de clasificación de Bayes ingenuo, con una normalización de las métricas y una combinación de tres métricas de código que permite realizar una buena identificación de *hotspots*.

**Palabras clave**: deuda técnica, *hotspots*, métricas de software, clasificadores.

# Capítulo 1

# Introducción

En 1992, fue introducida la metáfora de deuda técnica (DT) desde un punto de vista financiero, para reconocer los posibles efectos negativos a largo plazo y de largo alcance que tendrá el código inmaduro [2]. En 2012, los investigadores estimaron de manera conservadora que, por cada 100 mil líneas de código en una aplicación, en promedio se tienen que invertir $361,000 dólares en DT, es decir, este es el costo de eliminar los problemas de calidad estructural que amenazan la viabilidad de las aplicaciones [1]. La DT es generalizada, afecta a todos los aspectos de la ingeniería del software, desde cómo levantamos los requerimientos, cómo diseñamos el sistema y hasta cómo lo implementamos. El término deuda técnica fue acuñado por Howard Cunningham en 1992, no es nuevo y tampoco los conceptos que cubre.

Durante 35 años los ingenieros de software han conocido a la DT con otros nombres, tales como: mantenimiento de software, evolución, envejecimiento, decadencia, reingeniería, sostenibilidad, entre otras. Pero el progreso ha sido poco sistemático, el tema no se considera atractivo y rara vez es enseñado en las escuelas, hasta la fecha [1].

La deuda técnica arquitectónica (DTA) se presenta al optar por soluciones rápidas a corto plazo, pero que posteriormente comienzan a erosionar el diseño con lo cual se afectan los atributos de calidad (AC) tanto internos como externos. Los primeros AC que se ven afectados son los de mantenibilidad y de evolucionabilidad [19]. Al presentarse lo anterior, se comprometerán las siguientes entregas del sistema dado que se tiene un código complejo que deberá ser reparado.

Conforme evoluciona un sistema de software, ciertos módulos cambian de forma más frecuente. Si estos cambios son debido a *bugs*, entonces podemos pensar que estos módulos son propensos a un *bug* (fuentes de deuda técnica). Estos módulos que cambian frecuentemente son conocidos como *hotspots* [7], [15] y el objetivo de este trabajo es proponer su identificación.

Identificar los *hotspots* en los sistemas es de suma importancia para las empresas desarrolladoras de software, ya que podrán orientar parte de sus recursos a ellos para disminuir su DT y tener productos de software de mejor calidad y mantenibles.

## 1.1. Deuda técnica en el software

La DT en el software se va generando y acumulando cuando se elige dejar de hacer algo en un momento del diseño o desarrollo de un sistema y que posteriormente impedirá realizar desarrollos en tiempos aceptables si no es atendido esto que se dejó de hacer. Dejar de hacer algo podría ser no levantar los requerimientos adecuadamente o a tiempo, realizar documentación insuficiente de los requerimientos o de las especificaciones técnicas, comenzar a desarrollar antes de tener el diseño terminado y autorizado, realizar un conjunto de pruebas limitado que no permitan detectar fallas en el sistema antes de liberarlo a producción, entre otros [1].

Para poder detectar la DT en el software se requiere de herramientas que permitan identificar aquellos módulos en el software que, por sus características, nos puedan sugerir módulos que pueden requerir de atención para ser analizados y determinar si requieren de refactorización[1]. Al aplicar refactorización para disminuir la DT en el software, permitirá en el mediano plazo a los equipos de desarrollo, dedicar tiempo y recursos para realizar nuevas funcionalidades en el sistema. Gestionar la DT en el software puede aplicarse de diferentes formas, desde invertir recursos para disminuirla hasta decidir en algunos casos no hacer nada, dado que podría resultar más caro aplicar refactorización, por lo que se podría decidir vivir con la DT en algunas partes del software [4].

Una regla de la alta dirección nos dice que lo que no es medible, no es manejable (Peter Drucker), en este caso, para la DT aplica totalmente, ya que al poder identificarla se puede decidir qué hacer con ella y establecer prioridades para irla disminuyendo o controlando.

[1] La refactorización es una técnica de la ingeniería de software que permite la optimización de un código previamente escrito, por medio de cambios en su estructura interna sin esto suponer alteraciones en su comportamiento externo. La refactorización no busca ni arreglar errores ni añadir nueva funcionalidad, si no mejorar la comprensión del código, para facilitar así nuevos desarrollos, la resolución de errores o la adición de alguna funcionalidad al software.

## 1.2. *Hotspots*

En la actualidad, gestionar y controlar cambios a productos de software complejos es una tarea de suma importancia para la industria del software. Los cambios al software pueden deberse por mantenimiento, por nueva funcionalidad o por fallas en algunas partes de este. Nuestro estudio se centra en anticiparse a estas fallas que se pueden presentar y prevenirlas con mantenimientos correctivos al software. La anticipación de la que hablamos es detectar aquellos módulos que sean propensos al cambio o propensos al error y que, dentro del desarrollo de esta investigación, los denominaremos *hotspots* (puntos calientes) [7], [15].

Catalogar como *hotspot* a un módulo que sufre cambios sería una forma sencilla de hacerlo, pero estos cambios pueden deberse a una refactorización al módulo o por adición de nueva funcionalidad. A pesar de que los módulos cambien de tamaño o su interacción con otros módulos del sistema, no necesariamente es porque se esté generando DT en él. Por otro lado, podríamos tener módulos que cambian y que tiene DT, pero los cambios no van dirigidos a corregir la DT. Y podemos tener módulos que están cambiando para corregir la DT que pudieran tener.

En la actualidad existen en el mercado varias herramientas de análisis de código (HAC) de las aplicaciones. Estas ofrecen una serie de métricas, en algunos casos de cada uno de los archivos, en donde se encuentra el código de los módulos que componen la aplicación analizada. Cada métrica es calculada bajo las reglas que tiene definida cada HAC. En muchos de los resultados de las métricas de las diferentes herramientas, no coinciden [23], esto debido a que no existe una unificación de las reglas a aplicar para calcular cada métrica por las diferentes HAC. Aún con estas diferencias, se utilizarán aquellas métricas en las que sí coinciden.

En este trabajo nos centraremos en la detección de posibles *hotspots* que puedan existir en los sistemas con el fin de sugerir a los equipos de desarrollo dónde podrías dirigir sus recursos de manera óptima y mantener sus sistemas con la menor DT.

## 1.3. Problema de detección de *hotspots*

En esta búsqueda de mejorar la situación de los sistemas, las métricas que se pueden obtener de cada módulo que conforma la aplicación de software podrían servir de apoyo para tratar de clasificarlos como un *hotspot* o no. Esto debido a que tenemos métricas que nos ofrecen características de cada módulo como el tamaño por número de líneas de código (CLOC) u otras que han sido propuestas por Chidamber y Kemerer [14] las cuales se han utilizado en varios estudios, ya que ofrecen características por módulo como la suma de la complejidad ciclomática[2] de los métodos de una clase (WMC), profundidad del árbol de herencia (DIT), número de hijos (NOC), acoplamiento entre objetos (CBO) y respuesta por clase (RFC).

En la actualidad los equipos grandes de desarrollo de software se están apoyando de sistemas de control de versiones (SCV) para poder realizar su trabajo de manera integral y controlar los posibles empalmes en un módulo de software entre dos o más desarrolladores al mismo tiempo. También hacen uso de sistemas de control de incidentes (SCI) en los cuales se registran los módulos de software que estuvieron relacionados cuando se presenta un incidente en el sistema y que requirió de reparación.

Se define como “cambios no estructurales de un módulo” cuando se modifican líneas, se añaden líneas, se eliminan líneas, se amplían clases o se reducen clases. Se define “cambio estructural” cuando, hay cambio en la lista de parámetros de un método, se agregan clase o métodos, se eliminan clases o métodos [6]. Como ejemplo de estás métricas tenemos la complejidad, tamaño, dependencias, cohesión, entre otras.

Adicionalmente a las métricas anteriores, existen métricas que son "temporales" y que miden la cantidad de cambios de un módulo del sistema a lo largo del tiempo. Estas se pueden obtener a partir del análisis de bitácoras del SCV que, además, se puede relacionar con información del SCI para obtener métricas históricas relacionadas con errores (*bugs*). A esta relación que se pueda obtener entre ambos sistemas (SCV y SCI) la denominaremos un *bug commit* asociado a los módulos.

Con algunas métricas antes mencionadas o con otras, que se han sugerido en los estudios relacionados, lo que se busca en esta investigación es encontrar aquellos módulos

[2] La Complejidad Ciclomática es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa.

"propensos a cambios" [5], [6], [19], [20] o "propensos a errores" [17]. Estos módulos se definen de esta manera ya que requieren ser corregidos por un mal diseño o por una mala construcción del sistema. Algunos estudios buscan estos módulos antes de que el software esté disponible para su uso (para evitar DT) [17] y otros los buscan en el software que ya está disponible para su uso (para disminuir DT) [7], [9], [15]. En nuestro caso consideraremos los *hotspots* como los módulos (archivos de código fuente) propensos al cambio o al error y los buscaremos en software ya disponible para su uso.

Como lo comentamos previamente, hay varios esfuerzos por encontrar los *hotspots* de una aplicación de software, algunos ya integrados en herramientas disponibles en el mercado y otros en trabajos de investigación que buscan proponer una solución sistemática que pueda ser aplicada en cualquier proyecto con un porcentaje aceptable de clasificación de los *hotspots* [19], [20], [21].

El problema es que cada aplicación de software tiene sus propias características, por ejemplo, el objetivo para el cual está dirigido, el ámbito que cubre, el número de funcionalidades incorporadas, entre otras, entonces encontramos una diversidad en la construcción de cada módulo de software a pesar de que estén construidas bajo metodologías y patrones de desarrollo que dicta la ingeniería de software.

Lo anterior nos lleva a buscar la manera de caracterizar cada uno de los módulos del software. Cada módulo puede ser un archivo, el cual puede representar una clase en lenguaje java o un archivo de cabecera en lenguaje C++. Esta caracterización de cada archivo del sistema se puede realizar a través de la valoración de cada uno de ellos. Lo anterior ha sido tema de estudio y hoy en día existen herramientas que nos permiten realizar a nivel sistema, un análisis y entregar métricas a nivel archivo en algunos casos y a nivel paquete, en otros [15], [17], [19].

El problema real que estamos abordando en este estudio es que no queda claro qué métricas son las que mejor permiten identificar *hotspots*, por lo que, a través de clasificadores se realizará una exploración de un conjunto de métricas seleccionadas y podamos obtener una combinación de ellas que nos permitan obtener un porcentaje aceptable de clasificación de *hotspots*.

## 1.4. Problemas de clasificación

Teniendo la capacidad de recolectar métricas estructurales e históricas de los elementos que conforman un sistema de software, el siguiente problema es la clasificación de estos elementos en aquellos que son o no *hotspots*.

Este tipo de problemas tienen el propósito de clasificar un conjunto de módulos en diferentes subconjuntos (categorías), los cuales comparten las mismas o parecidas características (métricas). Este tipo de problemas han sido abordados por diferentes ramas de las ciencias de la computación y en específico, de la inteligencia artificial, viéndolo como un problema de regresión logística [19], uno de optimización [17] o uno más de clasificación con aprendizaje automático [5], [20].

En la actualidad son muy usados los algoritmos de aprendizaje automático para la clasificación de objetos, por ejemplo, la clasificación de documentos o correos electrónicos spam [35]. Cada uno de los algoritmos de clasificación toman un conjunto de datos que se desean clasificar en diferentes clases. Su forma de abordar los datos puede derivar en diferentes resultados de clasificación que podemos obtener de cada uno de ellos.

Una vez que se confirma que existe DT en una aplicación, el disminuirla o erradicarla llevará tiempo. Por lo que, el proceso de clasificación seguramente se aplicará varias veces, ya que difícilmente se disminuirá la DT en un solo evento. Además de que, la aplicación podría ser modificada por nuevas funcionalidades, refactorización o solución a incidentes, con lo cual la DT podría variar.

## 1.5. Objetivos del trabajo de investigación

### 1.5.1 Objetivo general

El objetivo general de esta investigación es el siguiente:

- Identificar potenciales *hotspots* en sistemas de software aplicando métodos de clasificación.

### 1.5.2 Objetivos específicos

Para alcanzar el objetivo general se plantean los siguientes objetivos específicos:

1. Hacer una revisión bibliográfica para identificar estudios enfocados en el reconocimiento y la reducción de la deuda técnica en el código.
2. Revisar los métodos de clasificación existentes y seleccionar uno que facilite la identificación de *hotspots*.
3. Implementar el método de clasificación seleccionado y aplicarlo a diferentes aplicaciones de código abierto.
4. Evaluar la efectividad del método seleccionado.

## 1.6. Aportaciones

Las aportaciones que se hicieron con este trabajo de investigación son las siguientes:

- Se propusieron dos métodos de clasificación de *hotspots* con una técnica de normalización de los datos de entrenamiento y prueba, lo que nos permitió la homogenización de estos y tener un comportamiento diferente en el aprendizaje de los algoritmos.
- Se desarrollaron dos algoritmos uno de aprendizaje automático (AA) y uno exhaustivo que permitieron la clasificación, evaluación de los resultados obtenidos y la comparación de resultados entre ellos.
- Se propuso una combinación de métricas reducidas de todas las utilizadas que ofrece los mejores resultados.
- Con el método que se propone se puede detectar en etapas tempranas la DT de los sistemas, ubicando los *hotspots* que pudieran irse manifestando.
- Se presentó el artículo titulado “Identificación de Potenciales Hotspots Usando Métodos de Clasificación” en el XIII Congreso Mexicano de Inteligencia Artificial, COMIA 2021.

## 1.7. Estructura del documento

El presente trabajo está estructurado de la siguiente manera. En el capítulo 2 se presentan las bases teóricas de la investigación. En el capítulo 3 se presentan los antecedentes, el estado del arte y una conclusión de la revisión sistemática de la literatura. En el capítulo 4 se presenta la metodología implementada en la investigación. Posteriormente, en el capítulo 5, se presenta la experimentación y análisis de los resultados obtenidos. Y, finalmente, en el capítulo 6 se presentan las conclusiones del presente trabajo.

# Capítulo 2

# Antecedentes

El presente estudio se relaciona con dos áreas del conocimiento que le dan forma y se relacionan con la investigación planteada. Por un lado, se tiene a la inteligencia artificial (IA), la cual es uno de los campos de la ciencia, que, en las últimas décadas ha avanzado más rápidamente debido a un mayor uso del método científico para experimentar y comparar enfoques. En ella, una de sus ramas que está teniendo mucho auge es el aprendizaje automático (AA), cuya tarea es hacer predicciones o tomar decisiones basadas en datos de entrenamiento [43]. La otra área es la ingeniería de software, cuya disciplina de arquitectura de software (AS) incluye métodos que permitan evaluar la arquitectura, el diseño o la construcción del código de un sistema. Dicha evaluación se realizaría a través de un conjunto de valores de métricas que pueden ser obtenidos de los sistemas a ser evaluados para tratar de detectar y disminuir su DT. En ese sentido, estas áreas del conocimiento se consideran adecuadas para fundamentar la instrumentación y sustentar la línea de investigación: aplicando métodos de clasificación para identificar *hotspots* en los sistemas.

## 2.1. Ingeniería de software

La ingeniería de software es una rama de la ingeniería que se interesa por todos los aspectos de la producción de software, desde las primeras etapas de la especificación del sistema hasta el mantenimiento del mismo después de que se pone en operación [36].

En la actualidad es imposible pensar en un mundo sin software. Muchas de las operaciones de las grandes empresas, de los gobiernos y hasta de las personas están asociadas a una computadora y a un software de control. La infraestructura y distribución industrial están computarizadas. El entretenimiento, la industria de la música, los videojuegos, el cine y la televisión, usan software de manera intensiva. Por lo tanto, la ingeniería del software es esencial para el funcionamiento de las sociedades.

Los sistemas de software son abstractos e intangibles. No están restringidos por las propiedades de los materiales, regidos por leyes físicas ni por procesos de fabricación. Esto simplifica la ingeniería de software, pues no existen límites naturales a su potencial. Sin embargo, debido a la falta de restricciones físicas, los sistemas de software pueden volverse rápidamente muy complejos, difíciles de entender y costosos de cambiar.

Algunos suponen que el término software es solo otro nombre para los programas de cómputo. No obstante, cuando se habla de ingeniería de software (IS), esto no solo se refiere a los programas en sí, sino también a todo el proceso de producción del software, a la documentación asociada y los datos de configuración requeridos para hacer que estos programas operen de manera correcta [36].

### 2.1.1 Calidad del software

La calidad del software se entiende como la conformidad con la especificación o satisfacer las necesidades del cliente. El desarrollo de software debe entregar al usuario la funcionalidad y el desempeño requeridos, además debe ser sustentable, confiable y utilizable, entre otras [36].

Cuando se habla de la calidad del software, se debe considerar que el software lo usan y cambian personas, además de sus desarrolladores. En consecuencia, la calidad no tiene que ver sólo con lo que hace el software, también incluye el comportamiento del software mientras se ejecuta, la estructura y organización de los programas del sistema y la documentación asociada. Esto se refleja en los llamados atributos de calidad o atributos no funcionales del software. Ejemplos de dichos atributos son modificabilidad, disponibilidad, escalabilidad, interoperabilidad, rendimiento, seguridad, testeabilidad (facilidad de probar el sistema), usabilidad, por mencionar algunos.

### 2.1.2 Deuda técnica

La evolución de la ingeniería de software (IS) ha permitido desarrollar software a través del uso de estructuras, herramientas y técnicas, pero aún en estos días con estos adelantos, se siguen teniendo diseños o códigos complejos en el software en los que se pueden quebrantar algunas reglas de diseño o de construcción.

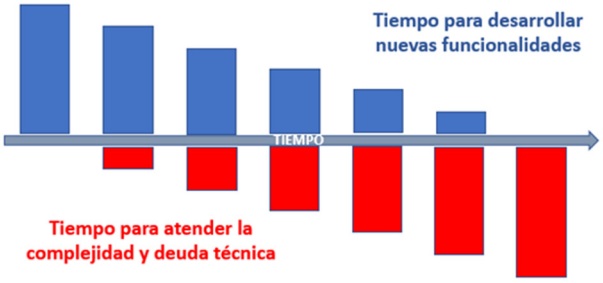



**Fig. 1 Representación de la DT en un sistema si no es atendida con el paso del tiempo.**[3]

La DT es "[...] una colección de constructos de diseño o de implementación que son convenientes a corto plazo, pero que establecen un contexto técnico que puede hacer que los cambios futuros sean más costosos o imposibles [...]" (Averigou et al. 2016). En la Fig. 1 podemos ver cómo un sistema cuándo comienza a desarrollarse está libre de DT (la mayoría de las veces). Al paso del tiempo, se van agregando nuevas funcionalidades y en algunas ocasiones se toman decisiones de diseño o codificación convenientes a corto plazo, pero si no son reparadas pronto, van erosionando de tal forma que, se hace presente la DT arquitectónica (DTA) y la DT en código, respectivamente [1]. Lo anterior provoca que los equipos de desarrollo consuman tiempo en reparar o lidiar con la DT en lugar de dedicar ese tiempo en desarrollar nuevas funcionalidades.

#### 2.1.2.1 Deuda técnica arquitectónica

La deuda técnica arquitectónica (DTA) es una metáfora utilizada para describir el diseño arquitectónico subóptimo y las opciones de implementación que brindan beneficios a corto plazo a costa del deterioro gradual a largo plazo de la calidad del software [42]. El deterioro de

[3] Fuente: How to deal with Technical Debt in Scrum, "*The Liberators*", Ene/2018, https://medium.com/the-liberators/how-to-deal-with-technical-debt-in-scrum-f4ec3481eabb

la calidad del software se manifiesta en módulos que serán en el futuro propensos a cambios o errores.

LA DTA en un sistema de software está impulsada por decisiones (poco analizadas o convenientes) de diseño sobre su estructura (capas, descomposición en subsistemas, interfaces), tecnologías (frameworks, paquetes, bibliotecas, enfoque de implementación), lenguajes, proceso de desarrollo o plataforma, teniendo un impacto en los atributos de calidad tanto internos como externos, por ejemplo, modificabilidad, escalabilidad y disponibilidad. La DTA es difícil de detectar y su corrección es amplia, desalentadora y a menudo evitada. Los sistemas grandes y de larga duración sufren de DTA, mientras que los pequeños y de corta duración mueren antes de que la DTA se convierta en un problema real [38].

Las causas probables que provocan la DTA son la presión del tiempo de entrega o el impulso comercial por liberar productos antes que la competencia. Las consecuencias de tener DTA en el sistema son velocidad de desarrollo reducida, mayor costo de mantenimiento, imposibilidad de implementar nuevas funcionalidades. Los síntomas de cómo se puede manifestar la DTA la encontramos en módulos del sistema recurrentes con errores, problemas de rendimiento o de desarrollo.

#### 2.1.2.2 Deuda técnica en el código

La DTA deriva inevitablemente en la implementación, de tal forma que se vincula ésta con el código fuente y por lo tanto se manifiesta la deuda técnica en el código (DTC) [1]. La DTC es generada por esas decisiones convenientes a corto plazo en la implementación (ineficiencias) y si éstas no son reparadas pronto se manifestarán en módulos problemáticos del sistema con errores recurrentes y que requerirán de un esfuerzo adicional para corregirlos. Uno de los atributos de calidad en el software que se verá impactado es el de la mantenibilidad.

La mantenibilidad del software se ha cuantificado a partir de valores de métricas que pueden ser obtenidos de él. Existen diversas métricas de código, de las cuales hablaremos más adelante. Varias de estas métricas se pueden calcular a nivel código fuente y usarse para evaluar propiedades de calidad bien conocidas como: herencia, acoplamiento, cohesión, complejidad, tamaño, entre otras [14].

Los valores de las métricas de manera aislada ubican un tipo de ineficiencia en un módulo, pero tener varios valores de métricas en un mismo módulo nos puede indicar con mayor probabilidad que es fuente de DT (es decir, un *hotspot*). Basándose en las métricas del software se han realizado múltiples estudios de investigación tratando de identificar la DT en el software de manera más precisa.

### 2.1.3 Métricas de los sistemas

En la industria de software, el mercado es cada vez más exigente y la calidad es una de las características que se debe de cumplir al desarrollar software.

La necesidad de ***medir*** procesos, software, productos o recursos es por cuatro razones, para caracterizar, evaluar, predecir y mejorar. La medición del software se ocupa de derivar un valor numérico para un atributo de un componente, sistema o proceso de software. Al comparar dichos valores unos con otros y con los estándares que se aplican, es posible extraer conclusiones sobre la calidad del software o valorar la efectividad de los procesos, las herramientas y los métodos de software [36]. La meta a largo plazo de la medición del software es usar la medición en lugar de revisiones para realizar juicios de la calidad del software [36] o en nuestro caso en la identificación de la DT.

Una ***métrica*** de software es una característica de un sistema de software, documentación de sistema o proceso de desarrollo que puede medirse de manera objetiva. Los ejemplos de métricas incluyen el tamaño de un producto en líneas de código, el número de fallas reportadas en un producto de software entregado o el número de días hombre requerido para desarrollar un componente del sistema. Las métricas de software pueden ser métricas de control o de predicción. Como sus nombres lo dice, las métricas de control apoyan la gestión del proceso y las métricas de predicción ayudan a predecir las características del software. Las métricas de control se asocian por lo general con procesos de software. Ejemplos de las métricas de control o de proceso son: el esfuerzo promedio y el tiempo requerido para reparar los defectos reportados. Las *métricas de predicción* se asocian con el software en sí y a veces se conocen como métricas de producto. Ejemplos de métricas de predicción son: la complejidad ciclomática de un módulo, la longitud promedio de los identificadores de un

programa y el número de atributos y operaciones asociados con las clases de objetos en un diseño [36].

El implementar el modelo adecuado de calidad al momento de desarrollar software es una de las decisiones más importantes. Existen diferentes modelos y estándares de calidad que se han desarrollado a lo largo del tiempo. Tenemos modelos o estándares que permiten medir la gestión de la calidad, las medidas de calidad, los requerimientos de calidad y la evaluación de la calidad. Por ejemplo, el modelo ISO 25000 (conocido con el nombre de SQuaRE -Software Quality Requirements and Evaluation-) o el estándar ISO/IEC 9126, entre otros. Los modelos y estándares brindan la posibilidad de transformar la calidad en algo concreto, tangible, objetivo que se puede definir, medir y mejorar. Dan pautas para implementar programas de mejoras en búsqueda de la calidad, identificando la DT en los sistemas para ser disminuida [45].

En esta investigación, para la identificación de la DT en los sistemas, nos apoyaremos de las métricas de predicción que dividiremos en *métricas estructurales y métricas históricas* de código. Estás se han utilizado ampliamente en investigaciones de la ingeniería del software [3], [16], [19], [20].

#### 2.1.3.1 Métricas estructurales

Las métricas estructurales son obtenidas a través de la recopilación de mediciones hechas de representaciones del sistema, como el diseño, el código fuente o la documentación. Ejemplos de estas son el tamaño del código y la longitud promedio de los identificadores que se usaron. Este tipo de métricas ayudan a valorar la complejidad, comprensibilidad y mantenibilidad de un sistema de software o de los componentes del sistema [36]. Algunos ejemplos de métricas estructurales:

- ***Fan-in*/*Fan-out***. *Fan-in* (abanico de entrada) es una métrica del número de funciones o métodos que llaman a otra función o método (por ejemplo, X). *Fan-out* (abanico de salida) es el número de funciones a las que llama la función X. Un valor alto para *fan-in* significa que X está estrechamente acoplado con el resto del diseño y que los cambios a X tendrán extensos efectos dominó. Un valor alto de *fan-out* sugiere que la

complejidad global de X puede ser alta debido a la complejidad de la lógica de control necesaria para coordinar los componentes llamados.

- **Longitud de código**. Ésta es una métrica del tamaño de un programa. Por lo general, cuanto más grande sea el tamaño del código de un componente, más probable será que el componente sea complejo y proclive a errores.
- **Complejidad ciclomática**. Ésta es una métrica de la complejidad del control de un programa. Tal complejidad del control puede relacionarse con la comprensibilidad del programa.
- **Longitud de identificadores**. Ésta es una métrica de la longitud promedio de los identificadores (nombres para variables, clases, métodos, etcétera) en un programa. Cuanto más largos sean los identificadores, es más probable que sean significativos y, por ende, más comprensible será el programa.
- **Profundidad de anidado condicional.** Ésta es una métrica de la profundidad de anidado de los enunciados *if* en un programa. Los enunciados *if* profundamente anidados son difíciles de entender y potencialmente proclives a errores.
- **Métricas específicas orientadas a objetos**. Unas de las más utilizadas son la suite de Chidamber y Kemerer (en ocasiones llamada suite C&K) de seis métricas orientadas a objetos (OO). Aunque se propusieron originalmente a principios de la década de 1990, aún son las métricas OO de más amplio uso [14], [22]. Algunas herramientas de diseño UML recopilan automáticamente valores para dichas métricas conforme se crean los diagramas UML.

#### 2.1.3.2 Métricas históricas

Las métricas históricas son obtenidas a través de las bitácoras de los sistemas de control de versiones (SCV, tales como Git) que guardan un histórico de los cambios que se hacen en los archivos asociados a los módulos de código. Una de ellas considera, adicionalmente, información de los sistemas de control de incidentes (SCI, tales como Jira) [23].

De las bitácoras se obtiene información muy valiosa. Del SCV, por cada *commit* (una nueva versión del código sistema), se puede obtener el *Identificador del commit*, los archivos que fueron modificados, las fechas de modificación, quiénes y porqué realizaron las

modificaciones, entre otras. Como lo podemos observar en la Fig. 2, la cual es un ejemplo de un *commit* en Git.

```
commit db1aece429e0fb46054c732546362a83930ac52c
Author: Tilman Hausherr <tilman@apache.org>
Date:   2019-11-17 14:25:16 +0000

    PDFBOX-4071: SonarQube fix: remove exception that won't be thrown

    git-svn-id: https://svn.apache.org/repos/asf/pdfbox/trunk@1869946
13f79535-47bb-0310-9956-ffa450edef68

1       2
examples/src/main/java/org/apache/pdfbox/examples/signature/CreateEmbeddedTimeStamp.java
1       1
examples/src/main/java/org/apache/pdfbox/examples/signature/CreateSignedTimeStamp.java
1       2       examples/src/main/java/org/apache/pdfbox/examples/signature/SigUtils.java
1       1
```

**Fig. 2 Ejemplo del registro de un *commit* en el SCV como Git.**

En el SCI cada incidente tiene un identificador único, además de información sobre el tipo de incidente, como qué lo generó (datos, usuario, sistema), fecha de registro del incidente, fecha de cierre del incidente, categoría, entre otros. Si para cada *commit* se le pudiera relacionar el identificador de incidente, si es que está relacionado con uno, se podría centrar el análisis de los archivos que se están siendo modificados en este tipo de *commits* y que son de nuestro interés. Cabe mencionar que la dificultad de obtener esta relación entre el *commit* y la incidencia es porque no todos los desarrolladores tienen la disciplina de incluir en el comentario del *commit* el ID del incidente ya que, en la mayoría de los SCV, como por ejemplo Git, no piden esta información al hacer *commit*. En la Fig. 3 se ejemplifica el registro de un *bug* y del que se obtiene el Identificador del incidente, en este caso identificador de *bug* (PDFBOX-4071).

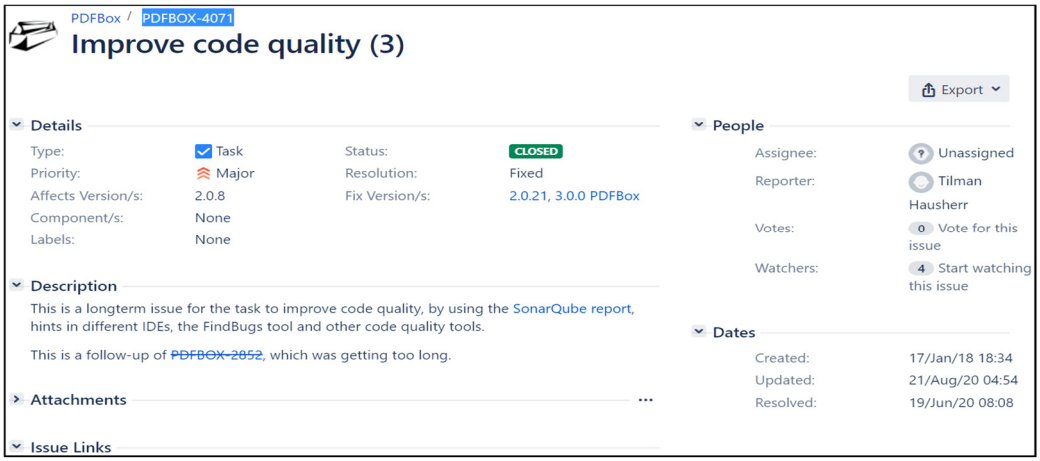



**Fig. 3 Ejemplo del registro de un incidente en el SCI como Jira[4].**

[4] Enlace del incidente registrado en Jira (SCI): https://issues.apache.org/jira/browse/PDFBOX-4071

Si se relaciona de manera adecuada el Identificador del *bug* al Identificador del *commit* entonces estaremos en posibilidades de obtener un *bug commit*, es decir, tener identificados los archivos que estuvieron relacionados a un error. En la Fig. 2 se ejemplifica cómo se relaciona el Identificador del *bug* (PDFBOX-4071) en el comentario del *commit*. Si lo anterior es posible medirlo en el tiempo, podríamos obtener una métrica de cada archivo que llamaremos Número de *Bug Commit* (NBC) en los que un archivo ha participado. El único impedimento para que podamos contar con está métrica, lo concluyen en [28], haría falta considerar no solo las bitácoras de los SCV y SCI, sino también las listas de correos electrónicos y chats donde se decide en muchas ocasiones la forma en la que se registrará y liberará la corrección de un error.

A continuación, se describen algunos ejemplos de métricas históricas [23], [17]:

- ***Co changes partners*.** Representa la cantidad de otros archivos con los que ha cambiado el archivo destino.
- ***Crossing***: Archivo con un alto abanico de entrada y salida que cambia a menudo con sus archivos relacionados (llama o lo llaman).
- ***Modularity violation***. Archivos que cambian juntos con frecuencia, pero que no tienen una relación estructural.
- ***Unstable Interface***. Archivo con muchos dependientes que cambia a menudo con todos ellos.
- **Número de *Bug commits***. Número de veces en las que un archivo ha participado en un *bug commit*.

#### 2.1.3.3 Proceso para obtener las métricas

En la actualidad existen en el mercado varias herramientas de análisis de código (HAC) de las aplicaciones. Estas ofrecen una serie de métricas de cada uno de los archivos que componen la aplicación analizada. Cada métrica es calculada bajo las reglas de cada HAC. En muchos de los resultados de las diferentes herramientas no coinciden entre ellas [23], esto debido a que no existe una unificación de las reglas que aplican las HAC para calcular cada métrica. En la Fig. 4 se ejemplifican las entradas (código fuente y compilado, listas de *commits* y listas de errores) para la HAC, las etapas aplicadas por las HAC y las salidas (métricas) de los sistemas

analizados. Las etapas principales que se realizan en las HAC para la medición e identificación de métricas son [36]:

**Elegir mediciones a realizar**. Esta etapa va directamente relacionada a las mediciones que pueda ofrecer la HAC que elegimos, las métricas que se pueden derivar de dichas mediciones y que nos interesan. De ahí que, elegiremos dicha HAC.

**Seleccionar componentes a valorar**. Algunas HAC ofrecen diferentes niveles de componentes a valorar. Por ejemplo, a nivel método, clase, módulo, archivo, entre otros. Será parte de la elección de la HAC, la que ofrezca el nivel de valoración de componentes que nos interese.

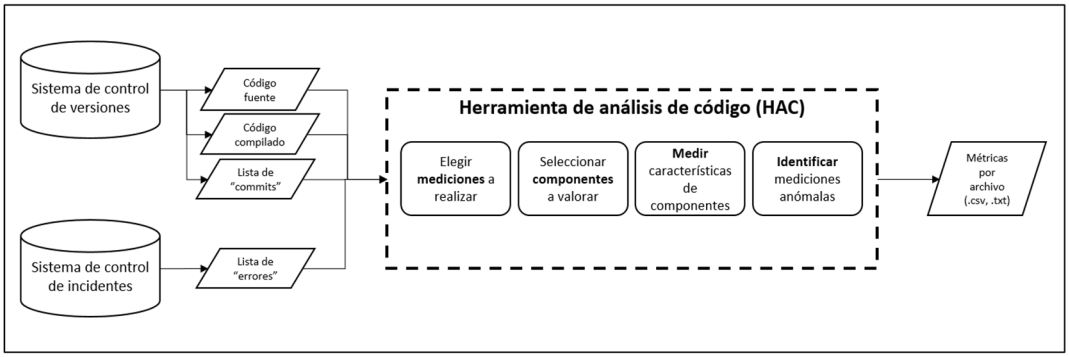



**Fig. 4 Diagrama general del funcionamiento de las herramientas de análisis de código.**

**Medir características de componentes**. Se miden los componentes seleccionados y se calculan los valores de métricas asociados.

**Identificar mediciones anómalas.** Después de hacer las mediciones de componentes, se hacen distintos tipos de análisis y se genera un reporte. Hay que observar los valores inusualmente altos o bajos para cada métrica, pues éstos sugieren que podría haber problemas con el componente que muestra dichos valores. Una vez que se obtuvieron los componentes con valores anómalos para sus métricas seleccionadas, deben examinarse para decidir si dichos valores de métrica anómalos significan que hay problemas con el componente.

### 2.1.4 Relación de deuda técnica con los atributos de calidad del software

La deuda técnica (DT) que se genera en los sistemas a través del tiempo por decisiones convenientes a corto plazo irremediablemente, impactarán en los atributos de calidad del software. La afectación puede reflejarse en cualquiera de los AC, pero el de mayor impacto es el de *mantenibilidad* [19], [20]. Lo anterior se debe a que, para reparar cualquiera de los otros AC afectados por la DT se deberá de cambiar código (probablemente complejo), probar el código modificado y validar su integración con el resto del sistema. Todas estás actividades adicionales representan tiempo y costos que se deben de invertir para aplicar las medidas correctivas para disminuir o eliminar la DT.

Pero no solo la mantenibilidad se puede ver afectada por la DT, puede afectar a otros AC como la comprensibilidad o facilidad de evolución, por lo que, es de suma importancia irla disminuyendo hasta tratar de extinguirla o reducirla.

## 2.2 Aprendizaje automático

La inteligencia artificial (IA) tiene tanto tiempo como la segunda guerra mundial. El campo de la IA intenta comprender y construir entidades inteligentes [35]. Actualmente, la IA abarca una gran variedad de subcampos, que van desde lo general (aprendizaje y percepción) hasta lo específico, como jugar al ajedrez, demostrar teoremas matemáticos, escribir poesía, conducir un automóvil en una calle concurrida y diagnosticar enfermedades. La IA es relevante para cualquier tarea intelectual; es verdaderamente un campo universal.

El aprendizaje automático (AA) es un área de la inteligencia artificial, que construye un modelo matemático basado en datos de muestra, conocidos como "datos de entrenamiento", para hacer predicciones o decisiones sin estar programado explícitamente para realizar la tarea [11], [43].

En cuanto al aprendizaje, según [43]

> "Se dice que un programa de computadora aprende de la experiencia E con respecto a alguna clase de tareas T y mide el desempeño P si su desempeño en las tareas en T, medido por P, mejora con la experiencia E."

El aprendizaje automático tiene como objetivo establecer un regresor o clasificador a través del aprendizaje del conjunto de datos de entrenamiento y luego evaluar el rendimiento del regresor o clasificador a través del conjunto de datos de prueba.

## 2.2.1 Divisiones del aprendizaje automático

En el aprendizaje automático, generalmente proporcionamos a los algoritmos un conjunto de datos de entrenamiento y un conjunto de datos de prueba. Por conjunto de datos de entrenamiento, se entenderá a la unión del conjunto de datos etiquetados y del conjunto de datos no etiquetados disponibles. El conjunto de datos de prueba consta de ejemplos nunca vistos y que el algoritmo intentará etiquetar.

Según la naturaleza de los datos de entrenamiento, podemos dividir el aprendizaje automático de la siguiente manera [43].

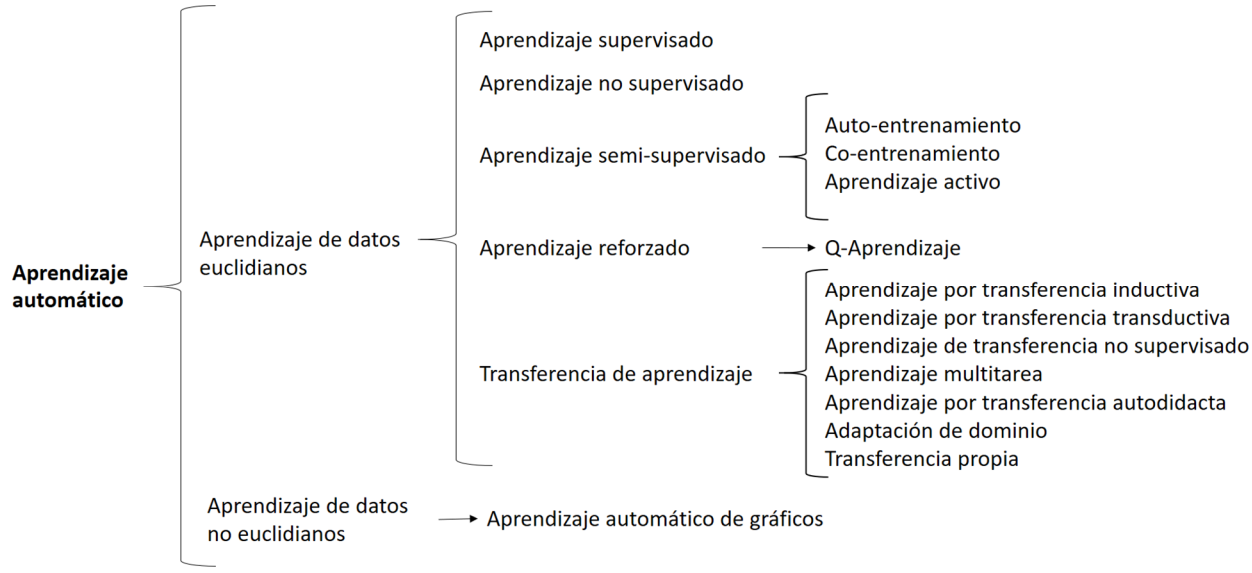



**Fig. 5 Árbol de aprendizaje automático.**

1. Aprendizaje de datos estructurados regular o euclidiano [43].

- **Aprendizaje supervisado:** dado un conjunto de entrenamiento que consta de datos etiquetados (es decir, entradas de ejemplo y sus salidas deseadas) $(X_{entrena}, Y_{entrena}) = \{(X_1, Y_1), \ldots, (X_l, Y_l)\}$, el aprendizaje supervisado aprende una regla general que relaciona las entradas con las salidas. Esto es como un "maestro" o un supervisor (experto en etiquetado de datos) que le da a un estudiante un problema (encontrar la relación de mapeo entre entradas y salidas) y sus soluciones (datos de salida etiquetados) y le dice a ese estudiante que descubra cómo resolver otros problemas similares: encontrar el mapeo de las características de muestras invisibles a sus etiquetas correctas o valores objetivo en el futuro [43].
- **Aprendizaje no supervisado**: en el aprendizaje no supervisado, el conjunto de entrenamiento consta únicamente del conjunto no etiquetado $X_{entrena} = X_u = \{X_1, \ldots, X_u\}$. La tarea principal del aprendiz automático es encontrar las soluciones por sí mismo (es decir, patrones, estructuras o conocimiento en datos no etiquetados) [43].
- **Aprendizaje semisupervisado**: dado un conjunto de entrenamiento $(X_{entrena}, Y_{entrena}) = \{(X_1, Y_1), \ldots, (X_l, Y_l)\}\, U\, \{X_{l+1}, \ldots, X_{l+u}\}\, con\, l/u$, es decir, se nos proporciona una pequeña cantidad de datos etiquetados junto con una gran cantidad de datos no etiquetados. El aprendizaje semisupervisado se encuentra entre el aprendizaje no supervisado (sin datos de entrenamiento etiquetados) y el aprendizaje supervisado (con datos de entrenamiento completamente etiquetados) [43]. Según cómo se etiqueten los datos, el aprendizaje semisupervisado se puede dividir en las siguientes categorías:
    - El autoaprendizaje es un aprendizaje semisupervisado que utiliza sus propias predicciones para enseñarse a sí mismo.
    - El coentrenamiento es un aprendizaje débilmente semisupervisado para datos de vista múltiple que utiliza la configuración de coentrenamiento y usa sus propias predicciones para aprender a sí mismo.
    - El aprendizaje activo es un aprendizaje semisupervisado en el que el alumno tiene algún papel activo o participativo en la determinación de qué puntos de datos pedirá que un experto o profesor los etiquete.
- **Aprendizaje por refuerzo**: los datos de entrenamiento (en forma de recompensas y castigos) brindan solo como retroalimentación a un agente de inteligencia artificial en un entorno dinámico. Esta retroalimentación entre el sistema de aprendizaje y la

experiencia de interacción es útil para mejorar el desempeño en la tarea que se está aprendiendo. El aprendizaje automático basado en la retroalimentación de datos se denomina aprendizaje por refuerzo. Q-learning es un algoritmo de aprendizaje automático basado en valores y se centra en la optimización de la función de valor según el entorno o el problema. La Q en el Q-learning representa la calidad con la que el modelo encuentra su próxima acción mejorando la calidad [43].

- **Transferencia de aprendizaje**: en muchas aplicaciones del mundo real, la distribución de datos cambia o los datos están desactualizados y, por lo tanto, es necesario aplicar transferencia de aprendizaje para considerar la transferencia de conocimiento desde el dominio de origen al dominio de destino. El aprendizaje de transferencia incluye, entre otros, el aprendizaje de transferencia inductiva, el aprendizaje de transferencia transductiva, el aprendizaje de transferencia no supervisado, el aprendizaje de tareas múltiples, el aprendizaje de transferencia autodidacta, la adaptación de dominio y la transferencia propia [43].

2. El aprendizaje de datos no euclidianos se refiere a la representación de elementos y conceptos más complejos y con mayor precisión. Algunos ejemplos de este tipo de datos pueden ser una molécula, una red, un árbol de datos, un grafo, entre otros. Los grafos son un tipo de estructura de datos que consta de nodos (entidades) que están conectados con bordes (relaciones). Esta estructura de datos abstracta se puede utilizar para modelar casi cualquier cosa. Los grafos nos permiten representar características individuales al tiempo que brindamos información sobre las relaciones y la estructura [43].

### 2.2.2 Métodos de aprendizaje automático

Existen diferentes métodos de aprendizaje automático para modelar los datos del problema subyacente. Estos métodos de aprendizaje automático se pueden dividir en los siguientes tipos:

1. **Métodos de aprendizaje automático basados en estructura de red.**

- **Redes neuronales artificiales** (ANN por sus siglas en inglés): las ANN se inspiran en el cerebro y se componen de neuronas artificiales interconectadas capaces de realizar ciertos cálculos en sus entradas [26]. Los datos de entrada activan las neuronas en la primera capa de la red cuya salida es la entrada a la segunda capa de neuronas en la red. De manera similar, cada capa pasa su salida a la siguiente capa y la última capa genera el resultado. Las capas entre las capas de entrada y salida se denominan capas ocultas. Cuando se utiliza una ANN como clasificador, la capa de salida genera la categoría de clasificación final [43].

- **Red bayesiana:** Una red bayesiana es un modelo gráfico probabilístico que representa las variables y las relaciones entre ellas [26]. La red se construye con nodos como las variables aleatorias discretas o continuas y aristas dirigidas como las relaciones entre ellas, estableciendo un grafo acíclico dirigido. Cada nodo mantiene los estados de la variable aleatoria y la forma de probabilidad condicional. Las redes bayesianas se construyen utilizando conocimiento experto o algoritmos eficientes que realizan inferencias [43].

2. **Métodos de aprendizaje automático basados en análisis estadístico**.

- **Reglas de Asociación**: El objetivo de la minería de reglas de asociación es descubrir reglas de asociación previamente desconocidas a partir de los datos. Una regla de asociación describe una relación X ⇒ Y entre un conjunto de elementos X y un solo elemento Y. Las reglas de asociación tienen dos métricas que indican con qué frecuencia ocurre una relación determinada en los datos: el *soporte* es una indicación de la frecuencia con la que aparece el conjunto de elementos en los datos establecidos, y la *confianza* es una indicación de la frecuencia con la que se ha encontrado que la regla es cierta [43].

- **Agrupamiento**: el agrupamiento es un conjunto de técnicas para encontrar patrones en datos no etiquetados de alta dimensión. Es un enfoque de descubrimiento de patrones no supervisado en el que los datos se agrupan en función de alguna medida de similitud [43].

- **Aprendizaje por conjuntos**: los algoritmos de aprendizaje supervisado, en general, buscan en el espacio de hipótesis, para determinar la hipótesis correcta que hará buenas predicciones acerca de un problema dado. Aunque es posible que existan buenas hipótesis, puede ser difícil encontrar una. Los métodos de conjunto combinan múltiples algoritmos de aprendizaje para obtener un mejor rendimiento predictivo que los algoritmos de aprendizaje constituyentes solos. A menudo, los métodos de conjunto utilizan múltiples alumnos débiles para construir un alumno fuerte [43].

- **Modelos ocultos de Markov** (HMM por sus siglas en inglés): un HMM es un modelo estadístico de Markov en el que se supone que el sistema que se modela es un proceso de Markov con estados no observados (es decir, ocultos). El principal desafío es determinar los parámetros ocultos a partir de los parámetros observables. Los estados de un HMM representan condiciones no observables que se están modelando. Al tener diferentes distribuciones de probabilidad de salida en cada estado y permitir que el sistema cambie de estado con el tiempo, el modelo es capaz de representar secuencias no estacionarias [43].

- **Aprendizaje inductivo**: la deducción y la inducción son dos técnicas básicas para inferir información a partir de datos. El aprendizaje inductivo es el aprendizaje supervisado tradicional y tiene como objetivo aprender un modelo a partir de ejemplos etiquetados y tratar de predecir las etiquetas de ejemplos que no hemos visto o que no conocemos. En el aprendizaje inductivo, a partir de observaciones específicas se comienzan a detectar patrones y regularidades, se formulan algunas hipótesis tentativas para explorar y finalmente se termina desarrollando algunas conclusiones generales o teorías. Varios algoritmos de aprendizaje automático (AA) son inductivos, pero por aprendizaje inductivo generalmente nos referimos a “poda incremental repetida para producir reducción de errores” y el algoritmo cuasi-óptimo (AQ por sus siglas en inglés) [43].

- **Bayes ingenuo**: Es bien conocido que un clasificador bayesiano sorprendentemente simple con fuertes supuestos de independencia entre características, llamado Bayes ingenuo, es competitivo con clasificadores de última generación como C4.5. En general, se supone que las características de entrada son independientes. Los clasificadores Bayes ingenuo pueden manejar un número arbitrario de características independientes, ya sean continuas o categóricas, al reducir una tarea de estimación de densidad de alta

dimensión a una estimación de densidad de kernel unidimensional, bajo el supuesto de que las características son independientes. Aunque el clasificador Bayes ingenuo tiene algunas limitaciones, es un clasificador óptimo, si las características son condicionalmente independientes dada la verdadera clase. Una de las mayores ventajas del clasificador Bayes ingenuo es que es un algoritmo en línea y su entrenamiento se puede completar en tiempo lineal [43].

3. **Métodos de aprendizaje automático basados en evolución.**

- **Computación evolutiva:** en informática, la computación evolutiva es una familia de algoritmos para la optimización global inspirados en la evolución biológica. El término computación evolutiva abarca algoritmos genéticos, programación genética, estrategias de evolución, optimización de enjambres de partículas, optimización de colonias de hormigas y sistemas inmunológicos artificiales [43].

### 2.2.3 Rendimiento de un algoritmo de aprendizaje automático

A continuación, se describen métricas para medir el rendimiento de un algoritmo de aprendizaje automático [43].

- **Escalabilidad**: este parámetro se puede definir como la capacidad de un algoritmo para manejar un aumento en su escala, como alimentar más datos al sistema, agregar más funciones a los datos de entrada o agregar más capas en una red neuronal. Sin que aumente ilimitadamente su complejidad.
- **Tiempo de entrenamiento**: esta es la cantidad de tiempo que tarda un algoritmo de aprendizaje automático en estar completamente entrenado y formar la capacidad de hacer sus predicciones.
- **Tiempo de respuesta**: esta métrica está relacionada con la agilidad de un sistema de aprendizaje automático y representa el tiempo que tarda un algoritmo para hacer una predicción después de haber sido entrenado.
- **Datos de entrenamiento**: esta métrica del algoritmo de aprendizaje automático es la cantidad y el tipo de datos de entrenamiento que necesita un algoritmo. Los algoritmos

compatibles con más datos de entrenamiento suelen tener una mayor precisión, pero también tardan más en entrenarse.

- **Complejidad**: La complejidad de un algoritmo se puede definir como la cantidad de operaciones matemáticas que se realizan para lograr una solución deseada.
- **Precisión**: En un algoritmo, la precisión se determina por la capacidad de este para pronosticar los casos positivos reales. Por ejemplo, si el algoritmo va a predecir si un conjunto de pacientes es propenso a diabetes, la precisión se medirá en función de los casos que el algoritmo indique como reales contra los casos reales.
- **Tiempo de Convergencia**: Esta métrica de un algoritmo, a diferencia del tiempo de respuesta, se relaciona con qué tan rápido acepta que la solución encontrada para ese problema en particular es la solución óptima en ese momento.
- **Confiabilidad de convergencia**: este parámetro representa la susceptibilidad de algún algoritmo a quedarse atascado en los mínimos locales y cómo las condiciones iniciales pueden afectar su desempeño.

## 2.3 Dificultad de usar la métrica número de *bug commits* para identificar *hotspots*

DV8 [33] es una de las HAC que permiten obtener la métrica NBC, la cual se definió en sección 2.1.3.2. Un valor alto en la métrica NBC significa que podríamos estar frente a un archivo que se está manifestando constantemente en errores del sistema y que requiere de atención, por lo que nos podría sugerir un *hotspot*.

Una característica que tiene la métrica NBC es que requiere de dos cosas, una es la disciplina de los desarrolladores para realizar la asociación del *bug* con el *commit* realizado. La otra es que, se requiere de tiempo para que NBC tenga un valor representativo de los *bug commits* y aporte en la detección de *hotspots* [23]. Lo anterior se debe a que no se puede marcar una tendencia con los primeros *bug commits* de que un archivo sea *hotspot*.

Si estuviéramos ante un sistema cuyo desarrollo ha comenzado recientemente y de cada archivo obtuviéramos la métrica NBC, ésta valdría cero para todos ellos, ya que no se ha presentado ningún *bug commit*. Ahora, si ya hubiera habido algunos *bugs* y estos si fueron

asociados a los commits, los archivos incluidos en el *commit* serían los primeros *hotspots* que tendría el sistema.

Si desde que se comienza a utilizar un sistema el valor de NBC se pudiera calcular y fuera confiable, estaríamos ante el *ground truth* (verdad fundamental -se refiere a la precisión de la clasificación del conjunto de entrenamiento para las técnicas de aprendizaje supervisado-) para la detección de los *hotspots*. Con lo anterior, podríamos tomarla al NBC como la métrica que nos ayudará en la detección de la DT en los sistemas. Sin embargo, por la falta de información que aún hay en ella y que requiere de tiempo y disciplina de los desarrolladores se deben buscar otras métricas que nos permitan encontrar los *hotspots* en los sistemas [23].

Dada la dificultad de disponer de la métrica NBC, en este trabajo se busca encontrar, mediante técnicas de aprendizaje automático, otras métricas que puedan ser obtenidas de forma temprana para ayudarnos a identificar los posibles *hotspots*.

# Capítulo 3

# Estado del arte

A continuación, se presenta un resumen del estudio sistemático de la literatura que se aplicó, desde el planteamiento de las preguntas de investigación, la revisión de los estudios existentes sobre el tema de investigación que hasta el momento de escribir este documento están publicados y hacer la propuesta de solución al problema de investigación.

## 3.1 Revisión de la literatura

Con el propósito de entender el problema de la detección de *hotspots* en los sistemas, conocer los trabajos de investigación existentes sobre el tema e identificar aspectos importantes a considerar en la propuesta de investigación, se realizó una revisión sistemática de la literatura.

La revisión sistemática de la literatura consistió en definir preguntas de investigación, establecer los motores de búsqueda de artículos (IEEE Xplore, Scopus, ACM Digital Library, Science Research, Google Scholar), definir palabras claves o sinónimos a buscar (“software development”, “hotspots”, “algoritmos”, “detección”, “heurística”, “optimización”, entre otras), clasificar los artículos encontrados. El resultado de este análisis arrojó una lista de cuarenta y cinco (45) artículos relacionados a las preguntas de investigación, palabras claves y sinónimos. Posteriormente se clasificaron en los que están relacionados con el tema, quedando treinta y cuatro (34). Al final se seleccionaron trece artículos que tratan el tema similar al de nuestra investigación.

Cabe señalar que existen trabajos donde solo hacen uso de métricas estructurales y algunos otros que combinan métricas estructurales con métricas históricas. Por otro lado, se encontraron trabajos con el uso de clasificadores de aprendizaje automático supervisados y no supervisados y muy pocos trabajos con el uso de algoritmos evolutivos. La mayoría de los trabajos revisados están dirigidos a aplicarse en sistemas desarrollados con lenguajes de programación OO, debido a que desde hace más de 3 décadas estos lenguajes han sido la punta de lanza de la mayoría de los sistemas desarrollados.

Dentro del resumen de los diferentes artículos revisados se mencionan una variedad de métricas que son propuestas en cada estudio por lo que se concentraron en la Tabla 1 para referenciarlas más adelantes.

**Tabla 1 Resumen de métricas de los diferentes trabajos revisados.**

| Id Métrica | Descripción |
|---|---|
| ACDF[9] | Aggregated change density normalized by frequency of changes |
| ATAF[9] | Aggregated change size Normalized by frequency of change |
| BOC[9] | Birth of a class |
| CAM | Cohesion Among Methods |
| CBO[14] | Coupling between Object Classes |
| CHD[9] | Change density |
| CHO[9] | Change occurred |
| CSB[9] | Changes since the birth |
| CSBS[9] | Changes since the birth normalized by size |
| DIT[14] | Depth of Inheritance Tree |
| FCH[9] | First time changes |
| FRCH[9] | Frequency of changes |
| LCA[9] | Last change amount |
| LCD[9] | Last change density |
| LCH[9] | Last time changes |
| LCOM[14] | Lack of Cohesion Methods |

| Id Métrica | Descripción |
|---|---|
| LOC | Lines of Code |
| NOC[14] | Number of Children |
| NOF | Number of Attributes |
| NOM | Number of Methods |
| NORM | Number of Overridden Methods |
| NSF[14] | Number of Static Attributes |
| NSM | Number of Static Methods |
| POM[24] | Probabilidad de modificación por clase |
| RFC[14] | Response for a Class |
| SIX | Specialization Index |
| SLOC | Source lines of code |
| WCD[9] | Weighted change density |
| WCH[9] | Weighted changes |
| WFR[9] | Weighted frequency of changes |
| WMC[14] | Weighted Methods per Class |

A continuación, se presenta un resumen de cada uno de los artículos revisados, una tabla comparativa entre ellos y un análisis de la comparativa, con lo cual se presenta el estado del arte, conclusiones y las preguntas de investigación que se plantean.

## A Novel UML based approach for early detection of change prone classes

Bura *et al*. [15] describen cómo la predicción de clases propensas a cambios es esencial para el desarrollo eficaz de software. La evaluación de los cambios de una versión de software a la siguiente puede mejorar la calidad del software. Utilizan métricas orientadas a objetos (MOO por sus siglas en inglés) de Chidamber y Kemerer (C&K) [14] internas del software y técnicas de abstracción que hacen única a una clase, de tal forma que indican que tan bien están definidas las clases y su impacto en la mantenibilidad del software. Junto con las MOO proponen otras métricas y utilizan diagrama de secuencia y de clases de UML 2.0 para obtener métricas de dependencias de clases y del código obtienen métricas de duración de ejecución, de tiempo de ejecución, de regularidad y popularidad. Estas métricas (MOO y propuestas) son utilizadas como entradas en una primera etapa a un algoritmo de Colonia de Abejas (ACB por sus siglas en inglés) [14] para refinar las mismas y posteriormente hacen uso del algoritmo de aprendizaje automático ID3, con lo que obtienen su resultado. Su algoritmo fue probado con

dos aplicaciones de código abierto Open Clinic y Open Hospital[5] y dos versiones de cada una de ellas.

Los autores concluyen que su trabajo de investigación construye un método de predicción de cambio óptimo utilizando la extracción frecuente de conjuntos de elementos. Adicionalmente, mencionan que a mayor valor de las métricas propuestas en las clases mayor la propensión al cambio [15].

### Predicting code hotspots in open-source software from object-oriented metrics using machine learning

Hilton *et al.* [24] presentan un método para extraer bases de datos de código abierto disponibles en Java para entrenar a un clasificador de Random Forest a fin de predecir que archivos probablemente sean *hotspots*. Se basan en métricas internas obtenidas del IDE de Eclipse y proponen una métrica externa por clase, llamada probabilidad de modificación (POM por sus siglas en inglés) para tipificar cada archivo en un *hotspot* o no *hotspot* y validar los resultados del modelo de clasificación. Los autores propusieron tomar el valor medio como el umbral para tipificar las clases en *hotspots* para aquellas que estuvieran por arriba del valor medio y como no *hotspot* las clases que estuvieran por debajo del valor medio. El POM por clase, es un cociente entre las veces que se ha modificado una clase entre el total de versiones que se han liberado del sistema. Las clases con valores altos en POM son clases con alta probabilidad de cambio. Las versiones previas de los sistemas sirvieron para entrenar al modelo, al momento de probar con nuevos sistemas, de estos solo se requiere la versión actual. Los autores utilizaron el modelo de clasificación de bosques aleatorios contenido en las clases de la biblioteca de aprendizaje automático de scikit-learn para Python. Se utilizó el 33% de los archivos para prueba, el 66% de los archivos para entrenamiento y se definió un umbral de POM para determinar si un archivo es *hotspot* o no. El modelo pudo predecir con un 75% de exactitud al conjunto de prueba que se reservó. El modelo se probó con trece aplicaciones de código abierto con un total de 7108 archivos a clasificar y con 789,750 líneas de código.

El modelo fue entrenado por cada una de las métricas para ver cuáles de ellas podrían predecir de manera individual los *hotspots* en cada archivo. La Tabla 2 muestra cada una de

[5]"The Complete Open-Source and Business Software Platform", Sourceforge, 2021. [Online]. Available: https://sourceforge.net/. [Accessed: 12-Oct-2021]

las métricas y sus resultados referente a esto, en donde observamos que la mejor métrica es la de métodos ponderados por clase (WMC por sus siglas en inglés).

**Tabla 2 Evaluación del modelo de Hilton et al. con medidas de rendimiento.**

| Metric | True low | False low | True high | False High | Accuracy |
|---|---|---|---|---|---|
| NSF | 399 | 244 | 287 | 125 | 65.024 |
| NSM | 450 | 277 | 254 | 74 | 66.73 |
| NOF | 407 | 361 | 170 | 117 | 54.692 |
| NOM | 410 | 270 | 261 | 114 | 63.602 |
| NORM | 440 | 309 | 222 | 84 | 62.749 |
| $WMC_{[14]}$ | 424 | 217 | 314 | 100 | 69.953 |
| $DIT_{[14]}$ | 343 | 267 | 264 | 181 | 57.536 |
| SIX | 444 | 313 | 218 | 80 | 62.749 |
| $LCOM_{[14]}$ | 452 | 399 | 132 | 72 | 55.355 |

De esto concluyen que hay una alta relación entre métricas internas y métricas externas, ya que en conjunto se llegó a una mejor predicción. Por otro lado, los autores ven que el modelo puede ayudar en las etapas de desarrollo del software para someter a mayores pruebas a los archivos que sean más propensos a ser *hotspots* y con ellos poder disminuir la DT [24].

### A new hybrid model for predicting change prone class

Godara *et al.* [5] proponen un modelo híbrido que combina características como la dependencia de comportamiento generada a partir de diagramas UML, el tiempo de ejecución y los eventos de seguimiento generados a partir del código fuente para predecir las clases propensas a cambios. Los autores utilizaron métricas de ejecución y de frecuencia de uso de los métodos de cada clase, obtenidas del código. Por otro lado, obtuvieron métricas adicionales basándose en los diagramas de clases y secuencia de UML, con lo que les permite definir una métrica llamada Dependencia Conductual, de tal forma que si una clase cambia otras clases pueden cambiar. El modelo utiliza el algoritmo de aprendizaje automático de árboles de decisión ID3. Las pruebas del modelo fueron realizadas con una sola aplicación de código abierto (Hospital Management) con un número grande de clases. Mencionan que fue un buen modelo de predicción [5].

### Automated change-prone class prediction on unlabeled dataset using unsupervised method

Yanb *et al.* [7] se plantearon como objetivo construir un modelo de predicción de clases propensas a cambios en conjuntos de datos sin etiquetar. Los autores presentan tres tipos de modelos que pueden ayudar a predecir los *hotspots* en las aplicaciones:

I. Predicción supervisada dentro del proyecto, que se entrena con datos históricos etiquetados del proyecto y se prueba con datos diferentes del mismo proyecto.

II. Predicción supervisada entre proyectos (diferentes), se entrena con datos etiquetados de un proyecto y se prueba con datos de otros proyectos. En este modelo de predicción sólo pudieron obtener hasta un 30% de precisión, dado que la distribución de las características entre un proyecto y otro pueden ser muy diferentes.

III. Predicción no supervisada que directamente aprende de los datos del proyecto.

Su trabajo se orienta sobre el modelo III. Se basan en métricas orientadas a objetos (MOO) [14] (estructurales) y un algoritmo no supervisado de aprendizaje automático llamado CLAMI y además de compararlo con una variación que hacen del mismo llamado CLAMI+. CLAMI se basa en tres fases: la primera es agrupación y etiquetado, la segunda es selección de métricas e instancias (datos entrenamiento y pruebas) y la tercera es aprendizaje (entrenamiento del algoritmo CLAMI/CLAMI+) y predicción con los datos de prueba [7].

El modelo se probó para 14 aplicaciones de código abierto, mostrando que los métodos no supervisados obtienen buenos resultados comparados con: *LogitBoost* (LB), *Multilayer perceptron* (MLP), *Radial basis function network* (RBF), *Support vector machine* (SVM) y *K-means* (SK). Utilizan 5 MOO [14]: Métodos ponderados por clase (WMC), Profundidad del árbol de herencia (DIT), Número de hijos (NOC), Respuesta para la clase (RFC), Falta de cohesión de los métodos (LCOM). Su modelo (CLAMI) lo compararon con un modelo que usa datos históricos del mismo proyecto (*within a project*) y con un modelo que usa datos históricos de otros proyectos (*across project*), concluyendo los autores que los métodos no supervisados dan resultados similares o mejoran a los supervisados [7].

### Identifying error proneness in path strata with genetic algorithms

Birt *et al.* [17] desde una perspectiva de detectar las clases propensas a error, mencionan los autores que la cobertura de las pruebas es un problema ya que alcanzar el 100% de las posibilidades es casi imposible ya que se vuelve exponencial el tiempo que se requiere y eso impacta en la rentabilidad de cualquier proyecto. Por esta razón es importante identificar qué partes del sistema se deben de probar con una buena carga de casos de prueba y cuáles no. De ahí que los autores definen el SoE (source of error) por cada clase. El valor SoE es la suma de todos los valores SoE que se asignan a cada sentencia de lenguaje: un *if*, *repeat*, *for* o un *while*, entre otras, atribuyendo pesos a las posibles fuentes de error. Se aplican algoritmos genéticos para determinar las clases propensas al error.

Los autores calculan una probabilidad de tener un error en cada clase basado en su SoE y la probabilidad de error de todo el sistema es la suma de todas las probabilidades de cada clase. Al tener las rutas del software como una matriz ponderada de SoE se les puede aplicar un algoritmo de búsqueda para encontrar las clases con mayor propensión al error.

En este estudio se utilizaron 10 aplicaciones de código abierto en C, a las cuales se les sembraron 40 errores por cada mil líneas de código para determinar qué tan bueno es el proceso para determinar clases propensas al error. Se ejecutaron 50 veces, variando la semilla de propagación del error para simular la variación en la distribución de los errores. Se intenta llegar al 80% de selección de clases con error. El algoritmo genético se compara con la selección de rutas aleatorias.

Los autores concluyen que, tener SoE similares hace que sea indiferente la selección que hace el algoritmo. Las partes más propensas al error son las partes más anidadas en el código (*if*, *while*, *for*, entre otros) y SoE no es la única métrica, pero tiene su mérito el aporte [17].

### An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes

Eski *et al.* [6] mencionan que el costo de mantenimiento del software suele ser más del 50% del costo del ciclo de vida total del software. Las pruebas iniciales del software juegan un papel

importante para reducir el costo de mantenimiento en el futuro. Se requiere ubicar las partes críticas del software para dirigir las pruebas de manera más adecuada.

Su método se basa en ocho métricas de diseño orientadas a objetos de C&K [14] y QMOOD (quality model for object oriented design) para determinar las partes críticas y propensas a cambios del software. Analizan las modificaciones en el software a lo largo de los cambios históricos de los proyectos de código abierto (SCV) para descubrir la relación entre las métricas del diseño orientado a objetos y los cambios en el software. Con su enfoque muestran que puede encontrar hasta el 80% de las partes (clases) propensas a cambios.

Los autores definieron un valor de "costo de cambio" (CC) para las clases, que se calcula de acuerdo con las diferencias entre dos versiones de la clase. En este cálculo, para obtener el posible efecto del cambio, las modificaciones se ponderaron de acuerdo con sus impactos arquitectónicos en el diseño. Por ejemplo, los cambios no estructurales como las modificaciones de comentarios no se tienen en cuenta, mientras que el cambio en la interfaz de un método tiene un mayor peso. Para evaluar su enfoque, los autores utilizaron tres aplicaciones de código abierto con sus respectivos historiales en el SCV de cada una de ellas, para mostrar la relación entre métricas estructurales y cambios en el software. Las ocho métricas utilizadas son las siguientes: CBO, DIT, LCOM, NOC, RFC, WMC, Número de atributos estáticos (NSF), número de líneas de código (LOC) y Cohesión entre métodos (CAM) (ver Tabla 1 o Apéndice).

Los autores concluyen que la combinación de métricas de WMC, LOC y RFC funciona para los sistemas de software analizados. También concluyen que, cada proyecto tiene su propia naturaleza y el historial de cambios puede ayudarnos a comprender esta naturaleza [6].

## Evaluation of software degradation and forecasting future development needs in software evolution

Maisikeli [8] observó en esta investigación que los componentes del software (clases, métodos y/o paquetes) que han sufrido cambios o modificaciones durante la evolución tienden a generar valores de entropía (medida de la degradación del software) más altos que aquellos con poco o ningún cambio; lo cual nos indica que las clases o métodos invocados con mayor frecuencia en el sistema son los que tienen las mayores posibilidades de ser cambiados o modificados.

En esta investigación, los autores seleccionaron como conjunto de datos para pruebas la aplicación JHotDraw (JHD por sus siglas en inglés), basada en Java de código abierto ampliamente utilizada y ampliamente probada, desarrollada con las mejores prácticas de ingeniería de software. Se tomaron seis versiones de este software y se recopilaron datos dinámicamente, de los cuales se utilizaron cuatro métricas: entropía, índice de madurez del software (IMS), esfuerzo COCOMO (EC) y periodo de duración para nuevas versiones (PDNV). Con ellas, se analizó la degradación y el nivel de madurez del software. Los resultados obtenidos se utilizaron como entrada para el análisis de series de tiempo y predecir el esfuerzo y el período de duración que pueden ser necesarios para el desarrollo de futuras versiones.

El autor concluye que cuando un sistema de software evoluciona y pasa de una versión a otra, se espera que la nueva versión supere a la anterior y que la nueva versión sea mejor estructuralmente y contenga menos defectos; sin embargo, este puede no ser el caso, ya que pueden introducirse nuevas funcionalidades no deseadas, la estructura puede degradarse y puede introducirse una medida de degradación y desorden. También el autor observó que los valores de entropía disminuyen constantemente a medida que el sistema de software evoluciona de una versión a otra, lo que indica que el sistema de software se estaba moviendo hacia su nivel de madurez óptimo [8].

### A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software

Elish *et al.* [9] establecen como objetivo de su investigación el derivar y validar un conjunto de métricas basadas en la evolución como indicadores potenciales de las clases propensas a cambios de un sistema orientado a objetos al pasar de una versión a la siguiente. Las métricas que utilizan son: MOO [14] y por otro lado métricas de GQM (métricas de pregunta objetivo) que se pueden ver como basadas en la evolución y que al combinarlas esperan que den mejores resultados. Utilizaron dos aplicaciones de código abierto (VSSPLUGIN y PeerSim) con trece y nueve versiones del software respectivamente.

Los modelos de predicción para la propensión al cambio de clase se construyeron de cuatro formas diferentes: usando las métricas basadas en la evolución y MOO [14] como variables independientes, usando sólo las métricas basadas en la evolución como variables independientes, usando sólo las métricas de MOO [14] como variables independientes o

usando sólo la métrica de probabilidad de cambios de clase (PCC) como variable independiente.

El análisis de componentes principales (PCA por sus siglas en inglés) es una técnica estándar para identificar dimensiones ortogonales y derivar componentes principales (PC por sus siglas en inglés), que son combinaciones lineales de las variables independientes estandarizadas. Para PCA se utilizó el método de normalización Varimax con Kaiser[6] como componentes de rotación. Se realizó análisis de correlación bivariado entre cada variable independiente y la propensión al cambio de la clase. También se hizo un análisis de correlación de *tau de Kendall* dado que se ajusta a múltiples observaciones que tienen el mismo valor y no dependen de la distribución de los datos.

Los autores concluyen, en primer lugar, que las métricas basadas en la evolución miden dimensiones diferentes a las MOO. Esto indica que no miden datos redundantes y por lo tanto, complementan a las MOO. En segundo lugar, varias métricas basadas en la evolución se correlacionan significativamente con la propensión al cambio de clase. En tercer lugar, un modelo de predicción basado en métricas basadas en la evolución a veces supera a un modelo basado en las MOO, y viceversa. Finalmente, se logra una predicción más precisa cuando las métricas basadas en la evolución se combinan con las MOO. En otras palabras, la inclusión tanto de las métricas propuestas basadas en la evolución como de las MOO en un modelo de predicción para la propensión al cambio de clase ofrece una precisión de predicción mejorada en comparación con un modelo que sólo incluye MOO o métricas basadas en la evolución [9].

### Cross-project change prediction using meta-heuristic techniques

Bansal *et al.* [16] realizaron varias predicciones de cambio *en proyectos* y *entre proyectos*. Los autores utilizan las características de distribución estadística del conjunto de datos para generar reglas que se pueden utilizar para una predicción de cambios exitosa. Los autores analizan la efectividad de los árboles de decisión metaheurísticos en la generación de reglas para la predicción exitosa de cambios entre proyectos. Los algoritmos metaheurísticos empleados son algoritmos genéticos de árboles de decisión híbridos y árboles de decisión oblicuos con aprendizaje evolutivo. Los autores comparan el rendimiento de estos algoritmos metaheurísticos con el modelo de árbol de decisión C4.5 utilizando medidas de rendimiento

[6] Método de rotación ortogonal que minimiza el número de variables que tienen cargas altas en cada factor. Simplifica la interpretación de los factores.

como exactitud, precisión, sensibilidad, exhaustividad y análisis ROC (por sus siglas en inglés receiver operating characteristic).

Las métricas que utilizan son las de [14] y número de líneas de código (LOC por sus siglas en inglés), las cuales se toman como las variables independientes, mientras que la variable dependiente es un valor nominal que indica si una clase cambió entre una versión y otra. Utilizan la herramienta SciTools para obtener las métricas. Utilizan 3 aplicaciones de código abierto: MyFaces, Struts y Wickets con 6 versiones de cada una de ellas, por lo que les da como resultado 15 conjuntos de entrenamiento.

Los autores precisan que los algoritmos evolutivos requieren de poco espacio de dominio, utilizan los principios de evolución y selección natural y se basan en una función objetivo la cual se evalúa iteración tras iteración hasta un número finito o hasta que se encuentra una solución lo suficientemente buena.

Para el método *en proyectos*, los autores seleccionan del mismo proyecto los datos de entrenamiento como se describió previamente y posteriormente se selecciona del mismo proyecto los datos de prueba. Para el método *entre proyectos,* los autores seleccionan los datos de entrenamiento de una aplicación A como ya se describió previamente y posteriormente de una aplicación B seleccionan el conjunto de datos de prueba.

Los autores concluyen que el algoritmo de árboles de decisión C4.5 proporciona los resultados de exactitud con 73.33%, sensibilidad con 74.14% y precisión con 83.69%. Mientras que el algoritmo evolutivo (árbol de decisión híbrido) obtiene los resultados de exactitud con 75.00%, sensibilidad con 93.97% y precisión con 74.15%. Por último, el algoritmo árbol de decisión oblicuo obtuvo los resultados de exactitud del 75.56%, sensibilidad con 79.31% y precisión con 82.14%. Los autores concluyen que el clasificador con los mejores rendimientos fue árboles de decisión híbrido [16].

### Identifying and understanding header file hotspots in C/C++ build processes

McIntosh *et al.* [18] mencionaron que un sistema de construcción rápida es el núcleo del desarrollo de software moderno. Sin embargo, los sistemas de software son grandes y

complejos y a menudo se componen de miles de archivos de código fuente que el sistema de compilación debe traducir cuidadosamente en entregables de manera oportuna.

Los archivos cabecera, que tienden a desencadenar procesos de reconstrucción lentos, son más problemáticos si también cambian con frecuencia durante el proceso de desarrollo y, por lo tanto, deben reconstruirse con frecuencia.

Los autores proponen en este trabajo un enfoque que analiza el árbol de dependencia de compilación (es decir, la estructura de datos utilizada para determinar la lista mínima de comandos que deben ejecutarse cuando se modifica un archivo de código fuente) y el historial de cambios de un sistema de software para identificar los *hotspots* cabecera en sistemas C/C++: archivos cabecera que cambian con frecuencia y desencadenan largos procesos de reconstrucción.

Los autores propusieron las métricas de entrada para su método, las cuales fueron: costo de reconstrucción (CR) y tasa de cambio (TC). El modelo para detectar los *hotspots* utilizó regresión logística y fue evaluado a través de correlación de datos Spearman y aplicando prueba de razón de verosimilitud.

El método propuesto por los autores para detectar los *hotspots* utilizan diferentes versiones de cuatro sistemas de código abierto: GLib, PostgreSQL, Qt y Ruby, desarrollados en lenguaje C/C++. El método como primer paso realizan la gráfica de dependencias de compilación, posteriormente calculan los costos de reconstrucción de cada archivo del sistema y por último realizan la evaluación para determinar los *hotspots* posibles. La evaluación se realiza graficando tasa de cambio vs. costo de reconstrucción de cada archivo. El plano es dividido en cuatro, dándoles la siguiente interpretación a cada uno de ellos, como se describe en la Tabla 3:

**Tabla 3 Interpretación a cada archivo por el cuadrante donde se ubican al graficarlos.**

| Tasa de cambio | | |
|---|---|---|
| | **B**<br>**Alta rotación**<br>Archivos que cambian con frecuencia y rápida compilación | **D**<br>**Hostpot**<br>Archivos que cambian con frecuencia y lenta compilación |
| | **A**<br>**Inactivos**<br>Archivos que no cambian con frecuencia y rápida compilación | **C**<br>**Compilación lenta**<br>Archivos que no cambian con frecuencia y lenta compilación |
| | | **Tasa de compilación** |

Los autores concluyen que los modelos de regresión construidos utilizando propiedades arquitectónicas y de código fuente pueden detectar entre el 32% y el 57% de los *hotspots*. Los autores también mencionan que su enfoque se puede utilizar para priorizar el esfuerzo de optimización de la construcción, lo que permite a los equipos centrar el esfuerzo en los archivos de encabezado que ofrecerán el mayor valor a cambio. Al aplicar continuamente su enfoque, los equipos de desarrollo pueden verificar que el esfuerzo de optimización de la construcción ha tenido un impacto [18].

### Active hotspot: an issue-oriented model to monitor software evolution and degradation

Feng *et al.* [21] mencionan que la degradación de la arquitectura tiene un fuerte impacto negativo en la calidad del software y puede resultar en pérdidas significativas. La degradación severa del software no ocurre de la noche a la mañana. El software evoluciona continuamente, a través de numerosos problemas, corrigiendo errores y agregando nuevas funciones. Las fallas de la arquitectura surgen silenciosamente y en gran parte pasan desapercibidas hasta que aumentan en alcance e importancia cuando el sistema se vuelve difícil de mantener.

Los autores propusieron un modelo llamado *hotspots* activo que se puede utilizar para monitorear la evolución del software y detectar degradación potencial, utilizando problemas como entidades de primera clase. Un *hotspot* activo está formado por archivos semilla que se modifican por varios problemas y los archivos que están conectados con ellos a través de uno de los 4 patrones de propagación de cambios (diseminación, concentración, dominó y dispersión). Utilizando los *hotspots* activos como monitores, los autores estudiaron el historial de cambios de 21 proyectos de código abierto. Los datos revelaron que, dentro de cualquier período de evolución, la mayoría de los archivos revisados para la solución de problemas simplemente se agrupan en unos pocos *hotspots* dominantes y la mayoría de estos puntos de acceso dominantes son persistentes y permanecen activos durante mucho tiempo, lo que implica que estos *hotspots* dominantes deberían ser los puntos focales de actividad y merecen una atención especial.

El método para detectar los *hotspots* activos se basa en los siguientes pasos. Primero, detección de archivos semilla tomando los registros del SCV y SCI como entradas, así como, un período de tiempo específico. Segundo, extracción de patrones de diseminación, concentración y dominó por cada *commit*. Tercero, extracción del patrón de dispersión con

herramientas de análisis de código fuente. Cuarto, cálculo de *hotspots* activos, utilizando la salida de los archivos semilla del paso 1 y la información de cuatro patrones de cambio de los pasos 2 y 3 como entradas, se determinan los *hotspots* activos.

Con este método se pueden detectar mucho más temprano los *hotspots* que con ciertas herramientas existentes para el mismo fin, pero, dependen de la calibración de sus umbrales para ser eficientes.

Los autores concluyen que, durante la mayoría de los períodos de evolución del software, por lo general, solo existe un puñado de grandes *hotspots*, que se pueden llamar *hotspots* dominantes, que unen la mayoría de los archivos activos. Estos *hotspots* dominantes deberían ser los puntos focales para el análisis. Los *hotspots* dominantes pueden durar mucho tiempo; son persistentes y longevos lo que, una vez más, muestra que deben ser los puntos focales de análisis y están únicamente relacionados con la intensidad de las interacciones y las relaciones arquitectónicas entre los archivos y no con los tamaños de los archivos o proyectos [21].

### Abductive network ensembles for improved prediction of future change-prone classes in object-oriented software

Al-Khiaty *et al.* [19] mencionan que los sistemas de software están sujetos a una serie de cambios debido a una variedad de objetivos de mantenimiento. Algunas partes del sistema de software son más propensas a cambios que otras. Estas partes propensas a cambios deben identificarse para que los recursos de mantenimiento se puedan asignar de manera efectiva.

Los autores proponen el uso de redes abductivas basadas en el método grupal de manejo de datos (GMDH por sus siglas en inglés) para modelar y predecir la propensión al cambio de clases en software orientado a objetos utilizando métricas estructurales del software [14] y el historial de cambios del software (SCV), cuantificado por un conjunto de métricas históricas.

Se construyeron varios modelos de predicción utilizando 3 tipos diferentes de métricas como predictores: métricas históricas, métricas estructurales y una combinación de estos dos tipos de métricas. Los autores informaron de la precisión de la predicción para cada modelo, así como de la fusión de los resultados de los tres modelos para formar un modelo de predicción por conjuntos.

Los autores reportan que los principales hallazgos de la investigación son los siguientes. Primero, los dos conjuntos de métricas (métricas históricas y estructurales) son predictores competitivos de la propensión al cambio. En segundo lugar, la combinación de los dos conjuntos de métricas como entradas a un clasificador abductivo basado en GMDH mejora la precisión de la clasificación en comparación con la obtenida con un único conjunto de métricas. En tercer lugar, el mejor rendimiento (83.58% de precisión en la clasificación) se obtuvo al fusionar las salidas de los tres modelos individuales utilizando la votación por mayoría para formar un clasificador abductivo de conjunto GMDH. Esto destacó una mejora de la precisión del 10.85% sobre la precisión del mejor miembro del conjunto, y una mejora de la precisión del 3.08% en comparación con el clasificador basado en regresión que utilizó el mismo conjunto de datos. Las métricas con mejor rendimiento fueron: edad y frecuencia de cambio (BCO), ocurrencia de cambio (FCH, LCH, CHO) y tamaño (LCOM) [19].

## Examining the effectiveness of machine learning algorithms for prediction of change prone classes

Malhotra *et al.* [20] indican que el cambio en las primeras etapas del ciclo de vida de un desarrollo de software es una estrategia eficaz para desarrollar un software de buena calidad a bajo costo. Para gestionar el cambio, los autores utilizaron modelos de calidad de software que pueden predecir de manera eficiente las clases propensas al cambio y, por lo tanto, guiar a los desarrolladores en la distribución adecuada de recursos limitados.

Su estudio examinó la efectividad de diez algoritmos de aprendizaje automático (AA) (máquina de soporte vectorial, programación de expresión genética, perceptrón multicapa con aprendizaje conjugado, método de agrupamiento para el manejo de datos, agrupación de K-means, TreeBoost, Bagging , Bayes ingenuo , árbol de decisión J48 y Random Forests) para desarrollar tales modelos de calidad de software en tres conjuntos de datos de software orientados a objetos (Robocode, Dspace y DrJava) con dos versiones de cada una de ellas. También compararon el rendimiento de los algoritmos de AA con la técnica de regresión logística ampliamente utilizada y clasificaron estadísticamente los algoritmos con la ayuda de la prueba de Friedman.

Las métricas propuestas para el estudio fueron las estructurales de [14] y número de líneas de código (NLOC por sus siglas en inglés).

Los autores concluyen que las métricas NLOC, CBO y WMC son predictores eficientes de cambio, ya que se incluyeron en el modelo de predicción de cambios utilizando los tres conjuntos de datos de código abierto después de la aplicación de la técnica de selección de características basada en correlación (CFS por sus siglas en inglés). Por otro lado, los resultados mostraron que el algoritmo de máquina de soporte vectorial es el mejor algoritmo AA que se puede utilizar para desarrollar modelos de predicción de cambios. Algunos otros algoritmos como perceptrón multicapa con aprendizaje conjugado, método de agrupamiento para el manejo de datos, árbol de decisión J48 y Bagging superan a la técnica estadística de regresión logística. Por lo tanto, los algoritmos AA son competitivos al tiempo que predicen cambios de software y pueden ser utilizados por la industria del software para planificar eficazmente la asignación de recursos y desarrollar software de buena calidad. El algoritmo de máquinas de soporte vectorial (SVM por sus siglas en inglés) obtuvo las mejores medidas de rendimiento con los que fueron evaluados los algoritmos de AA, con los siguientes valores: exactitud 77.44%, sensibilidad: 27.54%, especificidad: 94.92, precisión: 65.52 y f-medida 38.78% [20].

## 3.2. Análisis de la revisión bibliográfica

En la revisión bibliográfica realizada, se encuentra que la mayoría de los trabajos van en el mismo sentido, en el de ubicar aquellos módulos del sistema que son propensos al cambio o al error, con el objetivo de mantener o mejorar la calidad del software y las inversiones que se hagan en él, sea bien dirigidas en aquellas partes que lo requieran más. En todos los trabajos se hace uso de métricas del software, en la mayoría de los trabajos se hace uso de las métricas estructurales de [14], en algunos otros trabajos hacen uso de métricas históricas que son obtenidas de los SCV y en otros se proponen tipos de métricas como entropía [8]. En la gran mayoría de los trabajos los modelos son soportados por algoritmos de aprendizaje automático, algunos cuantos de algoritmos bioinspirados y otros con regresión logística.

A continuación, se muestra en la Tabla 4 una comparativa de los trabajos revisados, en ella se muestra un resumen de las características principales de cada uno de ellos, las cuales son: referencia del artículo revisado, como definen un *hotspot*, si hacen uso de métricas estructurales (código), métricas históricas a través de sistema de control de versiones, lenguaje

o herramientas que permitieron obtener las métricas, si hicieron uso de control de versiones, algoritmo de inteligencia artificial que utilizaron para la investigación, medidas de rendimiento con el que fue evaluado el resultado, número de aplicaciones con el que fue probado el método y porcentaje de exactitud reportado.

**Tabla 4 Resumen de la revisión sistemática de la literatura sobre la detección de *hotspots*.**

| Referencia | ¿Cómo definen un *hotspot*? | Uso de métricas estructurales | Uso de métricas históricas | Lenguaje o herramienta para obtener métricas | Uso de sistema de control de versiones | Algoritmo | Medidas de rendimiento | Número de aplicaciones para prueba | Porcentaje de exactitud reportado |
|---|---|---|---|---|---|---|---|---|---|
| D. Bura, M. Rachna, A. Choudhary, M. Surajmal, R. Kumar Singh, B. Tripathi Kumaon, (2017), "A Novel UML Based Approach for Early Detection of Change Prone Classes" [15] | Propensión al cambio | Dinámicas: (duración de ejecución, información de tiempo de ejecución, dependencia de clase, regularidad y popularidad)<br><br>UML 2.0<br>(Diagrama de Secuencia y Clases)<br>(IOM, COM, ART) | No | Java Understand | No | Colonia de Abejas (ABC) | No | 2 | No reportan |
| R. Hilton, E. Gethner, (2017), "Predicting Code Hotspots in Open-Source Software from Object-Oriented Metrics Using Machine Learning" [24] | Áreas del código que cambian repetidamente durante el desarrollo del software | Por cada clase: NSF, NSM, NOF, NOM, NORM, WMC, DIT, SEIS, LCOM | POM<br>(Probabilidad de Modificación por Clase) | Eclipse<br>Python<br>(scikit-lear) | No | Random Forest | Precisión | 13 | 75% |
| D. Godara, R.K. Singh, (2014), "A New Hybrid Model for Predicting Change Prone Class" [5] | Propensión al cambio | UML 2.0<br>(Diagrama de Secuencia y Clases)<br>Dependencia Conductual (directa e indirecta) | Tiempo de Ejecución y Cuantas veces se ejecutan las clases | Java Understand | No | ID3 | No indican | 1 | No reportan |
| M. Yanb, X. Zhanga, C. Liub, L. Xub, M. Yang, D. Yang, (2017), "Automated change-prone class prediction on unlabeled dataset using unsupervised method" [7] | Propensión al cambio | Métricas OO:<br>WMC<br>DIT<br>NOC<br>RFC<br>LCOM | Entrenamiento<br><br>MPC<br>DAC<br>NOM<br>SIZE1<br>SIZE2 | | NO<br>Solo para entrenar | Método no supervisado: CLAMI+ (Agrupación, etiquetado, selección de métricas y selección de instancias) | CCR<br>UAC (ROC)<br>F-medida (measure) | 14 | No reportan |
| J. R. Birt, Renate Sitte, (2005), "Identifying Error Proneness in Path Strata with Genetic Algorithms" [17] | Propensión al error | SoE (Source of Error)<br>Pesos a las clases = Sum(SoE) | | | No | CBOEvolutivo: Genético | Comparativo: AG vs Selección aleatoria de trayectorias | 10 | 85% de Predicción |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| S. Eski, F. Buzluca, (2011),"An Empirical Study on Object-Oriented Metrics and Software Evolution in Order to Reduce Testing Costs by Predicting Change-Prone Classes" [6] | Propensión al cambio | (Métricas OO)<br>(C&K): WMC, DIT, NOC, CBO, RFC, LCOM<br>QMOOD:<br>relación entre atributos de calidad y propiedades de diseño | Cambios de SW.<br>Factor: Costo del Cambio (pondera entre 2 versiones, por su impacto arquitectónico) | e-Quality | SI<br>Cambios Estructurales y No estructurales (Se define un peso) | Ecuaciones de Análisis de Cambio | HIT RATIO (%)<br>CHANGE COST<br>COST RATIO (%) | 3 | No reportan |
| S.Garba Maisikeli, (2016), "Evaluation Of Software Degradation And Forecasting Future Development Needs In Software Evolution" [8] | Degradación de clases | Esfuerzo COCOMO, Métricas de duración -degradación- y Periodo de duración | Índice de madurez del SW<br><br>Ecuaciones Shannon- y Entropía | AspectJ Weaver | Si | Ecuaciones | Índice de madurez Entropía | 1 | No reportan |
| M. O. Elish*,† M. Al-Rahman Al-Khiaty, (2012), "A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software " [9] | Propensión al cambio | Chidamber y Kemerer (C&K) | GQM, BOC, FCH, FRCH, LCH, WCH, WCD, WFR, TAC, ATAF, CHD, LCA, LCD, CSB, CSBS, ACDF, CHO | | Si | Ecuaciones para determinar las métricas externas.<br>Estadística | Tasa de clasificación correcta | 2 | No reportan |
| A. Bansal, S. Jajoria, (2019), "Cross-Project Change Prediction Using Meta-Heuristic Techniques" [16] | Propensión al cambio | Chidamber y Kemerer (C&K)<br>+<br>LOC | Características de Distribución: media, moda, mediana, mín, máx, varianza, desviación estándar y rango | SciTools<br><br>Java Understand | Si | C4.5<br><br>Evolutivos: AdaBoost, LogitBoost, Bagging | Exactitud<br>Exhaustividad<br>Precisión<br>ROC | 3 | Exactitud y Exhaustividad > 65% |
| S. McIntosh, B. Adams, M. Nagappan, A. E. Hassan, (2015), "Identifying and Understanding Header File Hotspots in C/C++ Build Processes" [18] | Archivos que más cambian con respecto al tiempo (versiones)<br><br>Cambios frecuentes y compilación lenta | C/C++<br>lh=hotspot<br>lr=mayor Cto reconstrucción<br>lc=tasa de cambio<br>lf=llamados include's<br><br>Contenido de código | Gráficos de dependencias de compilación<br><br>Costo de Reconstrucción<br><br>Tasa de cambio de cada archivo<br><br>Layout código | MAKAO<br><br>Java Understand<br><br>Layout cód.: subsistemas, depth, directory pan-in<br><br>SLOCCount<br>Contenido cód.: file fan-in, SLOC, Núm de Includes | Si | Modelos de<br><br>Regresión Logística: probabilidad de puntos calientes<br><br>Correlación de datos Sperman<br><br>Prueba de razón de verosimilitud | Costo de reconstrucción vs. tasa de cambio de cada archivo<br><br>con umbrales de 90 y 0.5 seg. respectivamente | 4 | 32% - 57% |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Q. Feng, Y. Cai, R. Kasman, D. Cui, T. Liu, H. Fang, (2019), "Active Hotspot: An Issue-Oriented Model to Monitor Software Evolution and Degradation" [21] | Puntos críticos dominantes de cambio | Difusión, concentración, dominó,<br>Dispersión | No | Java-Underst<br>Simiam | Si | | | 21 | |
| M. Al-Khiaty, R. Abdel-Aal, M. Elish, (2017) "Abductive Network Ensembles for Improved Prediction of Future Change-Prone Classes in Object-Oriented Software" [19] | Propensión al cambio de Clases | De Producto: OO (C&K) | De Evolución: Edad de las clases, frecuencia de los cambios | ExamDiff | Si | Redes Abductivas Método Grupal de Manejo de Datos (GMDH) | Exactitud<br>Exhaustividad<br>Precisión<br>ROC<br>F1-Medida | 1 | 83% de Exactitud |
| R. Malhotra, M. Khanna, (2014), "Examining the effectiveness of machine learning algorithms for prediction of change prone classes" [20] | Propensión al cambio | Chidamber y Kemerer (C&K)<br>+<br>LOC | | Java Understand | Si | SVM<br>GEP<br>MLP-CG<br>GMDH<br>K-means<br>TB<br>BG<br>NB<br>J48<br>RF | Sensibilidad<br>Especificidad<br>Exactitud<br>Precisión<br>ROC<br>F1-Medida<br>+<br>Prueba de Friedman | 3 | 64% al 85% de Exactitud |

### 3.2.1. Entendimiento del término *hotspot* en la literatura

Una vez que se ha realizado la revisión de la literatura, en la cual se encuentran diferentes interpretaciones de un *hotspot*, para nuestra investigación lo definiremos como *todo archivo asociado a un módulo de un sistema que sea propenso a un cambio o a un error*. Estos cambios o errores que pueden presentar los elementos del sistema son derivados de deficientes diseños arquitectónicos o de implementación ya sea en sus versiones iniciales o en la evolución de cada uno de ellos.

Algunas definiciones de *hotspot* que se encontraron en la literatura son:

- Aquellos elementos de un sistema que tienen alta probabilidad de tener un cambio [5], [6], [7], [9], [15], [16], [19], [20].
- Aquellos elementos de un sistema que tienen alta probabilidad de tener un error [7].
- Puntos críticos dominantes de cambio [21].
- Archivos que más cambian con respecto al tiempo [17].
- Áreas del código que cambian repetidamente durante el desarrollo del software [24].

### 3.2.2. Métricas que se proponen en la detección de *hotspots*

Hay varios tipos de métricas que pueden utilizarse en la realización de proyectos de software para gestionar, predecir y mejorar la calidad de software. La finalidad del uso de métricas es evaluar sistemas para conseguir alta calidad y robustez.

En casi todos los trabajos revisados se plantea el uso de métricas de calidad de código como el elemento principal para la identificación de los *hotspots* en los sistemas. Esto es, debido a que existe una gran variedad de éstas, que a lo largo del tiempo se han propuesto, que nos proporcionan características del sistema y de la calidad con la que está construido.

Tenemos trabajos que hacen uso de métricas estructurales ya bastante consolidadas y que nos indican el estado de cada elemento del software. Estas métricas son las siguientes:

- **Métricas estructurales de Chidamber y Kemerer** [14] métodos ponderados por clase (WMC), profundidad del árbol de herencia (DIT), número de hijos (NOC), número de hijos

(CBO), respuesta para la clase (RFC), falta de cohesión de los métodos (LCOM) las cuales analizan la estructura de las clases (elementos de software) y que en la mayoría de los estudios aportan significativamente a la detección de *hotspots*. De estas, en los estudios [4], [7], [19], [20] se concluye que las métricas: RFC, CBO, LCOM Y WMC son las que están más relacionadas con la propensión al cambio de las clases.

- **Métricas estructurales basadas en diagramas de UML**: las de dependencia de clases (CD), regularidad (Reg) y popularidad (Pop)[7] o las de dependencia de comportamiento de clases como duración de ejecución (ED), información de tiempo de ejecución (RTI) [15].
- **Métrica de tamaño**: en [6], [20] propone a número de líneas de código (NLOC) como una buena métrica predictora al cambio, dado que el tamaño está directamente relacionado a la propensión al cambio o al error.

Por otro lado, tenemos las métricas históricas, las cuales pueden ser obtenidas de los sistemas de control de versiones (SCV) que utilizan los equipos de desarrollo al momento de estar desarrollando la primera versión del sistema y posteriormente las múltiples versiones que pueda haber del sistema por nuevas versiones, relacionadas a nueva funcionalidad o la reparación de incidentes presentados en producción.

De este tipo de métricas se pueden obtener una variedad muy grande, esto depende de la herramienta para obtenerlas o de la propuesta que se haga para obtenerlas, ya que al contar con las bitácoras de los SCV del sistema se pueden proponer diferentes formas de medir cambios en los elementos del sistema.

**Métricas basadas en la evolución del software**: nacimiento de una clase (BOC), primera vez que cambió (FCH), frecuencia de cambio (FRCH), última vez que cambió (LCH), cambios ponderados (WCH), densidad de cambio ponderada (WCD), frecuencia ponderada de cambios (WFR), tamaño de cambio agregado normalizado por frecuencia de cambio (ATAF), densidad del cambio (CHD), cantidad de últimos cambios (LCA), densidad del último cambio (LCD), cambios desde el nacimiento (CSB), cambios desde el nacimiento normalizado por tamaño (CSBS), densidad de cambio agregado normalizada por frecuencia de cambios (ACDF), Se produjo un cambio (CHO), [9], POM (probabilidad de modificación por clase), veces de uso de clase, costo del cambio, entre otras. Estás métricas de igual forma que las estructurales tienen un importante aporte en

los trabajos que hicieron uso de ellas, como los trabajos en las que las usaron en [19] donde se concluye que las métricas: edad y frecuencia de cambio (BCO), ocurrencia de cambio (FCH, LCH, CHO) o como en [4] donde su POM, que, combinado con las métricas estructurales, obtiene un buen nivel de predicción de cambio. También tenemos la métrica de violación a la modularidad[7] (MV) la cual observan en [23] como una métrica que se correlaciona con otras con un alto porcentaje para detectar la deuda técnica.

### 3.2.3. Algoritmos aplicados para la detección de *hotspots*

Dentro de la revisión de la literatura se ubicaron diversas propuestas para la detección de *hotspots*, casi todos con propuestas de algoritmos de inteligencia artificial, tanto con algoritmos de optimización como con algoritmos de aprendizaje automático y algunas otras propuestas con métodos estadísticos. A continuación, se presenta un resumen de estos:

- **Algoritmos de aprendizaje automático** (AA), los cuales en algunos casos fueron empleados de apoyo en el proceso para seleccionar métricas [5], [15], como algoritmos de comparación con modelos de regresión [20], o como el algoritmo principal para la detección de *hotspots* [4], [5], [7], [15], [19], [20]. De estos últimos tenemos que sus resultados son bastante favorables en la detección de *hotspots* ya que manejan entre un 65% y un 80% de exactitud. Los algoritmos más utilizados son: ID3, Random Forest, C4.5, Redes Abductivas, Bagging, Máquinas de Soporte Vectoriales, CLAMI y CLAMI+.
- **Métodos de optimización** (MO). Los cuales se usan para selección de métricas en algunos casos [15] y para detección de *hotspots en otros* [16], [17]. De estos últimos se usaron los algoritmos de: AdaBoost, LogitBoost, Bagging y Genéticos, teniendo exactitud del 65% para Bagging [16] y de 85% para Genéticos [19].
- **Estadísticos**. Se usaron diversos modelos estadísticos para detectar *hotspots* para algunos casos [9], [18] y, por otro lado, como comparativo para evaluar el nivel de exactitud con respecto a algoritmos de aprendizaje automático. Tenemos trabajos que definen sus propias funciones para **análisis de cambio** [6] que combinan su modelo con

[7] La teoría de la regla de diseño de Baldwin y Clark propuso que los módulos independientes se pueden cambiar o incluso reemplazar sin influirse entre sí. Sunny Wong, introdujo el término violación de la modularidad, que describe dos módulos estructuralmente independientes que cambian juntos con frecuencia, lo que significa que no son verdaderamente independientes. Cuanto más a menudo cambien juntos dos archivos estructuralmente no relacionados, más probable es que existan dependencias implícitas entre ellos.

métricas estructurales y métricas de evolución y concluyen que entre más versiones de las aplicaciones se utilicen mayor la precisión, además de que, las métricas que mayor aporte tienen al cambio son: WMC, LOC y RFC. También tenemos el modelo de **series de tiempo** [8], que, combinado con métricas de evolución de índice de madurez y de entropía, obtienen tendencias de cambios en las clases y recursos necesarios para realizarlos. Y por último tenemos modelos de **regresión logística** [18], [20] usando probabilidad de puntos calientes, correlación de datos Spearman y prueba de razón de verosimilitud, tomando como variables independientes a las métricas estructurales y evolutivas, obteniendo resultados para un caso [18] una exactitud del 57% y por otro lado cuando fueron comparados con algoritmos de aprendizaje automático [20] su exactitud fue del 64%, quedando en la media con respecto de los algoritmos de aprendizaje automático.

### 3.2.4. Medidas de rendimiento para evaluar resultados

Para dar certeza de los resultados que se obtienen en las investigaciones se encuentran diferentes propuestas de medidas de rendimiento. Algunos trabajos proponen sus propias medidas de rendimiento para evaluar los resultados y algunos otros trabajos más recientes proponen métricas mínimas para evaluar los resultados de los algoritmos de clasificación.

- Lo que podemos observar es que las medidas de rendimiento que se mantienen en los trabajos [16], [19], [20] son las de: exactitud, precisión, exhaustividad y análisis ROC (área bajo la curva) y Medida-F.
- Tenemos otros trabajos que plantean sus propias medidas de evaluación, como son: Entropía, Índices de madurez [8], tasa de cambio de costo (CCR por sus siglas en inglés) [7] los cuales van muy dirigidos al modelo que están planteando, tasa de clasificación correcta en [9], comparativo AG vs selección aleatoria de trayectorias en [17] y costo de reconstrucción vs. tasa de cambio de cada archivo [18].

## 3.3. Conclusiones de la revisión bibliográfica

Con base en el análisis de los trabajos revisados previamente, podemos concluir lo siguiente con la intención de determinar la propuesta de investigación.

- En la mayoría de los trabajos revisados, los autores proponen la identificación de elementos del software (archivos) que son propensos a cambios o errores. Todos coinciden en que se considera como *hotspots* a cualquier elemento del sistema (archivo, cabecera, clase, entre otros) que pueda manifestarse con un grado alto de propensión al cambio o al error. En este trabajo de investigación se está de acuerdo con las definiciones de *hotspot* encontradas en la revisión de la literatura por lo que en lo sucesivo lo entenderemos como:

  **"Los elementos del software (archivos de código fuente) que son propensos al cambio o propensos al error durante la evolución de este."**

- Sobre las métricas, como pudimos ver en nuestra revisión de la literatura, algunos trabajos concluyen que tienen métricas estructurales o históricas que les ofrecen mejores resultados que otras. En los diferentes trabajos, la obtención de métricas depende de las herramientas que se utilizaron para obtenerlas. Y solo en pocos casos utilizaron las bitácoras de los SCV de manera adecuada, ya que no todos los equipos de desarrollo tienen la disciplina de vincular los *commits* con los issues del SCI y poder generar métricas históricas.
- Como lo podemos ver en la columna "Número de aplicaciones para prueba" de la Tabla 4, los diferentes estudios trabajaron con un conjunto de aplicaciones que pudieron obtener con las características que buscaban, en su gran mayoría son aplicaciones de código abierto, orientados a objetos y desarrolladas en java. Algunos trabajos como [7] y 16] utilizan los resultados obtenidos con un conjunto de aplicaciones en otras aplicaciones que no intervinieron en los primeros resultados, esto da mayor certeza a la propuesta de detección de *hotspots*, dado que eso es lo que realmente pasará, ya que con la propuesta de métricas y algoritmo de clasificación se deberá de validar que estás funcionan en nuevas aplicaciones.

- Para la clasificación de *hotspots* encontramos una variedad de métodos para realizarlo, desde plantearlo como un problema de optimización hasta de aprendizaje automático supervisados o no supervisados.

## 3.4. Preguntas planteadas para la investigación

Una vez realizada la revisión de la literatura y presentado el estado del arte, detectamos que existen varias propuestas sobre uso de métricas estructurales e históricas, pero no hay una evaluación de las posibles combinaciones de métricas y cuales podrían dar los mejores resultados en la detección de *hotspots*, por lo que nos planteamos las siguientes preguntas de investigación:

1. ¿De un conjunto de métricas dadas, existen una o varias combinaciones de métricas que puedan ofrecer mejores resultados que otras, en la detección de *hotspots* en las aplicaciones de software?
2. ¿Las métricas para detectar *hotspots* de una aplicación pueden funcionar para determinar *hotspots* en otras aplicaciones?
3. ¿Cuál será el desempeño de un método de clasificación teniendo un método de referencia (exhaustivo)?

# Capítulo 4

# Metodología para identificar un conjunto de métricas para detectar *hotspots*

Uno de los resultados que se obtuvieron de la revisión de la literatura fue la importancia que tienen las métricas estructurales e históricas para la detección de *hotspots* en las aplicaciones y se puedan tomar acciones que ayuden a disminuir y prevenir la DT.

Para la obtención de *hotspots* en los sistemas se propone la siguiente metodología, la cual comprende los siguientes pasos: selección de aplicaciones para el estudio, selección de herramientas de análisis de código, selección de métricas candidatas, identificación de hotspots, análisis de resultados de cada experimento y selección de métricas finales y validación de resultados. En la Fig. 6 podemos ver el flujo de la metodología paso a paso. El detalle de cada paso se describe a continuación.

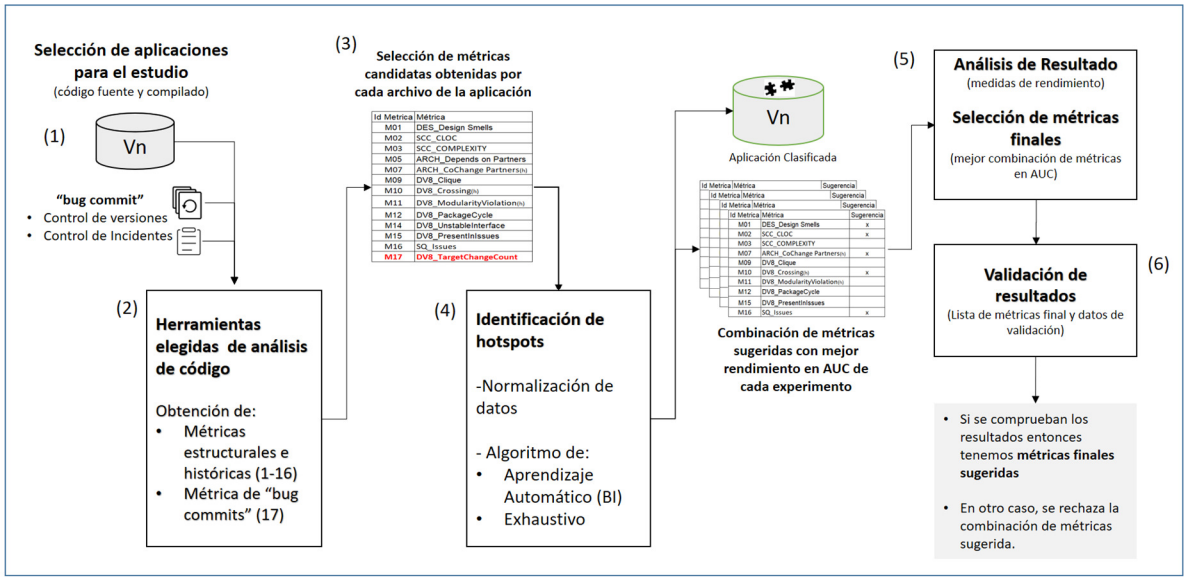



**Fig. 6 Metodología propuesta para identificar un conjunto de métricas que ayuden en la detección de *hotspots* en los sistemas.**

1. **Selección de aplicaciones para el estudio** (sistemas) que se utilizan para analizar el comportamiento de sus métricas. Las aplicaciones que se elijan deben cumplir con ciertas características que se detallan más adelante. Sin embargo, la característica más importante a cumplir es que utilicen los SCV y SCI y que los *commits* hagan referencia a las entradas en SCI para poder obtener la métrica NBC.
2. **Selección de herramientas de análisis de código** (HAC) que reciban como entradas código fuente, código compilado, historiales de su sistema de control de versiones (SCV) o historial de su sistema de control de incidentes (SCI). Las HAC elegidas deben de entregar como salida un conjunto de métricas por cada archivo fuente del sistema, puesto que el estudio se centra a ese nivel de detalle.
3. **Selección de métricas candidatas.** De todas las métricas obtenidas (disponibles) con las HAC que se eligieron, se realiza un listado de *métricas candidatas* en la que se incorporarán métricas que no estén repetidas (que no signifiquen lo mismo), que los valores devueltos por las métricas sean numéricos continuos (que no sean binarios), entre otros criterios. En las métricas candidatas debe de estar NBC ya que esta nos servirá para validar los resultados. De estás *métricas candidatas* se buscará una combinación de ellas que nos permita identificar los *hotspots* en las aplicaciones a las cuales llamaremos *métricas seleccionadas*.
4. **Identificación de *hotspots***. A través de la normalización de datos y un método de clasificación de Bayes ingenuo (BI), se realizan diferentes experimentos para la obtención de una combinación de métricas que ayuden a identificar los *hotspots* de los sistemas. Optamos por utilizar a BI como el clasificador puesto que los tiempos de entrenamiento son excepcionalmente rápidos, el rendimiento, la facilidad de implementación y los recursos computacionales mínimos requeridos lo hacen una opción apropiada para clasificar [27]. Además, con el fin de contrastar los resultados obtenidos por el clasificador BI, se considera el uso de un algoritmo exhaustivo.
5. **Análisis de resultados de cada experimento y selección de métricas finales**. De cada uno de los experimentos se selecciona el que obtenga los mejores resultados de clasificación basándose en las medidas de rendimiento, principalmente en el área bajo la curva (AUC por sus siglas en inglés). Teniendo

en cuenta que un experimento está representado por un algoritmo, una normalización de datos y una combinación de métricas resultante.

Con el análisis de los resultados que se ejecuta en el paso anterior, se hace la elección de la lista de *métricas finales* que se sugieren utilizar para la posible identificación de los hotspots en los sistemas. Esta lista de *métricas finales* se toma del experimento que resulte con el valor más alto en el AUC con respecto al resto de los experimentos.

6. **Validación de resultados**. Se aplica el algoritmo, la normalización de datos y la lista de *métricas finales* a las aplicaciones que fueron destinadas para este propósito. Estas aplicaciones son independientes a las que se utilizan en el paso 4 (fases de entrenamiento y pruebas). Se comprueban o se rechazan los resultados.

## 4.1 Selección de aplicaciones para el estudio

La selección de las aplicaciones es la base sobre la que se fundamenta el estudio, ya que de ellas saldrán los sujetos de estudio. El sujeto de estudio es cada archivo de código que es parte de una aplicación. De cada archivo se tomarán sus características (métricas) y a partir de ellas se tratará de etiquetarlo como *hotspot* o no *hotspot*. Entonces, se seleccionarán aplicaciones que sean de diferentes tamaños (cantidad de archivos). Lo anterior, para que las métricas que sean obtenidas de cada uno de los archivos de la aplicación ofrezcan una variedad de valores en ellas. Por ejemplo, entre más grandes sean las aplicaciones, serán mayores las dependencias entre sus elementos (archivos, clases, métodos, entre otros) y esto se verá reflejado en métricas como NOC (número de hijos -si estuviéramos trabajando con aplicaciones que utilizan un lenguaje de programación OO, por ejemplo, Java-) o LOC (número de líneas).

Se deberá elegir el conjunto de aplicaciones con las que se desea hacer el estudio, donde se deberá de buscar que todas estas cumplan con las mismas características, tales como:

- La organización que les da mantenimiento sea la misma. Esto para asegurar que las aplicaciones elegidas estén construidas sobre la misma cultura de la ingeniería de software.
- El mismo lenguaje con el que están desarrolladas. Por ejemplo, Java, C/C++, Python, Go, R, entre otros. Para cada una de las métricas que sean obtenidas de los archivos fuente de las aplicaciones representen lo mismo y no represente un sesgo en el estudio.
- Que utilicen los SCV y SCI y que los *commits* realizados hagan referencia al número de issue del SCI para poder obtener la métrica NBC.
- Que tengan un número considerable de *bug commits* (tomamos como mínimo 1000 mil o más) para que la métrica de NBC pueda ser considerada para realizar el etiquetado de archivos antes de comenzar con la experimentación con el algoritmo de clasificación.

Dada la importancia de validar los resultados de la experimentación, se deberán dividir las aplicaciones. Una parte para la etapa de *entrenamiento y pruebas* (sección 4.4) y otra parte para la etapa de *validación de los resultados* (sección 4.5). Con los resultados obtenidos (algoritmo, normalización de datos y combinación de métricas) en la etapa de *entrenamiento y pruebas*, se realizará la *validación de resultados* sobre el conjunto de aplicaciones reservadas para esta etapa. Si los resultados obtenidos en la etapa de *validación de resultados* fueran similares en AUC a los obtenidos en la etapa de *entrenamiento y pruebas*, estaremos en condiciones de confirmar nuestra hipótesis de que existen métricas que nos permiten encontrar archivos *hotspots* en aplicaciones bajo esta metodología.

Como parte de este primer paso, se deberá de obtener de cada una de las aplicaciones: el código fuente, el código compilado, las bitácoras del SCV y las bitácoras del SCI. Estas son las entradas para las HAC, de las cuales nos apoyaremos para la obtención de un conjunto de métricas. Si el código compilado no se tuviera, se deberá de generar a partir del código fuente de cada aplicación.

## 4.2 Selección de herramientas de análisis de código

La oferta de herramientas de análisis de código (HAC) ha crecido en el mercado en los últimos años. Por mencionar algunas de ellas, tenemos: SonarQube[8], Structure 101[9], Designite[10], CAST[11], SonarGraph[12], CodeInspector[13], CodeMRI[14], DV8[15], NDepend[16], SQuore[17], Symfony Insight[18], CodeScene[19], Archinaut[20], entre otras.

Lo que se busca que cumplieran las HAC seleccionadas:

- Estás estén relacionadas con la detección de la DT, parte de su misión del análisis de código que realizan sea tratando de buscar la DT y la plasmen en un conjunto de métricas que ofrezca para ser analizadas.
- Que sean ampliamente accesibles. Esto con el fin de que no se tenga inconveniente en poder hacer uso de ellas, es decir, se puedan obtener y exista documentación suficiente para utilizarlas.
- Los resultados de las métricas que ofrezcan sea a nivel archivo ya que a este nivel es donde buscamos identificar los *hotspots*.
- Acepten como entrada el lenguaje de programación en la que están desarrolladas las aplicaciones que sean elegidas en la sección 4.1.
- Sus entradas puedan ser el código fuente o el código compilado.
- De ser posible, que acepte como entrada bitácoras de un SCV y de un SCI para el cálculo de métricas históricas.
- Sus salidas que ofrezcan sean en formatos legibles y fáciles de manipular. Estos representarán los datos de entrenamiento, pruebas y validación, por lo que no deberá costar trabajo prepararlos para que sean tomados por los algoritmos a utilizar para el estudio.

[8] "Sonarquere." [En línea]. Disponible: https://www.sonarsource.com/products/sonarqube/
[9] "Structure 101." [En línea]. Disponible: https://structure101.com/
[10] "Designite." [En línea]. Disponible: https://www.designite-tools.com/
[11] "CAST." [En línea]. Disponible: https://www.castsoftware.com/
[12] "SonarGraph." [En línea]. Disponible: https://www.hello2morrow.com/
[13] "CodeInspector." [En línea]. Disponible: https://www.code-inspector.com/
[14] "CodeMRI." [En línea]. Disponible: https://codemri.com/
[15] "DV8." [En línea]. Disponible: https://archdia.com/
[16] "NDepend." [En línea]. Disponible: https://www.ndepend.com/
[17] "Squore." [En línea]. Disponible: https://codemri.com/
[18] "Symfony Insight." [En línea]. Disponible: https://insight.symfony.com/
[19] "CodeScene." [En línea]. Disponible: https://codescene.com
[20] "Archinaut." [En línea]. Disponible: https://doi.org/10.1145/3387906.3388633

## 4.3 Selección de métricas candidatas

Como lo pudimos ver en la revisión de la literatura, hay una variedad de métricas propuestas a utilizar en los diferentes trabajos revisados (ver Tabla 4 de la sección 3.2). En este estudio, se consideran algunas que se han usado en estudios previos y algunas otras existentes, pero que no se han usado hasta ahora.

Con el apoyo de las HAC se pueden obtener una diversidad de métricas de las cuales se deben de considerar como disponibles, aun así, hay algunas que se podrían simplificar, ya que representan la misma métrica. Por ejemplo, LOC (líneas de código) la cual se puede obtener de tres de las herramientas de código (DES, SCC y DV8). En la Tabla 5 se muestra la lista ejemplo de métricas, así como, la HAC de la que se puede obtener.

**Tabla 5 Ejemplos de métricas disponibles de algunas de las herramientas mencionadas.**

| Nombre | Heramienta | Descripción |
|---|---|---|
| Complexity | Designite | Compeljidad |
| Design Smells | Designite | Diseño oloroso |
| LOC | SCC | Tamaño en líneas de código |
| CLOC | SCC | Tamaño en Lineas de Código fisico (LOC) |
| Revisions | Archinaut | Número de commits en los que aparece un archivo |
| Dependent Partners | Archinaut | Número de archivos que dependen de este archivo (fan in) |
| Target Change Count | DV8 | Número de veces que aparece un archivo en un commit asociado con un error ("bug commit") |
| Modularity Violation | DV8 | Número de violaciones a la modularidad en las que aparece este archivo |
| Code Smells | SonarQube | Número de olores de código de SonarQube |
| Bugs | SonarQube | Número de errores de SonarQube |
| ClassTangles | Structure101 | Número de enredos de clase en los que participa este archivo |
| Size | Structure101 | Una medida de tamaño indefinida |

De la lista de métricas disponibles que se puedan obtener de las HAC se deberán elegir las *métricas candidatas* con las que se realizarán los experimentos previstos y que se describen más adelante en la sección 4.4.

Se deberá elegir hacia dónde se quiere enfocar el análisis de métricas, dado que tenemos de diferentes tipos, es decir, tenemos métricas que miden el tamaño en líneas de código, complejidad de cada archivo fuente, antipatrones, ciclos a nivel paquete, malos olores de diseño, violaciones a la modularidad, interfaces inestables, dependencia, malos olores de código[21], entre otras. También se debe de considerar que algunas de ellas son similares por lo que no se recomienda que se utilicen dos o más que representen lo mismo.

[21] El concepto de código con malos olores (code smells) fue introducido a fines de los años 90 y es una manera de referirse a ciertas características subjetivas en el código fuente que podrían repercutir en problemas de operación y mantenimiento.

Para las aplicaciones elegidas, se obtienen los valores de las métricas candidatas por archivo. Estos valores serán nuestros datos de entrenamiento, prueba y validación de resultados. Estas son las entradas para el paso de clasificación de *hotspots*.

## 4.4 Identificación de *hotspots*

Recordemos que los *hotspots* son los archivos de una aplicación que son propensos al cambio y propensos al error. Para llevar a cabo la clasificación de los archivos *hotspot* se propone el algoritmo de aprendizaje automático supervisado BI. Para hacer una validación y contrastar los resultados obtenidos por el clasificador, se propone un algoritmo exhaustivo.

Los clasificadores de aprendizaje automático supervisado se han aplicado con éxito en diferentes dominios como la medicina, psicología, marketing, ingeniería de software, entre otros [20].

Por otro lado, debido a la diferencia de magnitudes de las métricas, se propone normalizarlas dentro del intervalo [0,1]. También se analiza la métrica Número *Bug Commits* (NBC) para etiquetar cada archivo (datos de entrenamiento y pruebas) antes de aplicar los algoritmos. Se elige está métrica para etiquetar los archivos como *hotspots* ya que esta representa el número de veces que un archivo ha estado involucrado en un *bug commit*, es decir, son los archivos que más han cambiado por estar involucrado en un error.

Con lo anterior, tendremos un etiquetado real y un etiquetado propuesto por el algoritmo, lo que nos permitirá calcular las medidas de rendimiento a través de la matriz de confusiones que revisaremos más adelante y así poder evaluar la clasificación.

### 4.4.1 Selección de algoritmo de clasificación

La estructura básica y el funcionamiento de los algoritmos de aprendizaje automático es bastante diferente de las técnicas estadísticas tradicionales. Estos algoritmos se han aplicado con éxito a

la ingeniería de software, como la predicción de defectos, la predicción del esfuerzo y la predicción de la capacidad de mantenimiento [12], [20].

Entre los algoritmos de aprendizaje automático que podrían utilizarse para la clasificación de los datos están: máquinas de vectores de soporte, programación de expresión genética, perceptrón multicapa con aprendizaje conjugado, método de agrupamiento para el manejo de datos, agrupación de K-medias, treeboost, bagging, Bayes ingenuo, árboles de decisión J48 y bosques aleatorios (SVM, GEP, MLP-CG, GMDH, K-means, TB, BG, NB y RF por sus siglas en inglés respectivamente). Estos son algunos de los algoritmos que evaluaron en [20] y donde concluyen que en general los algoritmos de aprendizaje automático dan buenos resultados en la clasificación, de hecho, lo comparan con un algoritmo estadístico de regresión logística y algunos de ellos están por arriba de este.

Para contrastar los resultados del algoritmo de aprendizaje automático seleccionado, se propone el uso de un algoritmo exhaustivo. La propuesta de este algoritmo se plantea por dos razones: la primera, para explorar todo el espacio de soluciones que pueda haber con cada una de las combinaciones de métricas y la segunda, para contrastar los resultados obtenidos con el clasificador de aprendizaje automático supervisado seleccionado. Se sugiere el uso de un algoritmo exhaustivo dado que computacionalmente puede ser resuelto en un tiempo razonable. El tiempo depende del número de métricas a utilizar y del tamaño de los datos. Este algoritmo evalúa todas las combinaciones de métricas posibles dada la lista de *métricas candidatas*. En general, si tenemos un conjunto de n métricas candidatas, el número de combinaciones posibles son $2^n$. Por ejemplo, si n = 30, $2^{30} = 1.07x10^9$.

### 4.4.2 Etiquetado de los datos de entrenamiento

Una de las características principales de los algoritmos de clasificación de aprendizaje automático supervisado es que los datos de entrenamiento deben de estar etiquetados previamente con la clase a las que están asociados.

Los datos de entrenamiento se etiquetan en dos clases, una clase $C_0$ que indica que el dato no es *hotspot* y otra clase $C_1$ que indica que el dato es *hotspot*. Para realizar el etiquetado de los datos se utiliza la métrica Número *Bug Commits* (NBC), la cual representa el número de

veces que un archivo ha participado en un *commit* asociado a un error (*bug commit*). Si se consideran valores altos en esta métrica para un archivo con respecto al resto de los archivos de la misma aplicación, entonces se puede sugerir que el archivo sea etiquetado con la clase ***C1***, en otro caso se etiqueta con la clase ***C0***.

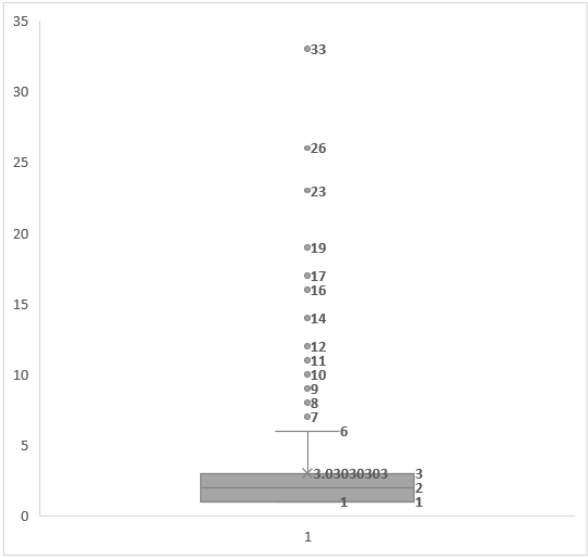



**Fig. 7 Ejemplo de distribución de la métrica NBC para la aplicación Tez.**

Para determinar el valor umbral de NBC para etiquetar los archivos como *hotspots* o no *hotspot*, se realiza un análisis de NBC a través de un diagrama de caja. Este diagrama nos permite representar gráficamente los valores de NBC a través de sus cuartiles, como lo podemos ver en la Fig. 7. El umbral para etiquetar los *hotspots* se seleccione de entre los valores de los cuantiles, de la media o la mediana.

Una vez que está definido el umbral para determinar si es *hotspot* se procede a etiquetar todos los datos para tenerlos listos para los diferentes experimentos que se ejecutarán. Cada aplicación tendrá tanto *hotspots* como no *hotspots*. Estos estarán intercalados y no tendrán ningún orden.

### 4.4.3 Tratamiento de los datos

Las métricas candidatas que puedan ser elegidas tienen diferentes escalas de valores. Algunas de ellas están en unidades, otras en decenas y algunas más en centenas o unidades de millar. Por ejemplo, la métrica *Crossing,* la cual mide el número de cruces en los que participa un archivo, su escala de valores está dada en unidades y la métrica *CLOC*, la cual mide el número de líneas físicas de código, su escala de valores está dada en centenas o unidades de millar.

Este tratamiento de homogeneización de los datos no se ha encontrado en ninguno de los trabajos revisados en el estado del arte, es decir, en todos los trabajos se han utilizado los valores de las métricas obtenidos con las diferentes técnicas y herramientas de cada trabajo.

Tratando de explorar algunas alternativas para que los datos tengan una escala común y puedan ser comparables, se propone aplicar algunas normalizaciones a los datos y revisar el comportamiento de los resultados considerando estas opciones. La normalización es un método de preprocesamiento. El preprocesamiento no es solo una técnica que se utiliza para convertir los datos sin procesar en un conjunto de datos limpio, sino que también mejora el rendimiento del aprendizaje automático [41]. Basándonos en lo anterior, se aplican los siguientes métodos:

- **Normalizar con máximo valor(nMáx)**. En este caso se normalizan los datos de las métricas seleccionadas con base en el valor más alto de cada una de ellas, de tal forma que se obtengan valores continuos entre cero y uno.
- **Normalizar con máximo valor de líneas de código (nLOC)**. La métrica de líneas de código es imprescindible en el subconjunto de métricas seleccionadas dado su aporte a la evaluación de los *hotspots* [23]. Supongamos que se seleccionó *CLOC* (tamaño de líneas de código físico), en este caso, se normalizan los datos de las métricas seleccionadas con respecto al valor más alto de la métrica *CLOC*. Con lo anterior, se obtiene una base común bajo la idea de que todos los archivos tienen el mismo tamaño. Para normalizar la base de datos se detecta el valor máximo de la métrica *CLOC* (vMax), posteriormente se obtiene un *factor de normalización por archivo* dividiendo vMax entre el valor CLOC de cada archivo y por último se multiplica el *factor de normalización por archivo* a cada uno de los valores de las métricas de ese archivo.

Con los dos conjuntos de datos normalizados (nMax y nLOC) y considerando el conjunto de datos original sin normalización (sNorm), se tienen tres conjuntos de datos para realizar los experimentos. Si a estos tres conjuntos de datos los analizamos con los dos algoritmos propuestos, tenemos seis experimentos a realizar para encontrar la combinación de métricas que nos ayude a identificar de mejor manera los *hotspots* en una aplicación.

En la Tabla 6 se presentan los seis experimentos que se pueden realizar. Tres de ellos para el algoritmo exhaustivo y tres de ellos para el clasificador Bayes ingenuo.

**Tabla 6 Experimentos propuestos para el estudio de detección de *hotspots*.**

| Id expermiento | Algoritmo | Normalización de datos |
|---|---|---|
| Exh (sNorm) | Exhaustivo | Ninguna |
| Exh (nMAX) | | Con máximo valor por métrica |
| Exh (nCLOC) | | Con máxima LOC en la base de datos |
| BI (sNorm) | Bayes Ingenuo | Ninguna |
| BI (nMAX) | | Con máximo valor por métrica |
| BI (nCLOC) | | Con máxima LOC en la base de datos |

De estos seis experimentos se seleccionará el que nos ofrezca la mejor combinación de métricas para la detección de *hotspots*. Junto con la combinación de métricas se obtendrá el algoritmo y la normalización de datos que resulte mejor en las medidas de rendimiento para la detección de *hotspots*.

### 4.4.4 Datos de entrenamiento y de prueba

Esta etapa de la metodología es de suma importancia, ya que se debe determinar la proporción de datos que se utilizará para el entrenamiento y para las pruebas del algoritmo de clasificación BI.

Para hacer la división de los datos se propone la técnica de validación cruzada de K iteraciones, también conocida como validación *K-fold cross validation* (FKCV por sus siglas en inglés). En esta técnica, los datos disponibles se dividen en K subconjuntos, uno de ellos se utiliza como datos de prueba y el resto (K-1) se utiliza como datos de entrenamiento. El proceso de validación se realiza K veces haciendo que cada vez varíe el subconjunto de prueba. Se toman todos los resultados y se saca la media aritmética para tener un resultado único [29].

La técnica FKCV permite que todos los datos se utilicen para obtener una estimación. Lo más común es que se desee estimar la precisión de un clasificador en un entorno de aprendizaje supervisado. En tal escenario, una cierta cantidad de datos etiquetados está disponible para entrenar al clasificador y posteriormente, con los datos no vistos (de prueba) probar el etiquetado que realice el clasificador. Usando una validación cruzada de 10 veces, uno usa repetidamente el 90% de los datos para construir un modelo y probar su precisión en el 10% restante. Es probable que la precisión promedio resultante sea algo subestimada para la precisión verdadera cuando el modelo se entrena con todos los datos y se prueba con datos no vistos, pero en la mayoría de los casos esta estimación es confiable. Particularmente si la cantidad de datos etiquetados es lo suficientemente grande y si los datos no vistos siguen la misma distribución que los ejemplos etiquetados [29].

Si tuviéramos $n$ aplicaciones para datos entrenamiento y pruebas, al aplicar la técnica FKCV tomaríamos cada aplicación como un subconjunto de datos. Si fuera $n = 10$, en la Fig. 8 se muestran las combinaciones de las 10 aplicaciones para entrenamiento y pruebas, en donde se tomaría una aplicación como datos de entrenamiento y las restantes nueve para pruebas. En total tendríamos 10 casos para aplicar el algoritmo de BI.

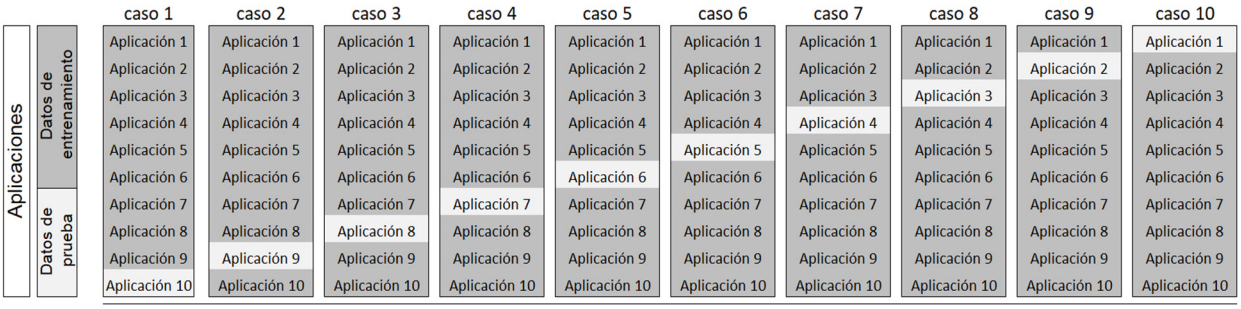



**Fig. 8 Ejemplo de casos de entrenamiento y prueba para el algoritmo BI.**

Se sugiere utilizar la técnica FKCV para la preparación de los datos de entrenamiento y prueba para la clasificación de BI con base en lo reportado en [16], donde concluyen que utilizar datos diferentes (no de las mismas aplicaciones) para entrenamiento que, para pruebas, favorece la clasificación. En nuestro caso, cada conjunto de datos de prueba será una aplicación, mientras que los datos de entrenamiento serán las demás aplicaciones, como lo podemos ver en cualquiera de los casos de la Fig. 8.

### 4.4.5 Medidas de rendimiento para evaluar algoritmos de clasificación

La validación de resultados obtenidos por los algoritmos en cada experimento se hace a través de las medidas de rendimiento más utilizadas recientemente en las investigaciones afines, como en [7], [16], [17], [19].

Las medidas de rendimiento que se considerarán son: exactitud, precisión, sensibilidad, especificidad y análisis ROC (del inglés Receiver Operating Characteristic) o área bajo la curva (AUC por sus siglas en inglés), las cuales reflejan la capacidad de clasificar los archivos como *hotspots*. Estás medidas de rendimiento han sido también utilizadas en [16], [19], [20], donde hacen uso de algoritmos de clasificación de aprendizaje automático para determinar archivos que sean propensos al cambio o al error.

Para poder calcular las medidas de rendimiento antes mencionadas, primero se calcula un matriz de confusión y posteriormente se obtienen los resultados de las medidas de rendimiento en función de esta.

#### Matriz de confusión

La matriz de confusión es una herramienta que permite visualizar el rendimiento de los clasificadores de aprendizaje automático. El principio de la matriz es contrastar los resultados reales contra los resultados del clasificador.

Se comparan las diferentes clases en las que se están clasificando los archivos. En nuestro caso tenemos dos clases, un cero (0) representa que un archivo no es *hotspot* y un uno (1) representa que es *hotspot*. La Fig. 9 muestra las cuatro combinaciones de resultados que se pueden obtener: TN (verdaderos negativos -no *hotspots* que son clasificados como no *hotspots*-), FP (falsos positivos -no *hotspots* clasificados como *hotspots*-), FN (falsos negativos -*hotspots* clasificados como no *hotspots*) y TP (verdaderos positivos -*hotspots* clasificados como *hotspots*).

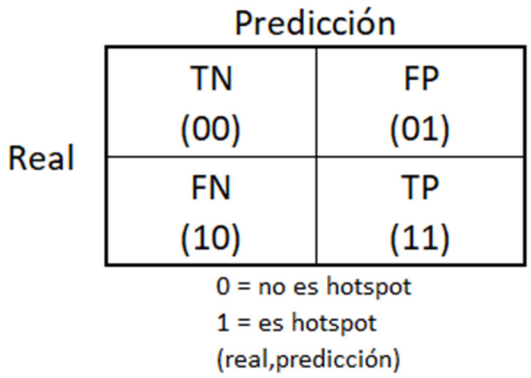



**Fig. 9 Matriz de confusión para evaluar los resultados de los algoritmos de clasificación.**

De manera general, se esperaría obtener valores altos en TN y TP y valores lo más cercanos a cero para FP y FN. Una vez que tenemos la matriz de confusión, en nuestro caso, calcularemos seis medidas de rendimiento para evaluar la clasificación. Estas medidas de rendimiento se presentan a continuación.

## Medidas de rendimiento

Las medidas de rendimiento con las que se evaluará el clasificador son precisión, exactitud, sensibilidad, especificidad, medida-f y área bajo la curva (AUC). Estás se han ocupado también en [16],[19], [20].

En la Tabla 7 se muestra el nombre de la *medida de rendimiento*, la *fórmula* con la que se calcula y su *interpretación.*

**Tabla 7 Medidas de rendimiento para evaluar clasificaciones.**

| Medida de rendimiento | Fórmula | Interpretación |
|---|---|---|
| **Precisión** (Precision) | $\frac{TP}{TP+FP}$ | Se define como la porción de archivos que se clasifican como hotspot (TP) y en realidad son hotspot |
| **Exactitud** (Accuracy) | $\frac{TP+TN}{TP+TN+FP+FN}$ | Es la proporción de archivos correctamente predichas como hotspot (TP) con respecto al número total de archivos |
| **Sensibilidad** (Recall) | $\frac{TP}{TP+FN}$ | Capacidad para determinar los verdaderos positivos (TP). |
| **Especificidad** | $\frac{TN}{TN+FP}$ | Capacidad para determinar los verdaderos negativos (TN). |
| **Medida-F** (f-measure) | $2 * \frac{Precisión * Exactitud}{Precisión + Exactitud}$ | Ponderación de precisión y exactitud. |
| **Análisis ROC** | Área bajo la curva (AUC)<br>Sensibilidad vs. 1 - Especificidad | Se busca un valor de área igual a 1. Se esperaría que la sensibilidad sea 1 y ls especificidad sea 1, de tal forma que tengamos un punto lo más cercano a la parte superior izquierda de la gráfica. |

Por cada combinación de métricas que sea evaluada, se obtienen los resultados de su matriz de confusión, posteriormente se calcula cada una de las medidas de rendimiento y se determina cuál combinación de métricas nos da los mejores resultados en ellas.

El AUC es el estadístico que proporciona una medida completa de la capacidad de clasificación de un instrumento o sistema diagnóstico (nuestro clasificador) [26]. También podemos decir que es la representación gráfica de la tasa de éxito (sensibilidad -eje x-) frente a la tasa de falsas alarmas (1 - especificidad -eje y-) para tareas de solo dos resultados (es *hotspot* o no es *hotspot*), según se varíe el umbral (Q3 de NBC), para determinar si es o no es *hotspot* [26]. Estas descripciones se muestran en la Fig. 10. Se deberá de buscar una clasificación que esté por arriba de la línea punteada (clasificación aleatoria) y lo más cercana a la esquina superior izquierda, en donde se dice que es una clasificación perfecta.

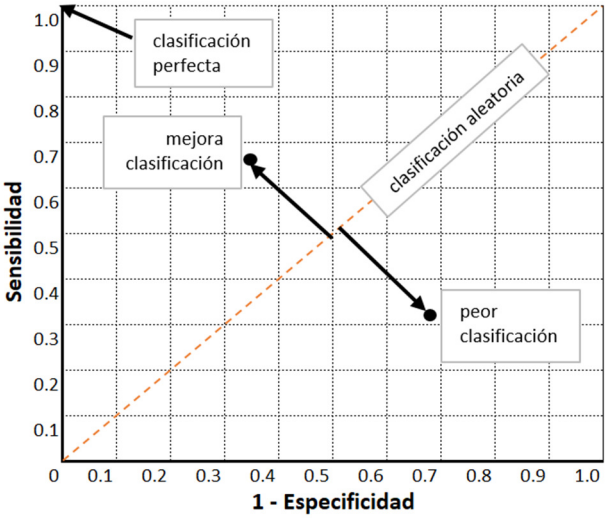



**Fig. 10 Elementos del análisis ROC (área bajo la curva).**

Se sugiere tomar el AUC como la medida de rendimiento para determinar la mejor combinación de métricas que clasifica los archivos en *hotspots*. Lo anterior, debido a que el AUC se basa en la sensibilidad y la especificidad (definidas en la Tabla 7), las cuales evalúan la capacidad de detectar los verdaderos positivos (*hotspots* reales), así como, los verdaderos negativos (no *hotspots* reales) respectivamente. Se considera que es tan importante poder predecir un *hotspot* (sensibilidad) como un no *hotspot* (1 - especificidad), de tal forma que si tomamos el AUC podremos evaluar ambas en una sola medida de rendimiento [25].

También en [39] puntualizan que el AUC es una medida de rendimiento con mayor alcance para evaluar los algoritmos de clasificación y debería ser usada con mayor frecuencia para este fin.

## 4.5 Análisis de resultados y selección de métricas finales

El análisis de los resultados de cada experimento se realiza con los siguientes pasos:

1. Se obtienen los resultados de las medidas de rendimiento al clasificar los *hotspots* en cada una de las aplicaciones por combinación de métricas.
2. Se determina cuál es la combinación de métricas que ofrece el mejor AUC promedio.
3. Se comparan los mejores resultados de cada experimento (algoritmo, normalización de datos y combinación de métricas) y se determina cuál de ellos es el que tiene los mejores resultados (en AUC).
4. El experimento seleccionado en el paso anterior es el que se sugiere como el que puede ayudar a los equipos de desarrollo para determinar los *hotspots* de una aplicación siguiendo esta metodología. Si además este tiene una AUC por arriba de 0.825, estaremos ante un mejor resultado que los obtenidos en los diferentes trabajos que se revisaron [19], [20].

Lo que se obtiene como resultado es:

- Un algoritmo para evaluar una aplicación e identificar potenciales *hotspots*.
- Una técnica de normalización de datos.
- Una combinación de métricas a utilizar.

Como consecuencia del análisis de resultados de los experimentos se obtiene una combinación a la que llamaremos *métricas finales*.

## 4.6 Validación de resultados

Con el resultado obtenido del paso anterior, se procede a validar el comportamiento de las *métricas finales* para clasificar los *hotspots* en otras aplicaciones. Estás aplicaciones se deben de reservar y no ser parte de los experimentos anteriores.

Para la validación de resultados, se realiza lo siguiente:

- Preparar los datos de prueba con las métricas finales de las aplicaciones reservadas.
- Preparar el algoritmo de BI con el método de normalización seleccionado.
- Preparar los datos de entrenamiento con las métricas finales, los cuales serán todos los datos que se utilizaron en la etapa de entrenamiento y pruebas de la sección 4.4.4.
- Ejecutar la clasificación de los *hotspots* con el algoritmo de BI.

Los resultados obtenidos de la clasificación de *hotspots* en las aplicaciones de validación se deberán comparar con los obtenidos en la etapa de entrenamiento y pruebas. Si los valores de AUC en este paso son mayores o iguales, estaríamos en posibilidades de confirmar que nuestra combinación de métricas finales y la normalización de datos seleccionado son favorables para la detección de *hotspots*. En caso contrario, estaríamos rechazando nuestra propuesta.

# Capítulo 5

# Implementación de la metodología

En este capítulo se presentará la implementación de la metodología para identificar un conjunto de métricas que permitan detectar *hotspots* en los sistemas. Se detalla cada una de las etapas de ella y las decisiones que se fueron tomando hasta llegar a la etapa de la experimentación y obtener los resultados, los cuales se analizan al final del capítulo.

## 5.1 Aplicaciones seleccionadas para el estudio

Después de realizar una búsqueda de aplicaciones que pudieran cumplir con las características sugeridas, se eligieron once aplicaciones para la ejecución de los diferentes experimentos planteados. Estas aplicaciones cumplieron con las siguientes características: son de código abierto, se les dan mantenimiento por parte de *the Apache Software Foundation*, están codificadas con el lenguaje de programación orientado a objetos Java, su código fuente está disponible en el repositorio colaborativo GitHub, hacen uso del sistema de control de versiones (SCV) Git, utiliza el sistema de control de incidentes (SCI) JIRA, han tenido como mínimo 1000 *bug commits* [23] y, dado que algunas HAC utilizan el código compilado para realizar su análisis, se tuvo que generar éste a partir del código fuente que se obtuvo del repositorio de Git.

**Tabla 8 Aplicaciones elegidas para el estudio.**

| Aplicación | Versión | Cantidad de Archivos | Líneas de código | Número de commits | Periodo | URL de descaga en github |
|---|---|---|---|---|---|---|
| Amqp-0x | 6.3.4 | 362 | 104181 | 1139 | 2016-01-25 - 2019-05-14 | https://github.com/apache/qpid-jms-amqp-0-x |
| Broke-j | 7.1.6 | 1865 | 321853 | 1148 | 2017-04-17 - 2019-12-02 | https://github.com/apache/qpid-broker-j |
| Deltaspike | 1.9.2 | 742 | 147125 | 1266 | 2014-03-16 - 2019-12-13 | https://github.com/apache/deltaspike |
| Flume | 1.9.0-rc3 | 441 | 80266 | 1082 | 2012-02-14 - 2018-12-17 | https://github.com/apache/flume |
| Hbase | 1.4.12 | 1786 | 439093 | 1088 | 2017-01-04 - 2019-11-20 | https://github.com/apache/hbase |
| Knox | 1.3.0 | 863 | 82549 | 1098 | 2016-01-30 - 2019-06-29 | https://github.com/apache/knox |
| Nifi | 1.10.0 | 4112 | 629298 | 1051 | 2018-05-03 - 2019-10-28 | https://github.com/apache/nifi |
| Oozie | 5.2.0 | 812 | 151221 | 1059 | 2014-05-20 - 2019-10-29 | https://github.com/apache/oozie |
| Sentry | 2.2.0 | 561 | 34768 | 1006 | 2015-02-18 - 2020-01-02 | https://github.com/apache/sentry |
| Tajo | 0.12.0-rc0 | 1557 | 265253 | 1095 | 2014-03-11 - 2019-09-11 | https://github.com/apache/tajo |
| Tez | 0.9.2 | 839 | 159251 | 1082 | 2015-05-18 - 2019-03-19 | https://github.com/apache/tez |

Estas aplicaciones se presentan en la Tabla 8, donde tenemos por cada aplicación el nombre con el que se hará referencia a cada una de ellas en el resto del trabajo, la versión que

fue utilizada para obtener el código fuente, la cantidad de archivos fuente que tiene cada aplicación, número de líneas de código, el número de *commit* ejecutados de los cuales mínimo debería de haber 1000 *bug commits*, el periodo analizado y la URL del SCV de donde se puede obtener los códigos fuentes.

Cabe mencionar que las bases de datos de las once aplicaciones utilizadas para esta investigación son retomadas del trabajo realizado por J. Lefever *et al.* [23], previa autorización por parte de ellos. Este trabajo aborda el problema de *On the Lack of Consensus Among Technical Debt Detection Tools* y concluyen que una necesidad de que la comunidad cree puntos de referencia y conjuntos de datos para evaluar la precisión de las herramientas de análisis de código en términos de medidas de uso común. Lo anterior se debe a que existe una discrepancia entre los resultados que ofrecen unas herramientas y otras en la detección de los archivos problemáticos pretendiendo ubicar la DT en los sistemas. Se contrasta en la sección 5.5 Análisis de resultados parte de las conclusiones [23] y de los resultados de esta investigación.

## 5.2 Herramientas de análisis de código seleccionadas

Dentro de los criterios que cumplieron las herramientas de análisis de código (HAC) para ser seleccionadas, tenemos que, dentro de sus objetivos principales estuvieron el detectar la deuda técnica, que pudieron analizar aplicaciones codificadas en Java, que recibieron código fuente o compilado, que admitieron (opcionalmente) las bitácoras de los SCV y SCI para obtener la métrica de *bug commit*, que las salidas de todas ellas las entregarán a nivel archivo en formato estructurado, abierto y no propietario (por ejemplo CSV, XML, JSON, entre otros) para preparar las entradas para los algoritmos.

**Tabla 9 Herramientas seleccionadas para la obtención de métricas disponibles.**

| Herramienta de análisis de código (HAC) | Entradas | Tipo de métricas de salida | URL de la HAC |
|---|---|---|---|
| **Archinaut (ARCH)** [30] | Código fuente<br>Bitácoras de SCV | Estructurales e históricas | [30] |
| **Designite (DES)** [31] | Bitácoras de SCV | Estructurales<br>(Identificación de "design smells") | https://www.designite-tools.com/ |
| **Succinct Code Counter (SCC)** [32] | Bitácoras de SCV | Estructurales<br>(Contador de código y complejidad) | https://github.com/boyter/scc |
| **DV8 (DV8)** [33] | Código fuente<br>Bitácoras de SCV<br>Bitácoras de SCI | Estructurales e históricas<br>(Identificación de problemas de arquitectura) | https://archdia.com/ |
| **SonarQube (SQ)** [34] | Código compilado | Estructurales (código "smell") | https://www.sonarqube.org/ |

Después de hacer la evaluación de las HAC, se eligieron cinco de ellas, las cuales son presentadas en la Tabla 9. De estas HAC elegidas, se obtuvieron una gama de métricas tanto estructurales como históricas de las cuales, se seleccionarán más adelante, cuáles de ellas se ocupan para los siguientes pasos. Structure101 fue una de las HAC que fue descartada porque varias de las métricas que ofrece se duplicaban con las ofrecidas por otras aplicaciones por ejemplo tamaño, complejidad, entre otras, además de que, en [23] sus métricas de Structure101 son evaluadas por debajo de las ofrecidas por otras herramientas.

A continuación, describimos cada una de las herramientas seleccionadas (algunas de ellas existentes en el mercado, otras son proyectos de investigación y unas más de código abierto) y se mencionan algunas de las métricas que se obtienen de ellas.

**Archinaut** [30] (ARCH): Archinaut toma varias fuentes de datos como entrada, incluida la información de códigos fuentes y las bitácoras de SCV de la aplicación a analizar. Archinaut produce varias medidas a nivel de archivo y puede identificar *hotspots* (bajo su definición de *hotspot*, la cual es la misma a la de este trabajo) mediante el análisis de tendencias en las métricas. En particular, para cada archivo, calcula *fan-in* (varios archivos llaman a un archivo), *fan-out* (un archivo llama a varios archivos), TotalDeps (la suma de *fan-in* y f*an-out*), así como, CoChangePartners (el número de otros archivos con los que el archivo de destino a cocambiado, según se registra en las bitácoras del SCV). Además, Archinaut es capaz de integrar resultados de otras herramientas. Compila resultados en un solo archivo CSV [23].

**Designite** [31] (DES): Designite toma el código fuente como entrada e identifica los malos olores (*code smells*) en los niveles de implementación, diseño y arquitectura. Debido a que estamos comparando métricas a nivel de archivo, no se consideran los olores de implementación, ya que identifican problemas dentro de métodos individuales. Designite identifica 20 tipos de

olores de diseño y 7 tipos de olores de arquitectura. Los olores de diseño se informan a nivel de archivo. La regla de modularización cíclicamente dependiente detecta ciclos entre archivos. Se crea una medida agregada, DesignSmells, que es la suma de los olores de diseño informados para un archivo determinado [23].

**Succinct Code Counter** [32] (SCC): El contador de código sucinto es una herramienta para medir el tamaño y la complejidad del código por cada archivo. Por sus criterios para discriminar comentarios o líneas en blanco su conteo de código es exacto. SCC utiliza una pequeña máquina de estado para determinar en qué estado se encuentra el código cuando llega a una nueva línea. Como tal, es consciente y capaz de contar: comentarios de una sola línea, comentarios de varias líneas, instrumentos de cuerda, cadenas de varias líneas y líneas en blanco. Debido a esto, puede determinar con precisión si un comentario está en una cadena o es realmente un comentario. También intenta contar la complejidad del código. Esto se hace comprobando las operaciones de bifurcación en el código.

**DV8** [33] DV8 recibe como entradas la bitácora de SCV, la bitácora de SCI y el código fuente de la aplicación a analizar. DV8 realiza el cálculo del nivel de desacoplamiento y el costo de propagación, la identificación de raíces arquitectónicas y la identificación de antipatrones de diseño [23]. Nuestro estudio considera los seis antipatrones: *Clique* (archivos que forman un componente fuertemente conectado), *Unhealthy Inheritance* (violaciones del principio de sustitución de Liskov[22]), *Package Cycle* (dos carpetas dependen mutuamente entre sí), *Unstable Interface* (un archivo con muchos dependientes que cambia a menudo con todos ellos), *Crossing* (un archivo con un gran número de archivos que dependen de él y con una dependencia de él con muchos otros archivos) y *Modularity Violation* (archivos que cambian juntos con frecuencia, pero no tienen una relación estructural). Además, ofrece la métrica de TargetChangeCount (número de veces que aparece un archivo en un *commit* asociado a un error -*bug commit*-).

**SonarQube** [34] (SQ): SonarQube es una herramienta de revisión automática de código de grado industrial ampliamente utilizada y reconocida, que define un vasto conjunto de reglas de calidad. SonarQube toma como entrada el código fuente y código compilado de la aplicación a analizar. SonarQube para proyectos Java, define reglas en 4 categorías: 377 reglas de olor de código, 128 reglas de errores, 59 reglas de vulnerabilidad y 33 reglas de puntos de acceso de seguridad. No es posible analizar las métricas de cada regla individual ya que los datos son

[22] Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

demasiado escasos, por lo tanto, se suman todas las métricas en SQ Issues. SonarQube también calcula el tamaño y la complejidad, pero no proporciona estos números a nivel de archivo.

Con las HAC procederemos a la elección de las métricas candidatas para buscar en ellas una combinación que nos ayude a identificar los *hotspots* en las aplicaciones.

## 5.3 Métricas candidatas

Derivado del análisis realizado a las once aplicaciones elegidas con las HAC se presenta a continuación en la Tabla 10 las métricas disponibles que se pudieron obtener.

**Tabla 10 Métricas disponibles obtenidas con las HAC de las once aplicaciones.**

| Nombre | Heramienta | Descripción |
|---|---|---|
| DES_Size | DES | Tamaño en líneas de código |
| DES_Complexity | DES | Compeljidad |
| DES_Design Smells | DES | Diseño oloroso |
| SCC_LOC | SCC | Tamaño en líneas de código |
| SCC_CLOC | SCC | Tamaño en Lineas de Código fisico (LOC) |
| SCC_COMPLEXITY | SCC | Complejidad |
| ARCH_Revisions | ARCH | Número de commits en los que aparece un archivo |
| ARCH_Dependent Partners | ARCH | Número de archivos que dependen de este archivo (fan in) |
| ARCH_Depends on Partners | ARCH | Número de archivos de los que depende este archivo (fan out) |
| ARCH_Total Dependencies | ARCH | Total de fan in + fan out |
| ARCH_CoChange Partners | ARCH | Número de archivos que cambiaron al mismo tiempo que el archivo en uno o más "commits" |
| DV8_LOC | DV8 | Tamaño en líneas de código |
| DV8_Target Change Count | DV8 | Número de veces que aparece un archivo en un commit asociado con un error ("bug commit") |
| DV8_Target Churn | DV8 | Rotación para todos los "bug commits" en los que aparece el archivo |
| DV8_Change Count | DV8 | Número de veces en los que un archivo aparece en un commit |
| DV8_Change Churn | DV8 | Rotación para todos los commits en los que aparece el archivo |
| DV8_Total Is sues | DV8 | Número total de problemas de DV8 para los archivos |
| DV8_Clique | DV8 | Número de componentes fuertemente conectado en las que participa este archivo |
| DV8_Crossing | DV8 | Número de cruces (fan in o fan out) en los que aparece este archivo |
| DV8_Modularity Violation | DV8 | Número de violaciones a la modularidad en las que aparece este archivo |
| DV8_Package Cycle | DV8 | Número de ciclos de paquetes en los que participa este archivo |
| DV8_Unhealthy Inheritance | DV8 | Número de jerarquías de herencias en mal estado en las que participa este archivo |
| DV8_Unstable Interface | DV8 | Número de interfaces inestables en las que participa este archivo |
| DV8_Present In Issues | DV8 | ¿En cuántos errores de las herramientas este archivo está presete? |
| SQ_Issues | SQ | Número de problemas de SonarQube (Issues + CodeSmells + Bugs + Vulnerabilities) |
| SQ_Code Smells | SQ | Número de olores de código de SonarQube |
| SQ_Bugs | SQ | Número de errores de SonarQube |
| SQ_Vulnerabilities | SQ | Número de vulnerabilidades de SonarQube |
| SQ_Security Spots | SQ | Número de puntos de seguridad de SonarQube |

A partir de la lista de métricas disponibles, se eligió el conjunto de *métricas candidatas* que se utilizaron en la ejecución de los experimentos de identificación de *hotspots*. Para ello se

aplicó una simplificación sobre las 29 métricas disponibles de las HAC. La simplificación de las métricas estuvo conducida por los siguientes criterios:

- Métricas que representan el tamaño de los archivos, es decir, número de líneas. La HAC que se utilizó como referencia dado que es una de sus principales virtudes fue SCC. Esta misma métrica la ofrece DES y DV8 (DES Size y DV8 LOC), pero no es confiable como lo comprobaron en [23], dado que dan resultados diferentes para un mismo archivo analizado.
- Métricas que representan la complejidad del código. En ésta, también se tuvo más de una opción, sin embargo, varias herramientas a menudo informan números drásticamente diferentes, lo que significa que los archivos más complejos informados por una herramienta pueden no ser los más complejos según otra herramienta. De ahí que se eligió sólo la métrica de complejidad de DES ya que en [23] es reportada como una de las métricas que mejor se correlaciona con otras (también consideradas en este estudio) y coincide con otras métricas en reportar archivos con potencial DT.
- Métricas que representan ciclos de paquete. Para estas métricas no fue muy diferente la situación de cómo fue para las métricas de tamaño y complejidad. Existe variación entre lo reportado por una herramienta que por otra. Se eligió la métrica de DV8.
- Consolidación de métricas. En algunos casos lo que se hizo fue consolidar en una sola métrica la suma de los valores de varias de ellas. Este es el caso de las métricas de SQ donde su análisis es bastante detallado y a veces los datos son muy escasos por lo que se decidió sumarlas en una sola. Otro caso es el de ARCH para las métricas de *fan-in* y *fan-out* se consolidaron en una sola a la que se llamó *Total de dependencias*.

Una vez aplicada la simplificación se obtienen 17 métricas candidatas a utilizar, las cuales se muestran en la Tabla 11. En esta, la primera columna es el identificador de la métrica que utilizaremos al mostrar los resultados; la segunda columna contiene la clave compuesta de la métrica que nos indica la herramienta de procedencia, la métrica y para algunas, un indicador *(h)* que nos dice si la métrica es obtenida del histórico de SCV y en la tercera, tenemos la descripción de lo que representa la métrica.

**Tabla 11 Métricas candidatas para el estudio.**

| Id Métrica | Clave de Métrica | Descripción |
|---|---|---|
| M01 | DES_Design Smells | Diseño oloroso |
| M02 | SCC_CLOC | Tamaño en Lineas de Código fisico (LOC) |
| M03 | SCC_COMPLEXITY | Complejidad |
| M04 | ARCH_Dependent Partners | Número de archivos que dependen de este archivo (fan-in) |
| M05 | ARCH_Depends on Partners | Número de archivos de los que depende este archivo (fan-out) |
| M06 | ARCH_Total Dependencies | Fan in + Fan out |
| M07 | ARCH_CoChange Partners(h) | Número de archivos que cambiaron al mismo tiempo que el archivo en uno o más "commits" |
| M08 | DV8_TotalIssues | Número total de problemas de DV8 para los archivos |
| M09 | DV8_Clique | Número de componentes fuertemente conectado en las que participa este archivo |
| M10 | DV8_Crossing(h) | Número de cruces en los que participa este archivo |
| M11 | DV8_ModularityViolation(h) | Número de violaciones a la modularidad en las que participa este archivo |
| M12 | DV8_PackageCycle | Número de ciclos de paquetes en los que participa este archivo |
| M13 | DV8_UnhealthyInheritance | Número de jerarquías de herencias en mal estado en las que participa este archivo |
| M14 | DV8_UnstableInterface(h) | Número de interfaces inestables en las que participa este archivo |
| M15 | DV8_PresentInIssues | ¿Cuántos tipos distintos de los problemas que identifica DV8 tiene? |
| M16 | SQ_Issues | Número de problemas de SonarQube |
| M17 | DV8_TargetChangeCount o número de "bug commits" **(NBC)** | Número de veces que aparece un archivo en un commit asociado con un error ("bug commit") |

Las primeras dieciséis métricas son las utilizadas para el estudio, dentro de las cuales se trata de encontrar una combinación que mejor nos clasifique los *hotspots* en las aplicaciones. La diecisieteava métrica, DV8 Target ChangeCount representa el número de *bug commits* por lo que en lo sucesivo nos referiremos a ella por NBC (Number of *Bug Commits*). Esta se utilizó para el etiquetado de los datos de entrenamiento y pruebas antes de cada experimento y poder validar los resultados del clasificador BI. Esta métrica se detalló en la sección 4.4.2 en la que se puntualizó que de existir la disciplina de los desarrolladores para relacionar un incidente con el *commit* que lo reparó, estaríamos ante el *ground truth* para la detección de *hotspots*.

Dentro del conjunto de *métricas candidatas* tenemos tanto métricas estructurales como métricas históricas. En cualquiera de los dos grupos de métricas, a valores altos en ellas, nos indica alta probabilidad de cambio o de error. Por ejemplo, si tuviéramos un archivo con un valor alto en la métrica M02 (número de líneas código) la probabilidad de que presente un error o un cambio es mayor al de un archivo (de la misma aplicación) con un número menor de líneas [23].

Una vez aplicados los pasos de la metodología de selección de aplicaciones, selección de herramientas de análisis de código y la selección de métricas a utilizar, ya se tenían las entradas para realizar la ejecución de los experimentos para la identificación de *hotspots*.

## 5.4 Identificación de *hotspots*

Para realizar la identificación de *hotspots*, es decir, la clasificación de archivos del sistema en *hotspots* o *no hotspots*, se procedió a elegir los algoritmos de clasificación. Como lo vimos en la sección 2.2.4, dentro de las ramas de la IA tenemos la de resolución de problemas por restricción de satisfacción y de aprendizaje automático. Estas fueron dos ramas seleccionadas dada su posibilidad de tener algoritmos de clasificación.

Se eligieron algoritmos que, basándose en la información disponible, pudieran distinguir entre los posibles estados para clasificar los archivos de un sistema en *hotspots* o no. Como se presentó en la sección 5.3, dentro de las métricas candidatas tenemos una (NBC) que nos ayudó a validar los resultados de los algoritmos de clasificación supervisada.

### 5.4.1 Algoritmos propuestos

Como parte de la experimentación se seleccionaron los siguientes algoritmos de clasificación:

- **Algoritmo genético**. Se planteó como un problema de optimización en el que con este algoritmo se aplicó una búsqueda ascendente estocástica en la que se mantiene una gran cantidad de soluciones locales con posibilidad de llegar a una solución óptima [17], [40]. La representación del problema en este caso fue que, cada cromosoma era uno de los archivos del sistema y cada gen eran las métricas candidatas (dieciséis) que se tienen de cada archivo. El resultado obtenido en la medida de rendimiento de exactitud al clasificar las nueve aplicaciones de prueba fue en promedio del 58%. Dado este resultado se decidió elegir otro algoritmo de clasificación.
- **Algoritmo de búsqueda exhaustiva**. Dada la incertidumbre de los resultados con el algoritmo anterior, se propuso utilizar un algoritmo exhaustivo que explorara todas las posibilidades. Esto fue posible debido a que las métricas utilizadas fueron 16, los que nos dio $2^{16}$ $(6.5x10^4)$ combinaciones a evaluar y se obtuvieron resultados en minutos. Esto no sería posible computacionalmente si en lugar de utilizar 16 métricas, utilizáramos, por ejemplo, 30 métricas pues entonces tendríamos $2^{30}(1.07x10^9)$ combinaciones para evaluar lo que resultaría en tiempos de horas o días. En este trabajo de investigación se

consideró emplearlo para contrastar los resultados con el algoritmo de aprendizaje automático seleccionado.

- **Algoritmo de aprendizaje supervisado**. Dada la efectividad que manifiestan los algoritmos de aprendizaje automático [20] y con la posibilidad de poder validar el resultado de clasificación del algoritmo (con la métrica NBC), se planteó utilizar un algoritmo de aprendizaje automático supervisado. Por otro lado, es parte del objetivo de la investigación hacer uso de algún algoritmo de aprendizaje automático.

  Como lo podemos ver en la revisión de la literatura (Tabla 4, columna *algoritmo* en la sección 3.2), en otros trabajos se han utilizado diferentes algoritmos de aprendizaje automático [7], [20]. En este trabajo, se consideraron los algoritmos para la tarea de clasificación a: máquinas de soporte vectorial (MSV), árboles de decisión (C4.4 y C4.5), Bayes ingenuo (BI), entre otros. Basándose en los resultados obtenidos por [39], en el cual hacen una comparación de los tres algoritmos y en el que concluyen que los tres obtienen resultados muy similares, por tanto, en este estudio se propone el uso del algoritmo de BI. Otras razones adicionales son:

  - Porque sólo requiere una pequeña cantidad de datos de entrenamiento (si el conjunto de aplicaciones elegidas y métricas candidatas son reducidas) para seleccionar los parámetros estimados necesarios en el proceso de clasificación. Debido a que se supone que es una variable independiente, solo se necesita la varianza basada en una variable en una clase para determinar la clasificación [27].
  - Porque los tiempos de entrenamiento son excepcionalmente rápidos, su rendimiento es aceptable, es fácil de implementar y los recursos computacionales requeridos son mínimos [27].
  - Porque es un algoritmo que sigue vigente y se continúa ocupando en diversos trabajos de investigación que requieren de clasificadores [27].

#### 5.4.1.1 Algoritmo clasificador de Bayes ingenuo

La estructura básica y el funcionamiento de los algoritmos de aprendizaje automático es diferente de las técnicas estadísticas tradicionales. La estadística tradicional desarrolla modelos (estocásticos) que permiten ajustar los datos y hacer inferencia. Mientras que el aprendizaje

automático desarrolla métodos computacionales que utilizan la experiencia (datos disponibles) para mejorar el rendimiento o hacer predicciones precisas, combinando conceptos fundamentales de la informática con ideas de estadística, probabilidad y optimización. Estos algoritmos se han aplicado con éxito a la ingeniería de software, como a la predicción de defectos, la predicción del esfuerzo y la predicción de la capacidad de mantenimiento [20], [40].

Bayes ingenuo es un algoritmo de clasificación para calcular la probabilidad mediante el cálculo de la frecuencia y combinación de valores en un dato. Se basa en el teorema de Bayes, el cual fue propuesto por el científico británico Thomas Bayes en 1973 (publicación póstuma) [27].

La clasificación bayesiana es una clasificación estática que minimiza la probabilidad de clasificación errónea. El algoritmo de BI crea un modelo probabilístico basado en el teorema de Bayes. En este algoritmo, cada una de las características (métricas en nuestro contexto) se considera de forma independiente. Solo requiere un pequeño conjunto de datos de entrenamiento para la clasificación y que los atributos sean razonablemente independientes [26].

El algoritmo de BI emplea un modelo que conecta atributos (métricas) con una clase (*hotspot* o no *hotspot* en nuestro problema) e incorpora conocimientos a priori para que la inferencia bayesiana pueda utilizarse y derivar la distribución a posteriori. La representación matemática del teorema de Bayes está definida por la Ec. (1), en la que se obtiene el valor más probable de clase a asignar a un archivo $\boldsymbol{p(Cj|M1,...,Mn)}$, basándose en el producto de la probabilidad $p(Cj)$ dada una clase $Cj$ por la productoria de las probabilidades condicionadas $p(Mi|Cj)$ de cada métrica $Mi$ dada una clase $Cj$ [27].

$$\boldsymbol{p(Cj|M1,...,Mn)} = p(Cj)\prod_{i=1}^{n} p(Mi|Cj) \quad (1)$$

El algoritmo de BI de Gauss se utiliza a menudo cuando se trata de datos continuos que tienen valores de clase asociados con una distribución normal. Los valores que se obtienen de las métricas de cada archivo tienen la característica de ser continuos. Por lo que la probabilidad condicionada $p(Mi|Cj)$ estará dada por la Ec (2) [27].

$$\boldsymbol{p(Mi|Cj)} = \frac{1}{\sqrt{2\pi\sigma_{Cj}^2}} * e^{-(Mi - \underline{\mu_{Cj}})^2/(2\sigma_{Cj}^2)} \quad (2)$$

Donde:

- $Cj$ son las clases que serán asignadas a cada archivo ($C1$=*hotspot* y $C0$=no *hotspot*)
- $Mi$ son los valores de las diferentes métricas que se están evaluando del archivo que se está clasificando
- $\mu_{Cj}$es la media de la clase $Cj$
- $\sigma_{Cj}^2$ es la desviación estándar de la clase $Cj$.

Para cada archivo se realiza la evaluación de la probabilidad condicional por clase: $p(C_j|M_1,...,M_n)$ y se clasificará como *hotspot* cuando $p(C_1|M_1,...,M_n) > p(C_0|M_1,...,M_n)$, en otro caso se clasificará como *no hotspot.*

#### 5.4.1.2 Algoritmo de búsqueda exhaustiva

Para contrastar los resultados del clasificador BI, se propone el uso de un algoritmo exhaustivo. Éste evalúa todas las combinaciones de métricas posibles dada la lista de *métricas candidatas*. Con las 16 métricas candidatas tendremos $2^{16}$(65,536) combinaciones a evaluar y obtener la combinación de métricas que obtenga mejores medidas de rendimiento en la elección de *hotspots*. Para este caso, donde tenemos 16 métricas, el tiempo de ejecución es razonable.

La implementación del algoritmo exhaustivo se realizó de la siguiente manera. Con la combinación de métricas en turno $CM$, se evalúa cada archivo del conjunto de datos. La evaluación se aplica con la función $Fcm$ descrita en la Ec. (3), en la que se suman los valores de cada una de las métricas que se está utilizando en la combinación $CM$.

$$Fcm\ (CM, f) = \sum_{m=1}^{16} (Valor(f, m)\ *\ CM(m)) \quad (3)$$

Una vez que son evaluados todos los archivos con la función $Fcm$, se ordenan de manera descendente. Se selecciona del mayor al menor el mismo número de archivos que fueron etiquetados como *hotspot* al inicio. Estos archivos seleccionados son los *hotspots* resultantes.

Con las métricas candidatas elegidas para el clasificador de BI y aplicando el algoritmo exhaustivo para cada combinación de métricas, se obtendrá la que ofrezca las mejores medidas de rendimiento, en específico el AUC [25]. Esto con el fin de poder contrastar los resultados del clasificador de BI.

### 5.4.2 Etiquetado de los datos de entrenamiento

El umbral que se consideró para etiquetar un archivo como *hotspots* fue el valor del tercer cuartil del diagrama de caja (como se explicó en la sección 4.4.2), es decir, serán etiquetados como *hotspots* todos los archivos que su NBC sea mayor o igual al valor del cuartil Q3 en el diagrama de caja para cada aplicación.

**Tabla 12 Análisis de la métrica NBC para definir el umbral para etiquetar a los archivos como *hotspot*.**

| Aplicación | Cantidad de archivos por aplicación | Archivos con NBC >= Q3 (Hotspots sugeridos) | % de Hotspot | Q3 de NBC | Máximo atípico NBC | Mediana (moda) NBC | Media NBC |
|---|---|---|---|---|---|---|---|
| **Deltaspike** | 742 | 28 | 3.8% | **2** | 12 | 1 | **2** |
| **Flume** | 441 | 58 | 13.2% | **4** | 25 | 2 | **3** |
| **Hbase** | 1,786 | 129 | 7.2% | **2** | 33 | 1 | **2** |
| **Knox** | 863 | 23 | 2.7% | **4** | 11 | 2 | **3** |
| **Nifi** | 4,112 | 137 | 3.3% | **2** | 15 | 1 | **1** |
| **Oozie** | 812 | 98 | 12.1% | **4** | 21 | 1 | **3** |
| **Sentry** | 561 | 12 | 2.1% | **3** | 10 | 1 | **2** |
| **Tajo** | 1,557 | 78 | 5.0% | **3** | 20 | 1 | **2** |
| **Tez** | 839 | 90 | 10.7% | **3** | 33 | 2 | **3** |
| **Amqp-0x** | 561 | 12 | 2.1% | **1** | 1 | 1 | **1** |
| **Broke-j** | 1,865 | 92 | 4.9% | **1** | 1 | 1 | **1** |
| Total | 14,139 | 757 | **5.4%** | | | | |

Se propuso el valor del cuartil Q3 como el umbral para etiquetar a los archivos como *hotspots* por tres razones:

- El valor del cuartil Q3 es mayor o igual a la *media* de NBC en la mayoría de las aplicaciones que se consideran para el análisis, como lo podemos ver en la Tabla 12. Esto nos indica que si tomamos el valor del cuartil Q3 (vQ3) como el umbral

para etiquetar a los archivos como *hotspot,* estaremos considerando a todos los archivos que en promedio han participado vQ3 veces en un *bug commit*.

- Si la *media* se aleja del cuartil Q3 y se acerca a la *mediana,* estaríamos frente a una distribución central de los datos, es decir, su desviación estándar es pequeña. Esto implica que tendríamos pocos valores atípicos (máximos o mínimos). Aún en este caso, si se tomara el umbral de NBC en el cuartil Q3, se estarían considerando a un conjunto reducido de archivos como *hotspots*. Y esto sería consistente con [21] donde concluyen que solo uno pocos archivos se vuelven *hotspots* dominantes y persistentes.
- Como vemos en la Tabla 12, por aplicación, hay archivos con máximo atípico muy grande, lo que sugiere que estos archivos deben ser etiquetados necesariamente como *hotspots* para ser evaluados por los algoritmos. Estos valores atípicos pueden indicar que fueron en algún momento *hotspot* o que están por serlo. Al seleccionar el cuartil Q3 como umbral, estos valores atípicos están incluidos en el análisis de *hotspots.*

Por ejemplo, en la Fig. 7 de la sección 4.4.2, que representa el diagrama de cajas de sus valores de NBC de la aplicación Tez, tenemos que el valor del cuartil Q3 es 3, entonces los archivos que serían etiquetados como *hotspot* son los que tengan un valor de NBC mayor o igual a 3.

En la Tabla 12 se resume el umbral del NBC (columna *Q3 de NBC*) que se tomó de cada aplicación para considerar el conjunto de archivos *hotspots*. También podemos ver, que de la cantidad total de archivos (13,940) se están sugiriendo como *hotspot* un total de 757, lo que equivale al 5.4%. Este es el porcentaje de archivos que los algoritmos deberán de clasificar como *hotspots*.

### 5.4.3 Tratamiento a los datos de las aplicaciones seleccionadas

A continuación, se muestran los datos en las tres representaciones en las que fueron utilizados para los experimentos. Primero, mostramos un ejemplo de datos sin normalizar (sNorm). Después mostramos un ejemplo de datos normalizados por máximo de cada métrica (nMáx). Por último, un ejemplo de datos normalizados por máximo de líneas de código (nLOC).

En general las tablas de datos que mostraremos para ejemplificar el tratamiento de los datos tendrán la siguiente estructura. Primero tenemos la columna #File que representa el identificador de cada archivo, las columnas *M01* a la *M16* representan las dieciséis métricas candidatas y por último la columna *M17,* la cual representa la métrica NBC que es el número de *commits* asociados a *bugs* y que se utilizó para etiquetar los archivos como *hotspots* antes de ejecutar los algoritmos.

**Datos sin normalizar (sNorm).**

**Tabla 13 Ejemplo de datos sin normalizar de la aplicación Tez.**

| #File | M01 | M02 | M03 | M04 | M05 | M06 | M07 | M08 | M09 | M10 | M11 | M12 | M13 | M14 | M15 | M16 | M17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 243 | 5 | 2326 | 302 | 6 | 92 | 98 | 56 | 36 | 1 | 4 | 6 | 11 | 10 | 4 | 6 | 159 | 33 |
| 358 | 4 | 3965 | 589 | 1 | 138 | 139 | 55 | 37 | 1 | 3 | 7 | 4 | 17 | 5 | 6 | 264 | 33 |
| 87 | 2 | 951 | 11 | 103 | 5 | 108 | 25 | 17 | 1 | 5 | 2 | 6 | 0 | 3 | 5 | 15 | 26 |
| 354 | 4 | 1543 | 149 | 4 | 67 | 71 | 35 | 21 | 1 | 2 | 4 | 4 | 6 | 4 | 6 | 68 | 23 |
| 608 | 0 | 623 | 61 | 2 | 32 | 34 | 13 | 5 | 0 | 0 | 3 | 0 | 1 | 1 | 3 | 5 | 19 |
| 342 | 4 | 2056 | 298 | 4 | 81 | 85 | 32 | 29 | 1 | 4 | 7 | 4 | 9 | 4 | 6 | 110 | 17 |
| 733 | 3 | 1165 | 194 | 3 | 25 | 28 | 16 | 14 | 1 | 0 | 5 | 2 | 4 | 2 | 5 | 77 | 16 |
| 15 | 5 | 892 | 100 | 24 | 27 | 51 | 16 | 18 | 1 | 1 | 6 | 4 | 4 | 2 | 6 | 56 | 16 |
| 355 | 3 | 1151 | 142 | 2 | 59 | 61 | 15 | 21 | 1 | 2 | 3 | 4 | 9 | 2 | 6 | 69 | 16 |
| 741 | 3 | 1174 | 183 | 1 | 24 | 25 | 14 | 11 | 1 | 0 | 4 | 2 | 2 | 2 | 5 | 73 | 14 |
| 729 | 3 | 1056 | 126 | 4 | 23 | 27 | 13 | 10 | 1 | 0 | 3 | 1 | 4 | 1 | 5 | 90 | 12 |
| 719 | 2 | 820 | 133 | 3 | 28 | 31 | 25 | 13 | 0 | 0 | 6 | 0 | 5 | 2 | 3 | 53 | 12 |
| 399 | 2 | 1863 | 375 | 0 | 11 | 11 | 22 | 7 | 0 | 0 | 5 | 0 | 1 | 1 | 3 | 147 | 11 |
| 664 | 4 | 542 | 65 | 4 | 14 | 18 | 0 | 3 | 0 | 0 | 1 | 0 | 2 | 0 | 2 | 10 | 11 |
| 746 | 3 | 1047 | 167 | 1 | 20 | 21 | 12 | 9 | 1 | 0 | 4 | 0 | 2 | 2 | 4 | 62 | 11 |
| 753 | 2 | 1141 | 197 | 2 | 25 | 27 | 11 | 8 | 0 | 0 | 4 | 0 | 2 | 2 | 3 | 68 | 10 |
| 590 | 3 | 304 | 23 | 20 | 14 | 34 | 9 | 9 | 1 | 4 | 1 | 1 | 2 | 0 | 5 | 19 | 10 |
| 467 | 1 | 658 | 54 | 1 | 28 | 29 | 13 | 5 | 0 | 0 | 3 | 0 | 1 | 1 | 3 | 4 | 10 |
| 836 | 1 | 16 | 4 | 2 | 2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 |
| 838 | 1 | 105 | 9 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 839 | 0 | 92 | 6 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Máximo** | **6** | **3965** | **589** | **103** | **138** | **139** | **56** | **37** | **1** | **5** | **12** | **11** | **17** | **5** | **6** | **264** | |

En esta representación, los datos son utilizados por los algoritmos directamente sin ningún tratamiento de normalización. La Tabla 13 es un ejemplo de los datos de la aplicación Tez. Esta está ordenada por la columna M17 (NBC) de manera descendente, con lo cual podemos observar que los archivos con el mayor valor NBC (33) son los que tienen los archivos 243 y 358. Como podemos ver en la Tabla 12, para la aplicación Tez el valor umbral de *Q3 de NBC* que se tomó para etiquetar a los archivos como *hotspots* o no, es mínimo de 3. Se fueron etiquetando como *hotspots* de manera descendente cada archivo hasta llegar al último con un valor de 3, el resto de los archivos fueron etiquetados como *no hotspot*. Los últimos en la Tabla 13 son aquellos que su NBC es igual a 0, es decir, estos archivos no han participado en un *bug commit*.

**Datos normalizados por máximo valor en cada métrica (nMáx).**

En este caso, los datos son utilizados por los algoritmos después de aplicarles la normalización por máximo valor en cada métrica (ver los máximos presentados en la parte inferior de la Tabla 13). La Tabla 14 es un ejemplo de la normalización de este tipo que se aplicó a los archivos de la aplicación Tez.

Como se describió en la metodología en la sección 4.4.3, se busca el máximo valor para cada una de las columnas de la M01 a la M16 y por columna con su máximo valor se divide cada valor de cada archivo entre el valor máximo. Esto lo podemos ejemplificar en la columna M02, en ésta, tenemos como máximo valor a 3965 (ver Tabla 13). Si dividimos el valor del archivo 243, el cual es 2326, tenemos un valor normalizado para este archivo de 0.587 ($\frac{2326}{3965}$). Esto se hace para cada archivo en esa columna. Y los pasos anteriores se repiten para las otras quince métricas.

**Tabla 14 Ejemplo de datos normalizados con máximo valor en cada métrica de la aplicación Tez.**

| #File | M01 | M02 | M03 | M04 | M05 | M06 | M07 | M08 | M09 | M10 | M11 | M12 | M13 | M14 | M15 | M16 | M17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 243 | 0.833 | 0.587 | 0.513 | 0.058 | 0.667 | 0.705 | 1.000 | 0.973 | 1.000 | 0.800 | 0.500 | 1.000 | 0.588 | 0.800 | 1.000 | 0.602 | 33 |
| 358 | 0.667 | 1.000 | 1.000 | 0.010 | 1.000 | 1.000 | 0.982 | 1.000 | 1.000 | 0.600 | 0.583 | 0.364 | 1.000 | 1.000 | 1.000 | 1.000 | 33 |
| 87 | 0.333 | 0.240 | 0.019 | 1.000 | 0.036 | 0.777 | 0.446 | 0.459 | 1.000 | 1.000 | 0.167 | 0.545 | 0.000 | 0.600 | 0.833 | 0.057 | 26 |
| 354 | 0.667 | 0.389 | 0.253 | 0.039 | 0.486 | 0.511 | 0.625 | 0.568 | 1.000 | 0.400 | 0.333 | 0.364 | 0.353 | 0.800 | 1.000 | 0.258 | 23 |
| 608 | 0.000 | 0.157 | 0.104 | 0.019 | 0.232 | 0.245 | 0.232 | 0.135 | 0.000 | 0.000 | 0.250 | 0.000 | 0.059 | 0.200 | 0.500 | 0.019 | 19 |
| 342 | 0.667 | 0.519 | 0.506 | 0.039 | 0.587 | 0.612 | 0.571 | 0.784 | 1.000 | 0.800 | 0.583 | 0.364 | 0.529 | 0.800 | 1.000 | 0.417 | 17 |
| 733 | 0.500 | 0.294 | 0.329 | 0.029 | 0.181 | 0.201 | 0.286 | 0.378 | 1.000 | 0.000 | 0.417 | 0.182 | 0.235 | 0.400 | 0.833 | 0.292 | 16 |
| 15 | 0.833 | 0.225 | 0.170 | 0.233 | 0.196 | 0.367 | 0.286 | 0.486 | 1.000 | 0.200 | 0.500 | 0.364 | 0.235 | 0.400 | 1.000 | 0.212 | 16 |
| 355 | 0.500 | 0.290 | 0.241 | 0.019 | 0.428 | 0.439 | 0.268 | 0.568 | 1.000 | 0.400 | 0.250 | 0.364 | 0.529 | 0.400 | 1.000 | 0.261 | 16 |
| 741 | 0.500 | 0.296 | 0.311 | 0.010 | 0.174 | 0.180 | 0.250 | 0.297 | 1.000 | 0.000 | 0.333 | 0.182 | 0.118 | 0.400 | 0.833 | 0.277 | 14 |
| 729 | 0.500 | 0.266 | 0.214 | 0.039 | 0.167 | 0.194 | 0.232 | 0.270 | 1.000 | 0.000 | 0.250 | 0.091 | 0.235 | 0.200 | 0.833 | 0.341 | 12 |
| 719 | 0.333 | 0.207 | 0.226 | 0.029 | 0.203 | 0.223 | 0.446 | 0.351 | 0.000 | 0.000 | 0.500 | 0.000 | 0.294 | 0.400 | 0.500 | 0.201 | 12 |
| 399 | 0.333 | 0.470 | 0.637 | 0.000 | 0.080 | 0.079 | 0.393 | 0.189 | 0.000 | 0.000 | 0.417 | 0.000 | 0.059 | 0.200 | 0.500 | 0.557 | 11 |
| 664 | 0.667 | 0.137 | 0.110 | 0.039 | 0.101 | 0.129 | 0.000 | 0.081 | 0.000 | 0.000 | 0.083 | 0.000 | 0.118 | 0.000 | 0.333 | 0.038 | 11 |
| 746 | 0.500 | 0.264 | 0.284 | 0.010 | 0.145 | 0.151 | 0.214 | 0.243 | 1.000 | 0.000 | 0.333 | 0.000 | 0.118 | 0.400 | 0.667 | 0.235 | 11 |
| 753 | 0.333 | 0.288 | 0.334 | 0.019 | 0.181 | 0.194 | 0.196 | 0.216 | 0.000 | 0.000 | 0.333 | 0.000 | 0.118 | 0.400 | 0.500 | 0.258 | 10 |
| 590 | 0.500 | 0.077 | 0.039 | 0.194 | 0.101 | 0.245 | 0.161 | 0.243 | 1.000 | 0.800 | 0.083 | 0.091 | 0.118 | 0.000 | 0.833 | 0.072 | 10 |
| 467 | 0.167 | 0.166 | 0.092 | 0.010 | 0.203 | 0.209 | 0.232 | 0.135 | 0.000 | 0.000 | 0.250 | 0.000 | 0.059 | 0.200 | 0.500 | 0.015 | 10 |

Una vez que es aplicada la normalización, todos los valores de las métricas están en el rango de [0,1]. Todos aquellos valores que tienen un 1 es que su valor antes de normalizar era el máximo valor en esa columna. Si observamos el archivo 358 en la Tabla 14, de las dieciséis métricas tiene 10 de ellas con valor de 1, esto nos indica que este archivo tiene valores muy altos con más del 67% de las métricas, además de que de acuerdo con M17 está etiquetado como *hots*pot.

En este caso el etiquetado de los datos se hace de la misma manera, definiendo el umbral para la aplicación y evaluando el valor de la métrica M17.

**Datos normalizados por máximo valor en líneas de código (nLOC).**

En este caso, los datos son utilizados por los algoritmos después de aplicarles la normalización por máximo valor en líneas de código. La métrica que se ocupó y que representa las líneas de código es la columna M02. La Tabla 15 es un ejemplo de la normalización de este tipo que se hizo a los archivos de la aplicación Tez.

**Tabla 15 Ejemplo de datos normalizados con máximo valor en líneas de código (M02) de la aplicación Tez.**

| #File | M01 | M02 | M03 | M04 | M05 | M06 | M07 | M08 | M09 | M10 | M11 | M12 | M13 | M14 | M15 | M16 | M17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 243 | 8.5 | 3965 | 514.8 | 10.2 | 156.8 | 167.1 | 95.5 | 61.4 | 1.7 | 6.8 | 10.2 | 18.8 | 17.0 | 6.8 | 10.2 | 271.0 | 33 |
| 358 | 4 | **3965** | 589 | 1 | 138 | 139 | 55 | 37 | 1 | 3 | 7 | 4 | 17 | 5 | 6 | 264 | 33 |
| 87 | 8.3 | 3965 | 45.9 | 429.4 | 20.8 | 450.3 | 104.2 | 70.9 | 4.2 | 20.8 | 8.3 | 25.0 | 0.0 | 12.5 | 20.8 | 62.5 | 26 |
| 354 | 10.3 | 3965 | 382.9 | 10.3 | 172.2 | 182.4 | 89.9 | 54.0 | 2.6 | 5.1 | 10.3 | 10.3 | 15.4 | 10.3 | 15.4 | 174.7 | 23 |
| 608 | 0.0 | 3965 | 388.2 | 12.7 | 203.7 | 216.4 | 82.7 | 31.8 | 0.0 | 0.0 | 19.1 | 0.0 | 6.4 | 6.4 | 19.1 | 31.8 | 19 |
| 342 | 7.7 | 3965 | 574.7 | 7.7 | 156.2 | 163.9 | 61.7 | 55.9 | 1.9 | 7.7 | 13.5 | 7.7 | 17.4 | 7.7 | 11.6 | 212.1 | 17 |
| 733 | 10.2 | 3965 | 660.3 | 10.2 | 85.1 | 95.3 | 54.5 | 47.6 | 3.4 | 0.0 | 17.0 | 6.8 | 13.6 | 6.8 | 17.0 | 262.1 | 16 |
| 15 | 22.2 | 3965 | 444.5 | 106.7 | 120.0 | 226.7 | 71.1 | 80.0 | 4.4 | 4.4 | 26.7 | 17.8 | 17.8 | 8.9 | 26.7 | 248.9 | 16 |
| 355 | 10.3 | 3965 | 489.2 | 6.9 | 203.2 | 210.1 | 51.7 | 72.3 | 3.4 | 6.9 | 10.3 | 13.8 | 31.0 | 6.9 | 20.7 | 237.7 | 16 |
| 741 | 10.1 | 3965 | 618.1 | 3.4 | 81.1 | 84.4 | 47.3 | 37.2 | 3.4 | 0.0 | 13.5 | 6.8 | 6.8 | 6.8 | 16.9 | 246.5 | 14 |
| 729 | 11.3 | 3965 | 473.1 | 15.0 | 86.4 | 101.4 | 48.8 | 37.5 | 3.8 | 0.0 | 11.3 | 3.8 | 15.0 | 3.8 | 18.8 | 337.9 | 12 |
| 719 | 9.7 | 3965 | 643.1 | 14.5 | 135.4 | 149.9 | 120.9 | 62.9 | 0.0 | 0.0 | 29.0 | 0.0 | 24.2 | 9.7 | 14.5 | 256.3 | 12 |
| 399 | 4.3 | 3965 | 798.1 | 0.0 | 23.4 | 23.4 | 46.8 | 14.9 | 0.0 | 0.0 | 10.6 | 0.0 | 2.1 | 2.1 | 6.4 | 312.9 | 11 |
| 664 | 29.3 | 3965 | 475.5 | 29.3 | 102.4 | 131.7 | 0.0 | 21.9 | 0.0 | 0.0 | 7.3 | 0.0 | 14.6 | 0.0 | 14.6 | 73.2 | 11 |
| 746 | 11.4 | 3965 | 632.4 | 3.8 | 75.7 | 79.5 | 45.4 | 34.1 | 3.8 | 0.0 | 15.1 | 0.0 | 7.6 | 7.6 | 15.1 | 234.8 | 11 |
| 753 | 7.0 | 3965 | 684.6 | 7.0 | 86.9 | 93.8 | 38.2 | 27.8 | 0.0 | 0.0 | 13.9 | 0.0 | 7.0 | 7.0 | 10.4 | 236.3 | 10 |
| 590 | 39.1 | 3965 | 300.0 | 260.9 | 182.6 | 443.5 | 117.4 | 117.4 | 13.0 | 52.2 | 13.0 | 13.0 | 26.1 | 0.0 | 65.2 | 247.8 | 10 |
| 467 | 6.0 | 3965 | 325.4 | 6.0 | 168.7 | 174.7 | 78.3 | 30.1 | 0.0 | 0.0 | 18.1 | 0.0 | 6.0 | 6.0 | 18.1 | 24.1 | 10 |
| 17 | 83.5 | 3965 | 0.0 | 751.3 | 0.0 | 751.3 | 333.9 | 208.7 | 0.0 | 41.7 | 83.5 | 0.0 | 41.7 | 41.7 | 166.9 | 41.7 | 10 |

Como se describió en la metodología en la sección 4.4.3, se busca el máximo valor en la métrica M02 (ver en la Tabla 13 para la aplicación TEZ, este valor es de 3965) y después se calcula el *factor de normalización* de cada archivo dividiendo el máximo valor entre el valor de cada archivo de la métrica M02. Por último, con el *factor de normalización* de cada archivo se multiplica a cada valor de cada métrica (de M01 a M16) para ese mismo archivo.

**Tabla 16 Ejemplo de datos normalizados con máximo valor en líneas de código por archivo.**

**Ejemplo 1 de normalización**

| | #File | M01 | M02 | M03 | M04 | M05 | M06 | M07 | M08 | M09 | M10 | M11 | M12 | M13 | M14 | M15 | M16 | M17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **sin noramlizar** | 358 | 4 | 3965 | 589 | 1 | 138 | 139 | 55 | 37 | 1 | 3 | 7 | 4 | 17 | 5 | 6 | 264 | 33 |
| | | | | | | | | | factor de normalización = (3965/3965) = 1.0 | | | | | | | | | |
| **normalizado** | 358 | 4 | 3965 | 589 | 1 | 138 | 139 | 55 | 37 | 1 | 3 | 7 | 4 | 17 | 5 | 6 | 264 | 33 |

**Ejemplo 2 de normalización**

| | #File | M01 | M02 | M03 | M04 | M05 | M06 | M07 | M08 | M09 | M10 | M11 | M12 | M13 | M14 | M15 | M16 | M17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **sin noramlizar** | 243 | 5 | 2326 | 302 | 6 | 92 | 98 | 56 | 36 | 1 | 4 | 6 | 11 | 10 | 4 | 6 | 159 | 33 |
| | | | | | | | | | factor de normalización = (3965/2326) = 1.704643 | | | | | | | | | |
| **normalizado** | 243 | 8.5 | 3965 | 514.8 | 10.2 | 156.8 | 167.1 | 95.5 | 61.4 | 1.7 | 6.8 | 10.2 | 18.8 | 17 | 6.8 | 10.2 | 271 | 33 |

Para ejemplificar lo anterior, observemos la Tabla 16, donde presentamos dos ejemplos de archivos de la aplicación Tez al aplicarles la normalización. El valor máximo que tenemos para la métrica M02 es 3965 (ver Tabla 13). Para el primer ejemplo, obtenemos el *factor de normalización* del archivo 358 donde obtenemos como resultado 1.0000 y al multiplicarlo por los valores de todas las métricas del archivo evaluado tenemos los nuevos valores, que para este caso son los mismos. Lo anterior se debe a que el valor máximo de líneas de código corresponde al archivo 358. Para el segundo ejemplo, obtener el *factor de normalización* del archivo 243 dividimos$\frac{3965}{2643}$,donde obtenemos como resultado 1.7046 y ahora lo multiplicamos por cada valor de todas las métricas del archivo evaluado, obteniendo nuevos valores de ellas asumiendo que el archivo es del mismo tamaño que el más grande.

Con esta normalización se está proyectando el valor de cada métrica suponiendo que todos los archivos tuvieran el mismo tamaño. Si vemos la Tabla 15, la columna M02 queda con un mismo valor para todos los archivos.

### 5.4.4 Selección de datos de entrenamiento y de prueba

La selección de los datos de entrenamiento y prueba que se describe a continuación, sólo se aplicó para el algoritmo de clasificación BI. Para el algoritmo exhaustivo, se tomaron las mismas aplicaciones elegidas para esta etapa descritas en la Tabla 17 y a ellas se aplicó directamente el algoritmo exhaustivo.

**Tabla 17 Aplicaciones elegidas para entrenamiento y prueba.**

| Aplicación | Deltaspike | Flume | Hbase | Knox | Nifi | Oozie | Sentry | Tez | Tajo | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| Cantidad de Archivos | 742 | 441 | 1786 | 863 | 4112 | 812 | 561 | 839 | 1557 | **11,713** |
| Cantidad de archivos etiquetados como hotspots (NBC) | 28 | 58 | 129 | 23 | 123 | 98 | 12 | 90 | 78 | **639** |
| Cantidad de archivos etiquetados como no hotspots (NBC) | 714 | 383 | 1657 | 840 | 3989 | 714 | 549 | 749 | 1479 | **11,074** |
| % Hotspots | 3.8% | 13.2% | 7.2% | 2.7% | 3.0% | 12.1% | 2.1% | 10.7% | 5.0% | **5.5%** |
| % No hotspots | 96.2% | 86.8% | 92.8% | 97.3% | 97.0% | 87.9% | 97.9% | 89.3% | 95.0% | **94.5%** |

A partir de las once aplicaciones listadas en la Tabla 8 (sección 5.1) que fueron elegidas se tomaron nueve de ellas para entrenamiento y pruebas, dejando para la fase de *validación de resultados* dos aplicaciones. Estas últimas aplicaciones se eligieron en función del tamaño que representan con base a la columna *Cantidad de archivos* de la Tabla 8, la primera fue Amqp-0x que es de tamaño pequeño con 362 archivos y un 3.3% de *hotspots* (que está por debajo de la media) y la segunda fue Broke-j que es de tamaño mediano con 1,865 archivos y un 4.9% de *hotspots* (que está cerca de la media). El tamaño se considera con respecto a la cantidad de los archivos que tienen las once aplicaciones.

Las aplicaciones para entrenamiento y prueba quedaron definidas como se muestra en la Tabla 17. En ella también se muestra la distribución de los archivos que se etiquetaron como *hotspots* conforme al criterio para etiquetar los archivos revisado en la sección 4.4.2.

Adicionalmente podemos observar de la Tabla 17 que se tuvieron aplicaciones muy diversas, como Nifi, que fue la aplicación más grande con 4112 archivos y solo tiene 123 etiquetados como *hotspots*, que corresponden al 3%. Otro ejemplo es Flume que fue la aplicación más pequeña con 441 archivos, de los cuales 58 están etiquetados como *hotspots*, que corresponden al 13.2%.

Una vez que fueron definidas las aplicaciones para entrenamiento y pruebas, se definieron los KFCV (técnica de validación cruzada de K iteraciones, también conocida como validación *K-fold cross validation* para hacer la división de los datos) que se aplicarían. Dado que se tuvieron nueve aplicaciones y siguiendo la segmentación de los datos sugeridos en la sección 4.4.4, se definió $k = 9$ , con lo cual se generaron nueve casos de entrenamiento y prueba para cada experimento, en donde se usó el algoritmo de BI.

**Tabla 18 Casos con distribución de los datos para entrenamiento y prueba.**

| | Datos de entrenamiento | | | | | Datos de prueba | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **# Caso (Aplicación de prueba)** | **Cantidad de archivos** | **Archivos hotspots (NBC)** | **Archivos no hotspot** | **% Hotspots** | **% No hotspots** | **Cantidad de archivos** | **Archivos hotspots (NBC)** | **Archivos no hotspot** | **% Hotspots** | **% No hotspots** |
| 1 (Tajo) | 10,156 | 561 | 9,595 | 4.8% | 81.9% | 1,557 | 78 | 1,479 | 0.7% | 12.6% |
| 2 (Tez) | 10,874 | 549 | 10,325 | 4.7% | 88.1% | 839 | 90 | 749 | 0.8% | 6.4% |
| 3 (Sentry) | 11,152 | 627 | 10,525 | 5.4% | 89.9% | 561 | 12 | 549 | 0.1% | 4.7% |
| 4 (Oozie) | 10,901 | 541 | 10,360 | 4.6% | 88.4% | 812 | 98 | 714 | 0.8% | 6.1% |
| 5 (Nifi) | 7,601 | 516 | 7,085 | 4.4% | 60.5% | 4,112 | 123 | 3,989 | 1.1% | 34.1% |
| 6 (Know) | 10,850 | 616 | 10,234 | 5.3% | 87.4% | 863 | 23 | 840 | 0.2% | 7.2% |
| 7 (Hbase) | 9,927 | 510 | 9,417 | 4.4% | 80.4% | 1,786 | 129 | 1,657 | 1.1% | 14.1% |
| 8 (Flume) | 11,272 | 581 | 10,691 | 5.0% | 91.3% | 441 | 58 | 383 | 0.5% | 3.3% |
| 9 (Deltaspike) | 10,971 | 611 | 10,360 | 5.2% | 88.4% | 742 | 28 | 714 | 0.2% | 6.1% |
| **Total** | 93,704 | 5,112 | 88,592 | **4.8%** | **84.0%** | 11,713 | 639 | 11,074 | **0.6%** | **10.5%** |
| | **% de datos de entrenamiento:** | | | 88.9% | | **% de datos de prueba** | | | 11.1% | |

La representación de los nueve casos y su distribución porcentual entre datos de entrenamiento y pruebas quedó como lo podemos ver en la Tabla 18. En ella vemos la columna *#Caso (Aplicación de prueba)* que nos indica el número de caso y la aplicación que se utilizó para pruebas. Posteriormente tenemos para los datos de entrenamiento por caso las columnas de *Cantidad de archivos*, *Archivos etiquetados como hotspots*, *Archivos no hotspots*, *porcentaje de Archivos hotspots* y *porcentaje de archivos no hotspots*. Para los datos de prueba se tienen las mismas columnas que para datos de entrenamiento. En el *Total* podemos ver que se tuvieron para entrenamiento 93,704 (88.9%) archivos, de los cuales 5,112 (4.8%) fueron etiquetados como *hotspots* y 88,592 (84.0%) fueron etiquetados como *no hotspots*. En total para prueba se tuvieron 11,713 (11.1%) archivos, de los cuales 639 (0.6%) se esperaba que el algoritmo de BI etiquetara como *hotspots* y 11,074 (10.5%) fueran etiquetados como *no hotspots.*

Una vez que fueron definidos los datos para entrenamiento y prueba para el algoritmo de clasificación supervisada BI, así como, las aplicaciones para el algoritmo exhaustivo se establecen las medidas de rendimiento que se utilizaron para evaluar la efectividad de los algoritmos.

### 5.4.5 Medidas de rendimiento elegidas para evaluar la clasificación

Con base en la revisión de la literatura y con aquellos trabajos en los que se usa esta medida de rendimiento, se encuentran estudios [16],[19], [20], en los que los resultados de la AUC en sus diferentes experimentos están por debajo de 0.825. Este valor se tomó como un punto de comparación para los resultados que se obtuvieron después de la experimentación.

Por cada combinación de métricas que sea evaluada, se obtienen los resultados de su matriz de confusión, para poder calcular la sensibilidad, especificidad y AUC, y determinar cuál combinación de métricas nos da los mejores resultados.

### 5.4.6 Experimentación para identificación de *hotspots*

Ahora se presentan los resultados obtenidos de cada uno de los experimentos ejecutados. Estos son mostrados en la Tabla 6 (sección 4.4.3). De cada uno de ellos, se muestran los resultados obtenidos y un análisis destacando la combinación de *métricas sugeridas* que ofrece los mejores resultados. Posteriormente, en la siguiente sección, se hará un análisis detallado de los resultados obtenidos de los seis experimentos y determinar si existe, una combinación de métricas (*seleccionadas),* que, apoyadas de un algoritmo de aprendizaje automático, puedan ofrecer una detección de *hotspots* aceptable en los sistemas analizados.

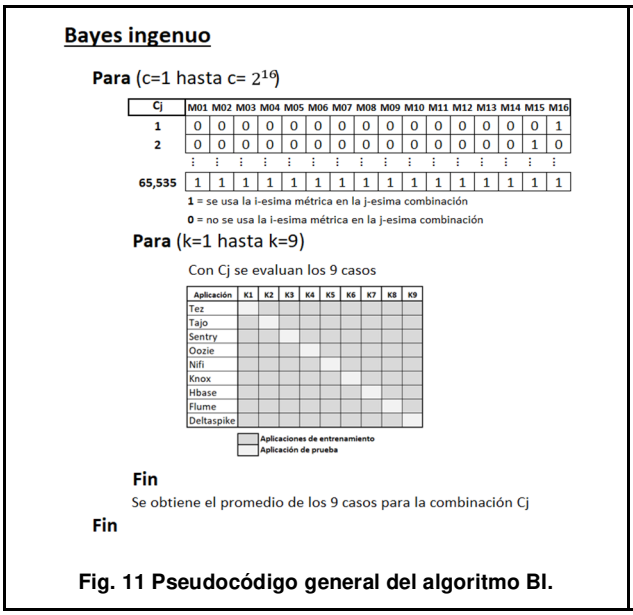



**Fig. 11 Pseudocódigo general del algoritmo BI.**

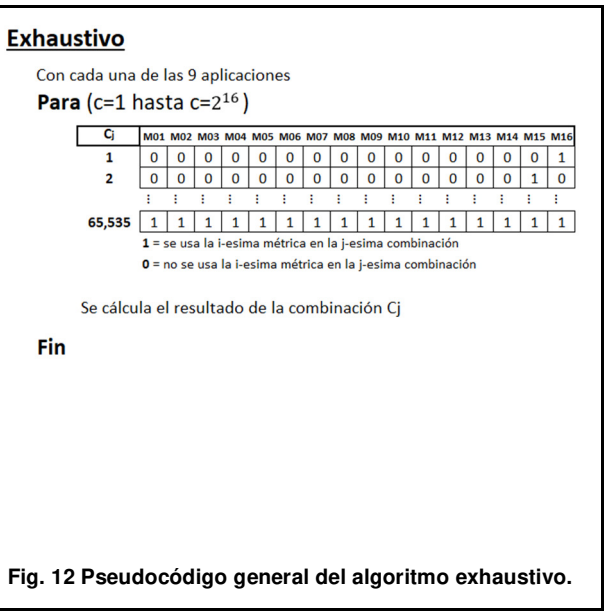



**Fig. 12 Pseudocódigo general del algoritmo exhaustivo.**

Para la ejecución de los experimentos, se desarrollaron programas en lenguaje C con los que se implementaron los algoritmos de BI y exhaustivo[23]. Las Fig. 11 y 12 muestran el pseudocódigo que se aplicó en cada algoritmo. Estos algoritmos fueron ejecutados con los datos sin normalizar y normalizados en los dos tratamientos explicados en la sección 5.4.3.

Un ejemplo del resultado de una ejecución del programa exhaustivo para la aplicación es el que se muestra en la Tabla 19. Los datos que tenemos como resultado son el nombre de la aplicación, el número de combinación, la combinación en representación binaria (recordemos que un 1 es que se usó la métrica y un 0 es que no se usó en la métrica), las siguientes cuatro columnas (D-G) es la matriz de confusión que se obtuvo con la combinación utilizada y de las siguientes columnas (H-M) son los cálculos de las medidas de rendimiento obtenidas con la combinación utilizada.

Las métricas utilizadas en una combinación son como sigue, si observamos el renglón veinte de la Tabla 19 tenemos que el código de la combinación es 1100100000000000. Si recorremos este de izquierda a derecha tenemos que las métricas que se utilizaron fueron M01 (primera posición), M02 (segunda posición) y M05 (quinta posición).

**Tabla 19 Ejemplo del resultado de una ejecución del programa exhaustivo.**

| | A | B | C | D | E | F | G | H | I | J | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | **Aplicación** | **NumCom** | **Combinacion** | **TP** | **TN** | **FP** | **FN** | **Exactitud** | **Precisión** | **Sensibilidad** | **Especificidad** | **Medida-F** | **UAC** |
| 2 | knox | 1 | 1000000000000000 | 2 | 819 | 21 | 21 | 0.951 | 0.087 | 0.087 | 0.975 | 0.159 | 0.531 |
| 3 | knox | 2 | 0100000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 4 | knox | 3 | 1100000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 5 | knox | 4 | 0010000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 6 | knox | 5 | 1010000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 7 | knox | 6 | 0110000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 8 | knox | 7 | 1110000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 9 | knox | 8 | 0001000000000000 | 2 | 819 | 21 | 21 | 0.951 | 0.087 | 0.087 | 0.975 | 0.159 | 0.531 |
| 10 | knox | 9 | 1001000000000000 | 2 | 819 | 21 | 21 | 0.951 | 0.087 | 0.087 | 0.975 | 0.159 | 0.531 |
| 11 | knox | 10 | 0101000000000000 | 8 | 825 | 15 | 15 | 0.965 | 0.348 | 0.348 | 0.982 | 0.511 | 0.665 |
| 12 | knox | 11 | 1101000000000000 | 8 | 825 | 15 | 15 | 0.965 | 0.348 | 0.348 | 0.982 | 0.511 | 0.665 |
| 13 | knox | 12 | 0011000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 14 | knox | 13 | 1011000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 15 | knox | 14 | 0111000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 16 | knox | 15 | 1111000000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 17 | knox | 16 | 0000100000000000 | 5 | 822 | 18 | 18 | 0.958 | 0.217 | 0.217 | 0.979 | 0.354 | 0.598 |
| 18 | knox | 17 | 1000100000000000 | 4 | 821 | 19 | 19 | 0.956 | 0.174 | 0.174 | 0.977 | 0.294 | 0.576 |
| 19 | knox | 18 | 0100100000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 20 | knox | 19 | 1100100000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |
| 21 | knox | 20 | 0010100000000000 | 8 | 825 | 15 | 15 | 0.965 | 0.348 | 0.348 | 0.982 | 0.511 | 0.665 |
| 22 | knox | 21 | 1010100000000000 | 9 | 826 | 14 | 14 | 0.968 | 0.391 | 0.391 | 0.983 | 0.557 | 0.687 |

Todos los experimentos fueron ejecutados en una máquina con las siguientes características: procesador Intel(R) Core(TM) i-7700 HQ con una velocidad de 2.8 GHz, 4 núcleos físicos, 8 núcleos lógicos, 16 GB de memoria RAM y sistema operativo Windows 10.

[23] La implementación de estos algoritmos se puede encontrar en https://github.com/hreyes7/deteccionHotspots

### 5.4.6.1 Algoritmo exhaustivo sin normalización de datos

Para este experimento, a los datos no se les aplicó ningún tratamiento de normalización. El algoritmo de búsqueda exhaustiva prueba todas las combinaciones $(2^{16})$ de métricas candidatas para identificar cuál de ellas es la que tiene la mejor medida de rendimiento de AUC. El algoritmo se ejecutó para cada una de las aplicaciones, se evaluó cada una de las combinaciones de métricas en cada aplicación y por último se determinó que combinación de métricas obtuvo, en promedio, el resultado más alto de AUC para todas las aplicaciones.

**Tabla 20 Resultado de algoritmo exhaustivo sin normalización de datos.**

| Experimento: **Exh (sNorm)** | | | Matriz de confusión | | | | Medidas de rendimiento | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Aplicación** | **Archivos evaluados** | **Combinación de métricas con mejor rendimiento** | **TP** | **TN** | **FP** | **FN** | **Exactitud** | **Precisión** | **Sensibilidad** | **Especificidad** | **Medida-F** | **AUC** |
| **Deltaspike** | 742 | | 8 | 700 | 17 | 17 | 0.95 | 0.32 | 0.32 | 0.98 | 0.48 | 0.65 |
| **Flume** | 441 | **M01 (DES_Design Smells)** | 28 | 353 | 30 | 30 | 0.86 | 0.48 | 0.48 | 0.92 | 0.62 | 0.70 |
| **Hbase** | 1786 | **M08 (DV8_TotalIssues)** | 61 | 1588 | 69 | 68 | 0.92 | 0.47 | 0.47 | 0.96 | 0.62 | 0.72 |
| **Knox** | 863 | **M09 (DV8_Clique)** | 7 | 824 | 16 | 16 | 0.96 | 0.30 | 0.30 | 0.98 | 0.46 | 0.64 |
| **Nifi** | 4112 | **M10 (DV8_Crossing(h))** | 42 | 3878 | 97 | 95 | 0.95 | 0.30 | 0.31 | 0.98 | 0.46 | 0.64 |
| **Oozie** | 812 | **M11 (DV8_ModularityViolation(h))** | 60 | 676 | 38 | 38 | 0.91 | 0.61 | 0.61 | 0.95 | 0.73 | 0.78 |
| **Sentry** | 561 | **M13 (DV8_UnhealthyInheritance)** | 7 | 544 | 5 | 5 | **0.98** | 0.58 | 0.58 | **0.99** | 0.73 | 0.79 |
| **Tajo** | 1557 | **M14 (DV8_UnstableInterface(h))** | 42 | 1443 | 36 | 36 | 0.95 | 0.54 | 0.54 | 0.98 | 0.69 | 0.76 |
| **Tez** | 839 | **M15 (DV8_PresentInIssues)** | 60 | 719 | 30 | 30 | 0.93 | **0.67** | **0.67** | 0.96 | **0.78** | **0.81** |
| | | **M16 (SQ_Issues)** | | | **Promedio:** | | **0.94** | **0.48** | **0.48** | **0.97** | **0.62** | **0.72** |

Los resultados obtenidos los podemos ver en la Tabla 20, en la cual se presentan la combinación de métricas con las mejores medidas de rendimiento del algoritmo por aplicación. Tenemos la columna de *Aplicación*, el número de *Archivos evaluados*, la combinación de métricas con la que se obtuvo el mejor resultado en promedio de AUC para todas las aplicaciones, la *matriz de confusión* y las *medidas de rendimiento* obtenidas, así como el *promedio* de las medidas de rendimiento de todo el experimento. Estos últimos resultados promedios fueron los que se contrastaron con el resto de los experimentos.

En este experimento encontramos como primeros resultados que la aplicación de Tez es la mejor evaluada en sensibilidad (capacidad de clasificar *hotspots*) con un 67%. Tenemos la aplicación de Sentry como la mejor evaluada en especificidad (capacidad de clasificar no *hotspots*) con un 99%. Tenemos a la aplicación Tez como la mejor evaluada en AUC con 81%, donde podemos observar que se está balanceando la capacidad de los verdaderos positivos (TP) como de los verdaderos negativos (TN). Tez tiene un 10.7% de *hotspots* etiquetados inicialmente el cual está por arriba del promedio de todas las bases de datos, que es de 5.4% (Tabla 12), lo

que nos dice que al menos para esta aplicación, la combinación de métricas seleccionada, la favorece. Esta combinación está conformada por seis métricas estructurales (M01, M08, M09, M13, M15, M16) y tres métricas históricas (M10, M11 y M14). De estas últimas, se destaca que son el 75% de las métricas históricas que se incluyeron para realizar los experimentos. Sin embargo, también se mencionó que las métricas históricas requieren de tiempo de evolución de las aplicaciones para que sus valores puedan aportar en la detección de *hotspots.*

Ahora, considerando el promedio de todos los resultados en este experimento *Exh (sNorm)* tenemos una exactitud del 94% y una AUC del 72% que se sustenta con un 48% en sensibilidad y un 97% en especificidad.

#### 5.4.6.2 Algoritmo exhaustivo normalizando datos con máximo valor por métrica

Para este experimento, los datos fueron normalizados por el máximo valor por métrica para homogeneizar la base de datos teniendo valores solo entre cero y uno. La normalización se aplica a todos los conjuntos de datos antes de aplicar el algoritmo exhaustivo. En la Tabla 14 podemos ver un ejemplo de una base de datos ya normalizada con máximo valor por métrica.

Aplicando el algoritmo exhaustivo con normalización de datos con máximo valor en cada métrica para cada una de las nueve aplicaciones, se obtuvieron los resultados presentados en la Tabla 21. En ellos, podemos ver que la aplicación que mejor fue evaluada en sensibilidad es Tez con un 72%, tenemos a Sentry como la mejor evaluada en especificidad con 99% y a Oozie como una de las aplicaciones con el mejor resultado en AUC con un 84%. Oozie es una de las aplicaciones que tiene un porcentaje más alto de *hotspots* etiquetados inicialmente con un 12.1%, mientras que el porcentaje promedio de las nueve aplicaciones de entrenamiento y prueba es de 5.4% (Tabla 12). También tenemos a Tez como un 84% en AUC, sustentado por 72% en sensibilidad y un 97% en especificidad. En este caso, Tez es la aplicación que fue mejor evaluada, dado que en las medidas de rendimiento de precisión y medida-F también están por encima de todas las demás.

**Tabla 21 Resultado de algoritmo exhaustivo normalizando datos por máximo valor en cada métrica.**

| Experimento: **Exh (nMAX)** | | | Matriz de confusión | | | | Medidas de rendimiento | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Aplicación** | **Archivos evaluados** | **Combinación de métricas con mejor rendimiento** | **TP** | **TN** | **FP** | **FN** | **Exactitud** | **Precisión** | **Sensibilidad** | **Especificidad** | **Medida-F** | **AUC** |
| **Deltaspike** | 742 | | 6 | 698 | 19 | 19 | 0.95 | 0.24 | 0.24 | 0.97 | 0.38 | 0.61 |
| **Flume** | 441 | **M01 (DES_Design Smells)** | 29 | 354 | 29 | 29 | 0.87 | 0.50 | 0.50 | 0.92 | 0.63 | 0.71 |
| **Hbase** | 1786 | **M02 (SCC_CLOC)** | 64 | 1591 | 66 | 65 | 0.93 | 0.49 | 0.50 | 0.96 | 0.64 | 0.73 |
| **Knox** | 863 | **M03 (SCC_COMPLEXITY)** | 11 | 828 | 12 | 12 | 0.97 | 0.48 | 0.48 | 0.99 | 0.64 | 0.73 |
| **Nifi** | 4112 | **M05 (ARCH_Depends on Partners)** | 56 | 3892 | 83 | 81 | 0.96 | 0.40 | 0.41 | 0.98 | 0.57 | 0.69 |
| **Oozie** | 812 | **M10 (DV8_Crossing(h))** | 70 | 686 | 28 | 28 | 0.93 | 0.71 | 0.71 | 0.96 | 0.81 | **0.84** |
| **Sentry** | 561 | **M11 (DV8_ModularityViolation(h))** | 8 | 545 | 4 | 4 | **0.99** | 0.67 | 0.67 | **0.99** | 0.80 | 0.83 |
| **Tajo** | 1557 | **M14 (DV8_UnstableInterface(h))** | 47 | 1448 | 31 | 31 | 0.96 | 0.60 | 0.60 | 0.98 | 0.74 | 0.79 |
| **Tez** | 839 | **M15 (DV8_PresentInIssues)** | 65 | 724 | 25 | 25 | 0.94 | **0.72** | **0.72** | 0.97 | **0.82** | **0.84** |
| | | **M16 (SQ_Issues)** | | | **Promedio:** | | **0.94** | **0.54** | **0.54** | **0.97** | **0.67** | **0.75** |

Los mejores resultados son obtenidos con la combinación de métricas seleccionada de nueve métricas presentadas en la Tabla 21, de las cuales seis métricas son estructurales (M01, M02, M03, M05, M15, M16) y tres métricas son históricas (M10, M11 y M14), lo que nos sigue confirmando que este tipo de métricas históricas son buenas predictoras para la detección de *hotspots*.

Considerando los resultados promedio obtenidos para este experimento *Exh (nMax)* tenemos una exactitud del 94% y una AUC del 75% que se sustenta con un 54% en sensibilidad y un 97% en especificidad.

### 5.4.6.3 Algoritmo exhaustivo normalizando datos con máximo valor en líneas de código

Para este experimento, los datos fueron normalizados con el máximo valor en la métrica líneas de código (M02) y posteriormente aplicado a toda la base de datos. De esta forma se proyectan todos los archivos al mismo tamaño y el efecto de proyección en el resto de las métricas.

Aplicando el algoritmo exhaustivo con normalización de datos con máximo valor en líneas de código se obtuvieron los resultados presentados en la Tabla 22. En ellos, podemos ver que la aplicación que mejor fue evaluada en la aplicación de Tez con un 81% en AUC, sustentado por un 66% de sensibilidad y un 96% de especificidad, además de que también en precisión y medida-F está por arriba de las demás. Sentry tiene un valor muy bueno de 99% en especificidad,

pero la sensibilidad es baja con un 58% por lo que obtiene un 79% en AUC, quedando por debajo de Tez.

**Tabla 22 Resultado de algoritmo exhaustivo normalizando datos por máximo valor en líneas de código.**

| Experimento: **Exh (nLOC)** | | | Matriz de confusión | | | | Medidas de rendimiento | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Aplicación** | **Archivos evaluados** | **Combinación de métricas con mejor rendimiento** | **TP** | **TN** | **FP** | **FN** | **Exactitud** | **Precisión** | **Sensibilidad** | **Especificidad** | **Medida-F** | **AUC** |
| **Deltaspike** | 742 | | 6 | 698 | 19 | 19 | 0.95 | 0.24 | 0.24 | 0.97 | 0.38 | 0.61 |
| **Flume** | 441 | | 28 | 352 | 30 | 31 | 0.86 | 0.48 | 0.47 | 0.92 | 0.62 | 0.70 |
| **Hbase** | 1786 | **M03 (SCC_COMPLEXITY)** | 64 | 1592 | 65 | 65 | 0.93 | 0.50 | 0.50 | 0.96 | 0.65 | 0.73 |
| **Knox** | 863 | **M08 (DV8_TotalIssues)** | 7 | 824 | 16 | 16 | 0.96 | 0.30 | 0.30 | 0.98 | 0.46 | 0.64 |
| **Nifi** | 4112 | **M09 (DV8_Clique)** | 39 | 3875 | 100 | 98 | 0.95 | 0.28 | 0.28 | 0.97 | 0.43 | 0.63 |
| **Oozie** | 812 | **M11 (DV8_ModularityViolation(h))** | 61 | 677 | 37 | 37 | 0.91 | 0.62 | 0.62 | 0.95 | 0.74 | 0.79 |
| **Sentry** | 561 | **M12 (DV8_PackageCycle)** | 7 | 544 | 5 | 5 | **0.98** | 0.58 | 0.58 | **0.99** | 0.73 | 0.79 |
| **Tajo** | 1557 | **M15 (DV8_PresentInIssues)** | 42 | 1443 | 36 | 36 | 0.95 | 0.54 | 0.54 | 0.98 | 0.69 | 0.76 |
| **Tez** | 839 | **M16 (SQ_Issues)** | 59 | 718 | 31 | 31 | 0.93 | **0.66** | **0.66** | 0.96 | **0.77** | **0.81** |
| | | | | | | **Promedio:** | **0.94** | **0.47** | **0.47** | **0.96** | **0.61** | **0.72** |

Los mejores resultados son obtenidos con la combinación seleccionada de siete métricas presentadas en la Tabla 22, de las cuales seis son métricas estructurales (M03, M08, M09, M12, M15, M16) y una métrica es histórica (M11). Podemos destacar que una de las métricas que destaca hasta este punto con los experimentos exhaustivos es M11.

Considerando los resultados obtenidos en las nueve aplicaciones con la combinación de métricas seleccionadas tenemos en promedio para este experimento *Exh (nLOC)* una exactitud de 94% y una AUC de 72% que se sustenta con un 47% en sensibilidad y un 96% en especificidad.

#### 5.4.6.4 Algoritmo Bayes ingenuo sin normalización de datos

Para este experimento, a los datos no se les aplicó ningún tratamiento. El algoritmo de clasificación BI recibió el conjunto de datos de entrenamiento y posteriormente los datos de prueba para realizar la experimentación por cada una de las combinaciones de métricas y determinar si existe una combinación de métricas que destaque de entre todas las combinaciones, en las medidas de rendimiento del algoritmo al clasificar los hotspots.

Aplicando el algoritmo de BI sin normalización de datos, se obtuvieron los resultados presentados en la Tabla 23. En ellos podemos ver que la aplicación que destaca en AUC es Sentry con un 94.0% sustentado con un 92% de sensibilidad y un 97%.

**Tabla 23 Resultado de algoritmo BI sin normalización de datos.**

| Experimento: **BI (sNorm)** | | | Matriz de confusión | | | | Medidas de rendimiento | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Aplicación de prueba** | **Archivos evaluados** | **Combinación de métricas con mejor rendimiento** | **TP** | **TN** | **FP** | **FN** | **Exactitud** | **Precisión** | **Sensibilidad** | **Especificidad** | **Medida-F** | **AUC** |
| **Deltaspike** | 742 | **M01 (DES_Design Smells)** | 0 | 711 | 6 | 25 | 0.96 | 0.00 | 0.00 | **0.99** | 0.00 | 0.50 |
| **Flume** | 441 | **M02 (SCC_CLOC)** | 26 | 358 | 25 | 32 | 0.87 | 0.51 | 0.45 | 0.93 | 0.64 | 0.69 |
| **Hbase** | 1786 | **M03 (SCC_COMPLEXITY)** | 77 | 1555 | 102 | 52 | 0.91 | 0.43 | 0.60 | 0.94 | 0.58 | 0.77 |
| **Knox** | 863 | **M07 (ARCH_CoChange Partners(h))** | 17 | 602 | 238 | 6 | 0.72 | 0.07 | 0.74 | 0.72 | 0.12 | 0.73 |
| **Nifi** | 4112 | **M09 (DV8_Clique)** | 32 | 3920 | 55 | 105 | 0.96 | 0.37 | 0.23 | **0.99** | 0.53 | 0.61 |
| **Oozie** | 812 | **M10 (DV8_Crossing(h))** | 70 | 659 | 55 | 28 | 0.90 | 0.56 | 0.71 | 0.92 | 0.69 | 0.82 |
| **Sentry** | 561 | **M11 (DV8_ModularityViolation(h))** | 11 | 534 | 15 | 1 | **0.97** | 0.42 | **0.92** | 0.97 | 0.59 | **0.94** |
| **Tajo** | 1557 | **M12 (DV8_PackageCycle)** | 68 | 1252 | 227 | 10 | 0.85 | 0.23 | 0.87 | 0.85 | 0.36 | 0.86 |
| **Tez** | 839 | **M15 (DV8_PresentInIssues)** | 66 | 717 | 32 | 24 | 0.93 | **0.67** | 0.73 | 0.96 | **0.78** | 0.85 |
| | | **M16 (SQ_Issues)** | | | **Promedio:** | | **0.90** | **0.36** | **0.58** | **0.92** | **0.48** | **0.75** |

En especificidad, tenemos a las aplicaciones Deltaspike y Nifi como las que destacan con un 99% y en sensibilidad tenemos a Sentry como la que más destaca con un 92%. A pesar de que Deltaspike y Nifi destacan en especificidad no alcanzan a ser consideradas como alguna de las mejor evaluadas dado que su sensibilidad en ambas aplicaciones es baja con un 0.0% y 23% respectivamente y con ello su AUC es baja también.

Sobre la combinación de métricas seleccionada, tenemos que, siete de ellas son estructurales (M01, M02, M03, M09, M12, M15 y M16) y tres de ellas son históricas (M07, M10, M11). Nuevamente las métricas M11, M15 y M16 siguen participando en la mejor combinación de métricas en los experimentos.

En este experimento con algoritmo BI se ve un cambio significativo positivo en el AUC para la mejor aplicación evaluada, esto con respecto a los tres experimentos anteriores con el algoritmo exhaustivo donde el porcentaje de AUC de la mejor aplicación evaluada alcanzó un 84% solamente.

#### 5.4.6.5 Algoritmo de Bayes ingenuo normalizando datos con máximo valor por métrica

Para este experimento, los datos fueron normalizados por el máximo valor por métrica para homogeneizar la base de datos de prueba y entrenamiento teniendo valores solo entre cero y uno. El algoritmo de clasificación BI recibió el conjunto de datos de entrenamiento y posteriormente los datos de prueba para realizar la experimentación por cada una de las

combinaciones de métricas y determinar si existe una combinación de métricas que destaque de entre todas las combinaciones, en las medidas de rendimiento del algoritmo al clasificar los *hotspots*.

**Tabla 24. Resultado de algoritmo BI normalizando datos por máximo valor en cada métrica.**

| Experimento: **BI (nMAX)** | | | Matriz de confusión | | | | Medidas de rendimiento | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Aplicación de prueba | Archivos evaluados | Combinación de métricas con mejor rendimiento | TP | TN | FP | FN | Exactitud | Precisión | Sensibilidad | Especificidad | Medida-F | AUC |
| Deltaspike | 742 | | 23 | 507 | 210 | 2 | 0.71 | 0.10 | 0.92 | 0.71 | 0.17 | 0.81 |
| Flume | 441 | | 39 | 317 | 66 | 19 | 0.81 | 0.37 | 0.67 | 0.83 | 0.51 | 0.75 |
| Hbase | 1786 | | 85 | 1558 | 99 | 44 | 0.92 | 0.46 | 0.66 | 0.94 | 0.62 | 0.80 |
| Knox | 863 | M02 (SCC_CLOC) | 18 | 782 | 58 | 5 | 0.93 | 0.24 | 0.78 | 0.93 | 0.38 | 0.86 |
| Nifi | 4112 | M10 (DV8_Crossing(h)) | 83 | 3781 | 194 | 54 | **0.94** | 0.30 | 0.61 | **0.95** | 0.45 | 0.78 |
| Oozie | 812 | M11 (DV8_ModularityViolation(h)) | 95 | 619 | 95 | 3 | 0.88 | **0.50** | 0.97 | 0.87 | **0.64** | 0.92 |
| Sentry | 561 | | 12 | 497 | 52 | 0 | 0.91 | 0.19 | **1.00** | 0.91 | 0.31 | **0.95** |
| Tajo | 1557 | | 70 | 1316 | 163 | 8 | 0.89 | 0.30 | 0.90 | 0.89 | 0.45 | 0.89 |
| Tez | 839 | | 75 | 659 | 90 | 15 | 0.87 | 0.45 | 0.83 | 0.88 | 0.60 | 0.86 |
| | | | | | | **Promedio:** | **0.87** | **0.32** | **0.82** | **0.88** | **0.46** | **0.85** |

Aplicando el algoritmo de BI normalizado datos por el máximo valor por métrica, se obtuvieron los resultados presentados en la Tabla 24. En ellos podemos ver que la aplicación que obtiene el mejor valor en AUC es la aplicación de Sentry con un 95.0% sustentado con un 100% (el primero con respecto a los anteriores experimentos) en sensibilidad y un 91% en especificidad. Se obtuvo un 1% más arriba en AUC que el experimento anterior aplicando el mismo algoritmo de BI sin normalización de datos. En este experimento se obtuvo en promedio un 85% en AUC, el cual es el más alto con respecto a los anteriores. Se obtuvieron cinco de las nueve aplicaciones con un AUC igual o mayor al 85%, lo cual no se había obtenido en los anteriores experimentos.

La combinación de métricas seleccionada que nos ofreció los resultados más altos en este experimento solo consta de tres métricas M02, M10 y M11. De esto, se destaca que el tamaño de los archivos al parecer sí influye en un elemento a considerar al tratar de ubicar los archivos *hotspots*, así como, la complejidad que puedan tener el código de cada archivo. También se sigue manteniendo de manera consistente la métrica M11 la cual ha estado presente en todos los experimentos hasta el momento.

En este experimento se obtuvo en promedio un 87% de exactitud y un 85% de AUC sustentado por un 82% en sensibilidad y un 84% en especificidad.

#### 5.4.6.6 Algoritmo Bayes ingenuo normalizando datos con máximo valor por métrica

Por último, para este experimento, los datos fueron normalizados con el máximo valor en la métrica líneas de código (M02) y posteriormente aplicado a toda la base de datos, de esta forma se proyectan todos los archivos al mismo tamaño y el efecto de proyección en el resto de las métricas. El algoritmo de clasificación BI recibió el conjunto de datos de entrenamiento y posteriormente los datos de prueba para realizar la experimentación por cada una de las combinaciones de métricas y determinar si existe una combinación de métricas que destaque de entre todas las combinaciones, en las medidas de rendimiento del algoritmo al clasificar los hotspots.

**Tabla 25 Resultado de algoritmo BI normalizando datos por máximo valor en líneas de código.**

| Experimento: **BI (nLOC)** | | | Matriz de confusión | | | | Medidas de rendimiento | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Aplicación de prueba | Archivos evaluados | Combinación de métricas con mejor rendimiento | TP | TN | FP | FN | Exactitud | Precisión | Sensibilidad | Especificidad | Medida-F | AUC |
| Deltaspike | 742 | M01 (DES_Design Smells) | 0 | 713 | 4 | 25 | **0.96** | 0.00 | 0.00 | **0.99** | 0.00 | 0.50 |
| Flume | 441 | M03 (SCC_COMPLEXITY) | 20 | 364 | 22 | 35 | 0.87 | 0.48 | 0.36 | 0.94 | 0.62 | 0.65 |
| Hbase | 1786 | M05 (ARCH_Depends on Partners) | 69 | 1550 | 114 | 53 | 0.91 | 0.38 | 0.57 | 0.93 | 0.53 | 0.75 |
| Knox | 863 | M07 (ARCH_CoChange Partners(h)) | 14 | 592 | 251 | 6 | 0.70 | 0.05 | 0.70 | 0.70 | 0.10 | 0.70 |
| Nifi | 4112 | M11 (DV8_ModularityViolation(h)) | 49 | 3853 | 122 | 88 | 0.95 | 0.29 | 0.36 | 0.97 | 0.44 | 0.66 |
| Oozie | 812 | M12 (DV8_PackageCycle) | 76 | 643 | 71 | 22 | 0.89 | **0.52** | 0.78 | 0.90 | **0.65** | 0.84 |
| Sentry | 561 | M13 (DV8_UnhealthyInheritance) | 9 | 527 | 22 | 3 | **0.96** | 0.29 | 0.75 | 0.96 | 0.45 | **0.85** |
| Tajo | 1557 | M14 (DV8_UnstableInterface(h)) | 65 | 1221 | 262 | 9 | 0.83 | 0.20 | **0.88** | 0.82 | 0.32 | **0.85** |
| Tez | 839 | M15 (DV8_PresentInIssues) | 45 | 710 | 60 | 24 | 0.90 | 0.43 | 0.65 | 0.92 | 0.58 | 0.79 |
| | | M16 (SQ_Issues) | | | | Promedio: | **0.88** | **0.29** | **0.56** | **0.91** | **0.41** | **0.73** |

Los resultados de este experimento se presentan en la Tabla 25, donde podemos observar que tenemos dos aplicaciones (Sentry y Tajo) con una AUC de 85%. De ellas podemos destacar a Sentry dado que obtuvo un 96% de exactitud, un 75% de sensibilidad y un 96% de especificidad. Tenemos a Deltaspike como la aplicación peor evaluada con un 50% en AUC, esto debido a que obtuvo un 0% en sensibilidad y un 99% en especificidad, es decir, no obtuvo ningún verdadero positivo (TP) y por el contrario casi obtuvo todos los verdaderos negativos (TN). En este experimento solo se obtuvieron dos aplicaciones con un AUC mayor o igual al 85%.

La combinación de métricas seleccionada que nos ofreció los mejores resultados en este experimento está conformada por diez métricas. De estas, siete (M01, M03, M05, M12, M14,

M15, M16) son métricas estructurales y tres (M07, M11 y M13) son métricas históricas. La métrica M11 mantiene la consistencia de estar presente en todos los experimentos.

## 5.5 Resultados obtenidos

A continuación, se muestra el resumen de los resultados obtenidos en la etapa de experimentación.

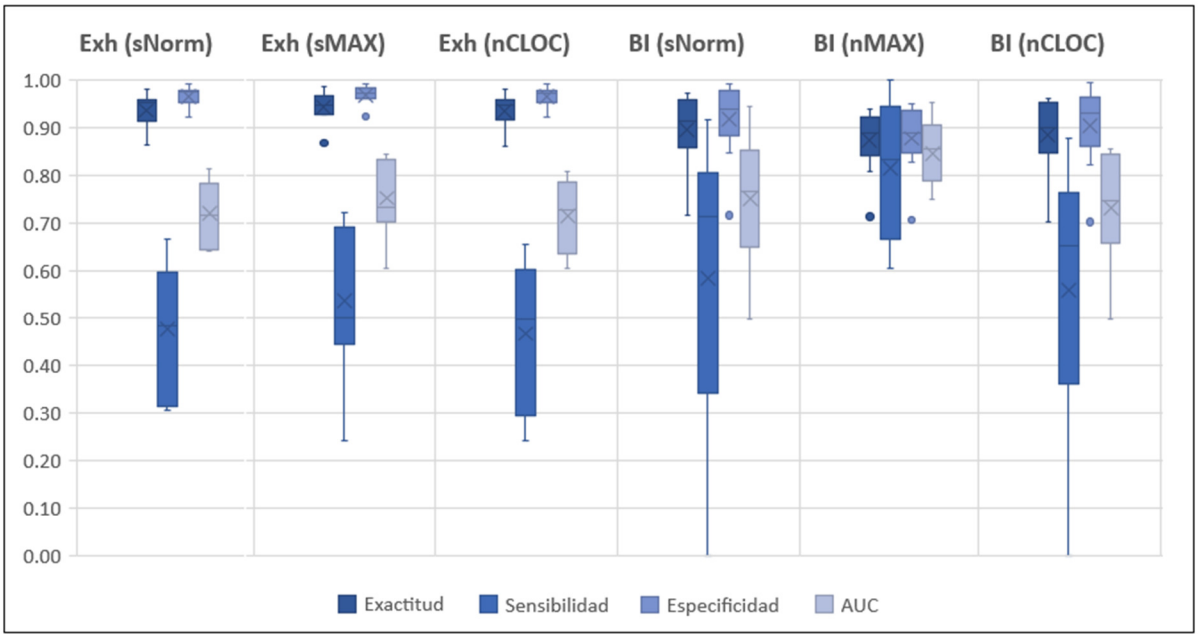
De cada uno de los experimentos se tomaron los promedios de las medidas de rendimiento obtenidos en ellos. Este resumen lo podemos observar en la Tabla 26.

**Tabla 26 Resumen de los resultados promedio de cada experimento.**

| | | Promedio de medidas de rendimiento | | | | | |
|---|---|---|---|---|---|---|---|
| **Experimento** | **Combinación de métricas seleccionada** | **Exactitud** | **Precisión** | **Sensibilidad** | **Especificidad** | **Medida-F** | **AUC** |
| **Exh (sNorm)** | M01, M08, M09, M10, M11, M13, M14, M15, M16 | 0.94 | 0.48 | 0.48 | 0.97 | 0.62 | 0.72 |
| **Exh (nMAX)** | M01, M02, M03, M05, M10, M11, M14, M15, M16 | 0.94 | 0.54 | 0.54 | 0.97 | 0.67 | 0.75 |
| **Exh (nCLOC)** | M02, M08, M09, M11, M12, M15, M16 | 0.94 | 0.47 | 0.47 | 0.96 | 0.61 | 0.72 |
| **BI (sNorm)** | M01, M02, M03, M07, M09, M10, M11, M12, M15, M16 | 0.90 | 0.36 | 0.58 | 0.92 | 0.48 | 0.75 |
| **BI (nMAX)** | M02, M10, M11 | 0.87 | 0.32 | 0.82 | 0.88 | 0.46 | 0.85 |
| **BI (nCLOC)** | M01, M03, M05, M07, M11, M12, M13, M14, M15, M16 | 0.88 | 0.29 | 0.56 | 0.91 | 0.41 | 0.73 |

Sobre las *combinaciones de métricas seleccionadas* tenemos dos experimentos con diez métricas, dos experimentos con nueve métricas, uno con siete métricas y uno con tres métricas.

En la Gráfica 1 mostramos un comparativo de las medidas de rendimiento que se obtuvieron en cada uno de los experimentos.

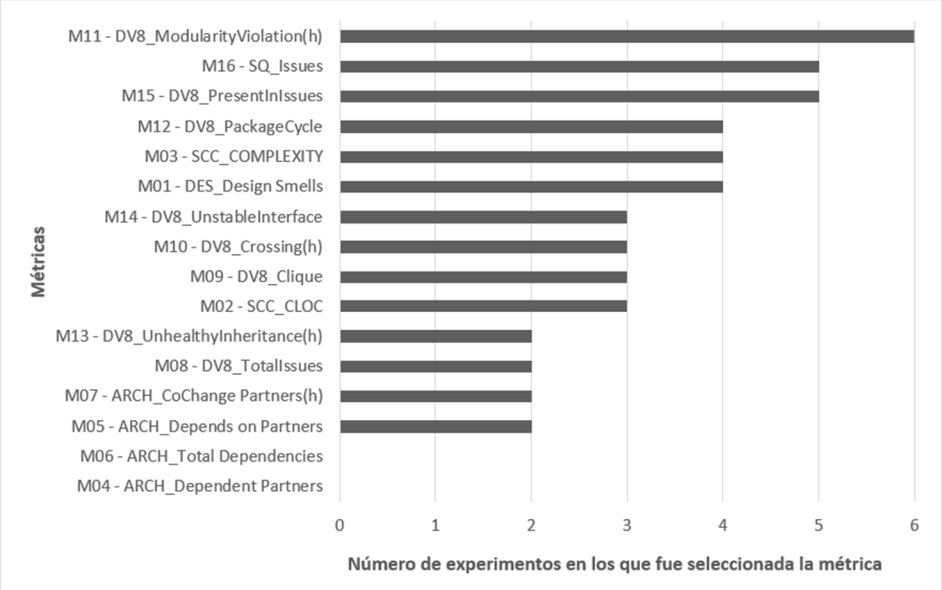



**Gráfica 1. Comparativo del desempeño de los algoritmos con normalización de datos (basado en Tabla 26).**

En esta gráfica podemos observar de manera general el resumen del desempeño obtenido de los algoritmos en las medidas de rendimiento de exactitud, sensibilidad, especificidad y AUC. Podemos ver qué algoritmo tiene los mejores resultados en AUC (BI nMAX) o en sensibilidad y especificidad (BI sNorm y BI nMAX).

En la Gráfica 2, podemos ver el número de veces que fue seleccionada cada métrica en los seis experimentos realizados. La métrica seleccionada en todos los experimentos fue M11, es histórica. M16 y M15 fueron elegidas en cinco experimentos, ambas son estructurales. Con cuatro selecciones tuvimos a las métricas M12, M02 y M01, siendo estas estructurales. De las dieciséis métricas tuvimos catorce que fueron escogidas, por lo menos en dos o más experimentos.

Por último, tuvimos a las métricas M04 y M06 que no fueron seleccionadas en ninguno de los experimentos. Estas métricas representan las dependencias entre archivos y con los resultados obtenidos es interesante ver que el número de dependencias entre archivos no parece tener relación con la DT.




**Gráfica 2. Métricas más seleccionadas en los experimentos.**

## 5.6 Análisis de resultados

A partir de los resultados obtenidos en la etapa de experimentación, se realizó un análisis y se determinó en cuál de los experimentos se obtuvo la combinación de métricas con los mejores resultados para la detección de *hotspots* en las aplicaciones.

### 5.6.1 Análisis desde la perspectiva del aprendizaje automático

Un problema importante en el aprendizaje automático es reducir la dimensionalidad D del espacio de características F (métricas) para superar el riesgo de sobreajuste. El sobreajuste de datos surge cuando el número n de métricas es grande y el número m de patrones de entrenamiento es comparativamente pequeño, según [43]. En este trabajo se tuvieron dieciséis métricas para clasificar y una para etiquetar, por lo que no presentamos sobreajuste en ese sentido. Se tuvieron en promedio 10,000 archivos etiquetados de entrenamiento por 1,000 archivos en promedio para pruebas, por lo que el número de archivos patrones de entrenamiento comparativamente no fue pequeño.

Los resultados obtenidos con el algoritmo exhaustivo y BI los presentamos a continuación:

**Algoritmo de búsqueda exhaustiva**. El cual seleccionó el mejor subconjunto de métricas que satisfizo el criterio de selección dado (mejores valores promedios en la medida de rendimiento AUC) mediante la enumeración exhaustiva de todos los subconjuntos de métricas.

Considerando los resultados utilizando las técnicas de normalización y basándonos en las medidas de rendimiento con las que fueron evaluados los algoritmos tenemos que:

- El algoritmo de búsqueda exhaustiva obtuvo resultados promedio entre 72% y 75% en AUC.

**Algoritmo de aprendizaje automático BI.** BI es un clasificador lineal el cual se considera una herramienta útil en el aprendizaje automático y la minería de datos. Nuestra clasificación fue binaria, es decir, clasificamos archivos como *hotspot* o *no hotspot*. Se realizó una clasificación supervisada dado que se etiquetaron los datos de entrenamiento con los valores de la métrica diecisiete NBC que se tenían disponibles para cada archivo.

Con BI se obtuvieron los mejores resultados en AUC con un 85% combinado con la técnica de normalización de datos nMAX y una combinación de métricas M02, M10 y M11 para la detección de *hotspots*.

Con estos resultados obtenidos, BI sigue siendo un algoritmo vigente y que es capaz de ofrecer buenos resultados aún y con pocos datos como se anticipó en la selección del algoritmo. Esto coincide con [43] donde BI tiene buenos resultados en AUC combinando métricas estructurales con métricas históricas.

### 5.6.2 Análisis desde la perspectiva de ingeniería de software

Desde un punto de vista de ingeniería de software podemos justificar la aparición de las siguientes métricas en los experimentos. Para recordar la referencia del origen de las métricas, indicaremos con “-e” cuando se trate de una métrica estructural y “-h” para indicar si es una métrica histórica.

- **M11-h (Número de violaciones a la modularidad en las que participa este archivo)**. Representa uno de los antipatrones que atentan sobre otro de los atributos de calidad de software que es la modularidad. Violación a la modularidad ocurre cuando dos módulos cambian de forma concurrente pero no tienen relación, es decir, existe una dependencia entre estos. Puede suceder, por ejemplo, cuando se hace *copy and paste* (copia y pega) de código en dos o más módulos y se tiene que cambiar en ambos módulos.

  Como lo mencionan en [23], M11 "brinda una capacidad distintiva para identificar archivos cargados de deuda técnica". Dado que los desarrolladores al tomar atajos, por ejemplo, aplicando *copy and paste* de funcionalidad en diferentes módulos, todos estos requerirán inevitablemente ser modificados, si hubiera cambios o errores en esa funcionalidad copiada y, por lo tanto, requerirá de tiempo para realizarlos, pudiendo ser menos el tiempo si fuera solo un módulo común que tuviera esta funcionalidad.

- **M16-e (Número de problemas de SonarQube)**. Para este estudio, esta métrica representó el total de reglas detectadas por SonarQube que incluyen *Code Smell*, *Bugs*, *Vulnerabilities* y *Security Hotspots*.

  En [23] reportan esta métrica como que exhibe una correlación positiva con otras métricas. Esto apoya los resultados obtenidos en esta investigación ya que esta métrica apareció en cinco de los seis (83.3%) experimentos realizados como una métrica seleccionada. Por su conformación de ésta con más de 597 reglas de calidad que aplica SonarQube, se podría ocupar como una más para la detección de *hotspots.*

  Hay que señalar que SonarQube pretende a través de sus métricas medir la DT por lo cual de alguna forma si está siendo efectivo con estos resultados obtenidos.

- **M15-e (¿Cuántos tipos distintos de los problemas que identifica DV8 tiene?)**. DV8 mide la modularidad del software y detecta antipatrones de arquitectura. Los problemas que detecta son: interfaces inestables, jerarquía de herencia no saludable, cruces, dependencias cíclicas y ciclo del paquete.

  También podemos ver que esta métrica se correlaciona al 100% con M16 dado que fueron seleccionadas en los mismos experimentos (cinco) y con M01 en cuatro de los cinco experimentos mencionados. Si tenemos archivos con valores altos en esta

métrica, estaremos ante un probable *hotspot* que potencialmente puedan presentar errores por los antipatrones que tiene. Este podría ser candidato a ser atendido en una etapa de mantenimiento para reducir la DT en el sistema.

- **M12-e (ciclos de paquete)**. Un paquete puede contener definiciones de tipos, clases, interfaces y otros paquetes (subpaquete), dando lugar a estructuras jerárquicas. Los ciclos en el paquete se presentan y contabilizan cuando en una clase c1 se importa un subpaquete (contenido en el mismo paquete que c1) que a su vez utiliza el subpaquete al que pertenece la clase c1. Idealmente, la estructura del paquete de un sistema de software debería formar una estructura jerárquica. Ciclos dentro de un mismo paquete reduce la comprensibilidad y mantenibilidad de un sistema.

  La estructura de paquetes en los sistemas permite a los desarrolladores organizar sus programas en subsistemas. Un sistema bien modularizado permite su evolución al admitir el reemplazo de sus partes sin afectar el sistema completo. Una buena organización de paquetes identificables y colaborativos facilita la comprensión, mantenimiento, prueba y evolución del software. Si tenemos archivos con valores altos en esta métrica nos estarían indicando que el software está decayendo. La modularización se desvía gradualmente, pierde calidad y necesitaría ser reestructurado.

- **M03-e (complejidad en el código)**. Representa tener código con un número alto de caminos lógicos individuales contenidos en un programa. Entre mayores cambios lógicos tenga un archivo mayor será su complejidad.

  Esta métrica con valores altos de cambios lógicos representaría la cantidad de pruebas que tendrían que realizarse al archivo para minimizar la posibilidad de error. Por otro lado, se reporta en [23] que las métricas basadas en el código como las líneas de código o la complejidad se relacionan poco con la propensión al cambio o al error. Pero lo que sí pudimos encontrar en este estudio, es que esta métrica combinada con otras más (M01, M11, M15 y M16), puede dar buenos resultados en la detección de *hotspots*. Esta métrica apareció acompañada de las otras métricas en los tres experimentos (Exh nMAX, BI sNorm y BI nCLOC) en los que fue seleccionada con los mejores resultados reportados para la detección de *hotspots*.

- **M01-e (Cantidad de malos olores de diseño).** Esta métrica representa los olores de diseño reportados por Designite, tales como la jerarquía profunda, la envidia de características y la regla de modularización dependiente del ciclo detecta ciclos entre archivos.

  Esta métrica en [23] comparada en sus resultados con respecto a M16 (número de problemas de SonarQube), para 10 proyectos analizados, tan sólo en uno, coinciden en sus 20 principales archivos seleccionados, lo que indica que los archivos con olores de diseño detectados por M01 son en su mayoría diferentes a los detectados por M16. Lo anterior se debe a dos razones, la primera, porque M01 mide únicamente *Code Smells* y M16 mide además *Vulnerabilities* y *Security Hotspots*. La segunda, es porque las reglas aplican las herramientas Designite y SonarQube para determinar la métrica *Code Smells* pueden ser diferentes.

- **M10-h (Cruces)**. Representa un archivo con una alta relación de entradas y salidas que cambia a menudo con los archivos que dependen de él o con los que él depende.

  Esta métrica representa las relaciones que puede tener un archivo con respecto a otros, por lo que, si este archivo manifiesta errores, los archivos que dependan de él también podrán presentar errores, incluso, si otros archivos de los cuales éste depende podrían de igual manera, presentar error. Entonces esta métrica en este resultado nos representa un buen indicador para detectar *hotspots*. Esto también coincide con [23] en el cual, se reporta como una métrica que se correlaciona de manera positiva con una métrica de cambio.

- **M02-e (conteo de líneas de código)**. Representa el tamaño (número de líneas) de código de cada archivo. Se discriminan comentarios o líneas en blanco con lo cual el conteo es exacto.

  Esta métrica proporciona una aproximación bruta de la complejidad o la cantidad de trabajo realizado. M02 no transmite la complejidad debida, por lo que es importante correlacionarla con la complejidad (M03), ya que podemos tener archivos largos y simples o archivos cortos y complejos como lo mencionan en [23], [43].

Como podemos observar en la revisión de las métricas que destacaron en los diferentes experimentos de este estudio, tenemos combinaciones de métricas tanto estructurales como históricas. De esta manera podemos aprovechar las virtudes de ambos tipos de métricas y no solo las de un tipo, como lo concluyen también en [43], [44]. Lo anterior lo tenemos representado en la combinación de métricas del experimento BI (nMAX) en el que se obtuvieron como métricas seleccionadas a M02, M10 y M11 (Tabla 24), donde la primera estructural y las otras dos son históricas. Dicha combinación funcionó bien debido a que de M02 se obtienen los archivos más grandes que potencialmente son problemáticos para su mantenimiento, de M10 se toman los archivos que tienen dependencias mutuas con cambios frecuentes al mismo tiempo y de M11 se ubican los archivos de cambian juntos sin tener relación alguna. Tenemos entonces que los *hotspots* podrían ser archivos grandes que cambian al mismo tiempo teniendo o no relación alguna. El algoritmo de BI no requiere que haya dependencias entre los archivos respecto a las métricas, de ahí que es posible esta combinación de métrica que nos sugiere.

### 5.6.3 Análisis sobre las medidas de rendimiento

Con base a los resultados de los experimentos y que podemos ver en la Gráfica 1, tenemos que en *exactitud* los tres experimentos con algoritmo exhaustivo obtienen los resultados más altos con un 94% y de los experimentos con el algoritmo de BI el que más destaca es el de sin normalización de datos (sNorm) con un 90%. En *precisión* detectamos al experimento Exhaustivo (nMax) con un 54%. En *sensibilidad,* al experimento Bayes Ingenuo (nMAX) como el que más destaca con un 82%. En *especificidad, media-f y AUC* encontramos al experimento Exhaustivo (nMAX) como el que más destaca con un 97%, 67% y 85% respectivamente.

Si analizamos los resultados desde el punto de vista de la **sensibilidad** (capacidad de detectar los verdaderos positivos -*hotspots*-) en los seis experimentos, tenemos que:

- Para los experimentos con el algoritmo exhaustivo (Tablas 19, 20 y 21) se alcanzaron resultados entre el 66% y 77%.
- Para los experimentos con el algoritmo de BI (Tablas 22, 23 y 24) se alcanzaron resultados entre 88% y 100%.
- El mejor experimento obtuvo un promedio del 82%: BI (nMax).

Esto nos sugiere que el algoritmo de BI ofrece en general mejores resultados para identificar los *hotspots* basándose solo en la sensibilidad. Esta conclusión coincide con los resultados promedio

de todos los experimentos (Tabla 25) donde el valor más alto en sensibilidad lo tenemos en el experimento de BI (nMax).

Ahora, si analizamos los resultados desde el punto de vista del **AUC**, como se expuso previamente, la medida de rendimiento de AUC es la que se tomaría para evaluar un algoritmo, dado que en ella se evalúa la capacidad de determinar tanto los *hotspots* (verdaderos positivos) como los no *hotspots* (verdaderos negativos). Por lo anterior, tenemos que:

- Para los experimentos con el algoritmo exhaustivo (Tablas 19, 20 y 21) se alcanzaron resultados entre el 81% y 84%.
- Para los experimentos con el algoritmo de BI (Tablas 22, 23 y 24) se alcanzaron resultados entre 85% y 95%.
- El mejor experimento obtuvo un promedio del 85%: BI (nMax).

Con los resultados desde la perspectiva de la sensibilidad y del AUC tenemos al experimento de BI (nMAX) como el que obtuvo los mejores resultados.

### 5.6.4 Análisis sobre la normalización de los datos

Con la implementación de los dos tipos de normalización de los datos se pudo aplicar una variación en el conocimiento que ofrecieron los datos de entrenamiento a los algoritmos.

Normalizando por **valor máximo en cada métrica** (nMAX) podemos ver en el resumen de los resultados de la Tabla 26 que en sensibilidad y AUC, para ambos algoritmos, los resultados son más altos.

- Para el algoritmo exhaustivo con normalización nMAX tenemos en sensibilidad un 54% y en AUC un 75%, siendo estos resultados mejores a los obtenidos sin normalizar los datos (48% y 72%) y normalizando datos por nLOC (47% y 72%).
- Para el algoritmo de BI con normalización nMAX tenemos en sensibilidad un 82% y en AUC un 85%, siendo estos resultados mejores a los obtenidos sin normalizar los datos (58% y 75%) y normalizando datos por nLOC (56% y 73%).

Como resultado tenemos que la técnica de normalización que mejores resultados nos ofreció fue la de normalizar por máximo valor en cada métrica (nMAX) en lugar de normalizar por el máximo número de líneas en toda la base de datos (nLOC). De hecho, esta última generó resultados por debajo de los obtenidos sin normalizar.

Con lo anterior, podemos confirmar que la normalización no es solo una técnica que se utiliza para convertir los datos sin procesar en un conjunto de datos limpio, sino que también mejora el rendimiento del aprendizaje automático.

## 5.7 Validación de resultados

Para validar la efectividad de los resultados que se obtuvieron en la sección 5.5, se procedió a ejecutar el algoritmo de BI normalizando datos por nMAX y con la combinación de métricas seleccionada M02, M10 y M11 a dos aplicaciones adicionales reservadas para clasificar sus archivos en *hotspots* o no, Amqp-0x y Broker-j. Estas aplicaciones no participaron en los experimentos realizados previamente en la sección 5.4. Como podemos observar en la Tabla 12, para Amqp-0x se tienen 561 archivos de los cuales 12 (2.1%) son etiquetados como *hotspots* y para Broker-j se tienen 1,865 archivos de los cuales 92 (4.9%) son etiquetados como *hotspots*.

Para la ejecución de la clasificación, se tomaron como datos de entrenamiento todos los datos de las nueve aplicaciones con las que se experimentó previamente y los datos de prueba fueron los de ambas aplicaciones a validar (por separado).

**Tabla 27 Resultados al evaluar Broker-j y Amqp-0x con todos los experimentos.**

| | | | Matriz de confusión | | | | Medidas de rendimiento | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Aplicación | Archivos a evaluar | Archivos Hotspots | TP | TN | FP | FN | Exactitud | Precisión | Sensibilidad | Especificidad | Medida-F | AUC |
| **Broker-j** | 1865 | 92 | 76 | 1602 | 171 | 16 | 0.90 | 0.31 | 0.83 | 0.90 | 0.46 | 0.86 |
| **Amqp-0x** | 561 | 12 | 12 | 497 | 52 | 0 | 0.91 | 0.19 | 1.00 | 0.91 | 0.31 | 0.95 |

Los resultados obtenidos los podemos observar en la Tabla 27, donde por un lado tenemos a la aplicación Broker-j para la cual se obtuvo una AUC de 86%, sustentado con un 83% de sensibilidad y un 90% de especificidad. Por otro lado, tenemos a la aplicación Amqp-0x para

la cual se obtuvo una AUC de 95.3%, sustentado por un 100% en sensibilidad y un 90.5% en especificidad.

Tenemos una exactitud (capacidad de predecir los archivos *hotspots* como los no *hotspots* con respecto del total de archivos revisados, ver Tabla 7) de 90% para Broker-j y del 91% para Amqp-0x, estos resultados destacan con respecto a los porcentajes reportados en los trabajos [16], [17], [18], [19] [20], [24] revisados en el estado del arte (Tabla 4). En ellos tenemos una exactitud como máximo del 85% y no reportan sensibilidad ni AUC.

Para Amqp-0x se obtuvo un 100% de sensibilidad, es importante mencionar que, cuando los archivos revisados fueron 561, en ellos solo se encontraron 12 (2.1%) archivos como *hotspots* y estos fueron clasificados en su totalidad.

Esto confirma los resultados obtenidos previamente en la experimentación. El algoritmo de aprendizaje automático BI, junto con la normalización nMAX y la combinación de métricas seleccionadas M02, M10 y M11, son una buena sugerencia para la detección de *hotspots*.

Esto puede ayudar a la ingeniería de software en la tarea continua de mantener los atributos de calidad en un nivel aceptable para tener software funcional a largo plazo. De manera específica, los atributos de calidad que se verían beneficiados son el de mantenibilidad y la capacidad de evolución, permitiendo que los desarrollos se realicen con mayor rapidez [1].

Por otro lado, la clasificación podrá orientar en varios sentidos a los ingenieros de software, no solo en las tareas de diseño, construcción o mantenimiento del software, sino también en otras áreas como la presupuestación, la organización y planeación de los equipos de trabajo [40].

## 5.8 Amenazas a la validez

Debido a que el estudio parte de algunos supuestos que podrían no darse y por lo tanto amenazar la validez del estudio. A continuación, describimos algunos de ellos:

- Partimos de la métrica de número de *bug commits* (NBC), la cual se genera si y sólo si los desarrolladores realizan la relación entre el *commit* y el número de incidente que

motivó cambios en los archivos modificados. De no encontrar aplicaciones con esta métrica, el estudio no podría replicarse.

- Para la obtención de las métricas se seleccionaron un conjunto de herramientas de análisis de código (HAC), pero podrían utilizarse otras para la obtención de las métricas. Como lo afirman en [23], hay métricas que son ofrecidas por diferentes HAC y no coinciden entre ellas en algunos casos ni en la métrica más básica como el número de líneas de código. Si se seleccionan otras HAC podrían variar los valores de las métricas y por lo tanto el resultado.
- Las aplicaciones con las que se realizó el estudio están desarrolladas en lenguaje de programación Java. Faltaría validar que las métricas finales para la detección de *hotspots* funcionan para aplicaciones desarrolladas en otros lenguajes de programación.

# Capítulo 6

# Conclusiones

En esta investigación se compararon los resultados obtenidos de un algoritmo exhaustivo con el algoritmo de Bayes ingenuo en su versión gaussiana para la identificación de *hotspots* en las aplicaciones. Utilizando métodos de normalización de los datos para probar variantes de estos. Y como solución buscamos una combinación de métricas que nos permita obtener los más altos resultados en la identificación de *hotspots*.

Ahora se procede a presentar las conclusiones a las que se llegaron después de presentar la metodología propuesta y su implementación.

## 6.1 Respuestas a las preguntas de investigación

Con los resultados obtenidos en las etapas de experimentación y validación se procedió a dar respuesta a las preguntas de investigación planteadas en la sección 3.4 e ir formulando conclusiones.

**¿De un conjunto de métricas dadas, existen una o varias combinaciones de métricas que puedan ofrecer mejores resultados que otras, en la detección de *hotspots* en las aplicaciones de software?**

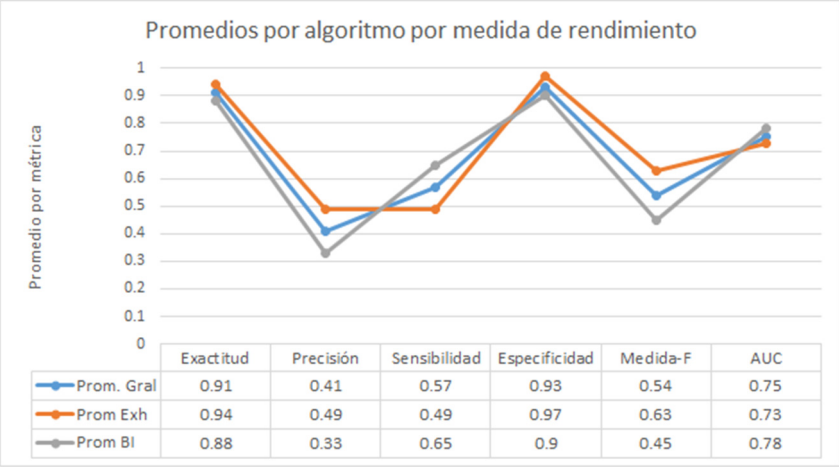
Como se pudo observar en los resultados promedio que se obtuvieron en cada experimento presentados en la Tabla 26, en cada uno de ellos hay una *combinación de métricas sugerida* que ofrecen los mejores resultados en la detección de *hotspots*. Encontramos combinaciones de métricas que utilizan de tres a diez métricas. Justo el experimento que utiliza el menor número de métricas (tres) es el que nos ofrece el mejor rendimiento para la obtención de *hotspots*, a saber, BI (nMAX). El hecho de que se utilicen pocas métricas favorece al uso de la metodología propuesta dado que la obtención de las métricas propuestas para la experimentación (Tabla 11) no es fácil obtenerlas en algunos casos.

**¿Las métricas para detectar *hotspots* de una aplicación pueden funcionar para determinar *hotspots* en otras aplicaciones?**

Para buscar y dar respuesta a esta pregunta se realizó una etapa de validación en la implementación. Se utilizaron dos aplicaciones, Broker-j y Amqp-0x, ajenas a todo el proceso de experimentación para validar los resultados obtenidos. Al realizar la validación sobre las dos aplicaciones con el mejor resultado en la experimentación, algoritmo de BI (nMAX) y combinación de métricas M02 (tamaño en líneas de código físico), M10 (número de cruces en los que participa este archivo) y M11 (número de violaciones a la modularidad en las que participa este archivo), obtuvimos los resultados mostrados en la Tabla 27. Podemos observar en ella que los resultados para ambas aplicaciones, Broker-j y Amqp-0x, están por arriba del mejor resultado obtenido (85%) en la experimentación con un 86% y 95% en AUC respectivamente. Esto nos lleva a sugerir el uso de esta combinación de métricas para la detección de *hotspots* en otras aplicaciones.

**¿Cuál será el desempeño de un método de clasificación teniendo un método de referencia (exhaustivo)?**

El algoritmo de clasificación que fue elegido fue Bayes ingenuo en su versión gaussiana y este fue contrastado con un algoritmo exhaustivo. De cada uno de ellos se plantearon tres experimentos realizando variaciones de los datos, normalizándolos.




| | Exactitud | Precisión | Sensibilidad | Especificidad | Medida-F | AUC |
|---|---|---|---|---|---|---|
| Prom. Gral | 0.91 | 0.41 | 0.57 | 0.93 | 0.54 | 0.75 |
| Prom Exh | 0.94 | 0.49 | 0.49 | 0.97 | 0.63 | 0.73 |
| Prom BI | 0.88 | 0.33 | 0.65 | 0.9 | 0.45 | 0.78 |

**Gráfica 2. Comparativo de rendimiento del algoritmo exhaustivo y Bayes ingenuo.**

Tomando los resultados promedio de cada medida de rendimiento para cada algoritmo de la Tabla 26 y graficando estos por medida de rendimiento como se muestra en la Gráfica 2, tenemos que el algoritmo exhaustivo está por arriba de BI y del promedio general en cuatros de las medidas de rendimiento (exactitud, precisión, especificidad y medida-f), pero dado el enfoque de la investigación, en el que se eligió a AUC como la medida de rendimiento decisoria, tenemos que BI es el que destaca en AUC (78%) y también en sensibilidad (capacidad de seleccionar los verdaderos positivos -*hotspots*-) con 65%. Entonces con los anteriores resultados podemos determinar que BI es el que mejor nos ayuda para la detección de *hotspots*, sin embargo, exhaustivo también ofrece buenos resultados y podría ser considerado.

## 6.2 Conclusiones

La importancia de ubicar los *hotspots* en las aplicaciones es de gran ayuda para los equipos de desarrollo al momento de decir en qué se invierten los presupuestos de mantenimiento. El trabajar en la disminución de *hotspots* abona de manera positiva a la disminución de la deuda técnica, lo cual ayuda a tener aplicaciones mantenibles y que se les pueda agregar nuevas funcionalidades fácilmente.

El objetivo general planteado en esta investigación fue “Aplicar métodos de clasificación para identificar *hotspots* y tener más elementos de decisión para reducir la deuda técnica en las aplicaciones de software”, fue alcanzado satisfactoriamente, ya que se propuso una metodología clara para la identificación de *hotspots*. Esta es apoyada de un par de algoritmos y determinar cuál de ellos resulta más apropiado para la detección de los *hotspots* en los sistemas. Se analizaron alternativas de normalización de los datos, donde se encontró que normalizar con base en el valor máximo de cada métrica nos dio mejores resultados que otra normalización o sin normalizar. Además de encontrar una combinación de tan solo tres métricas de dieciséis que fueron propuestas inicialmente, lo que podría hacer más sencilla la obtención de los datos para la detección de los *hotspots*. Para lograr lo antes descrito, se tuvo que completar cada uno de los objetivos específicos.

Se logró identificar en la literatura que existen diversos trabajos que tratan de identificar los archivos de un sistema que son propensos al cambio o propensos al error (*hotspots*). En

todos los casos tratan de identificar los *hotspots* a través de las métricas estructurales en la mayoría de los casos y solo algunos a través de métricas históricas que son obtenidas de los sistemas de control de versiones.

Posteriormente, se evaluó un par de algoritmos, uno exhaustivo que nos permitió explorar todas las soluciones posibles y contrastando éstos con los resultados de otro algoritmo de clasificación de aprendizaje automático supervisado Bayes ingenuo en su versión gaussiana. Para determinar cuál de los algoritmos sería el que finalmente se propondría, se utilizaron las medidas de rendimiento más comunes para evaluar algoritmos de clasificación. De las medidas de rendimiento calculadas la que se le dio el peso más importante fue a AUC, dado que en ella se refleja tanto la capacidad del algoritmo de identificar los *hotspots* como los *no hotspots* (sensibilidad y especificidad respectivamente). Con base en el AUC se determinó que el algoritmo que se propone para la identificación de *hotspots* es Bayes ingenuo. Junto con el algoritmo seleccionado se encontró la combinación de métricas que ofreció los mejores resultados en AUC, la cual está conformada por M02 (tamaño en líneas de código físico), M10 (número de cruces en los que participa este archivo) y M11 (número de violaciones de modularidad en las que participa este archivo). M11 fue la métrica que participa en todos los resultados de los diferentes experimentos lo que también se resalta como una métrica que favorece la detección de *hotspots*. La métrica M02 está presente en nuestros resultados como en la mayoría de los estudios realizados previamente y revisados, esto se debe a que a mayor número de líneas mayor la probabilidad de incurrir en errores [23].

Dado que solo se requieren de tres métricas (finales) para la detección de *hotspots*, entonces se requerirían de menos herramientas de análisis de código (HAC), facilitando la obtención de estas. Las herramientas de la que provienen las tres métricas finales son DV8 y SCC.

Por último, la efectividad de los resultados fue validada satisfactoriamente con dos aplicaciones ajenas a la etapa de experimentación, en los cuales se confirman los resultados obteniendo con un AUC por arriba (entre 86% y 95%) de lo obtenido en la etapa de experimentación (85%) y en los estudios previos similares a este [16], [19], [20].

## 6.3 Contribuciones

Las contribuciones que se realizaron con este trabajo son las siguientes:

1. Se presentan dos algoritmos para la identificación de *hotspots* con una técnica de manejo de los datos de entrenamiento y prueba.
2. Se aplicó la normalización de datos por máximo valor en cada métrica, máximo valor en líneas de código para obtener valores más homogéneos y validar si esto influye en la detección de *hotspots*.
3. La medida de rendimiento de AUC se propone como la más idónea para evaluar los algoritmos en su capacidad de detectar los *hotspots* y *no hotspots*.
4. Se obtuvo una combinación de métricas reducidas de todas las utilizadas, que ofreció los mejores resultados y lo más importante, es que solo consta de tres métricas (M02, M10 y M11). Lo anterior, resulta favorable puesto que, conseguir estas tres métricas resultará mucho más sencillo que las dieciséis métricas que fueron propuestas en la etapa de experimentación.
5. De los trabajos previamente realizados a esta investigación no se había utilizado el algoritmo de Bayes ingenuo como clasificador de *hotspots*, la normalización de datos ni las tres métricas propuestas aquí.

## 6.4 Trabajo futuro

Durante el desarrollo de este trabajo se utilizaron métricas de aplicaciones desarrolladas en lenguaje Java para ser analizadas y que nos permitieron llegar a los resultados presentados, sin embargo, se podría implementar la metodología establecida para aplicaciones desarrolladas en otros lenguajes de programación vigentes, tales como C#, Python, Go, entre otros.

Otra posible área de investigación sería aumentar las bases de datos para que el algoritmo de clasificación aumente su entrenamiento y pueda discernir de manera más apropiada la clasificación de los *hotspots*. La base de datos podría ser no solo de aplicaciones desarrolladas en lenguaje Java sino en otros lenguajes, ya que el principio básico es, obtener las tres métricas que se sugirieron como las que ayudan en la detección de *hotspots*.

También se podría establecer un método sistemático para la obtención de las métricas finales con mayor facilidad y con ellos los equipos de desarrollo puedan aplicar de manera rápida la detección de *hotspots*.

Este trabajo se basó en un *snapshot* de los cambios (que cada aplicación tuviera por lo menos 1,000 *bug commits*), ver Tabla 8, pero se podría hacerse un análisis temporal en el que se tomen las métricas por cada n *bug commits* o por cada x meses. Esto podría probablemente dar tendencia de *hotspots* por temporalidad.

# Referencias

1. P. Avgeriou, P. Kruchten, R.L. Nord , I. Ozkaya, C. Seaman, "*Reducing friction in software development*", THE IEEE COMPUTER SOCIETY, 0740-7459/16, pp. 66-73, 2016. Disponible: https://resources.sei.cmu.edu/asset_files/Article/2016_101_001_463905.pdf.

2. T. Besker, A. Martini, J. Bosch, "*Managing architectural technical debt: A unified model and systematic literature review*", The Journal of Systems and Software, 0164-1212, 2018, Disponible: https://doi.org/10.1016/j.jss.2017.09.025.

3. R. Shatnawi, W. Li, "*The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process*", The Journal of Systems and Software, pp. 1868–1882, 2008, Disponible: DOI:10.1016/j.jss.2007.12.794

4. F. Buschmann, "*To pay or not to pay technical debt*", IEEE Software, Vol. 28, No. 6, pp. 29-31, 2011, DOI: 10.1109/MS.2011.150.

5. D. Godara, R.K. Singh, "*New hybrid model for predicting change prone class*", International Journal of Computer Science and Telecommunications. Vol. 5, Issue 7, 2014, Disponible: ISSN:2047-3338

6. S. Eski, F. Buzluca, "*An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes*", Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, Disponible: DOI:10.1109/ICSTW.2011.43

7. M. Yanb, X. Zhanga, C. Liub, L. Xub, M. Yang, D. Yang, "*Automated change-prone class prediction on unlabeled dataset using unsupervised method*", ELSEVIER Information and Software Technology, 2017, Disponible: http://dx.doi.org/10.1016/j.infsof.2017.07.003

8. S. G. Maisikeli, "*Evaluation of software degradation and forecasting future development needs in software evolution*", International Journal of Software Engineering & Applications (IJSEA), Vol.7, No.6, 2016, Disponible: DOI : 10.5121/ijsea.2016.7604

9. M. O. Elish, M. Al-Rahman Al-Khiaty, "*A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software*", JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS (Online Library), 2012, Disponible: DOI: 10.1002/smr.1549

10. E. Tello-Leal, C. M. Sosa, D. A. "*Revisión de los sistemas de control de versiones utilizados en el desarrollo de software*", Ing. USBMed, Vol. 3, No. 1, pp. 74-81, 2012, Disponible: ISSN: 2027-5846.

11. T. M. Mitchell, "*Does machine learning really work?*," AI Magazine, Vol. 18 No. 3 (1997), DOI: https://doi.org/10.1609/aimag.v18i3.1303

12. I. Goodfellow, Y. Bengio, A. Courville, "*Deep learning*", MIT Press, 2016, Disponible: http://www.deeplearningbook.org

13. R. Martí, "*Procedimientos metaheurísticos en optimización combinatoria*", Departament d'Estadística i Investigació Operativa, Facultat de Matemátiques, Universitat de Valéncia, 2001.

14. S. R. Chidamber, C. F. Kemerer, "A metrics suite for object-oriented design", IEEE Transactions on Software Engineering, pp. 467–493, 1994.

15. D. Bura, M. Rachna, A. Choudhary, M. Surajmal, R. K. Singh, B. T. Kumaon, "*A novel UML based approach for early detection of change prone classes*", International Journal of Open Source Software and Processes, Volume 8, Issue 3, 2017, Disponible: DOI: 10.4018/IJOSSP.2017070101

16. A. Bansal, S. Jajoria, "*Cross-project change prediction using meta-heuristic techniques*", International Journal of Applied Metaheuristic Computing, Vol. 10, Issue 1, 2019, Disponible: DOI: 10.1109/APSEC.2005.69

17. J. R. Birt, R. Sitte, "*Identifying error proneness in path strata with genetic algorithms*", 12th Asia-Pacific Software Engineering Conference (APSEC'05), 2005, Disponible: DOI: 10.1109/APSEC.2005.69

18. S. McIntosh, B. Adams, M. Nagappan, A. E. Hassan, "*Identifying and understanding header file hotspots in C/C++ build processes*", Automated Software Engineering, 2015, Disponible: DOI: 10.1109/ASE.2019.00095

19. M. Al-Khiaty, R. Abdel-Aal, M. Elish, "*Abductive network ensembles for improved prediction of future change-prone classes in object-oriented software*", The International Arab Journal of Information Technology, 2017, Disponible: https://www.researchgate.net/publication/321678919

20. R. Malhotra, M. Khanna, "*Examining the effectiveness of machine learning algorithms for prediction of change prone classes*", International Conference on High Performance Computing & Simulation (HPCS), 2014, Disponible: DOI: 10.1109/HPCSim.2014.6903747

21. Q. Feng, Y. Cai, R. Kasman, D. Cui, T. Liu, H. Fang, "*Active hotspot: An issue-oriented model to monitor software evolution and degradation*", 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, Disponible: DOI: 10.1109/ASE.2019.00095

22. D. Rodríguez, R. Harrison, "*Medición para la gestión en la ingeniería del software*". Capítulo 4: Medición en la orientación a objetos, España, RA-MA, 2000.

23. J. Lefever, Y. Cai, H. Cervantes, R. Kazman, H. Fang, "*On the lack of consensus among technical debt detection tools*", 43d International Conference on Software Engineering, Software Engineering In Practice Track (ICSE 2021), Madrid, Spain (virtual), May 23-29, 2021.

24. R. Hilton, E. Gethnerd. "*Predicting code hotspots in open-source software from object-oriented metrics using machine learning*", International Journal of Software Engineering and Knowledge Engineering, Vol. 28, No. 3, pp. 311–331, 2018, Disponible: https://www.worldscientific.com/doi/abs/10.1142/S0218194018500110

25. P. Cornejo, "*Comparación de modelos de curvas ROC para la evaluación de procedimientos estadísticos de predicción en investigación de mercados*", tesis doctoral, Universidad Complutense de Madrid, UCM, M, España, 2004, Disponible: http://concejero.wikidot.com/tesis

26. K. Lai, N. Twine, A. O'Brien, Y. Guo, D. Bauer, "*Artificial intelligence and machine learning in bioinformatics*", Encyclopedia of bioinformatics and computational biology, Vol. 1, pp. 272-286, 2019, ISBN: 978-01-12-811414-8.

27. A. Kelly; M. A. Johnson, "*Investigating the statistical assumptions of Naïve Bayes classifiers*", 55th Annual Conference on Information Sciences and Systems (CISS), 2021, Disponible: DOI: 10.1109/CISS50987.2021.9400215

28. A. Bachmann, C. Bird, F. Rahman, P. Devanbu, A. Bernstein, "*The missing links: bugs and bug-fix commits*", Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Disponible: DOI:10.1145/1882291.1882308

29. P. Refaeilzadeh, L. Tang, H. Liu, "*Cross-validation*", Arizona State University, ASU, AR, USA, 2008.

30. H. Cervantes and R. Kazman, "*Software archinaut - a tool to understand architecture, identify technical debt hotspots and control its evolution,*" in International Conference on Technical Debt (TechDebt'2020), 2020, p. to be presented.

31. "*Reduce technical debt and improve maintainability*", Designite, 2021. [Online]. Available: https://www.designite-tools.com/. [Accessed: 12-Oct-2021].

32. "*SCC is a very fast accurate code counter with complexity calculations*", Succinct Code Counter, 2021. [Online]. Available: https://github.com/boyter/scc. [Accessed: 23-Feb-2021]

33. "*Software health management system*", Archdia, 2021. [Online]. Available: https://archdia.com/

34. Y. Cai, R. Kazman, "*DV8: automated architecture analysis tool suites*", ACM International Conference on Technical Debt (TechDebt), 2019, pp. 54-55, DOI: 10.1109/TechDebt.2019.00015.

35. S. Russell and P. Norvig, "*Artificial intelligence: a modern approach, 3rd ed.*", Pearson Education, Inc., New Jersey, NJ, USA, 2010.

36. I. Sommerville, "*Ingeniería de software, 9.ª ed.*", Pearson Education, Inc. México, MEX., 2011. Disponible: ISBN: 978-607-32-0603-7.

37. G. Booch, "*The history of software engineering*", IEEE Software, Vol. 35, Issue: 5, 2018, Deisponible: DOI: 10.1109/MS.2018.3571234.

38. D. Feitosa, A. Ampatzoglou, A. Gkortzis, S. Bibi, A. Chatzigeorgiou, "*Code reuse in practice: benefiting or harming technical debt*", Journal of Systems and Software, Vol. 167, 2020, Disponible: https://doi.org/10.1016/j.jss.2020.110618.

39. J. Huang, J. Lu, C.X. Ling, "*Comparing naive Bayes, decision trees, and SVM with AUC and accuracy*", Third IEEE International Conference on Data Mining, Melbourne, FL, USA, 2003, Disponible: DOI: 10.1109/ICDM.2003.1250975.

40. V. Escandon, "Aplicación de un algoritmo genético multiobjetivo para la replaneación de liberaciones en proyectos ágiles de software", tesis de maestría, Universidad Autónoma Metropolitana, UAM, México, 2019. Disponible: http://tesiuami.izt.uam.mx/uam/aspuam/presentatesis.php?recno=23488&docs=UAMII23488.pdf

41. J. Jou-Mo, "Effectiveness of normalization pre-processing of big data to the machine learning performance", Journal of the KIECS. pp. 547-552, Vol. 14, no. 3, 2019. Disponible: https://doi.org/10.13067/JKIECS.2019.14.3.547

42. R. Verdecchia, "Architectural technical debt identification: moving forward", IEEE International Conference on Software Architecture Companion (ICSA-C), Seattle, WA, USA, 2018. Disponible: DOI: 10.1109/ICSA-C.2018.00018.

43. X. Zhang, "A matrix algebra approach to artificial intelligence", Springer Nature Singapore, Beijing, CHI.,2020, Disponible: https://doi.org/10.1007/978-981-15-2770-8

44. F. Rahman, P. Devanbu, "How, and why, process metrics are better", International Conference on Software Engineering, 2013,
Disponible: https://www.researchgate.net/publication/261120271

45. W. Perdomo, C. M. Zapata, "Software quality measures and their relationship with the states of the software system alpha", Ingeniare. Revista chilena de ingeniería. Vol. 29, no. 2, 2021, Disponible: http://dx.doi.org/10.4067/S0718-33052021000200346

# Apéndice

Tabla de siglas de métricas mencionadas en el texto.

| Tipo métrica | Métrica | Descripción en inglés | Descripción |
|---|---|---|---|
| Estructural | WMC | Weighted Methods per Class | Métodos ponderados por clase |
| | DIT | Depth of Inheritance Tree | Profundidad del árbol de herencia |
| | NOC | Number of Children | Número de hijos |
| | CBO | Coupling Between Object Classes | Acoplamiento entre clases de objetos |
| | RFC | Response for class | Respuesta para la clase |
| | LCOM | Lack of Cohesion of Methods | Falta de cohesión de los métodos |
| | NLOC | Number lines of code | Número de líneas de código |
| | LOC | Lines of code | Líneas de código |
| | NSF | Number of Static Attributes | Número de atributos estáticos |
| | NSM | Number of Static Methods | Número de métodos estáticos |
| | NOF | Number of Attributes | Número de atributos |
| | NOM | Number of Methods | Número de métodos |
| | NORM | Number of Overridden Methods | Número de métodos sustituidos por una super clase |
| | SIX | Specialization Index | Índice de especialización: ((DIT*NORM)/NOM) |
| | SoE | Source of error | Suma de las fuentes de error |
| | CAM | Cohesion among methods | Cohesión entre métodos |
| | CCP | Co changes partners | Cantidad de archivos con los que cambia un archivo |
| | UI | Unstable Interface | Archivo con dependientes y que cambia junto con ellos |
| | DES_Size | Size according to DES | Tamaño en líneas de código según DES |
| | DES_Complexity | Complexity according to DES | Complegidad según DES |
| | DES_Design Smells | Design smell according to DES | Diseño "oloroso" según DES |
| | SCC_LOC | Lines of code according to SCC | Líneas de código según SCC |
| | SCC_CLOC | Count lines of code according to SCC | Cuenta de líneas de código según SCC |
| | SCC_COMPLEXITY | Complexity according to SCC | Complejidad según SCC |
| | ARCH_Dependent Partners | Fan in | Número de archivos que dependen de este archivo (fan in) |
| | ARCH_Depends on Partners | Fan out | Número de archivos de los que depende este archivo (fan out) |
| | ARCH_Total Dependencies | Toal fan in + fan out | Total de fan in + fan out |
| | DV8_LOC | Lines of code according to DV8 | Líneas de código según DV8 |
| | DV8_Total Is sues | Total number of DV8 issues for files | Número total de problemas de DV8 para los archivos |
| | DV8_Clique | Number of strongly connected components in which this file participates | Número de componentes fuertemente conectado en las que participa este archivo |
| | DV8_Crossing | Number of crosses in which a file participates | Número de cruces en los que participa un archivo |
| | DV8_Modularity Violation | Number of modularity violations in which this file appears | Número de violaciones a la modularidad en las que aparece este archivo |
| | DV8_Package Cycle | Number of packet cycles this file participates in | Número de ciclos de paquetes en los que participa este archivo |
| | DV8_Unhealthy Inheritance | Number of unhealthy inheritance hierarchies in which this file participates | Número de jerarquías de herencias en mal estado en las que participa este archivo |
| | DV8_Unstable Interface | Number of unstable interfaces this file participates in | Número de interfaces inestables en las que participa este archivo |
| | DV8_Present In Issues | In how many tool errors is this file present? | ¿En cuántos errores de las herramientas este archivo está presete? |
| | SQ_Issues | Number of SonarQube issues (Issues + Code Smells + Bugs + Vulnerabilities) | Número de problemas de SonarQube (Issues + CodeSmells + Bugs + Vulnerabilities) |
| | SQ_Code Smells | Number code smell of SonarQube | Número de olores de código de SonarQube |
| | SQ_Bugs | Number of SonarQube errors | Número de errores de SonarQube |
| | SQ_Vulnerabilities | Number of SonarQube vulnerabilities | Número de vulnerabilidades de SonarQube |
| | SQ_Security Spots | Number of SonarQube security points | Número de puntos de seguridad de SonarQube |
| Tiempo de ejecución | ED | Execution duration | Duración de ejecución |
| | RTI | Run time information | Información de tiempo de ejecución |
| | CD | Class dependency | Dependencia de clases |
| | Reg | Regularidad de ejecución | Regularidad |
| | Pop | Popularity | Popularidad |

| Tipo métrica | Métrica | Descripción en inglés | Descripción |
|---|---|---|---|
| Evolución del software | BOC | Birth of a class | Nacimiento de una clase |
| | FCH | First time changes | Primera vez que cambio |
| | FRCH | Frequency of changes | Frecuencia de cambio |
| | LCH | Last time changes | Última vez que cambio |
| | WCH | Weighted changes | Cambios ponderados |
| | WCD | Weighted change density | Densidad de cambio ponderada |
| | WFR | Weighted frequency of changes | Frecuencia ponderada de cambios |
| | ATAF | Aggregated change size normalized by frequency of change | Tamaño de cambio agregado normalizado por frecuencia de cambio |
| | TACH | Total amount of changes | Cantidad total de cambios |
| | CHD | Change density of a class | Densidad del cambio de una clase |
| | LCA | Last change amount | Cantidad de últimos cambios |
| | LCD | Last change density | Densidad del último cambio |
| | CSB | Changes since the birth | Cambios desde el nacimiento |
| | CSBS | Changes since the birth normalized by size | Cambios desde el nacimiento normalizado por tamaño |
| | ACDF | Aggregated change density normalized by frequency of changes | Densidad de cambio agregado normalizada por frecuencia de cambios |
| | CHO | Change occurred | Se produjo un cambio |
| | POM | Modification probability (of a class) | Probabilidad de modificación (de una clase) |
| | CC | Change cost | Costo de cambio |
| | Entropía | Entropy | Entropía |
| | IMS | Software maturity index | Índice de madurez del software |
| | EC | Effort COCOMO | Esfuerzo COCOMO |
| | PDNV | Duration period for new versions | Periodo de duración para nuevas versiones |
| | PCC | Class change probability | Probabilidad de cambio de clase |
| | CR | Reconstruction cost | Costo de reconstrucción |
| | TC | Exchange rate | Tasa de cambio |
| | ARCH_Revisions | Number of commits in which a file appears | Número de commits en los que aparece un archivo |
| | ARCH_CoChange Partners | Number of files that changed at the same time as the file in one or more "commits" | Número de archivos que cambiaron al mismo tiempo que el archivo en uno o más "commits" |
| | DV8_Target Change Count **(NBC)** | Number of times a file appears in a commit associated with an error ("bug commit") | Número de veces que aparece un archivo en un commit asociado con un error ("bug commit") |
| | DV8_Target Churn | Rotation for all "bug commits" in which the file appears | Rotación para todos los "bug commits" en los que aparece el archivo |
| | DV8_Change Count | Number of times a file appears in a commit | Número de veces en los que un archivo aparece en un commit |
| | DV8_Change Churn | Rotation for all commits the file appears in | Rotación para todos los commits en los que aparece el archivo |

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

# ACTA DE EXAMEN DE GRADO

No. 00110
Matrícula: 2192802549

Identificación de potenciales hotspots en software.usando métodos de clasificación.

En la Ciudad de México, se presentaron a las 10:00 horas del día 18 del mes de octubre del año 2023 en la Unidad Iztapalapa de la Universidad Autónoma Metropolitana, los suscritos miembros del jurado:

DR. EDUARDO FILEMON VAZQUEZ SANTACRUZ
MTRO. VICTOR HUGO ESCANDON BAILON
DR. ABEL GARCIA NAJERA

Bajo la Presidencia del primero y con carácter de Secretario el último, se reunieron para proceder al Examen de Grado cuya denominación aparece al margen, para la obtención del grado de:

MAESTRO EN CIENCIAS (CIENCIAS Y TECNOLOGÍAS DE LA INFORMACIÓN)

DE: HUMBERTO REYES SAN PEDRO

y de acuerdo con el artículo 78 fracción III del Reglamento de Estudios Superiores de la Universidad Autónoma Metropolitana, los miembros del jurado resolvieron:

Aprobar

Acto continuo, el presidente del jurado comunicó al interesado el resultado de la evaluación y, en caso aprobatorio, le fue tomada la protesta.

HUMBERTO REYES SAN PEDRO
ALUMNO

REVISÓ

MTRA. ROSALÍA SERRANO DE LA PAZ
DIRECTORA DE SISTEMAS ESCOLARES

DIRECTOR DE LA DIVISIÓN DE CBI

Roman Linares Romero
DR. ROMAN LINARES ROMERO

PRESIDENTE

DR. EDUARDO FILEMON VAZQUEZ SANTACRUZ

VOCAL

MTRO. VICTOR HUGO ESCANDON BAILON

SECRETARIO

DR. ABEL GARCIA NAJERA