UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

ANÁLISIS ESTÁTICO DE SOFTWARE

Idónea comunicación de resultados que
para obtener el grado de:
Maestro en Ciencias y Tecnologías de la Información

Presenta:

José Miguel Ortiz Roque

Asesores:

Dr. René Mac Kinney Romero
Ing. Luis Fernando Castro Careaga

Jurado calificador:

Presidente:	Dra. Perla Velasco-Elizondo
Secretario:	Dr. Humberto Cervantes Maceda
Vocal:	Dr. René Mac Kinney Romero

Ciudad de México, Septiembre 2017

Resumen

Uno de los aspectos más importantes a mitigar en el desarrollo de sistemas de alta calidad es el número de defectos presentes en el código fuente debido a que estos podrían manifestarse como fallos durante fases posteriores a la codificación. Por esta razón, se han diseñado múltiples prácticas de ingeniería de software encargadas de descubrir la mayor cantidad posible de dichos defectos, siendo el Análisis Estático Automatizado (ASA) una de las más prometedoras. Sin embargo, aquellas herramientas que se ocupan de llevar a cabo esta práctica presentan una gran desventaja, la cual es la generación de un elevado número de posibles defectos y cuya relevancia es imperceptible a la correcta funcionalidad del sistema (alertas no accionables), provocando un gran consumo de tiempo al momento de inspeccionar cada una de ellas.

Por lo anterior, el presente trabajo de investigación hace uso del aprendizaje maquinal para crear una Técnica de Identificación de Alertas Accionables (AAIT) como una forma de incorporar el ASA al Proceso de Desarrollo de Software (PDS). Para dos proyectos de software ajenos entre sí, se han generado múltiples reportes de alertas de análisis estático, los cuales han sido transformados en conjuntos de vectores de 46 Características de Alerta (CA) que sirven para construir y evaluar diferentes modelos de clasificación de alertas con el fin de aumentar el número de defectos relevantes descubiertos (alertas accionables) luego de concluir la fase de codificación y previo a la fase de pruebas. Los resultados obtenidos muestran que la utilización de modelos internos o externos al proyecto (es decir, la construcción de modelos con base en las alertas de un proyecto y su ejecución sobre las alertas del mismo proyecto o de otro) ofrecen un desempeño promedio (exactitud, precisión y sensibilidad) del 96.4% y del 71.1% respectivamente.

Adicionalmente, el análisis realizado sobre el impacto que produciría la ejecución de nuestro mejor modelo externo predice que se lograría una eficiencia de eliminación de defectos del 90.0% a costa de un aumento del 47.16% sobre el tiempo total invertido en la corrección de dichos defectos respecto a un PDS que no incorpore ASA, permitiendo aumentar el número de defectos relevantes descubiertos en un 42.9% y disminuir el número de alertas irrelevantes en un 56.0%.

Agradecimientos

A la Universidad Autónoma Metropolitana, Unidad Iztapalapa (UAM-I) y al Posgrado en Ciencias y Tecnologías de la Información (PCyTI) por darme la oportunidad de ingresar a la Maestría en Ciencias y Tecnologías de la Información (MCyTI) y proporcionarme una formación integral a lo largo de mis estudios. Asimismo, al Consejo de Ciencia y Tecnología (CONACyT) por otorgarme el financiamiento necesario para la realización de este proyecto de investigación y, en consecuencia, la obtención de este grado académico.

A mis asesores Dr. René Mac Kinney Romero e Ing. Luis Fernando Castro Careaga por depositar su total confianza en mi trabajo, por compartirme parte de su extenso conocimiento e invaluable experiencia profesional, por sus valiosos consejos y grandes enseñanzas, por su paciencia y por su tiempo invertido desde el inicio y hasta la culminación de este proyecto; ambos son excepcionales profesores y excelentes personas, mi admiración y respeto para cada uno de ellos. También, a la Dra. Perla Velasco-Elizondo y al Dr. Humberto Cervantes Maceda por su compromiso, tiempo y dedicación como parte del jurado calificador para la presente idónea comunicación de resultados, por sus valiosos comentarios y por sus sugerencias.

A mis padres Julia Leticia Roque y Celedonio Ortiz por su interminable cariño y por su compresión. A mis hermanos Luis Ángel Ortiz y Hugo Marcos Ortiz por creer siempre en mí, por su apoyo incondicional y por su infinita amistad; son para mí, el motor de mis constantes deseos de superación. Gracias a cada uno de mis profesores, a mis compañeros de generación y a mis amigos y, en general, gracias a aquellas personas que directa o indirectamente dedicaron un poco de su atención durante mi formación como Maestro en Ciencias y Tecnologías de la Información.

Contenido

Lista de acrónimos

Acrónimo	Significado
AAIT	Técnica de Identificación de Alertas Accionables
ASA	Análisis Estático Automatizado
ASAT	Herramienta de Análisis Estático Automatizado
SMT	Herramienta de Métricas de Software
CA	Características de Alerta
PDS	Proceso de Desarrollo de Software
VCA	Vector de Características de Alerta

Lista de figuras

Lista de tablas

1. INTRODUCCIÓN

En la actualidad, vivimos una época en la que las tecnologías de la información forman parte esencial de las actividades diarias de nuestra sociedad: empresarios o investigadores, trabajadores o estudiantes, adultos o jóvenes, todos en algún momento hemos interactuado con sistemas de software de todo tipo y, sin lugar a dudas, hemos experimentado un cierto grado de satisfacción en el uso de estos; un gran software para uso empresarial o para propósitos estadísticos, un cajero automático de un banco o un sistema de control escolar, son tan sólo algunos ejemplos del momento tecnológico por el que atravesamos.

En el contexto de la ingeniería de software, la calidad, entendida como el grado de satisfacción a las necesidades o expectativas de un cliente o usuario al utilizar un sistema de software [1], se convierte entonces en uno de los aspectos más importantes a observar dentro del Proceso de Desarrollo de Software (PDS) y al cual destinar grandes esfuerzos con el fin de generar productos de alta calidad. Por ello, uno de los elementos más importantes (quizá el más crítico) a considerar para producir sistemas de software de calidad es el número de defectos presentes en el código fuente, los cuales habrán de manifestarse como fallos del sistema en fases posteriores del desarrollo. Para contrarrestar esta situación, numerosas prácticas de ingeniería de software han sido diseñadas con el principal objetivo de disminuir el número de defectos y, en consecuencia, lograr una mayor calidad en el producto final. Ejemplos de dichas prácticas son: el análisis estático de software, las revisiones y las inspecciones [1], las pruebas automatizadas (unitarias y de sistema), las pruebas especializadas (de integración), entre otras; todas ellas enfocadas a descubrir el mayor número de defectos posible en fases tempranas del desarrollo de software, en particular, durante la fase de codificación [2].

Sin embargo, las prácticas más utilizadas resultan ser las menos efectivas y las más costosas, proporcionando una baja efectividad en la eliminación de defectos y un alto consumo de tiempo para repararlos una vez descubiertos [2, 3]. Hoy en día y de manera desafortunada, no todos los equipos de desarrollo de software incorporan una práctica tan valiosa (como lo es el análisis estático) a través de una o más herramientas que la automaticen (conocidas como Herramientas de Análisis Estático Automatizado -ASAT, *Automated Static Analysis Tools*-), privándose de los grandes beneficios que esta puede ofrecer: alta efectividad y bajo consumo de tiempo de reparación de defectos con un mínimo de esfuerzo. Definimos una ASAT como una herramienta de análisis estático que automatiza la inspección

del código fuente con el fin de encontrar posibles defectos antes de que estos tengan oportunidad de manifestarse como fallas en tiempo de ejecución [4, 5, 6, 7, 8], las ASAT más populares descubren una amplia variedad de errores, vulnerabilidades de seguridad y malas prácticas de programación [9, 10].

No obstante, una de las grandes desventajas que rodean a las ASAT es la generación de un gran número de posibles defectos y cuya relevancia es imperceptible a la correcta funcionalidad del sistema, provocando un gran consumo de tiempo al momento de inspeccionar cada una de las alertas que estas herramientas reportan [6, 4, 8, 11]; insistimos, no todos las alertas reportadas por la ASAT requerirán ser atendidas y el hecho de repararlas todas para descubrir aquellas que realmente afecten al sistema, provocaría un derroche de esfuerzos por parte del desarrollador. Por esta razón, resulta necesario contar con un mecanismo para mitigar dicho comportamiento y ayudar a los desarrolladores a optimizar sus procesos de desarrollo.

Es así como el presente trabajo de investigación propone insertar una Técnica de Identificación de Alertas Accionables (AAIT, *Actionable Alert Identification Technique*) al PDS con el fin de aumentar el número de defectos relevantes descubiertos y disminuir el número de defectos irrelevantes por reparar siendo estos insertados en la fase de codificación. La AAIT aquí presentada utiliza un enfoque de aprendizaje maquinal [7, 6] para construir modelos de clasificación de nuevas alertas a través de un proceso compuesto por 3 etapas: (1) generación de versiones, (2) generación de características y (3) generación de modelos. Adicionalmente, se presenta un análisis sobre la fiabilidad de incorporar dicha técnica al PDS con el fin de mostrar su impacto sobre el número de defectos descubiertos y su tiempo total de corrección en cada una de las fases del PDS.

1.1. Objetivos

1.1.1. Objetivo general

Presentar una propuesta para optimizar el proceso de desarrollo de software con el fin de aumentar la calidad del producto final a través del análisis de la información de los defectos introducidos durante la fase de codificación.

1.1.2. Objetivos específicos

- Identificar, seleccionar y construir versiones de dos proyectos de software actualmente en uso y con características similares.
- Generar información de los defectos introducidos en el código fuente basada en las alertas generadas por una herramienta de análisis estático automatizado.

- Identificar, analizar y seleccionar las principales características de cada alerta con base en su aportación a la clasificación de nuevos defectos.
- Construir, evaluar y analizar un modelo predictivo capaz de clasificar de manera efectiva nuevas alertas mediante técnicas de aprendizaje maquinal.
- Elaborar una propuesta para aumentar el número de defectos relevantes descubiertos y disminuir el número de defectos irrelevantes por reparar antes de llegar a la fase de pruebas y que han sido introducidos durante la fase de codificación dentro del proceso de desarrollo de software.

1.2. Motivación

Una de las principales características de la ingeniería de software, como una extensión de la ingeniería tradicional, es aplicar un enfoque cuantificable durante el proceso de desarrollo [1] con el fin de garantizar la calidad del mismo. Se estima que más del 40% del esfuerzo empleado para la producción de software [12], y hasta un 80% del costo total [13], se gasta en la fase de evolución, lo cual se traduce en trabajo adicional durante el mantenimiento del producto final. Por lo anterior, descubrir y eliminar el mayor número posible de defectos introducidos durante la fase de codificación, disminuiría el número de defectos detectados en fases posteriores y, en consecuencia, el costo asociado a la reparación de estos. Como se puede ver en la figura 1.1, los defectos introducidos, los defectos descubiertos y el costo de reparación se encuentran estrechamente relacionados entre sí [3], influyendo en los costos y tiempos finales, acortando o alargando un proyecto de software.

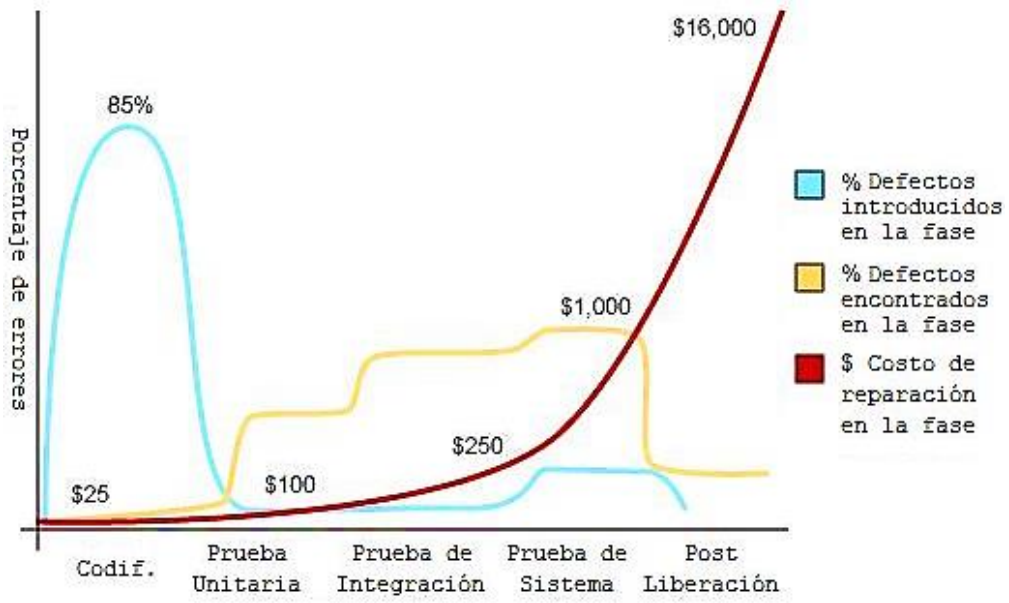


Figura 1.1 Relación entre defectos introducidos, defectos descubiertos y su costo de reparación.

Por otro lado, si centramos nuestra atención en la tabla 1.1 (adaptada del libro de Jones [2]), podemos observar que existen múltiples prácticas de ingeniería de software que se ocupan de aminorar el impacto del problema antes expuesto, proporcionando una alta eficiencia en la eliminación de defectos (es decir, un alto porcentaje de defectos descubiertos) durante las distintas fases del PDS [2]. Estamos interesados en descubrir el mayor número posible de defectos relevantes durante la fase de codificación y, en particular, el Análisis Estático Automatizado (ASA, *Automated Static Analysis*) nos ofrece la eficiencia más alta (con un 87%) de las prácticas señaladas, situándolo en el primer lugar dentro de las principales actividades para la eliminación de defectos, además de ser una de las 10 mejores prácticas de ingeniería de software (sexto lugar) [2]. Por estas razones, estamos motivados a plantear la incorporación de tan valiosa práctica al PDS por medio de nuestra propuesta.

Defect Removal Efficiency by Defect Type

	Activities	Req. defects	Des. defects	Code defects	Test defects
	Static Analysis				
1.	Automated static analysis		30.00%	87.00%	
2.	Requirements inspections	85.00%			
3.	Design inspections		85.00%		
4.	Code inspections			85.00%	
	General Testing				
47.	Unit test			30.00%	
48.	Integration test		10.00%	30.00%	
49.	System test	10.00%	20.00%	25.00%	

Tabla 1.1 Prácticas de ingeniería de software utilizadas para la eliminación de defectos.

Dicho lo anterior, resultaría natural pensar en el uso del ASA dentro del PDS con el fin de generar código fuente de mayor calidad a través de una herramienta (llamada ASAT) que automatice la detección de posibles defectos en el sistema construido. La pregunta es: ¿por qué una herramienta tan útil y prometedora no es utilizada ampliamente y de manera formal dentro del campo? Desafortunadamente, tan sólo el 28.7% de los desarrolladores hacen uso de estas herramientas de análisis estático dentro de sus PDS [14, 15] y, la mayoría de ellos, tiende a rechazar su uso por la siguiente razón: las altas tasas de falsos positivos que estas herramientas generan. Se estima que en promedio, el 64% de las alertas reportadas por una ASAT son falsos positivos [6, 4, 16, 8, 17], este comportamiento constituye una fuerte problemática en la aceptación de esta práctica en el desarrollo de software [18], pues impacta negativamente en el número de alertas que un desarrollador debe reparar para descubrir los defectos realmente relevantes en su sistema, es decir, aquellas alertas que necesiten una pronta atención. Esto podría llegar a ser poco alentador por la siguiente razón:

- Supongamos que una ASAT genera un total de 480 alertas y que inspeccionar cada una de ellas toma 5 minutos aproximadamente [4, 11, 19]. Dado que una jornada laboral es de 8 horas, pasarían 5 días de trabajo sin descanso para que el desarrollador termine con esta ardua labor, y en el peor de los casos, el 91% de dichas alertas serán consideradas de poca relevancia. En general, el desarrollador habría consumido sus esfuerzos de manera innecesaria poco más de 4 días y medio.

Finalmente, es importante mencionar que centramos nuestra atención en la aplicación del ASA al lenguaje de programación Java [20] ya que, de acuerdo con la compañía TIOBE Index [21] y como se puede ver en la figura 1.2, lenguajes como Java, C y C++, si bien han perdido cerca del 25% del mercado actual en la última década debido al surgimiento de nuevos lenguajes más o menos conocidos y a su utilización dentro de los PDS, de alguna forma han conseguido posicionarse como los más populares y los más utilizados; por esto, es importante estudiar una ASAT para el lenguaje de mayor popularidad con el fin de proporcionar soluciones vigentes en la construcción de nuevos sistemas de software.

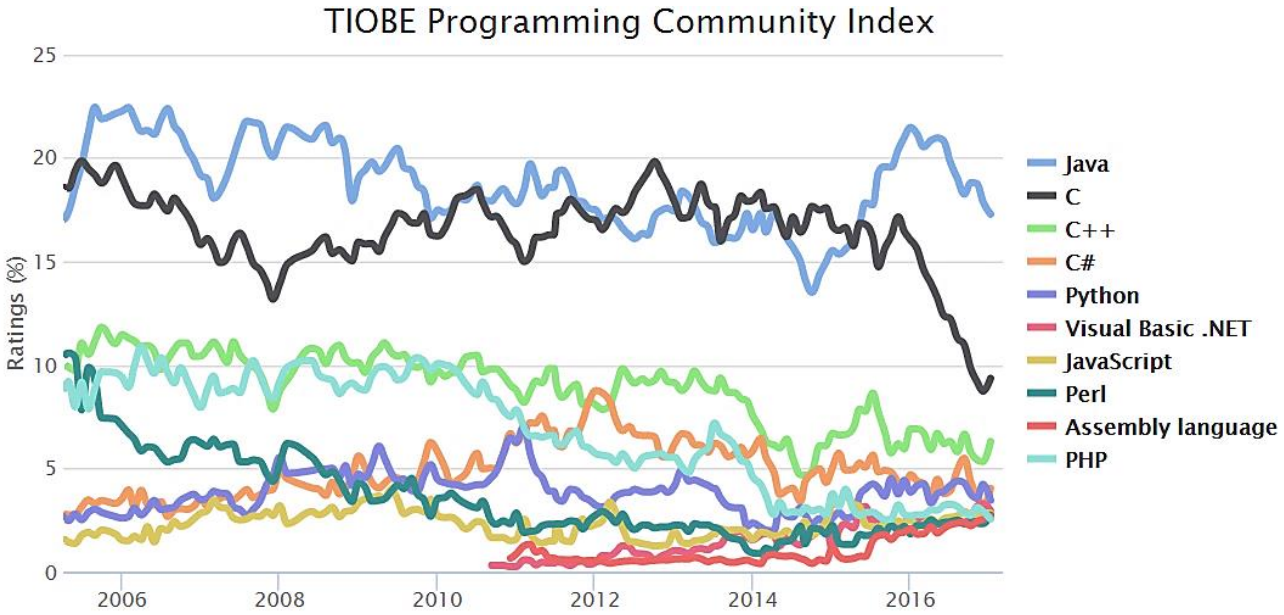


Figura 1.2 Lenguajes de programación más populares en los últimos 10 años.

Por todo lo anterior, el presente trabajo de investigación orienta sus esfuerzos a incidir de manera positiva en la aceptación de las ASAT en el desarrollo de software, a través de una propuesta cuyo objetivo es aumentar el número de defectos relevantes descubiertos y que han sido introducidos durante la fase de codificación, optimizando el PDS e incrementando la calidad del producto final.

1.3. Metodología de investigación

A lo largo de esta sección, mostraremos el proceso de investigación que hemos seguido para desarrollar este estudio, el cual se compone de las siguientes etapas (véase figura 1.3):

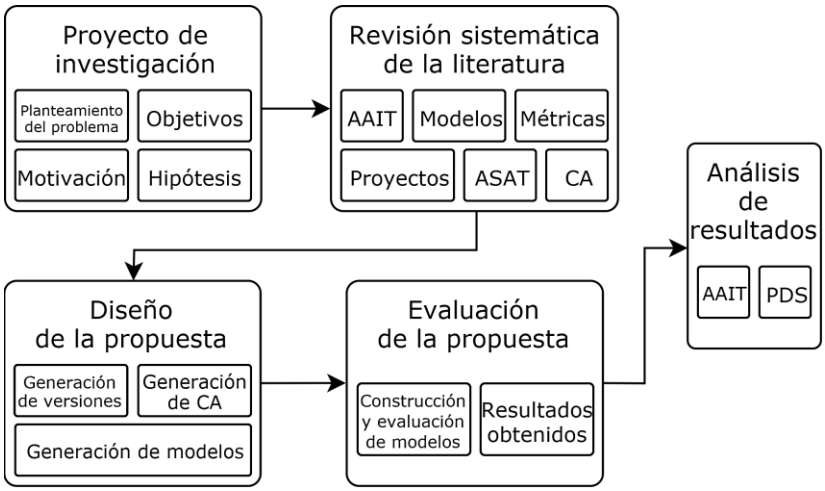


Figura 1.3 Metodología de investigación.

- **Formulación del proyecto de investigación.** Durante esta etapa se define la problemática a resolver y se plantean los objetivos a cumplir, además de establecer una planeación de las actividades a realizar a lo largo del estudio con base en una correcta delimitación del tema a investigar. Adicionalmente, se construye una hipótesis, la cual será sometida a un proceso de aceptación o refutación.
- **Revisión sistemática de la literatura.** A través de esta actividad, se estudian las diferentes técnicas que utilizan un modelo de mitigación de falsos positivos utilizando el enfoque de aprendizaje maquinal. También, se identifican y se analizan las principales etapas que componen a cada técnica (y que mejor se adapten al desarrollo de nuestra propuesta), las herramientas de software utilizadas, los casos de estudio explorados y las medidas de desempeño encargadas de evaluar dichas técnicas.
- **Diseño de la propuesta.** Con base en el conocimiento adquirido en la etapa anterior, se diseña una propuesta cuyo objetivo es aumentar el número de defectos relevantes descubiertos antes de llegar a la fase de pruebas dentro del PDS. Dicha propuesta se basa en las diferentes etapas que los autores utilizan en la creación de sus modelos y en las diversas propiedades de sus AAIT[1], además de incorporar nuevas características. La metodología *training and testing* [22] es parte esencial en el desarrollo de esta etapa.

[1] Las herramientas de software a utilizar durante la siguiente etapa es un ejemplo de dichas propiedades, seleccionamos aquellas que más se adecúen a nuestra propuesta.

- **Evaluación de la propuesta.** Una vez que la propuesta ha sido diseñada, se vuelve necesario implementarla con el fin de realizar una evaluación de la misma y obtener una serie de resultados que nos brinden información acerca del desempeño de todos los modelos generados y en general, del comportamiento de nuestra AAIT. Durante esta etapa, se genera, se identifica, se analiza y se selecciona la información necesaria para la construcción y evaluación de modelos de clasificación de alertas basados en diferentes algoritmos de aprendizaje maquinal. Todas estas actividades son la parte experimental de la propuesta.
- **Análisis de resultados.** Con base en la selección de los mejores modelos (es decir, aquellos que cuentan con el mayor poder predictivo), se realiza un análisis comparativo del desempeño logrado por el conjunto de modelos propios o internos al proyecto y el conjunto de modelos externos al mismo. Posteriormente, se estudia la fiabilidad de incorporar una técnica como la nuestra dentro del PDS; las ventajas, desventajas e implicaciones de la propuesta son discutidas a lo largo de esta etapa.

1.4. Propuesta

La propuesta que se aborda a lo largo de este estudio tiene como objetivo aumentar el número de defectos relevantes descubiertos y disminuir el número de defectos irrelevantes inmediatamente después de la fase de codificación y su contribución principal es la creación de una AAIT basada en la generación de un modelo externo de predicción de nuevos defectos para dos proyectos ajenos entre sí, el cual se encarga de clasificar nuevas alertas apoyado de lo que llamamos reetiquetado de alertas (véase sección 3.2.2.2) y a través de un conjunto de vectores que no incorporan características de temporalidad[2]; adicionalmente, proponemos su incorporación al PDS una vez culminada la fase de codificación, haciendo un análisis sobre su fiabilidad mediante una proyección de su impacto en el número de defectos descubiertos y en el tiempo total de su corrección en cada fase del PDS. Hasta ahora, estudios anteriores sólo han abordado las AAIT bajo la perspectiva de modelos propios o internos al proyecto estudiado, con características de temporalidad y sin tomar en cuenta el tipo de alerta inspeccionada, además de que ninguno de ellos ha intentado analizar de manera cuantitativa el impacto que las AAIT producirían sobre el PDS. Veremos entonces, que la AAIT desarrollada, junto a nuestra propuesta, nos favorecerá para cumplir con los objetivos planteados.

[2] Las características de temporalidad son todas aquellas características que provienen directa o indirectamente del historial del proyecto, algunos ejemplos son: el tiempo de vida de la alerta, la edad y la ranciedad del archivo.

En primer lugar, nos acercaremos a nuestra propuesta desde una visión general. Como podemos ver, en la figura 1.4 se muestra un PDS tradicional, el cual incorpora las distintas etapas de la fase de pruebas (esbozadas en color negro); queremos aprovechar la misma figura para mostrar el resultado de incorporar una nueva fase (esbozada en color rojo) basada en una AAIT inmediatamente después de la codificación y previo a las pruebas. Supongamos que la fase de codificación ha generado un artefacto[3] listo para entrar a la fase de pruebas y que dicho artefacto contiene un total de 100 defectos relevantes en el código fuente. Adicionalmente, asociamos cada una de las fases con su respectiva eficiencia de eliminación de defectos tomada del libro de Jones [2] tal y como se muestra en la figura 1.4.

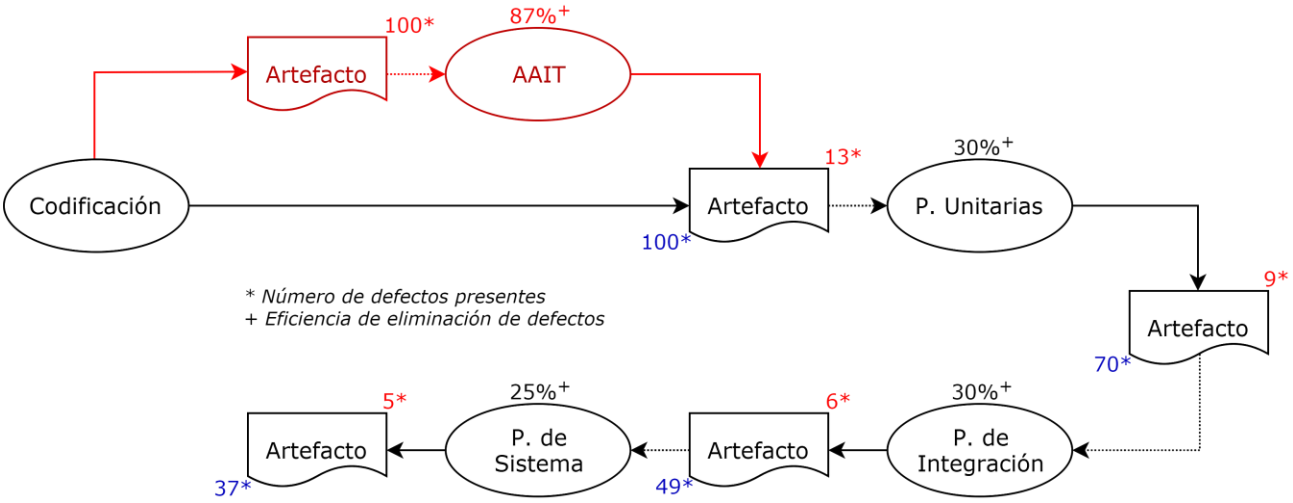


Figura 1.4 Visión general de la propuesta.

Enfoquemos particular atención al número de defectos presentes en cada uno de los artefactos (escritos en color rojo y azul), observemos que al avanzar por las distintas fases del desarrollo, ocurre una diferencia notoria entre el número de defectos presentes en cada artefacto cuando el PDS ha incorporado una AAIT (escrito en color rojo) y cuando no lo ha hecho (escrito en color azul), obteniendo una efectividad del 95% para cuando las AAIT es utilizada y del 67% para cuando no lo es, quedando 5 y 37 defectos presentes en el artefacto final respectivamente, defectos que habrán de ser descubiertos por el usuario final.

Claramente desearíamos que el usuario final encuentre el menor número posible de defectos en el sistema de software que utiliza, pero no sólo eso; llegar al artefacto final con un número reducido de defectos tiene sus implicaciones en cuanto a su costo de reparación. Recordemos que el tiempo requerido para reparar un defecto crece de manera exponencial conforme transcurren las distintas fases en el PDS [3] y supongamos que, el tiempo para reparar un defecto luego de aplicar una AAIT al

[3] Entendemos al término artefacto como un producto terminado o entregable, en particular, la fase de codificación producirá código compilado o un ejecutable listo para ser probado.

artefacto es de 0.5 horas, después de las pruebas unitarias es de 1 hora, con la llegada de las pruebas de integración será de 2 horas y finalmente, repararlo después de pasar por las pruebas de sistema tomará 4 horas. La tabla 1.2 muestra este comportamiento.

Tipo / Fase	AAIT	P. Unitaria	P. de Integración	P. de Sistema	Total
Con AAIT	43.5 hrs	4 hrs	6 hrs	4 hrs	**57.5 hrs**
Sin AAIT	0 hrs	30 hrs	42 hrs	48 hrs	**120 hrs**

Tabla 1.2 Tiempo de corrección de defectos.

Al observar la tabla 1.2, inicialmente podríamos pensar en una inversión mayor de tiempo para reparar los defectos presentes en el artefacto una vez que se ha incorporado una AAIT en el proceso de desarrollo, sin embargo, si revisamos el tiempo final invertido (columna Total), notaríamos que ocurre una disminución de tiempo del 52%, lo que haría completamente viable la utilización de una AAIT dentro del PDS.

Por último, resulta conveniente mencionar que las suposiciones hechas antes, tienen como único objetivo ilustrar nuestra idea de incorporar una AAIT al PDS y los cálculos realizados están basados en los valores reportados en la literatura, tanto los valores de eficiencia de eliminación de defectos, como los valores para los tiempos de corrección en cada una de las fases. A lo largo del capítulo 5, retomaremos nuevamente esta idea pero con valores debidamente ajustados con los resultados experimentales obtenidos, situación que nos ayudará a comprender las ventajas, desventajas e implicaciones de nuestra propuesta.

1.5. Hipótesis

Con base en la problemática a resolver, los objetivos planteados y la motivación que justifica nuestro estudio, establecemos la siguiente hipótesis: es posible aumentar el número de defectos relevantes descubiertos y disminuir el número de defectos irrelevantes por reparar dentro del código fuente antes de llegar a la fase de pruebas mediante la construcción y utilización de un modelo externo para la clasificación de nuevas alertas basado en la información generada por la ASAT.

1.6. Estructura del documento

A continuación, describimos de manera breve el contenido de este documento:

- En el capítulo 2 presentamos, relatamos y comparamos el trabajo relacionado a las AAIT que anteceden a este estudio, citando los resultados más relevantes descubiertos por los diferentes

autores. Examinamos las principales etapas de las cuales se compone cada AAIT, los proyectos de software analizados y sus características, las ASAT, los algoritmos y las medidas de desempeño utilizadas, además de las Características de Alerta (CA) seleccionadas. También, reconocemos y discutimos algunos puntos de mejora al conocimiento existente para posteriormente plasmarlos dentro de nuestra propuesta.

- En el capítulo 3 establecemos el diseño de nuestra propuesta con el fin de aumentar el número de defectos relevantes descubiertos antes de llegar a la fase de pruebas. Dicho diseño, se basa en algunas de las propiedades de las AAIT revisadas en el capítulo 2 e incorpora algunas modificaciones a los modelos ya existentes. Describimos cada una de las etapas que componen la propuesta planteada: (1) la generación de versiones del proyecto en cuestión dado un historial, (2) la generación de características que habrán de representar a las alertas generadas por la ASAT y (3) la generación de modelos a través de la selección de las principales CA y de la ejecución de distintos algoritmos de aprendizaje maquinal, además de evaluar su desempeño con base en su exactitud, precisión y sensibilidad.
- En el capítulo 4 detallamos la forma de implementar el diseño propuesto durante en el capítulo 3 para crear diferentes modelos de clasificación de nuevas alertas, mencionando los algoritmos de selección de características y de aprendizaje utilizados y los resultados conseguidos luego de evaluar cada uno de dichos modelos. También, damos detalle acerca de los proyectos de software, las alertas generadas por las ASAT y las preferencias de reetiquetado utilizadas. Por último, recreamos la construcción de modelos internos para cada proyecto (basados en el trabajo de Heckman y Williams [7, 6]), los evaluamos y reportamos los resultados alcanzados con el fin de complementar nuestra propuesta.
- En el capítulo 5 analizamos y discutimos los resultados obtenidos durante la selección del conjunto de mejores modelos generados en el capítulo 4, comparando el desempeño logrado entre el conjunto de modelos propios o internos al proyecto y el conjunto de modelos externos al mismo y posteriormente, con los modelos actualmente existentes; estableciendo ventajas, desventajas y su posible impacto una vez incorporados al PDS a través de nuestra propuesta.
- En el capítulo 6 concluimos y proporcionamos algunas recomendaciones para el trabajo a futuro con el fin de mejorar la propuesta aquí presentada.

2. ANTECEDENTES

El objetivo principal de este capítulo es presentar, analizar y comparar las principales características de los diferentes trabajos que anteceden a nuestro estudio en el contexto de las Técnicas de Identificación de Alertas Accionables (AAIT, *Actionable Alert Identification Technique*) [6, 7, 11, 19, 23, 24, 9, 10, 25], poniendo un particular énfasis en las etapas que las componen y en el modo en que funcionan, ya que serán de gran trascendencia en el diseño de nuestra propuesta. En la siguiente sección, presentamos los principales conceptos que se abordan y las principales herramientas que se utilizan a lo largo de este estudio. En la sección 2.2.1, exponemos los trabajos relacionados en el área de las AAIT, distinguimos sus aportaciones y resultados más importantes, los proyectos de software analizados y sus características, las herramientas, los algoritmos y las medidas de desempeño empleadas. Por último, en la sección 2.4 reconocemos y discutimos posibles puntos de mejora a los estudios seleccionados con el fin de plasmarlos dentro de nuestra propuesta.

2.1. Nociones preliminares

Hoy en día, los sistemas de software cambian constantemente durante su desarrollo debido a que se descubren y se corrigen nuevos defectos, surgen nuevos requerimientos y se añaden nuevas funcionalidades, entre otros factores. Por ello, es necesario contar con un esquema de versionamiento capaz de reflejar en qué punto se encuentra o cuál es el grado de avance del sistema, siendo el versionamiento semántico (a nuestro entender) [26] uno de los más adecuados dentro de la administración de la configuración. Dicho esquema, proporciona un conjunto de reglas que rige la manera en la que se nombran cada una de las versiones de un sistema de acuerdo a un patrón llamado "X.Y.Z" (por ejemplo, 2.6.3), el cual distingue 3 tipos de versiones que son de gran trascendencia en el desarrollo de este estudio. Por esta razón, estos tipos de versiones son descritos a continuación:

1. **Versión *major* (X).** Describe un cambio muy grande en el sistema al borrar o añadir múltiples funcionalidades, los cambios realizados no son compatibles con versiones anteriores [26].
2. **Versión *minor* (Y).** Describe un cambio mediando en el sistema, añadiendo nuevas funcionalidades o modificando las ya existentes pero respetando en todo momento la compatibilidad con versiones anteriores [26].

3. **Revisión (Z).** Describe un cambio pequeño en el sistema, pues tan sólo incluye corrección de errores menores o refactorizaciones al código ya existente. Es compatible con revisiones anteriores y nunca es utilizado para añadir o modificar funcionalidades [26].

Por otro parte, ya hemos mencionado que el análisis estático de software nos permite identificar posibles defectos dentro del código fuente mediante la ejecución de una Herramienta de Análisis Estático Automatizado (ASAT, *Automated Static Analysis Tool*), la cual se encarga de realizar esta inspección de manera automatizada [4, 5, 6, 7, 8]. Una vez que la ASAT es ejecutada sobre el código fuente del proyecto en cuestión, la herramienta genera un conjunto de posibles defectos o alertas que pueden ser representadas por medio de lo que llamamos Características de Alerta (CA), las cuales pueden ser vistas como propiedades que rodean explícita o implícitamente a una alerta. Ejemplo de dichas características son el tipo y categoría de la alerta, el nombre y tamaño del artefacto alerta, el número de línea de la alerta y la complejidad ciclomática, por mencionar algunos [4, 6, 7].

Antes de continuar, debemos mencionar que la descripción y categorización de cada una de las CA utilizadas dentro de este estudio será abordada a lo largo de la sección 3.2.1.1. No obstante, por medio de la figura 2.1 podemos observar un fragmento de código fuente escrito en el lenguaje de programación Java [20] con el fin de describir una alerta de análisis estático cuya categoría es *Performance* [27] y su tipo es *SBSC_USE_STRINGBUFFER_CONCATENATION* [27], además de ejemplificar algunas de las CA que le rodean. Como se podrá ver, el número de línea de la alerta es 73, el nombre del archivo es ServiciosUsuario.java y el nombre del método es generaContrasenia() con un tamaño de 8 LLOC (*Logical Lines Of Code*).

ServiciosUsuario.java

```
62    public String generaContrasenia() {
63        String contrasenia = "";
64        String base[] = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
65                "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
66                "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x",
67                "y", "z", "A", "B", "C", "D", "E", "F", "G", "H", "I", "J",
68                "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V",
69                "W", "X", "Y", "Z", "@", "!", "#", "$", "|" };
70        int tamBase = base.length;
71        Random r = new Random();
72        for (int i = 0; i < 15; i++)
73            contrasenia += base[r.nextInt(tamBase)];
74        return contrasenia;
75    }
```

Figura 2.1 Código fuente que rodea a una alerta de análisis estático.

Dicho lo anterior, con base en la revisión de las características asociadas a cada alerta generada por la ASAT, un desarrollador habrá de determinar si dicha alerta es lo suficientemente

importante como para tratarla. Aparece entonces uno de los conceptos más importantes en nuestro trabajo, la accionabilidad de una alerta, la cual se define así:

- Si el desarrollador determina que la alerta en cuestión es de suficiente relevancia para la correcta funcionalidad del sistema, es tratable (es decir, que es posible corregirla) y debe ser corregida lo más pronto posible, decimos que la alerta es accionable o es un verdadero positivo [10, 9, 6, 7, 8, 4, 11, 19].
- Por el contrario, si el desarrollador decide que la alerta no es relevante o, de acuerdo a su percepción, es una alerta poco importante y de poca o nula trascendencia para la correcta funcionalidad del sistema, entonces decimos que la alerta es una alerta no accionable o es un falso positivo [10, 9, 6, 7, 8, 4, 11, 19].

Sin embargo, las ASAT (incluyendo a FindBugs [27]) generan hasta 40 alertas por cada mil líneas de código [6] (cuando el código fuente debe contener como máximo dos [11, 19]) de las cuales, el 64% en promedio son alertas no accionables [6, 4, 16, 8, 17], sin duda, este hecho podría desmotivar al desarrollador a utilizarlas. Por lo anterior, se vuelve necesario hacer uso de técnicas que permitan mitigar el número excesivo de alertas no accionables que reportan las ASAT, siendo el aprendizaje maquinal uno de los enfoques más utilizados en el área de las AAIT [4, 7, 11, 23, 25, 10]. Antes de continuar con la definición de aprendizaje maquinal y de algunos de los conceptos relacionados a esta área del conocimiento, mostramos una breve reseña de lo que es FindBugs.

2.1.1. FindBugs

Dado que uno de los objetivos de este estudio es generar la mayor cantidad posible de información acerca de los defectos introducidos en el código fuente con base en las alertas generadas por una ASAT, retomaremos el trabajo de Heckman y Williams [6, 7] para elegir aquellas herramientas que nos ayuden a obtener resultados más favorables, siendo FindBugs una de ellas.

Desarrollada en la Universidad de Maryland (Estados Unidos) por William Pugh, FindBugs es una herramienta de código abierto que implementa Análisis Estático Automatizado (ASA, *Automated Static Analysis*) sobre código compilado Java [28]. Para lograrlo, dicha herramienta utiliza diferentes estrategias para la detección de patrones de código que pudieran reflejar un defecto dentro del sistema, algunas de ellas son: exploración del Java Bytecode[4] sobre los métodos de las clases analizadas, control y análisis de flujo de datos [28].

[4] Java Bytecode es un archivo de código binario que contiene un programa ejecutable escrito en el lenguaje Java. En general, se trata de la transformación del código fuente al código máquina entendible por la máquina virtual de Java [20].

FindBugs es capaz de descubrir más de 426 tipos de alertas, los cuales son clasificados en 9 categorías (por ejemplo, malas prácticas de programación, exactitud del código, internacionalización, concurrencia, desempeño y seguridad) y priorizados de acuerdo a su nivel de gravedad (alta, media y baja) [28]. La categoría y la descripción de algunos tipos de alerta reportados por dicha herramienta son mostradas en la tabla A.1 al final de este documento (véase apéndice A).

2.2. Aprendizaje maquinal

Dado un conjunto de CA, el aprendizaje maquinal es uno de los enfoques más prometedores y utilizados para la predicción de alertas accionables mediante la aplicación de un modelo de aprendizaje [4, 7, 11, 23, 25, 10]. Surgen entonces dos conceptos fundamentales en el desarrollo de nuestro trabajo: (1) el aprendizaje maquinal, definido como el proceso de identificar posibles patrones (de manera semiautomatizada o automatizada) con el fin de extraer información potencialmente útil y previamente desconocida a partir de un conjunto de datos [7, 22] y (2) el modelo de aprendizaje, el cual es generado una vez que se aplica un algoritmo de aprendizaje sobre algún conjunto de datos para calcular dichos patrones [22]. Una breve descripción acerca de los algoritmos de aprendizaje utilizados para generar modelos durante nuestros experimentos puede ser consultada en la tabla A.4 al final de este documento (véase apéndice A).

Con esto en mente, es así como las AAIT hacen uso del aprendizaje maquinal para determinar la accionabilidad de nuevas alertas dado un conjunto de entrenamiento (el cual contiene alertas previamente etiquetadas como accionables o no accionables), intentando disminuir el número de falsos positivos que las ASAT generan [4, 8]. En general, estas técnicas pueden ser divididas en dos tipos:

1. **AAIT de clasificación.** Dividen las alertas generadas en dos grupos, alertas accionables y alertas no accionables [6, 5, 4, 8].
2. **AAIT de priorización.** Priorizan el conjunto de alertas con base en un valor de probabilidad que describe su grado de accionabilidad, situando en la parte superior a aquellas con mayor posibilidad de ser alertas accionables, este tipo de técnicas se convierten en AAIT de clasificación a medida que la probabilidad decrece [6, 5, 4, 8].

Es importante mencionar que, dichas técnicas pueden clasificar y/o priorizar alertas utilizando uno o varios enfoques: basado en el tipo o la información contextual de la alerta, teoría de grafos, matemático-estadístico, fusión de datos y aprendizaje maquinal [6, 4, 29]. Diferentes autores han utilizado uno o más enfoques para desarrollar cada una de sus técnicas; en particular, nuestra

propuesta está basada en la creación de una AAIT para la construcción y evaluación de modelos de clasificación de nuevas alertas utilizando algoritmos de aprendizaje maquinal.

Dicho lo anterior, podemos observar que el resultado de implementar una AAIT sobre un proyecto de software es un conjunto de modelos de clasificación de nuevas alertas, los cuales (luego de haber sido construidos) necesitan ser evaluados de alguna manera. Para ello, diferentes autores se apoyan de 4 posibles valores de clasificación que resultan de comparar los valores predichos por cada modelo y los valores reales para cada una de las alertas generadas [6, 7, 8, 4, 5], estos valores se resumen en la tabla 2.1 y son descritos a continuación:

- **Verdadero Positivo (VP):** describe una alerta como accionable cuando la alerta efectivamente es una alerta accionable.
- **Verdadero Negativo (VN):** describe una alerta como no accionable cuando la alerta efectivamente es una alerta no accionable.
- **Falso Positivo (FP):** describe una alerta como accionable cuando la alerta en realidad es una alerta no accionable.
- **Falso Negativo (FN):** describe una alerta como no accionable cuando la alerta en realidad es una alerta accionable.

Alertas predichas por el modelo			
Accionables	**No accionables**		
Verdadero Positivo (VP)	Falso Negativo (FN)	**Accionables**	**Alertas que son observadas**
Falso Positivo (FP)	Verdadero Negativo (VN)	**No accionables**	

Tabla 2.1 Valores de clasificación de alertas.

Una vez definidos estos valores de clasificación (véase tabla 2.1), es posible calcular una serie de medidas de desempeño con el fin de evaluar qué tan bien los distintos modelos de clasificación de alertas accionables realizan dicha predicción y determinar su grado de fiabilidad. Diferentes investigadores se han enfocado en 3 mediciones fundamentales [6, 7, 8, 4, 23, 24, 19, 11], la cuales son:

1. **Exactitud (Ex):** es la proporción de alertas accionables y no accionables clasificadas de forma correcta. En la ec. 2.1 presentamos la forma de calcular la exactitud, veamos:

$$Ex = \frac{VP + VN}{VP + VN + FP + FN} \qquad \text{(Ec. 2.1)}$$

2. **Precisión (Pr):** es la relación de alertas accionables clasificadas de forma correcta (VP) y el total de

las alertas predichas como accionables (VP + FP). En la ec. 2.2 presentamos la forma de calcular la precisión, veamos:

$$Pr = \frac{VP}{VP + FP} \quad \text{(Ec. 2.2)}$$

3. **Sensibilidad (Se):** también llamada tasa de VP, es la relación de alertas accionables clasificadas de forma correcta (VP) y el total de alertas realmente accionables (VP + FN). En la ec. 2.3 presentamos la forma de calcular la precisión, veamos:

$$Se = \frac{VP}{VP + FN} \quad \text{(Ec. 2.3)}$$

A continuación, presentamos una breve reseña acerca de otra de las herramientas (llamada WEKA [30]) que será utilizada de manera recurrente a lo largo de nuestro estudio y que implementa cada uno de los conceptos presentados anteriormente.

2.2.1. Waikato Environment for Knowledge Analysis (WEKA)

Dado que otro de nuestros objetivos es seleccionar, construir y evaluar modelos de clasificación de nuevas alertas mediante diferentes algoritmos de aprendizaje maquinal y dado que nuestro propósito fundamental no está enfocado en proponer, mejorar o adaptar algún(os) algoritmo(s) de aprendizaje para lograr dicho resultado, utilizaremos el Ambiente de Waikato para el Análisis del Conocimiento (WEKA, *Waikato Environment for Knowledge Analysis*) [30] para llevar a cabo parte de nuestros experimentos.

Desarrollada en la Universidad de Waikato (Nueva Zelanda) [31] por Witten y Frank, WEKA es una herramienta de software libre que contiene una colección de algoritmos de aprendizaje maquinal, los cuales permiten realizar diferentes tareas como lo son: selección de atributos, construcción y evaluación de modelos utilizando validación cruzada [32, 33, 22]. Es importante mencionar que, en la técnica de validación cruzada, el conjunto de datos es dividido en un cierto número de particiones (por ejemplo, WEKA divide la información en 10 particiones disjuntas) y cada una de ellas es utilizada para evaluar el desempeño del modelo (el cual ha sido entrenado con las particiones restantes); el resultado final habrá de ser el promedio del desempeño obtenido en cada evaluación [22]. De manera general, la validación cruzada puede ser entendida como una metodología *training and testing*, en la cual una porción de los datos es utilizada para entrenar al algoritmo de aprendizaje (*training*) y la parte restante es utilizada para evaluar su desempeño en la clasificación de nuevas instancias de datos (*testing*) [6, 22]

Para seleccionar las características más relevantes en la accionabilidad de una alerta dado un conjunto de datos, esta herramienta proporciona diversos algoritmos de selección de atributos, los

cuales son ejecutados mediante la combinación de un método de evaluación y una estrategia de búsqueda. Por otro lado, WEKA implementa múltiples algoritmos de aprendizaje (agrupados en 9 categorías) para la construcción de modelos y para su evaluación en términos de exactitud, precisión y sensibilidad. La descripción de cada uno de los métodos, estrategias y algoritmos de aprendizaje utilizados a lo largo de este estudio es mostrada en la tabla A.2, tabla A.3 y tabla A.4 respectivamente, la cual puede ser encontrada al final de este documento (véase apéndice A).

2.3. Trabajos relacionados

Heckman y Williams [6, 7] presentan un proceso para la construcción de modelos de identificación de alertas accionables llamado SAAI (*Systematic Actionable Alert Identification*), el cual utiliza un enfoque de aprendizaje maquinal para identificar las CA más importantes y con ellas, generar un conjunto de modelos encargados de clasificar las alertas generadas para dos proyectos de software. Dicho proceso se compone de 4 etapas, las cuales son descritas a continuación:

1) **Recopilación de datos.** La información asociada a cada una de las alertas generadas por FindBugs a lo largo de las distintas revisiones de cada proyecto (JDOM [34] y Runtime [35]) es representada como un conjunto de vectores de valores compuestos por un total de 51 CA candidatas. Cabe mencionar que, cada alerta es etiquetada con base en su estado actual dentro del historial de revisiones: no accionable si la alerta ha perdurado a lo largo de las revisiones y accionable si esta ha desaparecido. Lo anterior, permite crear un conjunto completo de predictores.
2) **Selección de CA.** Basados en dicho conjunto de vectores, se crean diferentes subconjuntos de las CA más importantes, no redundantes y no relacionadas entre sí, esto es logrado mediante diferentes algoritmos de evaluación de atributos. Estos subconjuntos se componen de entre 3 y 14 características.
3) **Creación de modelos.** Una vez que se cuenta con los subconjuntos de CA independientes entre sí, 15 algoritmos de aprendizaje maquinal son utilizados para construir una serie de modelos de clasificación de alertas aplicando validación cruzada.
4) **Selección de modelos.** Los modelos generados son evaluados de acuerdo a 3 medidas de desempeño: precisión, exactitud y sensibilidad. Esto permite distinguir aquellos modelos con un mayor poder de predicción.

   A lo largo de su investigación, Heckman y Williams observaron que el tiempo de vida es una de

las características más importantes en la accionabilidad de una alerta y, en general, el 50% del total de características son comunes a ambos proyectos. Ellas han utilizado WEKA para desarrollar las etapas 2, 3 y 4, y así, seleccionar al mejor de los modelos mediante los algoritmos K* [22, 32] para JDOM y k-NN [22, 32] para Runtime, con una exactitud de 98.7% en ambos proyectos.

Yüksel y Sözer [11] unen sus esfuerzos con el fin de etiquetar más de 3000 alertas de análisis estático (como accionables o no accionables), las cuales provienen de un sistema de televisión digital de aproximadamente 1500 KLOC (*Kilo Lines Of Code*) utilizando el enfoque de aprendizaje maquinal. Ellos proponen una técnica de clasificación de 3 etapas:

1) **Selección de CA.** Una ASAT es ejecutada semanalmente sobre el proyecto, lo cual genera un conjunto de alertas. Basados en esta información, estos autores generan un vector de valores por cada alerta, el cual se compone de 10 características; una de ellas proviene de la retroalimentación por parte del desarrollador al estado de cada alerta (la alerta es etiquetada con 4 posibles estados: reparar, no es un problema, ignorar y analizar) y las 9 características restantes de la ASAT. Cabe mencionar que, la accionabilidad de cada alerta es determinada con base en la experiencia de los autores del estudio, la figura 2.2 (tomada de [19]) muestra el proceso antes descrito. Una vez realizado esto, se evalúa la relevancia de cada característica mediante 10 algoritmos de selección de atributos de WEKA.

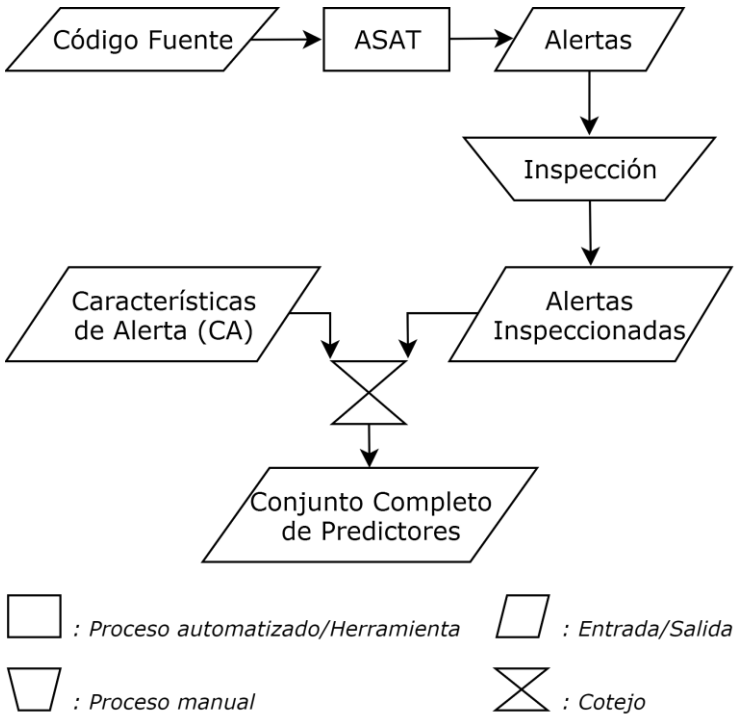


Figura 2.2 Proceso para la creación de un conjunto completo de predictores.

2) **Clasificación con validación cruzada.** Teniendo en cuenta lo anterior, se utiliza validación cruzada para evaluar la exactitud de 34 algoritmos de aprendizaje maquinal sobre el conjunto total de alertas, la mejor exactitud (86.4%) ocurre con el algoritmo Comité Aleatorio [22, 32].
3) **Clasificación durante el Proceso de Desarrollo de Software (PDS).** El algoritmo de aprendizaje es entrenado con las alertas generadas por la ASAT durante 5 ejecuciones continuas y es probado con las restantes, esto con el fin reflejar un punto en el tiempo del PDS y así, simular situaciones más reales dentro del desarrollo. La mejor precisión (89.1%) ocurre con el algoritmo Híbrido DT-NB [32, 22].

Una contribución importante de este estudio es la de utilizar la idea del desarrollador como una característica relevante para el etiquetado de alertas, hecho que posteriormente es demostrado mediante la selección de atributos. El tiempo de vida, el tipo y la prioridad de la alerta también fueron características relevantes.

Más tarde, Yüksel *et al.* propondrían un enfoque basado en la fusión o combinación de clasificadores [19], utilizando el mismo conjunto de datos generado en su estudio anterior y buscando mejorar sus propios resultados. En dicho estudio, se examina la exactitud de 3 algoritmos de aprendizaje que combinan múltiples clasificadores; sin embargo, los resultados obtenidos no aumentan la exactitud de la clasificación en más de un 4%.

Moriggl [23] presenta una AAIT cuyos modelos, generados mediante diversos algoritmos de aprendizaje maquinal, son capaces de clasificar nuevos defectos a partir de un conjunto de vectores de características extraído del código fuente de varios proyectos Java. Para lograrlo, esta técnica lleva a cabo las siguientes etapas:

1) **Extracción y análisis de CA.** El prototipo JJParse es un programa Java que analiza estáticamente el código fuente y extrae más de 150 características por operación. Provenientes de varios proyectos de código abierto, estas CA describen la siguiente información: localización y tipo de la operación, información de ramificación, de la variable principal, de la auxiliar, del método que la contiene y de sus parámetros. Una vez extraídas dichas características, estas son representadas mediante un vector de valores y describen una instancia para un posible fallo, los principales defectos a considerar son aquellos que hacen que un programa en ejecución se bloquee: referencia a punteros nulos, errores de conversión de tipos, desbordamiento de memoria, uso inadecuado de operadores, entre otros.

2) **Selección de CA.** El conjunto total de características es evaluado con el fin de obtener un conjunto reducido de las 127 CA más relevantes. Las características de localización, información de las variables y de los parámetros, fueron algunas de las CA descartadas por no ser de trascendencia en la clasificación de alertas.
3) **Ejecución de aprendices.** Con base en las más de 650 instancias generadas por el prototipo, diferentes algoritmos de aprendizaje (implementados por WEKA) son entrenados utilizando validación cruzada para crear un conjunto de modelos candidatos.
4) **Evaluación de modelos.** El desempeño de cada modelo construido es evaluado sobre 38 instancias de código defectuoso (resultado de introducir de manera intencional errores en el código fuente de los proyectos estudiados) con el fin de conocer a los mejores modelos. Los algoritmos Perceptrón Multicapa, Árbol C4.5 y Árbol LM [22, 32] lograron exactitudes de 97.19%, 96.25% y 96.09% respectivamente.

Dos años más tarde, Moriggl *et al.* [24] enriquecerían la AAIT antes desarrollada, incorporando un nuevo prototipo llamado CMore para la extracción de CA en proyectos escritos en lenguaje C. La metodología seguida, las características recopiladas y el tipo de defectos buscados son relativamente similares a su estudio anterior. Nuevamente, el algoritmo Perceptrón Multicapa [22, 32] obtuvo la mejor exactitud de 96.98%, lo cual muestra que las AAIT son adaptables a cualquier lenguaje de programación y en consecuencia, la metodología puede mantenerse idéntica aun cuando la AAIT presente ligeras modificaciones.

Hanam *et al.* [9, 10] proponen una técnica para diferenciar las alertas accionables de las no accionables mediante el hallazgo de alertas con patrones de código fuente similares. Aunque se trata de una AAIT de priorización, este estudio plantea la idea de priorizar la mayor cantidad de alertas accionables dentro del 5% superior para luego utilizarlas como un conjunto de entrenamiento y, con ello, obtener un modelo para la clasificación de nuevas alertas. Dicho enfoque, puede ser resumido como sigue:

1) **Generación de CA.** Un programa se encarga de crear un vector de 19 características asociadas a cada una de las 2249 alertas generadas por FindBugs y al código fuente que les rodea. Las CA que rodean a cada alerta (patrones de alerta) son extraídas de las sentencias de código fuente potencialmente más relevantes y más cercanas a la ubicación de la alerta; para lograrlo, una herramienta de software (llamada WALA [36]) distingue aquellas características que afectan a cada alerta mediante la construcción de una árbol de sintaxis abstracta. Los autores recomiendan

limitar el número de sentencias a explorar (5 es el número recomendado), esto permite optimizar tiempos, disminuir el tamaño final del vector creado y aumentar el desempeño del clasificador.

2) **Análisis de CA.** Con base en un conocimiento previo, se define la accionabilidad de cada una de las alertas de forma manual, lo que permite tener un conjunto completo de entrenamiento. Dichos autores sostienen que, en todo proyecto existen ciertos patrones de código que se encuentran asociados a la accionabilidad de una alerta y que no son detectados por las ASAT.
3) **Priorización de alertas.** Una vez que se tiene un conjunto de alertas etiquetadas como accionables y no accionables, se genera un modelo para priorizarlas, de tal forma que todas las alertas accionables queden dentro del 5% superior y en el 95% restante las no accionables.
4) **Creación y evaluación de modelos.** Posteriormente, esta AAIT toma las primeras 225 alertas (es decir, el 10% superior) de la lista de alertas priorizadas en el paso anterior y las utiliza para construir diferentes modelos de clasificación basados en los algoritmos Árbol AD, Bayes Ingenuo y Red de Bayes [22, 32], los cuales son implementados por la herramienta WEKA.

A pesar de que este estudio no tiene en cuenta la exactitud como medida de desempeño y su funcionamiento principal se basa en una AAIT de priorización (lo que dificulta compararlo con otros trabajos), nos ayuda a observar que las alertas generadas por las ASAT podrían ser clasificadas de acuerdo al patrón de alerta buscado, lo que daría una alta configurabilidad a la AAIT, inclusive sería posible comenzar con un conjunto de patrones pequeño e irlo expandiendo a través del tiempo, logrando mejorar la clasificación de alertas.

Liang *et al.* [25] proponen una AAIT que basa su funcionamiento en la idea de identificar una o más Líneas Relacionadas a un Error Genérico[5] (LREG) dentro del código fuente con el objetivo de construir de manera automática un conjunto de entrenamiento apoyado en el previo etiquetado de alertas y en su posterior priorización, situando aquellas que sean accionables al inicio de una lista de alertas priorizadas. Esta técnica es aplicada a 5 proyectos de código abierto y, como se puede ver en la figura 2.3, es construida siguiendo las siguientes etapas:

1) **Identificación de Revisiones de Reparación de un Error Genérico (RREG).** Una RREG es aquella revisión que ha sido dada de alta en el repositorio de código fuente como consecuencia de la reparación de un error genérico (por ejemplo, abrazos mortales, punteros nulos o pérdida de recursos). Con esto en mente, partiendo del historial de revisiones y de la base de datos de seguimiento de errores de cada proyecto, un algoritmo identifica un total de 358 RREG con base

[5] Los distintos tipos de errores genéricos son equivalentes a los tipos de errores que una ASAT puede encontrar.

en los mensajes de error reportados y en el número de archivos modificados durante la revisión: a menor número de archivos, mayor probabilidad de ser una RREG.

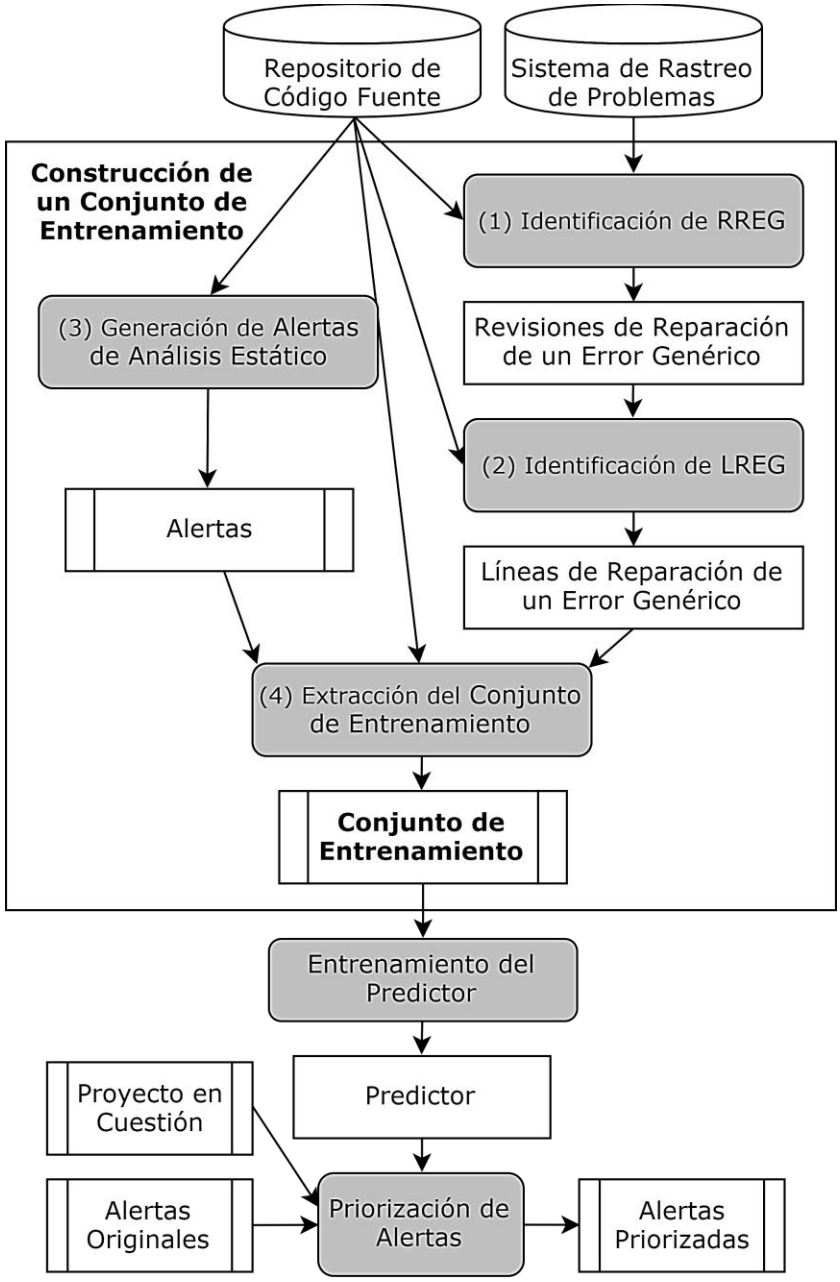


Figura 2.3 Proceso para la construcción de un conjunto de entrenamiento.

2) **Identificación de LREG**[6]**.** Una LREG se define como aquella línea que ha sido modificada dentro de una RREG. Con esto en mente, un algoritmo toma la información asociada a la RREG e identifica la(s) LREG, recordemos que una LREG son aquellas líneas que han sido modificadas durante una RREG y son parte fundamental en la definición de la accionabilidad de un alerta.

[6] Una buena ASAT es capaz de generar alertas que indican la(s) línea(s) a corregir, esta información es utilizada para definir a la LREG.

3) **Generación de alertas de análisis estático.** Un conjunto de alertas es generado mediante la ejecución de 4 diferentes ASAT sobre todas las revisiones del proyecto bajo análisis.
4) **Extracción del conjunto de entrenamiento.** Para cada proyecto bajo análisis, se construye un conjunto de vectores compuestos por 22 atributos de entrada o factores de impacto y un atributo de salida (la accionabilidad de la alerta). Dichos factores de impacto provienen de 3 categorías: (1) descriptores de la alerta, (2) estadísticas de la alerta y (3) características del código fuente. Por otro lado, el atributo de salida es etiquetado con base en la información recopilada sobre las LREG y de acuerdo a la siguiente heurística: una alerta que es detectada en una RREG y que desaparece en revisiones posteriores y que además contiene una o más LREG, es etiquetada como accionable; si lo anterior no sucede, la alerta es etiquetada como no accionable. Antes de llegar a la siguiente etapa, 7 algoritmos de selección de atributos de WEKA son ejecutados para conocer aquellos factores de impacto o CA menos relevantes.
5) **Entrenamiento del predictor.** Los algoritmos Red de Bayes, Regresión Logística Multinomial, k-NN, Empaquetado, Bosques Aleatorios y Tablas de Decisión [22, 32] (implementados por WEKA) son entrenados para generar diferentes modelos de clasificación con el fin de seleccionar al más adecuado en cada proyecto analizado. Adicionalmente, cada modelo graba una probabilidad de accionabilidad para cada una de las alertas.
6) **Priorización de alertas.** El mejor modelo clasificador es utilizado para generar una lista priorizada de alertas de acuerdo a su probabilidad de accionabilidad, situando a las más propensas a ser accionables en la parte superior del listado.

Liang *et al.* [25] comparten la implementación de su Servicio de Análisis de Defectos de Código (CODAS, *Code Defect Analysis Service*), el cual integra múltiples herramientas de análisis estático (como lo es FindBugs) y prioriza las alertas generadas de acuerdo a su probabilidad de accionabilidad.

2.4. Resumen y análisis

Para apoyarnos en el reconocimiento de algunas áreas de mejora y proporcionar un análisis sobre los puntos de interés acerca de los trabajos relacionados descritos en la sección anterior, mostramos una tabla comparativa (véase tabla 2.2) que resume sus propiedades y características más importantes: los proyectos estudiados, las herramientas de software, técnicas y algoritmos utilizados, además de los resultados obtenidos. Dicho lo anterior, procedemos a realizar un análisis de la literatura seleccionada.

Autor(es)	Proyecto	Tamaño (KLOC)	No. de revisiones	ASAT	No. de alertas	No. de CA candidatas	Método de etiquetado
Heckman y Williams [5, 6]	JDOM	13.2+	29	FindBugs y JavaNCSS	420	51	Comparación entre revisiones
	Runtime	15.5+	41		853		
Yüksel y Sözer [20]	Digital TV	1500+	19	No mencionada	1147	10	Clasificación manual
			5 y 14		1068 y 79		
Yüksel *et al.* [21]	Digital TV	1500+	19	No mencionada	1147	10	
Moriggl [23]	Varios	-	-	JJParse	650+ y 38	150	
Tribus *et al.* [24]	Varios	-	-	Cmore	4000+ y 44	-	
Hanam *et al.* [8, 9]	Tomcat6	122+	11	Findbugs y WALA	1971	19	
	Commons	43+	14		329		
	Logging	19+	11		237		
Liang *et al.* [35]	Varios	-	-	CODAS	-	22	

Tabla 2.2 Trabajos relacionados a las AAIT.

Autor(es)	Técnica	Mejor clasificador	Exactitud	Precisión	Sensib.	Observaciones
Heckman y Williams [5, 6]	Validación cruzada	K*	98.70%	-	-	Heurística para determinar la accionabilidad de una alerta. 50% de las CA son comunes a ambos proyectos. Tiempo de vida como una CA relevante.
		k-NN	98.70%	-	-	
Yüksel y Sözer [20]	Validación cruzada	Comité Aleatorio	86.40%	86.40%	86.40%	Idea del desarrollador, tiempo de vida, tipo y prioridad de la alerta como CA relevantes.
	Train & Test	Híbrido DT-NB	89.10%	90.70%	89.10%	
Yüksel *et al.* [21]	Validación cruzada	Empaquetado	86.70%	87.00%	87.00%	
Moriggl [23]	Train & Test	Perceptrón Multicapa	97.19%	-	-	Una ASAT propia que brinda la idea de buscar defectos particulares.
Tribus *et al.* [24]	Train & Test	Perceptrón Multicapa	96.98%	-	-	
Hanam *et al.* [8, 9]	Validación cruzada	Árbol AD	-	100.00%	31.00%	Los patrones de código están asociados con los defectos dentro del sistema.
		Bayes Ingenuo	-	45.00%	84.00%	
		Bayes Ingenuo	-	10.00%	62.00%	
Liang *et al.* [35]	Validación cruzada	k-NN	-	98.70%	98.70%	Las LREG son información fundamental en la clasificación de una alerta.

Tabla 2.2 Trabajos relacionados a las AAIT.

A pesar de la existencia de AAIT con resultados verdaderamente buenos (como las presentadas en la sección anterior), hemos observado que en todas ellas se construyen modelos internos, los cuales se basan en el entrenamiento de clasificadores con alertas propias del proyecto estudiado y que se prueban sobre posibles nuevos defectos del mismo sistema; hasta ahora, este ha sido el enfoque utilizado para la identificación de alertas accionables. Surgen entonces una serie de cuestiones alrededor de esta idea, las cuales son:

- Supongamos que un desarrollador está trabajando en un nuevo sistema y que naturalmente querrá corregir el mayor número posible de defectos relevantes introducidos en el código fuente con el fin de evitar que su sistema presente fallas en un futuro. Asumimos también que, él no cuenta con un historial de revisiones y que no cuenta con la experiencia necesaria para decidir si una alerta es accionable o no. Ciertamente, el no podrá hacer uso de una AAIT y la única alternativa que tiene será ejecutar una ASAT de manera directa sobre el código fuente y hacer lo que a su consideración sea pertinente. Dicho lo anterior, ¿el desarrollador debe reparar cada una de las alertas generadas por la ASAT? ¿debe esperar a contar con un historial de revisiones o contar la experiencia necesaria para lograr distinguir entre alertas accionables y no accionables?
- Pensamos que la respuesta a las preguntas planteadas anteriormente es un no y queremos explicar por qué. Si bien el trabajo de Heckman y Williams afirma que las CA más importantes son específicas para cada proyecto [6, 7], en dicho estudio también se hace la observación de que el 50% de las CA son comunes a los proyectos explorados, dada esta observación surge la siguiente inquietud: si existen CA que son comunes entre dos proyectos ajenos entre sí, entonces debería ser posible crear un modelo de clasificación basado en la información de uno para utilizarlo en otro.
- Yüksel y Sözer, al igual que Heckman y Williams, también observan que el tiempo de vida de una alerta es una CA muy importante en la accionabilidad de una alerta, sin embargo, ellos plantean otra CA de igual importancia y que no necesita de un repositorio de código fuente: la idea del desarrollador, que en realidad es un preetiquetado de la alerta. Otros estudios analizan el código que rodea a la alerta para construir algo similar a esta "idea del desarrollador" [23, 24], en particular, Hanam *et al.* sostienen que existen ciertos patrones de código que inevitablemente se convierten en un defecto [9, 10]. Una vez observado esto, pensamos que sería posible utilizar este tipo de características para determinar si una alerta es accionable o no, logrando dar una alta configurabilidad a la AAIT aun cuando la metodología sea modificada [23, 24].

Por último, deseamos mencionar que nuestra propuesta está inspirada en el trabajo de Heckman y Williams [6, 7] debido a que consideramos que es uno de los estudios más completos, algunas de nuestras razones son: (1) es una AAIT aplicada a sistemas de software actualmente en uso, lo suficientemente grandes como para incorporar control de versiones mediante un repositorio de código fuente, (2) en ella se genera un conjunto de alertas con características variadas y extraídas de diversas fuentes, lo que permite tener información lo suficientemente descriptiva acerca de un posible defecto en el sistema, (3) incorpora una heurística bastante objetiva para el etiquetado de alertas y (4) se trata de la AAIT con el mejor desempeño logrado. Por lo anterior, nuestra AAIT retoma gran parte de dichas propiedades, adaptando algunas de ellas para cumplir con los objetivos planteados en este estudio.

3. DISEÑO DE LA PROPUESTA

Ya hemos discutido acerca de los avances logrados hasta el momento en el área de las Técnicas de Identificación de Alertas Accionables (AAIT, *Actionable Alert Identification Technique*) y de las principales aportaciones de los investigadores en el tema, diferentes formas de diseñar, implementar y evaluar cada uno de sus enfoques, proyectos estudiados, características analizadas, herramientas y algoritmos utilizados; todas ellas dedicadas a generar el mejor modelo para la identificación de alertas accionables utilizando el enfoque de aprendizaje maquinal [7, 6, 19, 11, 29, 24, 25, 10, 9, 8]. Inspirados en esto, el objetivo principal de nuestra propuesta es optimizar el Proceso de Desarrollo de Software (PDS) aumentando el número de defectos relevantes descubiertos antes de llegar a la fase de pruebas mediante una AAIT.

La figura 3.1 muestra la forma en que nuestra técnica puede ser insertada dentro del PDS, justo después de concluir la codificación y antes de llegar a la fase pruebas, lo que permitiría obtener los mayores beneficios del Análisis Estático Automatizado (ASA, *Automated Static Analysis*). No debemos olvidar que, nuestra propuesta centra su atención en dar un tratamiento al artefacto generado por la fase de codificación (el código fuente), por lo que omitimos mencionar la existencia de otras fases en el PDS, como lo son el análisis de requerimientos, el diseño de la solución o la implantación del sistema [37, 2, 38, 39].

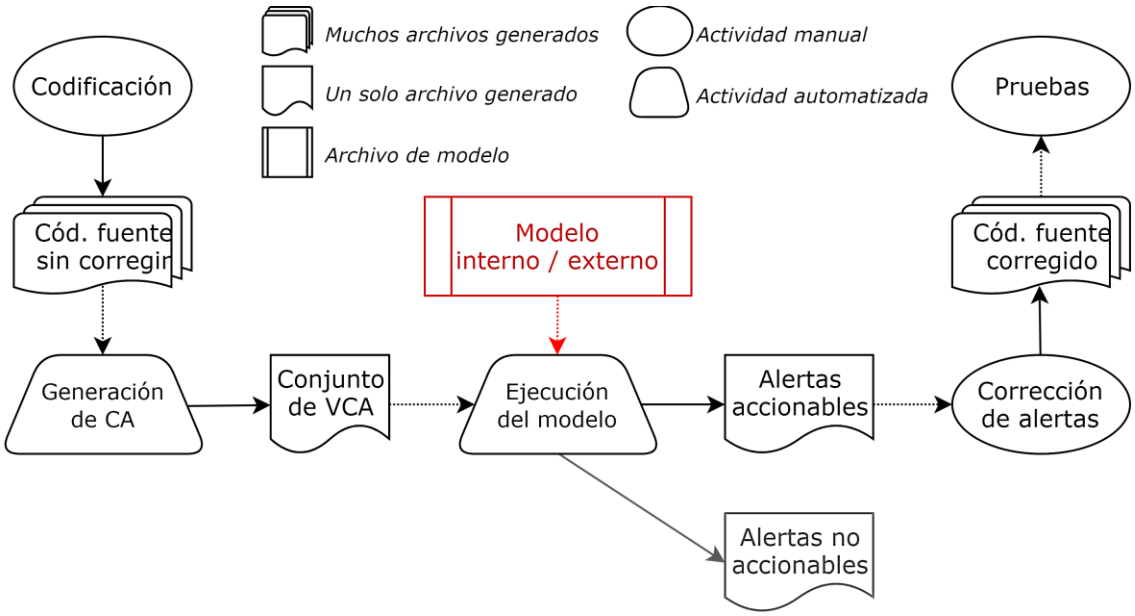


Figura 3.1 Visión de conjunto de la propuesta.

Como se puede ver en la figura 3.1, como resultado de la fase de codificación se tiene un conjunto de archivos que contienen el código fuente del sistema en desarrollo (sin corregir), a los cuales se les aplica un procesamiento llamado Generación de Características de Alerta (CA) con el fin de obtener un conjunto de vectores de características. Recordemos que, dichos vectores describen a cada una de las alertas reportadas por la ASAT para el proyecto en cuestión, que no incorporan características de temporalidad por tratarse de un sistema de reciente creación (es decir, no cuenta con un historial de versiones) y, que hasta ese momento, se desconoce su accionabilidad.

Gracias a un modelo externo (es decir, un modelo construido con base en las alertas de un proyecto ajeno al estudiado) creado mediante la ejecución de nuestra AAIT (véase figura 3.2), es posible clasificar cada alerta como accionable o no accionable aplicando dicho modelo al conjunto de Vectores de Características de Alerta (VCA) generado, lo que permitiría dirigir los esfuerzos de los desarrolladores a corregir los defectos o alertas más relevantes y de mayor impacto a la correcta funcionalidad del sistema para contar con un nuevo conjunto de archivos que almacenen un código fuente corregido. Creemos que este enfoque permitiría aumentar el número de defectos relevantes descubiertos y corregirlos antes de llegar a la fase de pruebas, observando una notoria diferencia entre un PDS que no incorpore nuestra técnica y uno que sí lo haga. Para verificar esta idea, en el capítulo 5 haremos un análisis sobre la fiabilidad de utilizar nuestra AAIT a través de una proyección de su impacto en el número de defectos presentes y en su tiempo total de corrección en cada fase del PDS.

A lo largo de este capítulo, detallaremos las diferentes etapas de las que se compone nuestra AAIT, la cual se inspira y retoma las principales características de los trabajos revisados en la sección 2.2.1, además de incorporar algunas modificaciones. Como se puede ver, la figura 3.2 muestra una visión general acerca de nuestra técnica, la cual se compone de diversas actividades que forman parte de las siguientes etapas:

1. **Generación de versiones.** Dado un proyecto de software, el cual cuenta con un repositorio de código fuente, se define un listado con las versiones a explorar (historial de versiones) con el fin de construirlas y obtener el código fuente a analizar, esto implica una compilación completa del proyecto en cada una de las versiones elegidas.
2. **Generación de características.** Una Herramienta de Análisis Estático Automatizado (ASAT, *Automated Static Analysis Tool*) y una Herramienta de Métricas de Software (SMT, *Software Metrics Tool*) son ejecutadas sobre cada una de las versiones construidas del proyecto (ejecución de herramientas), lo cual genera múltiples reportes de alertas que serán procesados con el fin de

obtener un conjunto de VCA previamente etiquetados como accionables o no accionables que describan a las alertas del proyecto estudiado.

3. **Generación de modelos.** Con ayuda de la herramienta WEKA [30] y el conjunto de VCA generado, se generan y se seleccionan los subconjuntos de características que mayor desempeño le brindan al clasificador (selección de CA) con el fin de crear y evaluar diferentes modelos para la clasificación de nuevas alertas [32]. El desempeño de los modelos generados es medido por su exactitud, precisión y sensibilidad.

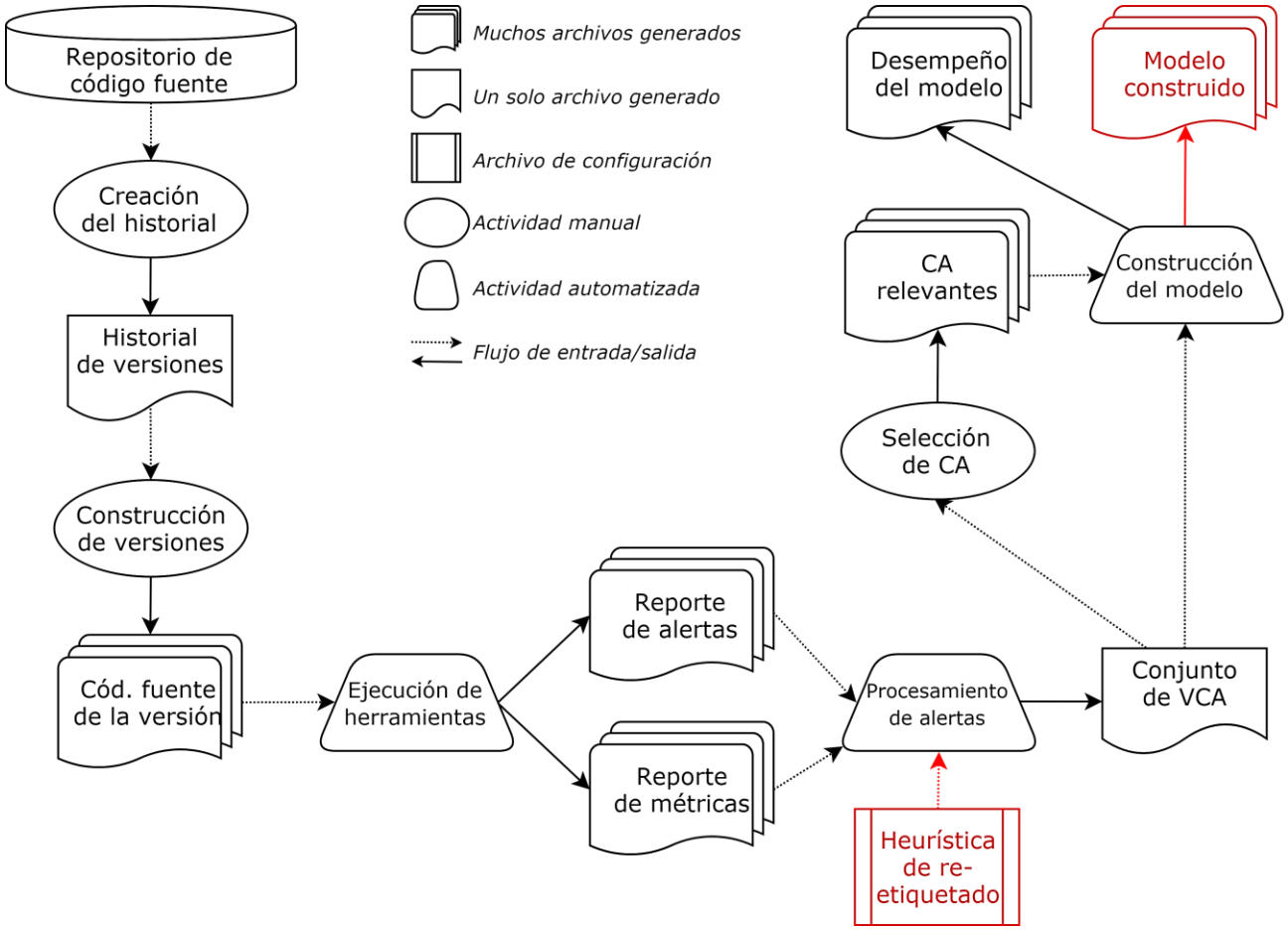


Figura 3.2 Visión de conjunto de la AAIT.

3.1. Generación de versiones

Dado que los sistemas de software están en constante cambio durante el proceso de desarrollo debido a que los requerimientos, el hardware y el software utilizados cambian, e inclusive porque se descubren nuevos defectos y se corrigen, se vuelve necesario implementar dichos cambios en nuevas

versiones del sistema[7] [39]. En el contexto de la ingeniería de software, una de las tareas fundamentales de la administración de la configuración es la de gestionar los cambios incorporados por el equipo de desarrollo en cada versión del sistema mediante una herramienta que facilite el manejo de esta información [39, 38].

Los repositorios de código fuente son herramientas que permiten mantener un registro de los cambios realizados en el código, guardando información sobre quién cambió qué, cuándo, cómo y por qué, además de guardar las múltiples versiones del sistema. En los últimos años, la mayoría de los equipos de desarrollo utilizan este tipo de herramientas y, en el caso de los proyectos de código abierto, hacen pública dicha información a través de internet [25, 4, 8, 24]. Queremos aprovechar este hecho para seleccionar y construir distintas versiones de dos proyectos actualmente en uso y con características similares entre sí, lo anterior, con el objetivo de recopilar información acerca de los defectos introducidos en su código fuente mediante las alertas generadas por una ASAT.

Como se puede ver en la figura 3.3, el proceso de generación de versiones se compone de dos actividades principales, las cuales son descritas a continuación:

1. **Creación del historial de versiones.** Estamos interesados en explorar sistemas de software que sean lo suficientemente grandes como para utilizar una herramienta de gestión de versiones como lo es SVN (*Subversion*) [40] o CVS (*Concurrent Versions System* o Sistema Concurrente de Versiones) [41], porque dentro de sus repositorios (independientemente de la herramienta utilizada) es posible acceder a la información relacionada a cada una de sus versiones, en particular, al código fuente del sistema. Por medio de esta actividad, nos daremos una idea acerca del tamaño del sistema al observar el seguimiento que se la ha dado a cada uno de sus componentes y, de forma paralela, podremos definir un listado de las versiones a explorar, llamaremos a este listado el historial de versiones (véase figura 3.3).
2. **Construcción de versiones.** Basados en el historial generado, comprobaremos y reuniremos todos los componentes necesarios para construir cada una de las versiones del proyecto en cuestión. Estamos interesados en lograr una compilación completa de cada una de ellas porque aquellos proyectos que no pueden construirse en su totalidad proporcionan información de análisis estático inconsistente [6, 7], en caso de no lograr construir alguna versión del historial, simplemente la omitiremos y continuaremos con la siguiente. En la figura 3.3 podemos observar

[7] Esta idea aplica para versiones tipo *major* y *minor* además de revisiones, las cuales también son consideradas como versiones (véase sección 2.1).

el resultado que se persigue al término de esta actividad, el cual es contar con un conjunto de archivos de código fuente compilado que integren a cada versión construida del proyecto.

Como veremos en la siguiente sección, cada una de estas versiones será de gran importancia para la heurística de etiquetado de alertas.

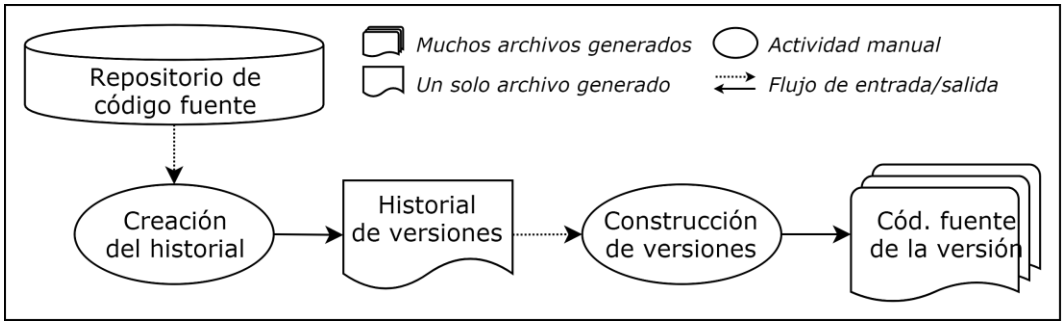


Figura 3.3 Proceso de generación de versiones.

3.2. Generación de características

La creación de un conjunto de VCA (también llamado conjunto de entrenamiento) etiquetado de manera apropiada juega un papel muy importante en el entrenamiento de un modelo para la clasificación de nuevas alertas. Por ello, es necesario (1) seleccionar el mayor número de características posible con el fin de describir ampliamente el comportamiento de cada alerta y (2) utilizar una o varias heurísticas para etiquetar las alertas generadas por la ASAT de acuerdo a su accionabilidad dentro el sistema en cuestión.

Dicho conjunto de VCA puede ser construido de forma manual o automática. Construirlo de forma manual puede ser muy eficaz, pero poco eficiente (es decir, podríamos obtener un conjunto que refleje el resultado esperado pero con un consumo excesivo de tiempo y esfuerzo); por el contrario, construirlo de forma automática nos brindaría alta eficiencia, pero seguramente poca eficacia [25]. Dado que la información acerca de los defectos introducidos durante la fase de codificación crece de manera acelerada, optaremos por automatizar una parte esencial de esta etapa que es el procesamiento de alertas. Como se puede ver en la figura 3.4, el resultado final que se persigue una vez concluida la etapa de generación de características es precisamente este conjunto de VCA.

A lo largo de esta sección, describiremos las actividades que deberán ser realizadas para obtener un conjunto de VCA para cada proyecto en cuestión (véase figura 3.4). Comenzaremos con la ejecución de dos herramientas (una ASAT y una SMT) sobre el conjunto de archivos de código fuente compilado que componen a cada una de las versiones, pero antes, definiremos las CA que, a nuestro saber y de acuerdo con trabajos anteriores [9, 7, 4, 25, 23, 24, 11, 19, 10, 6], describen de manera apropiada a cada

una de las alertas generadas por la ASAT. Posteriormente, procesaremos cada uno de los reportes generados por dichas herramientas para determinar la accionabilidad de cada una de las alertas y así, llegar al resultado esperado: un conjunto de VCA que represente a nuestro conjunto de alertas etiquetadas como accionables y no accionables.

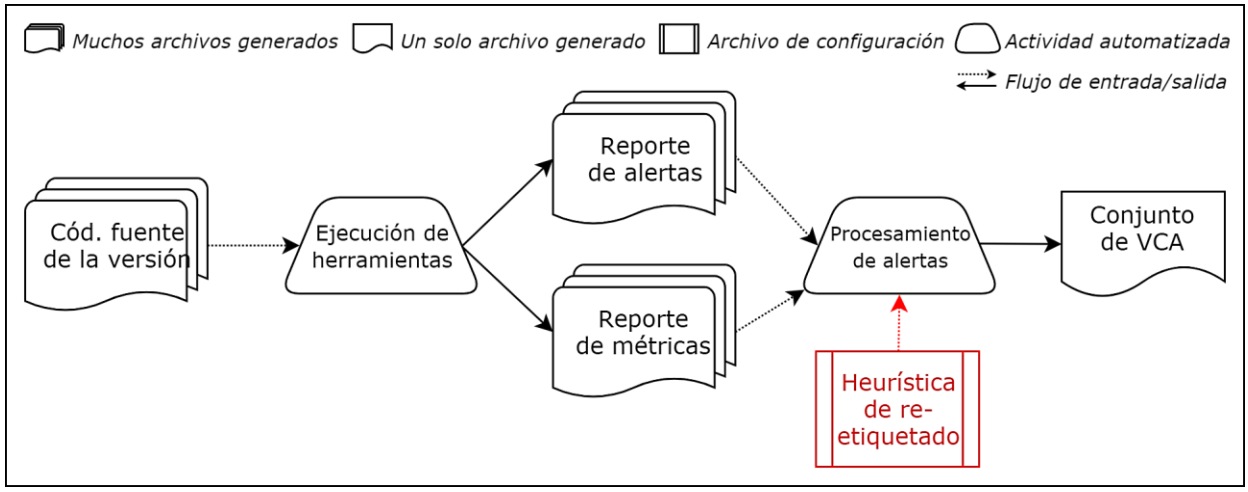


Figura 3.4 Proceso de generación de características.

3.2.1. Ejecución de Herramientas de Análisis Estático Automatizado

Una alerta reportada por una ASAT puede tener diferentes posibles arreglos, sin embargo, una buena ASAT debe ser capaz de indicar al menos uno de ellos e idealmente indicar las líneas que el desarrollador debe corregir [25]. Estamos interesados en utilizar una ASAT que pueda conseguir lo antes dicho y una de ellas es FindBugs [27], la cual es una herramienta lo suficientemente útil como para ser incorporada en una AAIT y mediante esta, obtener excelentes resultados (como los logrados en [6, 7]). Adicionalmente, dado que las diferentes ASAT son complementarias entre sí en cierto grado [25], al igual que Heckman y Williams [6, 7], complementaremos la ejecución de FindBugs con una herramienta de métricas orientadas a objetos llamada Metrics [42] con el fin de enriquecer nuestro conjunto de CA, veremos más adelante que también de la información almacenada en el repositorio de código fuente es posible extraer una serie de CA.

Como se puede ver en la figura 3.4, estas dos herramientas son ejecutadas sobre cada una de las versiones del proyecto en cuestión (recordemos que cada versión se compone de un conjunto de archivos que almacenan código fuente compilado), de lo cual se obtiene un conjunto reportes de alertas generados por la ASAT y un conjunto de reportes de métricas de software generados por la SMT. Estos reportes, contienen de manera implícita una serie de características que habrán de ser extraídas mediante el procesamiento de dicha información (véase sección 3.2.2). A continuación, describiremos el conjunto de CA que hemos decidimos recopilar.

3.2.1.1. Características de alerta

Queremos representar de la mejor manera posible el comportamiento de cada uno de los defectos presentes en el código fuente con base en las alertas generadas por la ASAT. Para definir correctamente la estructura de nuestro VCA, debemos tener presente el conjunto de características a explorar y es por ello que nos basamos en las múltiples características que las diferentes AAIT han utilizado para representar cada una de sus alertas (por ejemplo, el tipo de alerta [19, 6, 4, 5, 11, 7], la prioridad [43, 19, 6, 11, 7], el tamaño del método [4, 6, 25, 7] y la complejidad ciclomática [6, 4, 7]). Hemos establecido un total de 46 características distribuidas en 4 categorías y recopiladas de 3 fuentes distintas: una ASAT (FindBugs), una SMT (Metrics) y el repositorio de código fuente. A continuación, proporcionamos la descripción de cada una de las CA seleccionadas y reconocemos los trabajos en los que nos hemos inspirado para definirlas, la descripción es omitida si consideramos que el nombre asociado a la característica es lo suficientemente representativo.

1. **Identificadores de alerta.** Entre la información que la ASAT genera, se encuentran las siguientes 7 CA que identifican de manera única a cada una de las alertas reportadas. Dichas características son descritas a continuación:
   - **Nombre del artefacto** [11, 19, 4, 5, 6, 7]**:** recopilado a nivel de proyecto, paquete, archivo, clase y método. El nombre del paquete es generalizado a la carpeta que contiene al archivo fuente, el nombre del método incluye el(los) parámetro(s) que recibe incluyendo su tipo. Una alerta podría no encontrarse dentro de un método o de una clase.
   - **Ruta del archivo.**
   - **Categoría de la alerta** [6, 7]**:** definida por la ASAT, describe una categorización de alto nivel para cada patrón de alerta (por ejemplo, malas prácticas de programación, concurrencia, desempeño y seguridad).
   - **Patrón de la alerta** [25]**:** definida por la ASAT, describe una categorización de nivel medio para cada tipo de alerta (por ejemplo, mal uso de excepciones, archivos o cadenas de texto, inconsistencia de tipos, referencias a punteros nulos y objetos, métodos o campos sin utilizar).
   - **Tipo de la alerta** [11, 19, 4, 5, 6, 7]**:** definida por la ASAT, describe un posible defecto dentro del código fuente (por ejemplo, excepción lanzada y no declarada, variable entera con conversión de tipos inconsistente, archivo sin cerrar, referencia a un objeto nulo y variable sin utilizar).
   - **Prioridad de la alerta** [43, 11, 19, 6, 7]**:** definida por la ASAT, describe una categorización numérica para el nivel de atención que debe darse a cada alerta.

- **No. de línea de la alerta** [6, 7, 4, 25]**:** describe el número de línea en que se ubica la alerta dentro del código fuente.

2. **Métricas de software.** Diversos investigadores han descubierto múltiples ventajas en la predicción de alertas accionables al utilizar algunas métricas de software [6, 7, 4, 25], en particular, la complejidad ciclomática de McCabe [44] y el número de líneas de código fuente. Lo anterior, atiende el siguiente razonamiento: si las métricas de software permiten descubrir aquellos métodos propensos a sufrir una falla y las alertas accionables son consideradas como una falla dentro del código fuente, entonces dichas métricas podrían aportar a la predicción de alertas accionables. A continuación describimos estas CA, las cuales serán recopiladas a distintos niveles de granularidad, veamos:
   - **Tamaño del artefacto** [6, 7, 4, 25]**:** definida por la SMT, describe el número de LLOC (*Logical Lines Of Code*) a nivel de proyecto, paquete, archivo, clase y método.
   - **No. de paquetes en el proyecto.**
   - **No. de archivos en el artefacto:** recopilados a nivel de proyecto y paquete.
   - **No. de clases en el artefacto** [6, 7]**:** recopilados a nivel de proyecto, paquete y archivo.
   - **No. de métodos en el artefacto** [6, 7]**:** recopilados a nivel de proyecto, paquete y archivo.
   - **Complejidad ciclomática** [6, 7, 4]**:** es una métrica de software que proporciona una evaluación cuantitativa de la complejidad lógica de un programa, definiendo el número de rutas independientes dentro de un método [38]. Dicho valor numérico es definido por la SMT.
3. **Características de repositorio de código fuente.** Los modelos creados por diferentes investigadores han utilizado CA que son recopiladas desde el repositorio de código fuente del proyecto para describir sus alertas [6, 7, 11, 19, 8, 25]. Utilizamos el número de versión en lugar de su fecha de lanzamiento debido a que este dato no sólo permite conocer el grado de avance del proyecto, sino también, el esfuerzo realizado hasta el momento. Estas características son mencionadas a continuación:
   - **No. de versión del proyecto** [11, 19, 6, 7, 4, 8, 25]**.**
   - **Fecha de lanzamiento de la versión** [6, 7]**.**
   - **No. de versión de creación del archivo** [6, 7]**.**
   - **No. de versión de eliminación del archivo** [6, 7]**.**

- **No. de versión de modificación del archivo** [6, 7]**:** describe la última modificación realizada al archivo. Una versión de modificación es registrada cuando el número de líneas de código en el archivo cambia.

4. **Características adicionales.** Estas características son calculadas a partir de las anteriores y proporcionan una mayor descripción de cada alerta en cada uno de sus distintos niveles de granularidad. El tiempo de vida de la alerta, la edad y la ranciedad del archivo son expresadas en días, describimos su definición a continuación:
   - **No. de alertas en el artefacto** [11, 19, 6, 7, 25]**:** describe el número de alertas reportadas por la ASAT para cada versión construida a nivel de proyecto, paquete, archivo, clase y método.
   - **No. total de alertas no accionables en la versión** [6, 7]**:** describe el número total de alertas no accionables presentes hasta el momento. Por ejemplo, si en una versión anterior existen 5 alertas no accionables y durante la versión actual se detectan 3 nuevas alertas, el número total de alertas en la versión es 8.
   - **No. de alertas no accionables en la versión** [6, 7, 9, 10]**:** describe el número de nuevas alertas no accionables identificadas durante la versión analizada, alertas que ya han sido reportadas en versiones anteriores y cuyo estado no accionable se mantiene hasta el momento no son contabilizadas.
   - **No. total de alertas accionables en la versión:** el número total de alertas accionables presentes hasta el momento. Esta característica es calculada de manera similar al número total de alertas no accionables en la versión.
   - **No. de alertas accionables en la versión:** el número de nuevas alertas etiquetadas como accionables dentro de la versión analizada, alertas que ya han sido etiquetadas en versiones anteriores no son contabilizadas.
   - **Tiempo de vida de la alerta** [6, 7, 9, 10, 11, 29]**:** define el tiempo transcurrido (en días) entre la versión de aparición y de corrección de una alerta, es decir, cuántos días transcurrieron desde la generación de la alerta hasta su etiquetado como una alerta accionable. Si la alerta nunca es corregida, el tiempo de vida son los días transcurridos entre la versión de aparición de la alerta y la última versión explorada del proyecto. Esta característica proporciona una idea acerca del tiempo que toma la corrección de una alerta, en caso de haberla corregido.
   - **No. de modificaciones a la alerta** [6, 7, 9, 10]**:** contabiliza el número de veces que ha sido cambiado el número de línea o la prioridad de una alerta durante su tiempo de vida.

- **Tiempo de vida del archivo** [6, 7]**:** esta característica es obtenida de manera similar al tiempo de vida de la alerta. Si un archivo ha sido eliminado, el tiempo de vida es calculado mediante la diferencia entre la versión de creación y la versión de eliminación del archivo en cuestión. Si un archivo nunca es eliminado, será la diferencia entre la versión de creación y la última versión explorada del proyecto.
- **Ranciedad del archivo** [6, 7, 9, 10]**:** describe el tiempo transcurrido entre la versión de creación y la versión de modificación del archivo. Si el archivo desaparece debido a una eliminación, la ranciedad es calculada por la diferencia entre las versiones de creación y de eliminación del archivo.

3.2.2. Procesamiento de alertas

Como se puede ver en la figura 3.4, el resultado que se obtiene una vez culminado el procesamiento de alertas es un conjunto de VCA, los cuales serán predictores de la accionabilidad de una alerta y además, servirán para desarrollar la siguiente etapa (generación de modelos), en donde las CA más relevantes son seleccionadas, se construyen y se evalúan múltiples modelos de aprendizaje maquinal. Deseamos construir dicho conjunto de VCA mediante un programa que automatice esta actividad dado el conjunto de reportes de alertas y el conjunto de reportes de métricas de software generados por la ASAT y la SMT respectivamente. Recordemos que estos dos conjuntos de reportes son el resultado de ejecutar dichas herramientas sobre cada una de las versiones construidas del proyecto en cuestión.

A continuación, describimos las funciones que nuestro programa debe realizar , es decir (1) la heurística de etiquetado de alertas, con la cual se define una forma muy sencilla de etiquetar una alerta como accionable o no accionable, (2) la heurística de reetiquetado de alertas, una vez que se ha definido la accionabilidad de una alerta, algunas permanecen como no accionables hasta al final del análisis, aprovecharemos este hecho para decidir si mantenemos el estado actual o no, a través de un valor de probabilidad que define qué tan relevante es dicha alerta y (3) establecer la estructura que cada VCA debe seguir. Estas 3 funciones nos permitirán generar un conjunto completo de VCA.

3.2.2.1. Heurística de etiquetado de alertas

El problema de etiquetar cada una de las alertas generadas por la ASAT como accionables y no accionables, indica de forma clara la variable dependiente a utilizar en el aprendizaje maquinal: la accionabilidad. Por tanto, resulta muy conveniente contar con un programa que, basado en los VCA generados (los cuales representan a las alertas), se encargue de automatizar la heurística de etiquetado

de alertas (tal como se muestra en la figura 3.5) apoyándose de las características que las identifican, las cuales son: tipo de alerta, nombre del método, nombre de la clase, nombre del archivo y nombre del paquete.

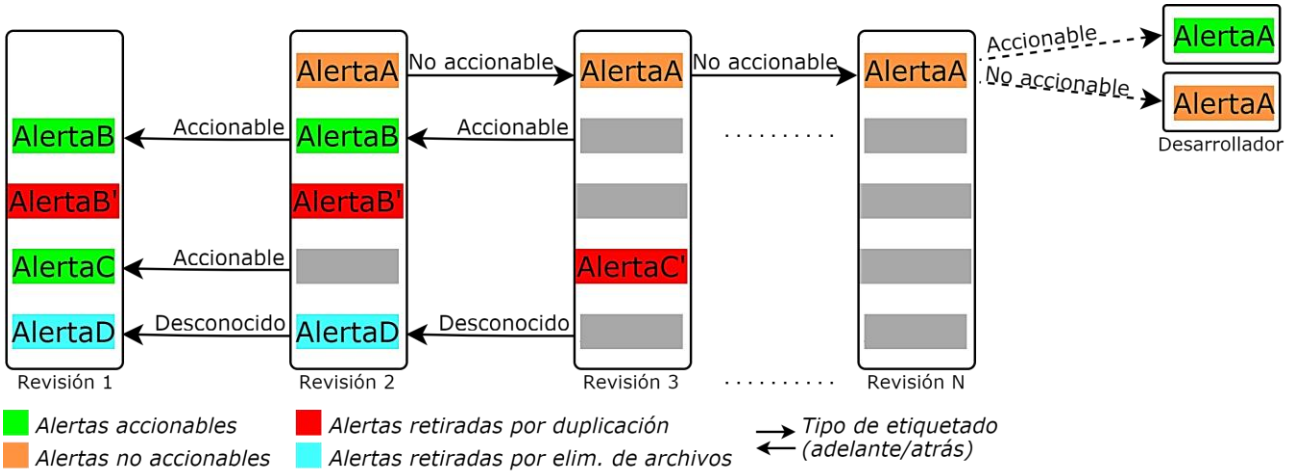


Figura 3.5 Heurística de etiquetado de alertas.

Es importante mencionar que, antes de comenzar el proceso de etiquetado, aquellas alertas que se encuentran dentro de una misma versión y que comparten los mismos identificadores son consideradas como alertas duplicadas y deben retirarse del conjunto, conservando sólo a una de ellas [6, 7, 5]. Lo anterior debe realizarse para cada una de las versiones del proyecto.

Una vez eliminadas las alertas duplicadas en cada versión, el etiquetado es realizado mediante comparaciones de los conjuntos de VCA provenientes de dos versiones sucesivas en el proyecto, comenzando con la versión más antigua y utilizando los identificadores de alerta. Dicho proceso (mostrado en la figura 3.5) se describe a continuación [6, 7, 5]:

- Estando en una versión posterior, una alerta es etiquetada como no accionable si la alerta no está presente en versiones anteriores. Las alertas de la versión inicial son consideradas no accionables por defecto y, aquellas alertas cuya estado no accionable permanece hasta la última versión pueden ser tratadas de dos formas: (1) un desarrollador inspecciona cada una de ellas y, de acuerdo a su experiencia, mantiene el estado no accionable de la alerta o bien, la etiqueta como accionable y (2) todas las alertas son consideradas no accionables sin distinción alguna.
- Una alerta se vuelve accionable cuando dicha alerta estuvo presente en versiones anteriores y desaparece en versiones posteriores. Es posible que una alerta que ha sido etiquetada como accionable, desaparezca sólo en algunas versiones y más tarde vuelva a aparecer; si lo anterior sucede, únicamente se conserva la primera aparición y su estado como una alerta accionable, las apariciones posteriores son retiradas del conjunto. Alertas que hayan sido etiquetadas como

accionables como consecuencia de la eliminación de un archivo guardan un estado desconocido y también deben retirarse del conjunto.

La idea fundamental detrás de esto es que: si una alerta ha estado presente a lo largo de las versiones y jamás ha desaparecido, es probable que dicha alerta no haya sido tan importante para los desarrolladores como para tratarla, o bien, que no repercuta en la correcta funcionalidad del sistema.

3.2.2.2. Heurística de reetiquetado de alertas

Una vez que las alertas han sido etiquetadas como accionables o no accionables, la naturaleza misma de la heurística de etiquetado (descrita en la sección anterior) nos lleva a tener un subconjunto de alertas cuyo estado es desconocido, aunque en un principio indicamos que aquellas alertas que permanecen hasta la última versión son consideradas no accionables, en realidad se trata de una clasificación incierta, pues no tenemos forma de definir su comportamiento dentro del proyecto. Siguiendo con la idea propuesta por Heckman y Williams [6, 7], este subconjunto de alertas puede ser inspeccionado por un desarrollador para decidir (de acuerdo a su experiencia) si la alerta en realidad es accionable, situación que nos lleva a definir una heurística de reetiquetado de alertas, la cual es descrita a continuación:

- Una alerta es reetiquetada como accionable con base en un valor de probabilidad, el cual define qué tan relevante es dicha alerta en el correcto funcionamiento del sistema. Dicha alerta será accionable si su probabilidad de relevancia es mayor a 0.6.
- El cálculo de la probabilidad proviene de 3 fuentes distintas: (1) de la idea del desarrollador acerca de la accionabilidad de la alerta, (2) de las priorizaciones de alertas hechas en trabajos anteriores y de la retroalimentación de falsos positivos de la ASAT[8] y (3) de la proporción de alertas del mismo tipo etiquetadas como accionables dentro del proyecto ajeno al proyecto de estudio.

4. Cada una de estas 3 fuentes contribuye con un peso a la probabilidad de accionabilidad de cada alerta. Por medio de la ec. 3.1 presentamos la forma de calcular dicha probabilidad donde W_i expresa el peso que se le asigna a cada fuente de probabilidad $Prob_i$, veamos:

$$Prob_{Acc} = W_1 * Prob_1 + W_2 * Prob_2 + W_3 * Prob_3 \quad \text{(Ec. 3.1)}$$

5. El valor de W_i es definido al ponderar cada una de las fuentes por su nivel de relevancia. Con

[8] FindBugs [27] utiliza un sistema de control de errores, en el cual, un usuario puede reportar la existencia de un falso positivo y el equipo de desarrollo, después de realizar un análisis exhaustivo del problema, determina si el problema reportado procede o no, lo cual nos da una idea acerca de la accionabilidad de una alerta.

base en el trabajo de Yüksel y Sözer [11], creemos que el mayor peso debe ser asignado a la idea del desarrollador en el cálculo total. Por ello, sugerimos utilizar 0.45 para la idea del desarrollador, 0.20 para las priorizaciones de trabajos anteriores y 0.35 para la proporción de alertas en el proyecto ajeno. La ec. 3.2 muestra la forma de calcular el valor de $Prob_i$ como la proporción de alertas accionables y el total de alertas encontradas dentro de la fuente seleccionada.

$$Prob_i = \frac{No.de\ A_{Acc}}{No.de\ A_{Acc} + No.de\ A_{No\ acc}} \quad \text{(Ec. 3.2)}$$

6. Por último, debemos mencionar que la búsqueda de las priorizaciones realizada en la literatura y la búsqueda de alertas accionables dentro del proyecto ajeno, además de darnos una idea acerca de la accionabilidad de la alerta (idea del desarrollador), está basada en el tipo de la alerta del VCA en cuestión. Por ello, decimos que esta heurística de reetiquetado de alertas se basa en un reetiquetado de tipos.

En general, esta heurística funciona bajo la idea de etiquetar alertas no accionables (o cuyo estado sea desconocido) como alertas accionables de acuerdo a su impacto en la correcta funcionalidad del sistema y en la idea de explorar los tipos de alerta de cada VCA, atendiendo el objetivo de aumentar el número de defectos relevantes descubiertos en el código fuente del sistema.

3.2.2.3. Vectores de características de alerta

Ya hemos dicho antes que, el conjunto de entrenamiento utilizado para la construcción de modelos no es más que un conjunto de VCA, donde cada uno de ellos representa una alerta que se compone de (1) N características de entrada, las cuales se encuentran agrupadas en 4 categorías (identificadores de alerta, métricas de software, características del repositorio de código fuente y características adicionales) y (2) una característica de salida, clase o accionabilidad de la alerta (accionable o no accionable). La figura 3.6 muestra la forma en que concebimos cada VCA, con N características de entrada y la clase de la alerta como salida.

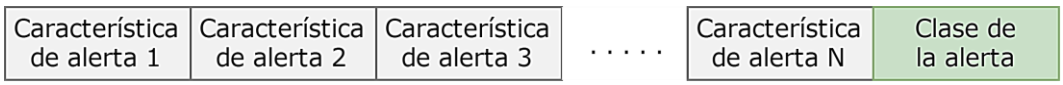


Figura 3.6 Estructura del Vector de Características de Alerta (VCA).

Sabemos también que, este conjunto de VCA deberá ser almacenado en un formato adecuado para poder ser utilizado en el entrenamiento del algoritmo de aprendizaje. En particular, para llevar nuestro conjunto de VCA al formato de archivos requerido por WEKA [32, 22, 45], dicho conjunto deberá ser almacenado en un Formato de Archivo de Relación de Atributos (ARFF, *Attribute-Relation*

File Format); por ello, aprovecharemos la aplicación web que ha diseñado Kuzovkin [46] para transformar un archivo CSV[9] en un archivo ARFF. De manera similar a un archivo CSV, en un ARFF cada vector o instancia del archivo está representada por una fila, mientras que los atributos o características que describen a las alertas están representados por columnas separadas por comas [32, 22, 45]. Una vez que se tiene esto, sería posible entrenar y evaluar diferentes algoritmos de aprendizaje maquinal sobre el mismo conjunto de VCA (situación que será discutida en la siguiente sección).

3.3. Generación de modelos

En el campo de la minería de datos, decimos que un modelo es creado cuando se aplica un algoritmo de aprendizaje a un conjunto de datos con el fin de identificar posibles patrones y obtener nuevo conocimiento a partir de dicha información [22]. Utilizaremos este hecho para construir, analizar y evaluar una serie de modelos candidatos y así, seleccionar al mejor modelo capaz de clasificar de manera efectiva nuevas alertas, aunque esta idea no necesariamente es una tarea sencilla, veremos que, una herramienta capaz de realizar todas estas actividades disminuirá dicha complejidad.

La figura 3.7 muestra la forma en que utilizaremos WEKA para generar nuestros modelos utilizando la técnica de validación cruzada [22] sobre el conjunto de VCA antes generado y sobre los diferentes subconjuntos de las CA más relevantes. El desempeño de cada modelo será evaluado con base en las medidas de desempeño identificadas en la sección 2.2 (exactitud, precisión y sensibilidad). Queremos insistir en que el objetivo principal de este estudio no es proponer, mejorar o adaptar algún algoritmo de aprendizaje a nuestro problema, por ello, estaremos interesados en explorar la mayor cantidad posible de algoritmos (tanto para la selección de CA, como para la construcción y evaluación de modelos) utilizando la configuración por defecto que dicha herramienta ofrezca.

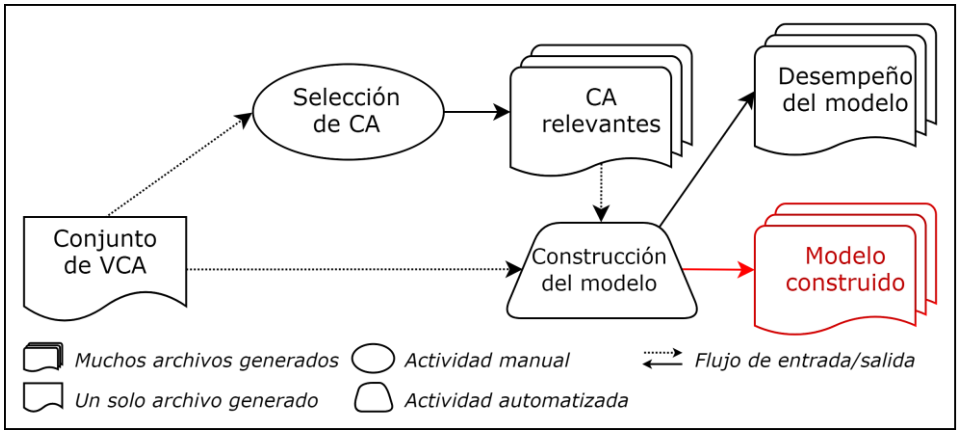


Figura 3.7 Proceso de generación de modelos.

[9] Valores Separados por Comas (CSV, *Comma-Separated Values*).

3.3.1. Selección de características de alerta

La selección de las características más relevantes en la accionabilidad de una alerta es de vital importancia para llevar el aprendizaje maquinal al contexto de nuestro problema, la razón es porque aquellas CA redundantes e irrelevantes en el conjunto de datos podrían distraer o confundir a algunos algoritmos de aprendizaje maquinal [6, 7], afectando directamente en la clasificación de nuevas alertas; de hecho, a mayor número de CA irrelevantes menor el desempeño logrado (hasta un 10% menor [22]), además de que cierta CA podría requerir de mucho tiempo para su recopilación y aportar muy poco al poder predictivo del modelo [6, 7, 22].

Witten y Frank [22] sugieren que la mejor manera de seleccionar las CA más relevantes es mediante una inspección manual basada en una comprensión profunda de la problemática, sin embargo, esto implicaría un incremento en el tiempo requerido para realizarlo, por lo que se vuelve necesario contar con métodos que automaticen dicha actividad. WEKA proporciona una serie de algoritmos de selección de atributos basados en múltiples métodos de evaluación y estrategias de búsqueda para identificar las CA con mayor poder predictivo dentro de un conjunto de datos [33, 11, 19, 23, 22, 45, 32].

Como se puede ver en la figura 3.7, dado un conjunto de VCA conseguiremos todos los subconjuntos posibles de CA al combinar todos los métodos y estrategias que WEKA nos ofrezca bajo su configuración por defecto [22, 32]. Cabe mencionar que, estaremos interesados en seleccionar aquellos subconjuntos con un número reducido de CA (entre 3 y 16 características), pues trabajos anteriores han demostrado obtener excelentes resultados (un mejor desempeño del modelo y una reducción en el tiempo de construcción) bajo dicha condición [7, 6, 25, 23, 24, 11, 19, 10, 9].

3.3.2. Construcción y evaluación de modelos

Como vimos en la sección 1.3, en la metodología *training and testing* una porción de los datos es utilizada para entrenar un algoritmo de aprendizaje (*training*) y la parte restante es utilizada para evaluar su desempeño en la clasificación de nuevas instancias de datos (*testing*) [6, 22]. Al igual que en el procesamiento de alertas detallado en la sección 3.2.2, estamos interesados en ejecutar de manera automatizada estas dos actividades a través de la herramienta WEKA, ya que esta relaciona dichas actividades de manera adecuada mediante la implementación de múltiples algoritmos de aprendizaje maquinal que utilizan la técnica de validación cruzada para la construcción y evaluación de modelos.

Como se muestra en la figura 3.7, una vez que se ha generado un conjunto de VCA previamente etiquetados como accionables o no accionables y múltiples subconjuntos de CA relevantes, se

construyen y se evalúan diferentes modelos candidatos para cada uno de dichos subconjuntos utilizando los valores de configuración por defecto que WEKA asigna a cada uno de sus algoritmos de aprendizaje [32, 22]. Si bien hemos mencionado que construiremos el mayor número posible de modelos para cada uno de los subconjuntos de CA, también debemos señalar la forma en que será evaluado el desempeño de cada uno de ellos: centraremos particular atención en la exactitud, precisión y sensibilidad de todos nuestros modelos debido a su amplia aceptación en la investigación de las AAIT [6, 7, 8, 4, 23, 24, 19, 11] y por su aplicabilidad a la mayoría de los algoritmos de aprendizaje maquinal. Estas medidas nos apoyarán en la elección del mejor modelo capaz de predecir la accionabilidad de nuevas alertas y en el análisis de su fiabilidad al ser incorporado al PDS.

4. EVALUACIÓN DE LA PROPUESTA

Como ya hemos mencionado en el capítulo anterior, el diseño de nuestra Técnica de Identificación de Alertas Accionables (AAIT, *Actionable Alert Identification Technique*) propone llevar a cabo una metodología compuesta de 3 etapas, la cuales son: (1) generación de versiones, (2) generación de características y (3) generación de modelos. Por ello, a lo largo de este capítulo describiremos la manera en que hemos implementado dichas etapas y los resultados obtenidos en cada una de ellas, siguiendo en todo momento la metodología antes mostrada (véase capítulo anterior) y puntualizando en los detalles más relevantes de cada actividad: los proyectos estudiados, las versiones construidas, las herramientas utilizadas, los reportes de alertas generados, el conjunto de Vectores de Características de Alerta (VCA) creado, las Características de Alerta (CA) más relevantes y por último, el desempeño de los modelos construidos.

Adicionalmente, debemos mencionar que la evaluación de nuestra propuesta es llevada a cabo desde dos perspectivas diferentes al variar el conjunto total de CA[10] empleadas para el entrenamiento del modelo y el proyecto en que dicho modelo es evaluado. Con esto en mente, partimos de la recreación del experimento de Heckman y Williams [7, 6] (basado en la construcción de modelos internos al proyecto) y luego lo adaptamos para lograr aplicarlo en la construcción de modelos externos al proyecto, lo cual nos permitirá realizar un análisis comparativo con los resultados obtenidos en cada una de dichas configuraciones con el fin de establecer ventajas, desventajas e implicaciones una vez que nuestra AAIT sea incorporada al Proceso de Desarrollo de Software (PDS). En particular, veremos que la segunda configuración nos llevará a validar nuestra hipótesis, la cual sostiene que es posible disminuir el número de defectos irrelevantes por reparar antes de llegar a la fase de pruebas dentro del PDS a través de la construcción de un modelo externo para la clasificación de nuevas alertas generadas por una ASAT.

A continuación, describiremos cada una de las configuraciones evaluadas dentro de nuestro estudio:

1. **Modelos internos.** Decimos que un modelo interno es construido cuando entrenamos un algoritmo de aprendizaje maquinal con base en el conjunto de VCA propio del proyecto y lo evaluamos sobre un conjunto de VCA con posibles nuevos defectos del mismo sistema.

[10] La descripción del conjunto total de CA puede ser consultado en la sección 3.2.1.1.

Utilizamos un total de 46 CA recopiladas de 3 fuentes distintas (una Herramienta de Análisis Estático Automatizado -ASAT, *Automated Static Analysis Tool*- como lo es FindBugs [27], una Herramienta de Métricas de Software -SMT, *Software Metrics Tool*- como lo es Metrics [42] y el repositorio de código fuente como lo es SVN -*Subversion*- [40]) para generar un conjunto de VCA y, mediante la técnica de validación cruzada [22], entrenar y evaluar más de 540 modelos candidatos para la clasificación de nuevas alertas. Basados en su exactitud, precisión y sensibilidad, determinamos a los 12 modelos que logran el mayor desempeño posible para dos proyectos de software escritos en Java [20]. Cabe mencionar que, esta configuración explota al máximo las propiedades identificadas dentro del trabajo de Heckman y Williams [6, 7] (quienes han diseñado el estudio más completo hasta el momento en el área de las AAIT) y cuya principal aportación fue incorporar características de temporalidad a cada uno de los VCA.

Este enfoque de modelos internos ha sido ampliamente utilizado debido a que se ha comprobado que son capaces de obtener los mejores resultados una vez que son llevados a la práctica [6, 7, 11, 19], por ello, el objetivo de esta configuración es mostrar la capacidad máxima que estos modelos pueden tener en la clasificación de nuevas alertas y, con base en sus ventajas y desventajas, migrar hacia nuestra propuesta.

2. **Modelos externos.** Decimos que un modelo externo es construido cuando entrenamos un algoritmo de aprendizaje con base en el conjunto de VCA de un proyecto y lo evaluamos sobre el conjunto de VCA de otro; como vimos en la sección 2.4, una de las desventajas de utilizar características de temporalidad en la generación de modelos es que el proyecto debe contar con un historial de versiones. Por esta razón, en esta configuración suponemos el caso en que se tiene un proyecto de software de reciente creación y que por tanto, no se cuenta con CA asociadas a dicho historial. Dicho lo anterior, omitimos aquellas CA que representan características de temporalidad, características del repositorio del código fuente y nombres de artefactos (véase sección 3.2.1.1) y utilizamos sólo 26 CA para describir cada una de las alertas que componen al conjunto de entrenamiento, construir más de 540 modelos candidatos utilizando validación cruzada y exportar a los 12 mejores con su base en su desempeño (exactitud, precisión y sensibilidad). Esta configuración atiende la idea de utilizar modelos externos para la clasificación de nuevas alertas.

4.1. Generación de versiones

Durante el diseño de nuestra propuesta, discutimos la importancia de estudiar proyectos de software que utilicen una herramienta para gestionar sus cambios, es decir, que cuenten con un repositorio de código fuente para almacenar cada una de las versiones del sistema (véase sección 3.1). Este hecho es de gran importancia para el desarrollo de nuestra AAIT al momento de definir la accionabilidad de una alerta mediante la heurística de etiquetado de aleras propuesta por Heckman y Williams [6, 7] (véase sección 3.2.2.1).

Debido a la naturaleza de nuestro problema y con el fin de simular condiciones semejantes en ambos proyectos, debemos estudiar dos sistemas de software actualmente en uso y de características similares con la idea de ejecutar los modelos generados de un proyecto en otro. A continuación, describimos los aspectos generales de cada uno de ellos:

1. **SAPCyTI** o Sistema de Administración de Posgrado en Ciencias y Tecnologías de la Información [47]. Es una aplicación web que automatiza el proceso de inscripción al Posgrado en Ciencias y Tecnologías de la Información (PCyTI) dentro de la Universidad Autónoma Metropolitana, Unidad Iztapalapa (UAM-I) [48]. Dirigido por el Dr. Humberto Cervantes Maceda, este proyecto (codificado en lenguaje Java) aplica la metodología Scrum [38, 39] y se apoya de distintos framewoks de desarrollo (principalmente Spring [49]), además de contar con un repositorio de código fuente como lo es SVN (*Subversion*) [40].
2. **JDOM** [34]**.** Es una biblioteca de código abierto creada exclusivamente para ser utilizada en Java, la cual proporciona la posibilidad de leer, manipular y escribir de manera sencilla y eficiente archivos XML[11]. Este proyecto fue fundado por Jason Hunter en el año 2000 y actualmente gestiona su desarrollo mediante un CVS (*Concurrent Versions System* o Sistema Concurrente de Versiones) [41], además de hacerlo público a través de internet.

Dicho lo anterior, recordemos que el artefacto final que se obtiene al término de la etapa de generación de versiones (como se mostró en la sección 3.1), es un conjunto de archivos de código fuente compilado que componen a cada una de las versiones del proyecto y para lograrlo, hemos realizado dos actividades muy importantes: la creación del historial de versiones y la construcción de versiones.

Creación del historial de versiones.

Con el objetivo de organizar la construcción de las distintas versiones de cada proyecto,

[11] Desarrollado por el Consorcio World Wide Web o W3C, el Lenguaje de Marcado Extensible (XML, *eXtensible Markup Language*) es un metalenguaje que se utiliza para almacenar grandes cantidades de información de manera estructurada.

guardamos (dentro de un archivo CSV[12]) un conjunto de registros con la información relacionada a dichas versiones (llamado historial de versiones), un registro por cada versión. Los datos que conforman a cada registro son muy sencillos, el número de versión del proyecto, su fecha y su hora de lanzamiento. Basados en el trabajo de Heckman y Williams [6, 7], sabemos que es posible explorar versiones tipo *major* o *minor* en lugar de revisiones[13] y viceversa de manera indistinta; al ser SAPCyTI un sistema de reciente creación, este no está compuesto por un gran número de versiones, por esta razón y por practicidad en nuestro estudio, optamos por crear de manera manual el historial de las 32 versiones que lo componen.

Por otro lado, el historial para JDOM no ha sido generado por nosotros. Hemos extraído del benchmark FAULTBENCH v0.3 (un punto de referencia para evaluar y comparar AAIT [5, 6, 7, 50]) el historial de revisiones de JDOM, el cual está compuesto por más de 1000 revisiones del proyecto, selecciona las revisiones 1, 26, 51, 76, 101... y las incluye en el listado hasta formar un total de 48, la última revisión también es incluida. La figura 4.1 muestra un fragmento del historial generado para JDOM, dicha información (al igual que en SAPCyTI) es almacenada en un archivo CSV. De acuerdo a Heckman y Williams [6, 7], la razón de seleccionar sólo un subconjunto de revisiones radica en la sobrecarga que podría ocurrir al momento de realizar el procesamiento de alertas, en SAPCyTI, esto no ha sido necesario debido a nuestra decisión de trabajar sobre las versiones tipo *major* y *minor* del proyecto. Cabe mencionar que, el historial generado durante esta actividad sólo refleja la previa selección de las versiones del proyecto a construir.

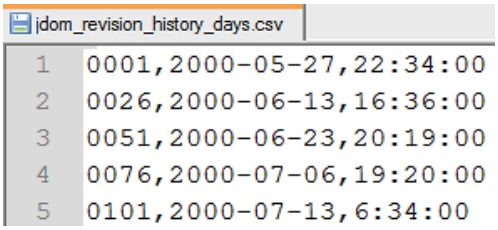


Figura 4.1 Historial de revisiones generado (fragmento).

Construcción de versiones.

Ya que hemos generado el historial de versiones para cada uno de los proyectos en cuestión, debemos construirlas, lo cual implica reunir todos los componentes necesarios para lograr una compilación completa de cada una de ellas, recordemos que una construcción parcial proporcionaría un análisis estático inconsistente [6, 7] y en caso de ocurrir esto, la versión debe ser omitida. En

[12] Valores Separados por Comas (CSV, *Comma-Separated Values*).
[13] La diferencia entre versiones tipo *major*, *minor* y revisiones puede ser consultada en la sección 2.1.

particular, los componentes necesarios para construir cada una de las versiones del proyecto SAPCyTI fueron: la instalación del sistema operativo Ubuntu 14.04 [51], Java SE 7 [52] y Maven 3.0.5 [53], además de Spring Tool Suite 3.6.1 [54] y Subclipse 1.10.13 [55]. Una vez instalados todos estos componentes, conseguimos acceder al repositorio de código fuente de SAPCyTI[14] para descargar las versiones indicadas en el historial y comenzar con la construcción de las mismas. La figura 4.2 muestra la última versión de SAPCyTI luego de haber accesado a su repositorio.

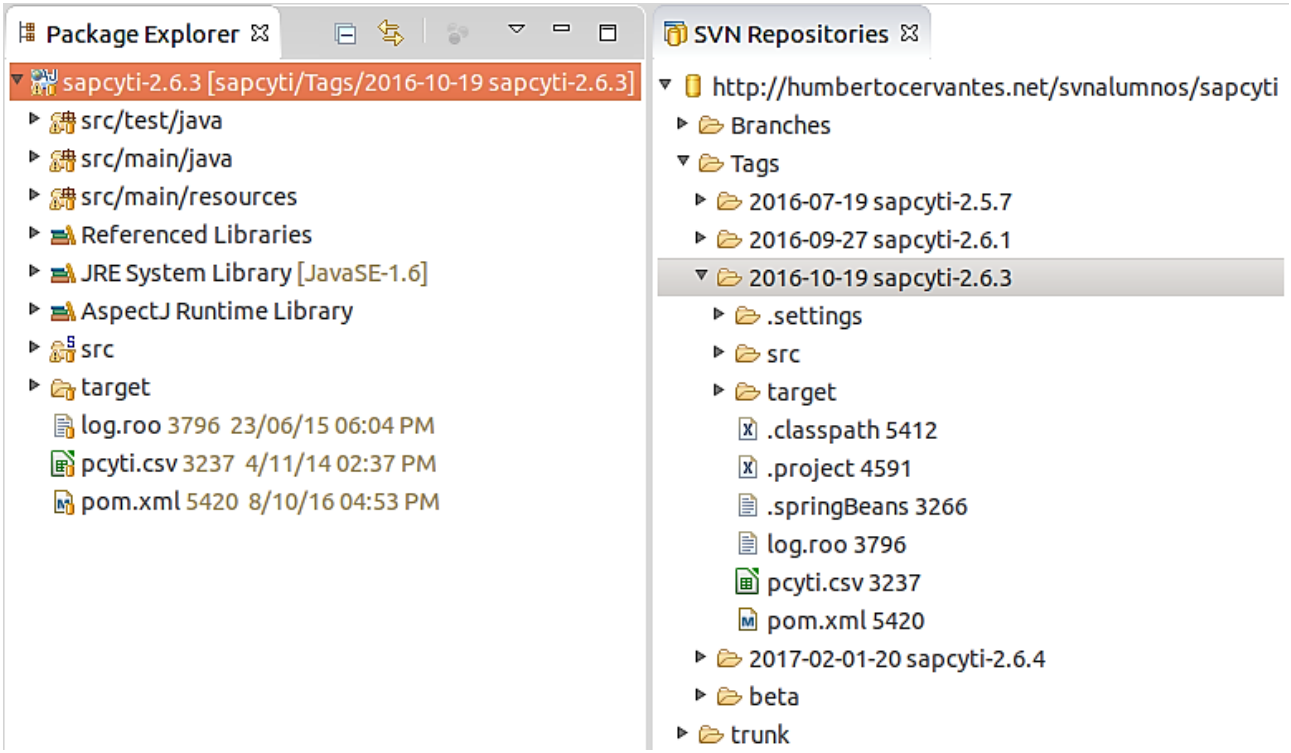


Figura 4.2 Acceso al repositorio de código fuente de SAPCyTI.

La actividad de construcción de versiones para SAPCyTI resulta ser muy sencilla, pues una vez descargada la versión a construir, el entorno nos brinda la posibilidad de actualizar todas las dependencias del proyecto de manera automática y, en caso de concluir satisfactoriamente dicha actualización (lo cual indica una compilación completa del proyecto), se crea una carpeta con todos los archivos Java ya compilados; si lo anterior sucede, copiamos dicha carpeta y la etiquetamos de acuerdo al historial de versiones. Cabe mencionar que, optamos por realizar esta actividad de manera manual debido a la simplicidad de la misma y a que la descarga y construcción de cada versión no toma más de 5 minutos, lo cual nos ayuda a evitar la curva de aprendizaje que implicaría construir cada versión de manera automatizada.

Recordemos que el objetivo principal de construir cada versión del proyecto es contar con un

[14] SAPCyTI es un proyecto de software privativo y por ello, el acceso a su repositorio de código fuente está controlado por el administrador del sistema.

conjunto de archivos de código fuente compilado, los cuales habrán de ser analizados por una ASAT con el fin de obtener un reporte de alertas de análisis estático para dicha versión. Para el caso de JDOM, estos reportes también forman parte del benchmark FAULTBENCH v0.3 [50] y por tanto, encontramos redundante la realización de esta actividad y decidimos omitirla.

Por último, presentamos una tabla con la información que resume la etapa de generación de versiones para cada uno de los proyectos bajo análisis (véase tabla 4.1). Logramos construir 21 de 32 versiones para SAPCyTI debido a que algunas versiones o no se construyeron de manera satisfactoria o se encontraban duplicadas dentro del historial. De acuerdo a Heckman y Williams [6, 7], para JDOM se construyeron 30 de 48 revisiones, lo cual concuerda con los reportes de análisis estático extraídos desde FAULTBENCH v0.3 [50]. Adicionalmente, observamos que el proyecto SAPCyTI contiene más de 10 KLOC (*Kilo Lines Of Code*) distribuidas en más de 100 archivos y en más de 150 clases, características que son similares a JDOM.

	SAPCyTI	**JDOM**
Dominio	Institucional	Formateo
Repositorio	SVN	CVS
Versiones extraídas	32	48
Versiones construidas	21	30
Tamaño	6683 - 10731	9122 - 12846
No. de archivos	75 - 103	86 - 114
No. de clases	110 - 164	109 - 153

Tabla 4.1 Información de los proyectos bajo análisis.

4.2. Generación de características

Continuando con la metodología esbozada en nuestro diseño (véase sección 3.2), queremos insistir en que el resultado esperado después de concluir la etapa de generación de características es un conjunto de VCA que nos permita construir y evaluar modelos para la clasificación de nuevas alertas. Durante esta sección, describiremos a detalle la manera en que hemos llegado a dicho conjunto de VCA a través de la ejecución de una ASAT y una SMT sobre el conjunto de versiones construidas de cada proyecto, además del procesamiento dado a los reportes generados por cada una de estas herramientas.

Liang *et al.* [25] sostienen que para generar un conjunto de entrenamiento eficiente y eficaz, este debe ser construido bajo un enfoque manual y automatizado. Siguiendo esta idea, optamos por crear nuestro conjunto de VCA de manera híbrida, donde la heurística de etiquetado de alertas es realizada de forma automatizada y la ejecución de herramientas y la heurística de reetiquetado de alertas de forma manual.

Ejecución de herramientas.

Una vez que se cuenta con el conjunto de archivos de código fuente compilado, los cuales construyen una versión específica de SAPCyTI, ejecutamos de manera manual una ASAT (FindBugs 3.0.1 [27]) y una SMT (Metrics 1.6.0 [42]) sobre cada una de las versiones construidas con el fin de obtener un conjunto de reportes de alertas de análisis estático y un conjunto de reportes de métricas de software orientadas a objetos, un reporte de cada herramienta por cada versión del proyecto (en total 42). La figura 4.3 y la figura 4.4 muestran el análisis que estas herramientas realizan dada una versión construida de SAPCyTI, hemos utilizado la configuración por defecto que ofrecen FindBugs y Metrics respectivamente.

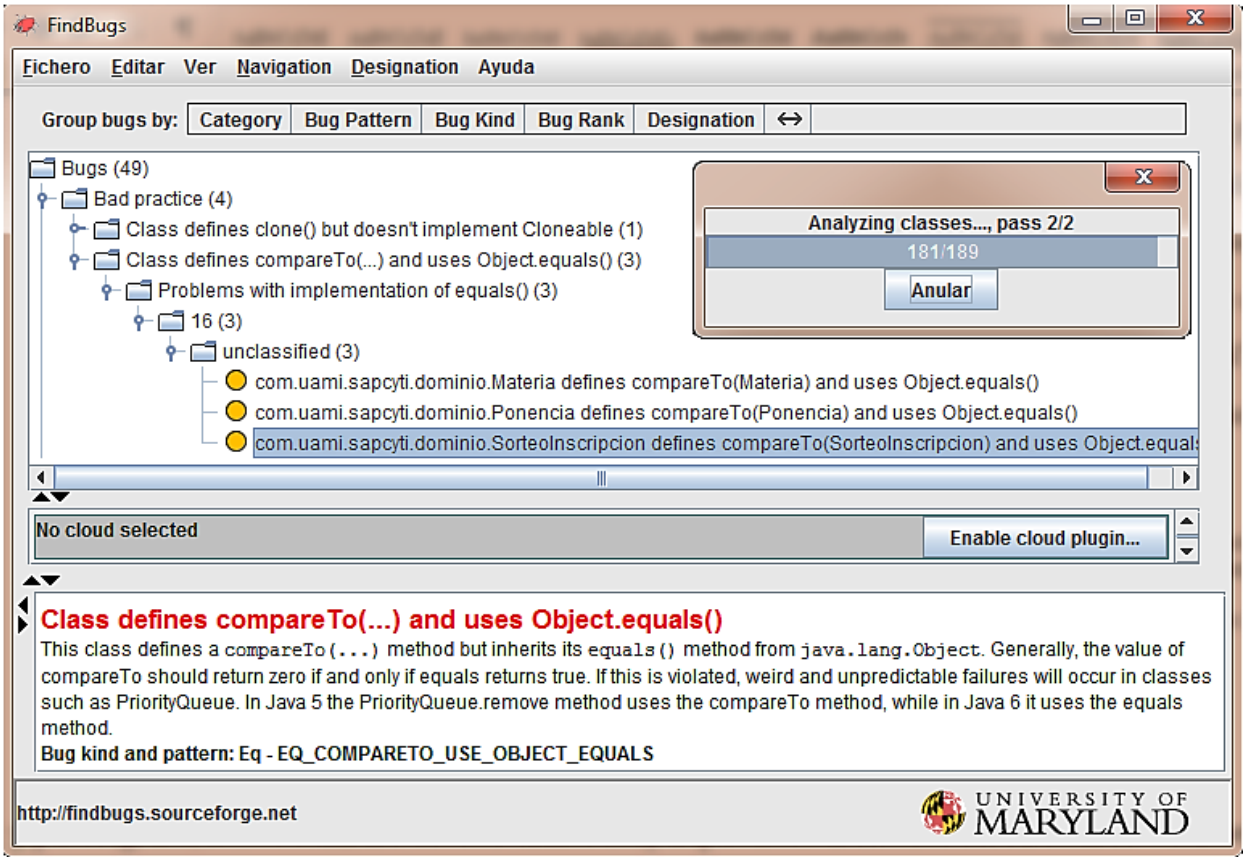


Figura 4.3 Ejecución de FindBugs 3.0.1 sobre una versión construida.

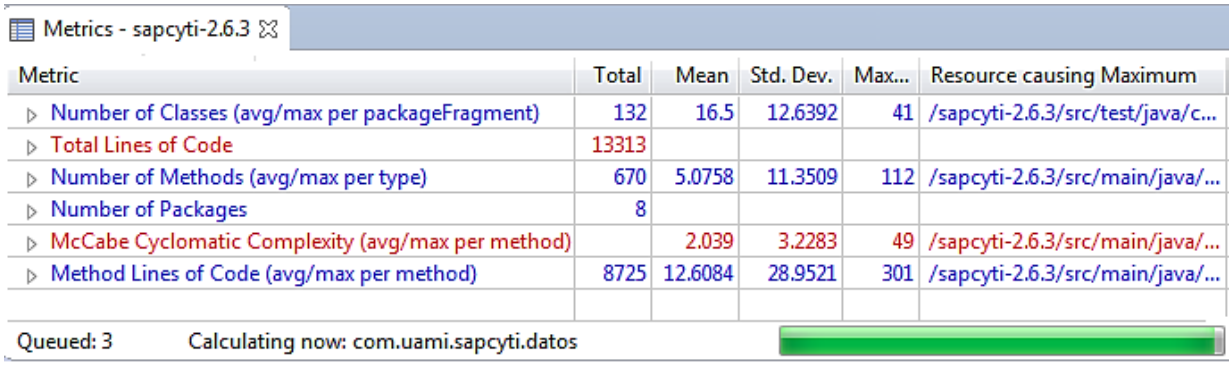


Figura 4.4 Ejecución de Metrics 1.6.0 sobre una versión construida.

Hasta ahora, sólo hemos mencionado que de la ejecución de la ASAT y de la SMT se crea un conjunto de reportes de alertas o de métricas según sea el caso, sin embargo, no hemos profundizado en la estructura de cada uno de ellos. Una de las ventajas que tienen FindBugs y Metrics es que los resultados que cada herramienta genera pueden ser exportados en archivos XML, este hecho hace que el procesamiento de la información almacenada en dichos archivos o reportes sea más sencillo. Recordemos que de estos reportes, habremos de extraer la mayor parte de las características que describirán a cada una de las alertas de los proyectos en cuestión.

Para el caso de JDOM, esta actividad ha sido omitida debido a que el benchmark FAULTBENCH v0.3 [50] contiene 30 reportes de alertas generados por FindBugs 1.3.7 y 30 reportes de métricas de software generados por la herramienta JavaNCSS 21.41 [56]. Por ello, optamos por retomar dichos reportes para la realización de la siguiente actividad: el procesamiento de alertas.

Procesamiento de alertas.

Hasta aquí, hemos realizado diferentes actividades de manera aislada para cada uno de los proyectos en cuestión: utilizamos versiones tipo *major* y *minor* en lugar de revisiones para SAPCyTI, instalamos su entorno de desarrollo, construimos cada una de dichas versiones y generamos sus respectivos reportes. Para el caso de JDOM, omitimos estas actividades porque tanto el historial como los reportes, forman parte del benchmark creado por Heckman y Williams [5], situación que nos ha permitido ahorrar un poco de tiempo y esfuerzo; sin embargo, el alcance de este benchmark sólo nos permite llegar hasta este punto. Afortunadamente, llegamos a esta actividad (el procesamiento de alertas) bajo la misma idea: contar con un reporte de alertas y un reporte de métricas de software para cada una de las versiones. Por ello, a partir de ahora, todas las actividades que se describan son aplicables a ambos proyectos, por lo que detallaremos el proceso para SAPCyTI y obviaremos los detalles para JDOM.

Ya hemos discutido que el procesamiento de alertas tiene como principal objetivo generar un conjunto de VCA previamente etiquetados de acuerdo a su accionabilidad (véase sección 3.2) y para lograrlo, hemos desarrollado un programa encargado de automatizar la estructuración y el etiquetado de cada uno de los vectores de características, tomando como entrada el historial de versiones, los reportes de alertas de análisis estático y los reportes de métricas de software asociados a cada versión, generando como salida el conjunto total de VCA del proyecto en cuestión; insistimos, cada uno de los vectores de características que componen a dicho conjunto representan a cada una de las alertas generadas por FindBugs y a su etiquetado correspondiente (accionable o no accionable). A

continuación, describiremos las dos etapas por las que atraviesa nuestro programa una vez que es puesto en ejecución (es decir, la extracción y el cálculo de características), veamos:

1. **Extracción de CA.** Durante esta etapa, el programa toma como entrada el historial de versiones de SAPCyTI, un reporte de alertas generado por FindBugs y un reporte de métricas de software generado por Metrics para extraer 35 CA como lo son identificadores de alerta, métricas de software e información acerca de la versión (la descripción de estas características puede ser consultada en la sección 3.2.1.1).

   Para lograr recopilar estas características, el programa realiza una lectura (línea por línea) de los archivos de entrada, interpretando cada uno de sus campos y, para el caso del reporte de métricas de software, estableciendo las relaciones que dichos campos guardan con cada alerta generada; en esencia, se realiza un barrido de la información almacenada en cada archivo y conforme se identifican características, estas son almacenadas dentro de un repositorio. Por último, el programa extrae del repositorio todas las CA identificadas y las organiza de acuerdo a su nivel de granularidad (es decir, a nivel de alerta, método, clase, archivo, paquete y proyecto) para después ser almacenadas en un archivo CSV parcial. Este proceso se realiza de manera iterativa hasta procesar cada una de las versiones en el historial.

   En la figura 4.5 podemos observar un fragmento de la información almacenada en uno de los múltiples archivos CSV generados a lo largo de esta etapa (por ejemplo, A0015-sapcyti-alerts.xml-VECTORS.csv). Queremos hacer hincapié en el siguiente detalle: cada archivo CSV generado representa una versión del sistema y, cada VCA (es decir, cada fila dentro del archivo) representa una alerta reportada por FindBugs dentro de la versión.

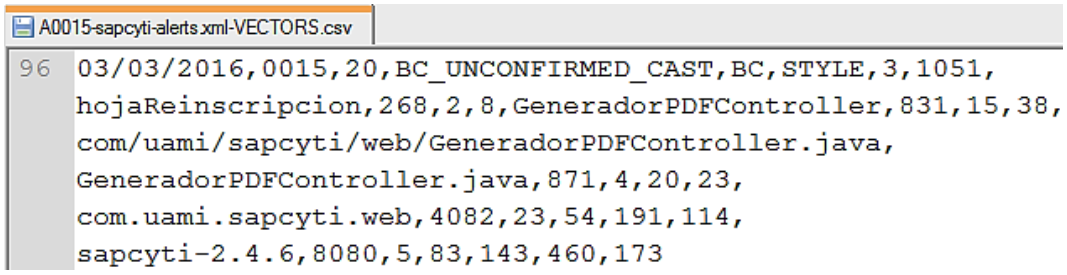


Figura 4.5 Archivo CSV generado durante la extracción de CA (fragmento).

2. **Cálculo de CA.** En esta etapa, el programa toma como entrada los archivos CSV generados en la etapa anterior (uno a la vez) y calcula 11 características de temporalidad (es decir, aquellas CA que son calculadas a partir de otras o con base en el historial de versiones) para cada uno de los VCA allí almacenados. Con esto en mente, el programa continúa el procesamiento aplicando la

heurística de etiquetado de alertas propuesta por Heckman y Williams [6, 7] (descrita en la sección 3.2.2.1) y etiquetando cada VCA (o fila dentro del archivo) como accionable o no accionable. De manera simultánea, realiza el cálculo de 11 CA adicionales, como lo son: el tiempo de vida de la alerta, la edad y la ranciedad del archivo, el número de modificaciones a la alerta, entre otras (véase categoría de características adicionales en la sección 3.2.1.1). Similarmente a la etapa anterior, estas CA son almacenadas dentro de un repositorio para después ser extraídas e insertadas en forma de vectores dentro de otro archivo CSV.

Finalmente, nuestro programa crea un archivo CSV final (ConjuntoVCA_SAPCyTI.csv) que contiene el conjunto total de VCA del proyecto. La figura 4.6 muestra la estructura que hemos dado a las 46 CA que componen a cada vector del conjunto, estas características se encuentran divididas en características extraídas (en color verde), características calculadas (en color azul) y valor de accionabilidad asignado. Adicionalmente, la figura 4.7 muestra un ejemplo de la información almacenada en dicho archivo CSV.

Fecha_Lanz_Vers	Num_Vers_Proy	ID_Alert	Tipo_Alert	Patron_Alert	Categ_Alert	Prioridad_Alert
Num_Linea_Ini	Nom_Metod	Tam_Metod	Num_Alertas_Metod	CC_Metod	Nom_Clase	Tam_Clase
Num_Metods_Clase	Num_Alerts_Clase	Ruta_Arch	Nom_Arch	Tam_Arch	Num_Clases_Arch	
Num_Metods_Arch	Num_Alerts_Arch	Nom_Paq	Tam_Paq	Num_Archs_Paq	Num_Clases_Paq	
Num_Metods_Paq	Num_Alerts_Paq	Nom_Proy	Tam_Proy	Num_Paqs_Proy	Num_Archs_Proy	
Num_Clases_Proy	Num_Metods_Proy	Num_Alerts_Proy				
Num_Vers_Create	Num_Vers_Delete	Num_Vers_Modify	Staleness_Arch	Num_Total_AlertsAcc		
Num_Alerts_Acc	Num_Total_AlertsNoAcc	Num_Alerts_NoAcc	Num_Mods_Alert	Alert_Lifetime_R		
Edad_Archivo_R						
IsActionable						

Figura 4.6 Estructura de cada uno de los VCA del conjunto final.

ConjuntoVCA_SAPCyTI.csv

```
3  03/03/2016,0015,20,BC_UNCONFIRMED_CAST,BC,STYLE,3,1051,
   hojaReinscripcion,268,2,8,GeneradorPDFController,831,15,38,
   com/uami/sapcyti/web/GeneradorPDFController.java,
   GeneradorPDFController.java,871,4,20,23,
   com.uami.sapcyti.web,4082,23,54,191,114,
   sapcyti-2.4.6,8080,5,83,143,460,173,
   4,0,32,0,60,30,172,23,4,17,229,11,369,
   UnActionable
4  28/02/2015,0004,20,BC_UNCONFIRMED_CAST_OF_RETURN_VALUE,BC,STYLE,3,270,
   actualizaDatosDeAlumno,10,1,3,ServiciosAlumno,115,14,4,
   com/uami/sapcyti/negocio/ServiciosAlumno.java,
   ServiciosAlumno.java,115,1,14,4,
   com.uami.sapcyti.negocio,1421,16,35,80,22,
   sapcyti-2.0,6683,5,75,110,315,136,
   4,0,4,369,0,0,136,136,5,11,369,0,0,
   Actionable
```

Figura 4.7 Archivo CSV generado luego del cálculo de CA (fragmento).

Hasta este punto, hemos generado un conjunto de 308 alertas para el proyecto SAPCyTI, 97 son alertas accionables y 207 son alertas no accionables, 4 alertas han sido retiradas del conjunto por tratarse de alertas duplicadas y no son consideradas dentro del experimento. Para JDOM, se han generado 436 alertas, de las cuales 205 son accionables, 197 son no accionables y 34 han sido retiradas del conjunto. La tabla 4.2 sintetiza esta información acerca de los conjuntos de VCA generados para SAPCyTI y JDOM, cabe mencionar que estos datos aún no reflejan el resultado de aplicar nuestra heurística de reetiquetado de alertas a dicho conjunto de vectores.

	SAPCyTI	JDOM
Alertas totales	308	436
Alertas accionables	97	205
Alertas no accionables	207	197
Alertas retiradas	4	34

Tabla 4.2 Información parcial de los conjuntos de VCA generados.

Recordemos que una de las desventajas de la heurística de etiquetado de alertas radica en el hecho de tener un número finito de versiones del proyecto. Debido a que existe una última versión en el historial, se vuelve insostenible continuar comparando las alertas del conjunto actual y las alertas generadas en una versión posterior (ya que esta no existe), lo que provoca que dentro del conjunto de VCA (parcialmente etiquetado) resida un subconjunto de alertas etiquetadas como no accionables, pero que en realidad se desconoce su accionabilidad. Por ello, aplicamos sobre dicho subconjunto la heurística de reetiquetado de alertas descrita en la sección 3.2.2.2 con el fin de precisar la accionabilidad de cada una de estas alertas.

Como ya hemos dicho antes, nuestra heurística de reetiquetado de alertas reetiqueta una alerta no accionable como accionable con base en un valor de probabilidad, el cual describe qué tan relevante es la alerta en el correcto funcionamiento del sistema mediante la inspección de su tipo (véase sección 3.2.2.2). La tabla 4.3 muestra los valores de probabilidad para los 13 tipos de alerta más propensos a revelar un defecto dentro del sistema y cuya descripción puede ser consultada en la tabla A.1 al final de este documento (véase apéndice A), recordemos que dicho valor se obtiene de la suma de las ponderaciones provenientes de 3 fuentes distintas, las cuales son: (1) idea del desarrollador, (2) retroalimentación de trabajos anteriores y (3) proporción de alertas accionables dentro del proyecto. Cabe mencionar que no hemos inspeccionado los más de 400 tipos de alertas que FindBugs es capaz de detectar [27], tan sólo hemos explorado los 51 tipos de alertas que han sido reportados para el proyecto SAPCyTI.

No.	Tipo	Idea	Literat.	Proyecto	Probab.
1	**ES_COMPARING_STRINGS_WITH_EQ**	0.45	0.10	0.35	0.90
2	**DMI_BAD_MONTH**	0.35	0.10	0.35	0.80
3	**BC_UNCONFIRMED_CAST_OF_RETURN_VALUE**	0.26	0.10	0.35	0.71
4	**NP_NULL_PARAM_DEREF**	0.45	0.09	0.18	0.71
5	**ICAST_IDIV_CAST_TO_DOUBLE**	0.35	0.00	0.35	0.70
6	**REC_CATCH_EXCEPTION**	0.15	0.18	0.35	0.67
7	**OS_OPEN_STREAM_EXCEPTION_PATH**	0.16	0.14	0.35	0.65
8	**RV_RETURN_VALUE_IGNORED_BAD_PRACTICE**	0.25	0.05	0.35	0.65
9	**DM_EXIT**	0.26	0.03	0.35	0.63
10	**OBL_UNSATISFIED_OBLIGATION_EXCEPTION_ …**	0.35	0.10	0.18	0.63
11	**RCN_REDUNDANT_NULLCHECK_WOULD_HAVE …**	0.19	0.09	0.35	0.63
12	**MS_PKGPROTECT**	0.18	0.09	0.35	0.62
13	**LI_LAZY_INIT_STATIC**	0.20	0.06	0.35	0.61

Tabla 4.3 Tipos de alerta con mayor relevancia (probabilidad > 0.6).

Una vez que hemos identificado los tipos de alerta que probablemente perjudicarían al sistema, abrimos el archivo CSV generado anteriormente (véase figura 4.7) y, de forma manual, etiquetamos como accionables a cada uno de los VCA descritos por alguno de estos 13 tipos seleccionados, ignorando aquellos que ya han sido etiquetados por la heurística de etiquetado de alertas. La figura 4.8 muestra la forma en que dos VCA cuyo tipo es BC_UNCONFIRMED_CAST_OF_RETURN_VALUE son modificados de acuerdo a dicha heurística, manteniendo intacta o ignorando la accionabilidad del primero (línea 4) y etiquetando como accionable al segundo (línea 5). De manera conjunta, la tabla 4.4 muestra el detalle de las modificaciones realizadas a los archivos CSV generados para ambos proyectos: partiendo de las 304 y 402 alertas generadas para SAPCyTI y JDOM respectivamente, 55 alertas han sido reetiquetadas para SAPCyTI y 15 para JDOM como alertas relevantes, quedando un total de 152 alertas accionables y 152 alertas no accionables para SAPCyTI y 220 accionables y 182 no accionables para JDOM.

Por último, utilizamos la aplicación CSV2ARFF [46] para convertir los dos archivos CSV generados (los cuales contienen a los conjuntos de VCA de cada proyecto) al Formato de Archivo de Relación de Atributos (ARFF, *Attribute-Relation File Format*) utilizado por la herramienta WEKA [32, 22, 45]. La figura 4.9 muestra la forma en que realizamos dicha conversión: (1) la aplicación toma como entrada el archivo en formato CSV, (2) le indicamos que la primera fila del archivo contiene los nombres de los atributos de entrada y el nombre de la clase y (3) la aplicación genera un archivo ARFF.

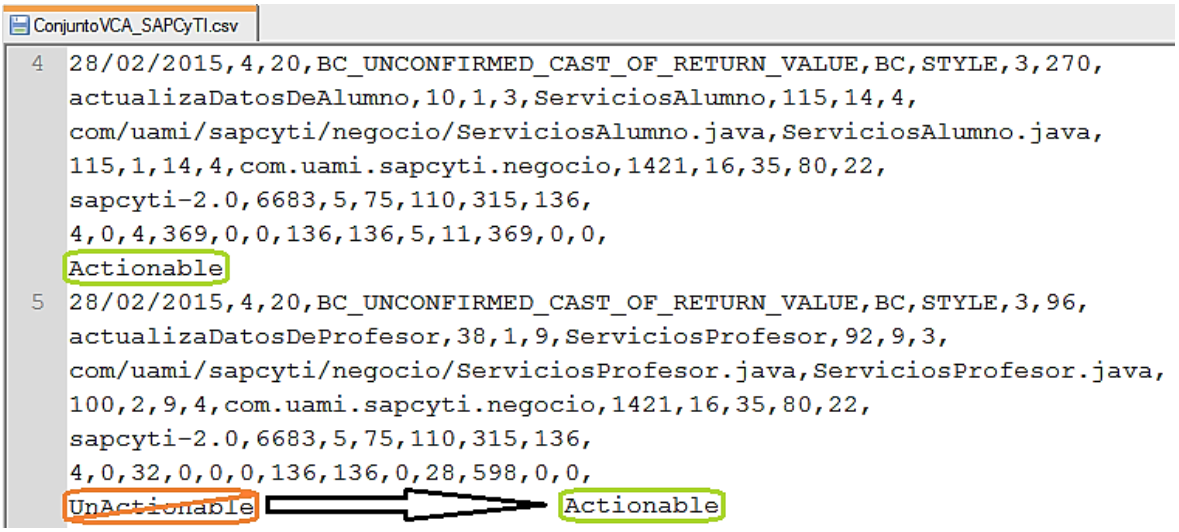


Figura 4.8 Modificación al archivo CSV generado luego del cálculo de CA (fragmento).

	SAPCyTI	JDOM
Tipos explorados	51	51
Tipos reetiquetados	13	8
Alertas totales	304	402
Alertas accionables	97	205
Alertas reetiquetadas	55	15
Alertas no accionables	152	182

Tabla 4.4 Información final de los conjuntos de VCA generados.

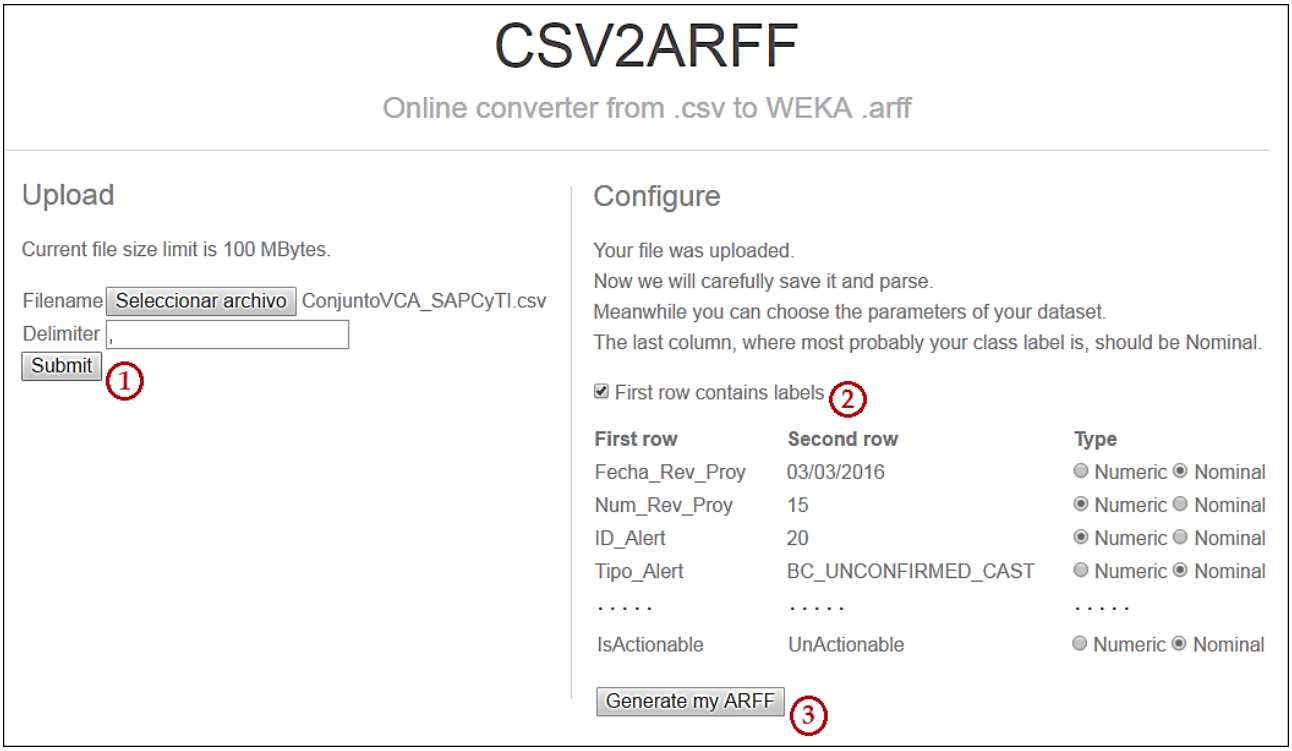


Figura 4.9 Proceso de conversión de un archivo CSV a un archivo ARFF.

4.3. Generación de modelos

Decimos que generamos un modelo cuando aplicamos un algoritmo de aprendizaje sobre un conjunto de datos. De acuerdo a la metodología señalada a lo largo de la sección 3.3, la etapa de generación de modelos se compone de diferentes actividades, las cuales pueden ser resumidas de la siguiente manera: basados en el conjunto de VCA antes generado (por ejemplo, de SAPCyTI), seleccionamos los subconjuntos de CA más relevantes para crear nuevos conjuntos de vectores y con ellos, construir y evaluar el desempeño de una serie de modelos candidatos de acuerdo a su exactitud, precisión y sensibilidad; el resultado será un conjunto de los 12 mejores modelos de clasificación de nuevas alertas para cada uno de los proyectos en cuestión. Utilizaremos la interfaz de usuario proporcionada por la herramienta WEKA 3.9.1 [30, 32, 33] para efectuar la selección de atributos (es decir, para obtener los subconjuntos de CA más relevantes) y la interfaz de aplicaciones para la construcción y evaluación de modelos [45].

A través de esta sección, presentaremos el resultado final de la evaluación de nuestra propuesta para dos configuraciones del experimento, las cuales son: (1) entrenamiento y evaluación de modelos internos al proyecto y (2) entrenamiento de modelos externos al proyecto y su evaluación sobre el conjunto de VCA de dicho proyecto. A continuación, proporcionaremos sólo un resumen de los componentes utilizados a lo largo de nuestro experimento y el detalle de los resultados obtenidos en cada una de las configuraciones del mismo; las actividades de selección de CA más relevantes, construcción y evaluación de modelos candidatos y selección de los mejores modelos, aplican de igual manera para cada una de las configuraciones del experimento y sólo serán descritas una sóla vez.

Selección de características de alerta.

WEKA contiene un conjunto de algoritmos de selección de atributos, el cual nos permite generar subconjuntos de aquellas CA más relevantes en la accionabilidad de una alerta [32], para lograrlo, exploramos el mayor número posible de métodos de evaluación y estrategias de búsqueda mediante la interfaz de usuario que dicha herramienta ofrece. La figura 4.10 muestra el proceso que hemos seguido para extraer cada uno de los subconjuntos de CA más relevantes, el cual es descrito de la siguiente manera: (1) abrimos el explorador de la aplicación y, en la pestaña de preprocesamiento, cargamos el archivo ARFF generado anteriormente (véase sección anterior), (2) vamos a la pestaña de selección de atributos, (3) elegimos el método de evaluación y (4) la estrategia de búsqueda y (5) ejecutamos la selección de CA utilizando la configuración por defecto asignada por la herramienta.

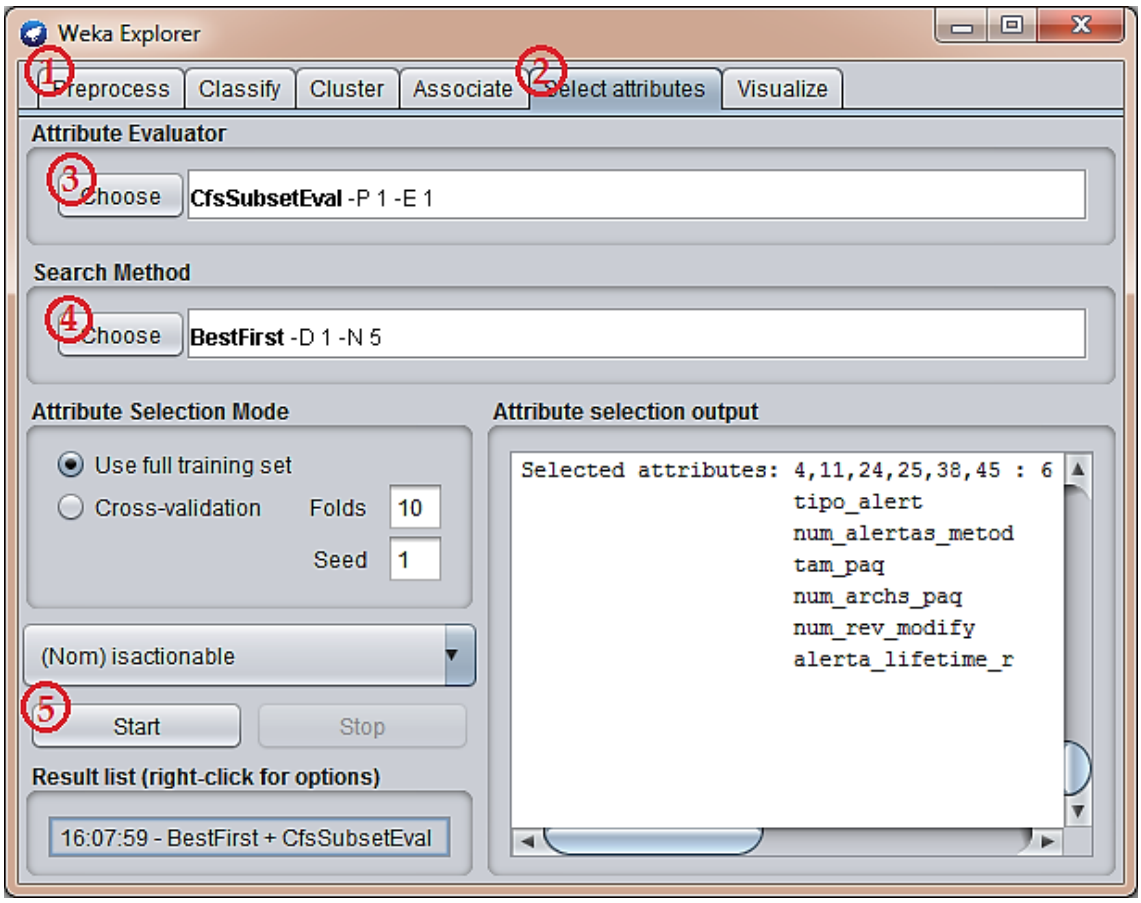


Figura 4.10 Proceso de selección de subconjuntos de CA más relevantes.

Una vez realizado lo anterior, sólo han prevalecido aquellas combinaciones conseguidas por tan sólo 5 métodos de evaluación (CfsSubsetEval, ClassifierSubsetEval, ConsistencySubsetEval, FilteredSubsetEval y WrapperSubsetEval) y 5 estrategias de búsqueda (BestFirst, GeneticSearch, GreedyStepwise, RaceSearch y RankSearch) [32, 22] ya que las demás combinaciones, o siempre definen subconjuntos de CA con un alto número de características (mayor que 16), o no conseguir concluir su ejecución de manera satisfactoria. De los 25 subconjuntos de CA conseguidos por cada conjunto de VCA generado, algunos de ellos han sido ignorados debido a un par de razones: o el subconjunto generado es el duplicado de otro ya existente, o bien, el subconjunto generado está compuesto por menos de 3 o más de 16 características. Cabe mencionar que, la descripción de cada uno de estos métodos de evaluación y estrategias de búsqueda es mostrada en la tabla A.2 y tabla A.3 respectivamente, la cual puede ser encontrada en el apéndice A al final de este documento.

Adicionalmente, WEKA nos brinda la posibilidad de sobrescribir las instancias de un archivo ARFF cuyos atributos de entrada sean tan sólo un subconjunto del total de los atributos declarados en el archivo original [32, 45]. Hacemos uso de esta característica y, por cada uno de los 25 subconjuntos de CA encontrados, abrimos el archivo ARFF que almacena al conjunto de VCA, seleccionamos las

características del subconjunto identificadas (además de la clase de la alerta) y guardamos un nuevo conjunto de VCA, etiquetándolo con la combinación método-estrategia utilizada para generarlo.

Construcción y evaluación de modelos.

Hasta este momento, la evaluación de nuestra propuesta estaría por concluir una vez que se construyan y se evalúen diversos modelos candidatos capaces clasificar nuevas alertas con ayuda de la herramienta WEKA, la cual cuenta con la implementación de múltiples algoritmos de aprendizaje que nos permiten generar dichos modelos. De manera similar a la selección de CA, hacemos uso del mayor número posible de estos algoritmos al explorar un total de 60 algoritmos distribuidos en 6 categorías (Bayesianos, Funciones, Perezosos, Metaclasificadores, Reglas y Árboles [32, 22]), lo cual implica la construcción de 60 modelos por cada subconjunto de CA identificado. Como vimos antes, la selección de características de alerta fue realizada de manera manual debido a la simplicidad y al poco tiempo que toma realizarla, sin embargo, esta misma condición no aplica para la construcción de modelos candidatos y, por esta razón, utilizamos la interfaz de programación de WEKA [30, 45] para desarrollar un programa encargado de automatizar dicha actividad.

Para hablar acerca del funcionamiento de nuestra implementación (tanto para modelos internos como para modelos externos), recordemos que una metodología *training and testing* es aquella donde una porción de los datos es utilizada para construir el modelo y la parte restante para evaluar su desempeño. De manera particular, para la generación de modelos internos a SAPCyTI, una vez que el programa entra en ejecución, este toma como entrada una lista de 60 algoritmos de aprendizaje y la ruta del directorio donde fueron almacenados cada uno de los subconjuntos de CA generados para SAPCyTI y comienza con la construcción de modelos, en la cual, para cada subconjunto de CA y para cada algoritmo de aprendizaje, se construye y se evalúa un modelo candidato utilizando validación cruzada. La información acerca del desempeño del modelo (exactitud, precisión y sensibilidad) es almacenada dentro de un repositorio, el cual contiene la información de todos los modelos generados hasta el momento. Como se puede ver, la característica de este tipo de modelos (internos) es que son entrenados y evaluados con base en el mismo archivo ARFF.

Para el caso de la construcción y evaluación de modelos externos, además del directorio que almacena los subconjuntos de entrenamiento (entrenando con JDOM), el programa debe contar con la ruta del directorio que almacena los subconjuntos de prueba para realizar el proceso antes mencionado (evaluando con SAPCyTI). Insistimos, ambas listas de subconjuntos de CA provienen de proyectos ajenos entre sí.

En ambos casos, el programa extrae del repositorio la información de todos los modelos generados una vez que se han agotado todos los archivos ARFF de entrenamiento y que se han explorado todos los algoritmos de aprendizaje. Para cada modelo candidato generado, el nombre del archivo ARFF utilizado, el nombre del algoritmo de aprendizaje explorado, la exactitud, la precisión y la sensibilidad, es almacenada dentro de un archivo CSV (modelsSAPCyTIvsSAPCyTI_FULL_R.csv) y cuyo ejemplo se muestra en la figura 4.11.

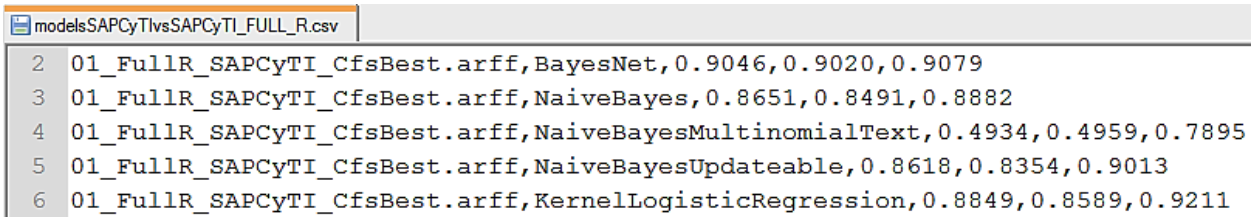


Figura 4.11 Archivo CSV de modelos candidatos generado (fragmento).

Selección de mejores modelos.

Como ya hemos dicho antes, los subconjuntos de CA se obtienen de la combinación de 5 métodos de evaluación y 5 estrategias de búsqueda, lo que provoca la creación de más de 1500 modelos por proyecto, tomando en cuenta que se generan 60 modelos candidatos por cada combinación. Supongamos que en lugar de generar 25 subconjuntos de CA tan sólo generamos 5 y, en lugar de construir 60 modelos sólo construimos 10, esto daría un total de 50 modelos candidatos. En la figura 4.12 podemos observar esta suposición con el fin de explicar nuestro proceso de selección, los valores reportados (filas) corresponden a la exactitud lograda por cada algoritmo de aprendizaje aplicado a cada uno de los subconjuntos de CA generados (columnas Comb1, Comb2, Comb3, Comb4 y Comb5), también, reportamos el promedio obtenido por cada modelo y combinación. Es importante mencionar que, la información contenida en dicha tabla proviene del archivo CSV generado en la sección anterior.

Una vez que se tiene dicha tabla (véase figura 4.12), seleccionamos las dos combinaciones cuya exactitud promedio sea mayor (Comb3 y Comb5) y de estas dos combinaciones, a los 5 modelos cuya exactitud promedio también sea mayor (Regresión Logística Lineal, k-NN, Listas de Decisión, Árbol AD y Árbol LM). Dicho lo anterior, obtenemos el desempeño promedio logrado por cada uno de los algoritmos y nos quedamos con los 5 mejores, es así como realizamos la selección del conjunto de mejores modelos de clasificación.

Por último, y fuera de esta suposición, debemos mencionar que hemos seleccionado a los 5 mejores subconjuntos de CA y a los 12 mejores modelos de clasificación para cada uno de los proyectos estudiados con el fin de proporcionar a los desarrolladores una visión general acerca de cuáles son los

métodos de evaluación, las estrategias de búsqueda y los algoritmos de aprendizaje que mejor se adaptan a las necesidades planteadas dentro de este estudio, tanto para la identificación y selección de las CA más relevantes, como para la construcción y evaluación de modelos de clasificación de nuevas alertas. A continuación, reportamos los resultados obtenidos para los dos tipos de modelos generados (internos y externos).

①

Modelo basado en	Exactitud					Promedio
	Comb1	Comb2	Comb3	Comb4	Comb5	
Bayes Ingenuo	0.865	0.898	0.865	0.901	0.875	0.881
Regresión Log. Lineal	0.938	0.974	0.951	0.941	0.974	0.956
k-NN	0.918	0.882	0.941	0.908	0.921	0.914
K*	0.901	0.826	0.947	0.803	0.891	0.874
Tablas de Decisión	0.809	0.852	0.796	0.803	0.806	0.813
Listas de Decisión	0.928	0.934	0.964	0.928	0.928	0.936
Árbol AD	0.951	0.957	0.974	0.970	0.967	0.964
Árbol C4.5	0.928	0.918	0.832	0.898	0.928	0.901
Árbol LM	0.941	0.970	0.957	0.951	0.980	0.960
Bosques Aleatorios	0.911	0.895	0.905	0.898	0.911	0.904
Promedio	0.909	0.911	0.913	0.900	0.918	0.910

②

Modelo basado en	Exactitud		Promedio
	Comb3	Comb5	
Bayes Ingenuo	0.865	0.875	0.870
Regresión Log. Lineal	0.951	0.974	0.963
k-NN	0.941	0.921	0.931
K*	0.947	0.891	0.919
Tablas de Decisión	0.796	0.806	0.801
Listas de Decisión	0.964	0.928	0.946
Árbol AD	0.974	0.967	0.971
Árbol C4.5	0.832	0.928	0.880
Árbol LM	0.957	0.980	0.969
Bosques Aleatorios	0.905	0.911	0.908
Promedio	0.913	0.918	0.916

③

Modelo basado en	Exactitud		Promedio
	Comb3	Comb5	
Regresión Log. Lineal	0.951	0.974	0.963
k-NN	0.941	0.921	0.931
Listas de Decisión	0.964	0.928	0.946
Árbol AD	0.974	0.967	0.971
Árbol LM	0.957	0.980	0.969
Promedio	0.957	0.954	0.956

Figura 4.12 Proceso de selección de mejores modelos (fragmento).

4.3.1. Resultados de modelos internos

La tabla 4.5 muestra la suma de las ocurrencias de cada CA en cada uno de los subconjuntos de CA generados, en ella podemos observar aquellas características con mayor número de apariciones (por ejemplo, el tiempo de vida de la alerta, el número de versión de modificación del archivo y, en el caso de SAPCyTI, el tipo de la alerta) y por el contrario, aquellas que nunca fueron seleccionadas dentro de los subconjuntos de CA (por ejemplo, el tamaño del archivo, el número de paquetes en el proyecto y el número de versión de creación del archivo). Es importante mencionar que no hemos contabilizado las características que han sido seleccionadas por subconjuntos compuestos por menos de 3 y más de 16

CA, recordemos que son subconjuntos inválidos dentro de nuestro experimento, por lo que no tienen relevancia para los resultados finales.

No.	Característica de alerta	SAPC.	JDOM	No.	Característica de alerta	SAPC.	JDOM
1	Fecha lanzam. de la versión	1	0	**24**	Tamaño del paquete	6	2
2	No. de versión del proyecto	0	2	**25**	No. de archivos en el paquete	4	1
3	ID de la alerta	3	1	**26**	No. de clases en el paquete	3	4
4	Tipo de la alerta	9	0	**27**	No. de métodos en el paquete	4	1
5	Patrón de la alerta	2	0	**28**	No. de alertas en el paquete	2	1
6	Categoría de la alerta	3	0	**29**	Nombre del proyecto	1	0
7	Prioridad de la alerta	3	2	**30**	Tamaño del proyecto	0	1
8	No. de línea de la alerta	4	2	**31**	No. de paquetes en el proy.	0	0
9	Nombre del método	6	0	**32**	No. de archivos en el proy.	1	1
10	Tamaño del método	4	0	**33**	No. de clases en el proyecto	0	2
11	No. de alertas en el método	6	0	**34**	No. de métodos en el proy.	1	0
12	Complejidad ciclomática	0	0	**35**	No. de alertas en el proyecto	0	1
13	Nombre de la clase	1	0	**36**	No. versión creación del arch.	0	0
14	Tamaño de la clase	5	2	**37**	No. versión elim. del archivo	0	1
15	No. de métodos en la clase	2	2	**38**	No. versión de mod. del arch.	8	4
16	No. de alertas en la clase	0	4	**39**	Ranciedad del archivo	2	2
17	Ruta del archivo	1	0	**40**	Total alertas acc. en la versión	0	0
18	Nombre del archivo	0	1	**41**	No. alertas acc. en la versión	2	3
19	Tamaño del archivo	0	0	**42**	Total alertas no acc. en la versión	0	2
20	No. de clases en el archivo	1	2	**43**	No. alertas no acc. en la versión	0	2
21	No. de métodos en el archivo	0	0	**44**	No. de mods. a la alerta	1	2
22	No. de alertas en el archivo	1	0	**45**	Tiempo de vida de la alerta	6	8
23	Nombre del paquete	0	1	**46**	Edad del archivo	2	1

Tabla 4.5 Características de alerta (CA) seleccionadas para modelos internos.

Para el caso de SAPCyTI, hemos seleccionado 11 subconjuntos de CA, 7 han sido omitidos por ser subconjuntos inválidos, 4 por no concluir la selección de manera adecuada y 7 por ser subconjuntos duplicados. Para JDOM, elegimos 9 subconjuntos de CA, 4 han sido omitidos por ser subconjuntos inválidos, 4 por no concluir adecuadamente su ejecución y 8 por ser subconjuntos duplicados. La tabla 4.6 y la tabla 4.7 muestran las combinaciones de los métodos de evaluación y las estrategias de búsqueda utilizadas para generar dichos subconjuntos de CA para SAPCyTI y JDOM respectivamente, los números separados por comas describen las características que componen a cada subconjunto de acuerdo a la estructura del VCA detallada en la sección 4.2 (véase figura 4.6).

Por otro lado, la tabla 4.8 muestra la información acerca de los modelos candidatos generados para cada uno de los proyectos en cuestión. Hemos generado 60 modelos candidatos por cada subconjunto de CA, lo cual implica un total de 660 y 540 modelos generados para SAPCyTI y JDOM

respectivamente. La descripción de cada uno de los algoritmos de aprendizaje utilizados para generar dichos modelos puede ser consultada en la tabla A.4 al final de este documento (véase apéndice A).

Combinación utilizada			Subconjunto generado
No.	**Método de evaluación**	**Estrategia de búsq.**	
1	**CfsSubsetEval**	**BestFirst**	4,11,24,25,38,45
2		**GeneticSearch**	4,5,6,9,10,11,14,15,17,24,25,27,32,34,38,45
3		**RankSearch**	4,8,9,11,14,24,25,26,27,28,38,45
4	**ClassifierSubsetEval**	**BestFirst**	3,5,7,8,9,14,38,41,44
5		**GreedyStepwise**	7,8,9
6		**RaceSearch**	4,38,39
7		**RankSearch**	4,8,9,11,14,24,25,26,27,28,38,45,46
8	**ConsistencySubsetEval**	**BestFirst**	4,9,13
9		**GeneticSearch**	3,4,9,10,13,14,20,22,24,26,27,29,41,46
10	**FilteredSubsetEval**	**BestFirst**	4,11,24,38,45
11	**WrapperSubsetEval**	**BestFirst**	1,3,4,6,7,10,11,15,38,39,45

Tabla 4.6 Subconjuntos de CA seleccionados para SAPCyTI.

Combinación utilizada			Subconjunto generado
No.	**Método de evaluación**	**Estrategia de búsq.**	
1	**CfsSubsetEval**	**BestFirst**	16,38,41,45
2		**GeneticSearch**	16,23,26,38,41,43,45
3	**ClassifierSubsetEval**	**BestFirst**	7,8,15,16,24,26,45
4		**GeneticSearch**	2,8,14,27,30,32,33,39,44,45,46
5		**GreedyStepwise**	7,24,26,45
6		**RaceSearch**	25,38,42,45
7		**RankSearch**	14,15,16,20,26,28,35,38,39,41,42,43,45
8	**ConsistencySubsetEval**	**GeneticSearch**	3,18,37
9	**WrapperSubsetEval**	**BestFirst**	2,20,33,44,45

Tabla 4.7 Subconjuntos de CA seleccionados para JDOM.

	SAPCyTI	**JDOM**
Algoritmos utilizados	60	60
Subconjuntos seleccionados	11	9
Modelos generados	660	540

Tabla 4.8 Resumen de la generación de modelos candidatos.

Como ya hemos dicho antes, debemos seleccionar a los 12 mejores modelos de clasificación de cada proyecto con base en su desempeño dentro de los 5 mejores subconjuntos de CA. Para el caso de SAPCyTI, hemos seleccionado las combinaciones número 1, 2, 6, 7 y 11 (véase tabla 4.6) y los modelos basados en los algoritmos Árbol AD, Árbol LAD, Árbol LM, Regresión Logística Lineal, Real AdaBoost, Árbol NB, Reglas Ripple-Down, Listas de Decisión, Perceptrón Multicapa Optimizado, k-NN, Clasificador Ordinal y Bosques Aleatorios [22, 32]. El promedio del desempeño logrado por cada uno de ellos (exactitud, precisión y sensibilidad) es mostrado en tabla 4.9.

Modelo basado en	Exactitud	Precisión	Sensib.
Árbol AD	96.4%	98.5%	94.2%
Árbol LAD	96.1%	97.7%	94.3%
Árbol LM	96.0%	97.6%	94.3%
Regresión Logística Lineal	95.5%	96.9%	94.1%
Real AdaBoost	95.3%	97.2%	93.4%
Árbol NB	94.3%	95.8%	92.6%
Reglas Ripple-Down	93.9%	98.6%	89.2%
Listas de Decisión	93.6%	98.8%	88.3%
Percep. Mult. Optimizado	93.3%	93.3%	93.3%
k-NN	91.4%	90.3%	92.8%
Clasificador Ordinal	90.7%	92.0%	89.2%
Bosques Aleatorios	90.4%	90.1%	90.8%
Promedio	**93.9%**	**95.6%**	**92.2%**

Tabla 4.9 Desempeño de los 12 mejores modelos para SAPCyTI.

Para JDOM, seleccionamos las combinaciones número 1, 2, 6, 7 y 9 (véase tabla 4.7) y los modelos basados en los algoritmos Bosques Aleatorios, Comité Aleatorio, K*, Árbol Best-First, Clasificador Ordinal, Árbol LM, Empaquetado, Árbol LAD, k-NN, Listas de Decisión, Árbol Aleatorio y Árbol AD [22, 32]. La tabla 4.10 muestra el desempeño logrado por cada uno de ellos.

Modelo basado en	Exactitud	Precisión	Sensib.
Bosques Aleatorios	93.9%	95.7%	93.0%
Comité Aleatorio	93.1%	94.3%	93.1%
K*	92.5%	94.2%	92.0%
Árbol Best-First	92.4%	94.1%	91.9%
Clasificador Ordinal	92.3%	94.8%	91.0%
Árbol LM	92.3%	95.2%	90.5%
Empaquetado	92.2%	94.4%	91.3%
Árbol LAD	92.2%	95.3%	90.3%
k-NN	92.0%	93.8%	91.5%
Listas de Decisión	91.8%	93.6%	91.4%
Árbol Aleatorio	91.7%	93.1%	91.7%
Árbol AD	91.5%	94.1%	90.2%
Promedio	**92.3%**	**94.4%**	**91.5%**

Tabla 4.10 Desempeño de los 12 mejores modelos para JDOM.

4.3.2. Resultados de modelos externos

La generación de modelos externos ocurre cuando entrenamos un algoritmo de aprendizaje con base en el conjunto de VCA de JDOM y lo evaluamos sobre el conjunto de VCA de SAPCyTI, esto también aplica de manera inversa (es decir, de SAPCyTI hacia JDOM). Para lograr esto, generamos nuevos archivos ARFF (uno por cada proyecto) a partir de los archivos utilizados en la generación de modelos internos. En estos nuevos archivos, las instancias que los componen eliminan 20 atributos o

CA, los cuales representan características de temporalidad, características del repositorio de código fuente y nombres de artefactos.

Dicho lo anterior, para cada proyecto obtenemos nuevos subconjuntos de CA y, de manera similar a la generación de modelos internos, obtenemos el número total de ocurrencias sobre los subconjuntos generados. Sintetizamos el resultado de esta actividad a través de la tabla 4.11, en la cual se pueden observar las características comunes a ambos proyectos (por ejemplo, el número de métodos en la clase y el número de archivos en el proyecto), aquellas que difieren (por ejemplo, el tamaño del proyecto y la categoría de la alerta) y aquellas que nunca son seleccionadas (por ejemplo, el número de métodos en el archivo).

No.	Característica de alerta	SAPC.	JDOM	No.	Característica de alerta	SAPC.	JDOM
4	Tipo de la alerta	14	6	**21**	No. de métodos en el archivo	0	0
5	Patrón de la alerta	5	2	**22**	No. de alertas en el archivo	3	1
6	Categoría de la alerta	3	0	**24**	Tamaño del paquete	4	3
7	Prioridad de la alerta	4	0	**25**	No. de archivos en el paquete	7	2
8	No. de línea de la alerta	7	5	**26**	No. de clases en el paquete	5	6
10	Tamaño del método	7	3	**27**	No. de métodos en el paquete	2	11
11	No. de alertas en el método	7	1	**28**	No. de alertas en el paquete	3	4
12	Complejidad ciclomática	0	3	**30**	Tamaño del proyecto	0	4
14	Tamaño de la clase	4	3	**31**	No. de paquetes en el proy.	1	1
15	No. de métodos en la clase	3	2	**32**	No. de archivos en el proyecto	4	4
16	No. de alertas en la clase	0	5	**33**	No. de clases en el proyecto	0	1
19	Tamaño del archivo	3	1	**34**	No. de métodos en el proy.	0	5
20	No. de clases en el archivo	3	4	**35**	No. de alertas en el proyecto	5	6

Tabla 4.11 Características de alerta (CA) seleccionadas para modelos externos.

Basados en el conjunto de VCA de SAPCyTI (archivo ARFF con instancias compuestas de 26 atributos), hemos seleccionado 14 subconjuntos de CA, 4 son omitidos por no crearse de manera correcta y 7 por ser subconjuntos duplicados. Para el caso de JDOM, elegimos 12 subconjuntos, 5 son descartados por invalidez y 8 por duplicación. La tabla 4.12 y la tabla 4.13 muestran las combinaciones de los métodos y estrategias utilizadas para generar dichos subconjuntos de CA para SAPCyTI y JDOM respectivamente. En total, hemos generado 840 y 720 modelos para SAPCyTI y JDOM respectivamente y la información acerca de la generación de dichos modelos es mostrada en la tabla 4.14.

Al igual que con los modelos internos, deseamos seleccionar a los 12 mejores modelos de clasificación utilizando la siguiente idea: un modelo creado con las alertas de JDOM será probado con las alertas de SAPCyTI y viceversa. Para el caso de SAPCyTI, hemos seleccionado las combinaciones 4, 5, 6, 10 y 14 y los modelos creados mediante los algoritmos k-NN, Clasificación Vía Regresión,

Regresión Logística Lineal, Árbol de Hoeffding, Regresión Logística de Kernel, Umbral de Punto Medio, Regresión Logística Multinomial, Clasificador Multiclase, Árbol LM, Tablas de Decisión, Árbol Aleatorio y Perceptrón Multicapa Optimizado [22, 32]. Para JDOM, elegimos las combinaciones 5, 6, 7, 8 y 9 y los modelos Empaquetado, Clasificación Vía Regresión, Bosques Aleatorios, Árbol REP, Comité Aleatorio, Árbol Aleatorio, Árbol de Hoeffding, Clasificador Multiclase, Umbral de Punto Medio, k-NN, Regresión Logística Lineal y Perceptrón Multicapa Optimizado [22, 32]. La tabla 4.15 y la tabla 4.16 muestran el promedio del desempeño logrado por los 12 mejores modelos externos a JDOM y a SAPCyTI respectivamente.

Combinación utilizada			**Subconjunto generado**
No.	**Método de evaluación**	**Estrategia de búsq.**	
1	**CfsSubsetEval**	**BestFirst**	1,7,16,17
2		**GeneticSearch**	1,2,5,6,7,17,18
3		**RankSearch**	1,2,5,6,7,9,16,17,18,19,20
4	**ClassifierSubsetEval**	**BestFirst**	1,5,12,15,17,23,26
5		**GeneticSearch**	1,2,5,6,9,10,12,15,17,26
6		**GreedyStepwise**	1,5,15,17,23
7		**RaceSearch**	1,3,4,9,20
8		**RankSearch**	1,5,7,9,16,17,18,19,20
9	**ConsistencySubsetEval**	**BestFirst**	1,4,6,10,23
10		**GeneticSearch**	1,3,4,6,10,12,23
11	**FilteredSubsetEval**	**BestFirst**	1,7,16
12	**WrapperSubsetEval**	**BestFirst**	1,2,5,6,7,13,18,26
13		**GeneticSearch**	1,2,3,4,7,13,18,22,26
14		**GreedyStepwise**	1,6,13,26

Tabla 4.12 Subconjuntos CA seleccionados para SAPCyTI (parcial).

Combinación utilizada			**Subconjunto generado**
No.	**Método de evaluación**	**Estrategia de búsq.**	
1	**CfsSubsetEval**	**BestFirst**	5,11,18,19,20,25,26
2		**GeneticSearch**	5,11,18,19,20,26
3		**RankSearch**	5,9,10,11,13,15,18,19,20,25,26
4	**ClassifierSubsetEval**	**BestFirst**	1,2,8,10,11,13,19,21,23
5		**GeneticSearch**	1,2,6,9,12,13,16,21,23
6		**GreedyStepwise**	1,8,13,19,23
7		**RaceSearch**	1,17,19,23
8	**ConsistencySubsetEval**	**BestFirst**	1,17,19,25,26
9		**GeneticSearch**	1,8,9,19,25,26
10	**WrapperSubsetEval**	**BestFirst**	5,6,16,18,19,21,25
11		**GeneticSearch**	5,7,11,16,18,19,20,21,22,24,26
12		**GreedyStepwise**	6,18,19

Tabla 4.13 Subconjuntos de CA seleccionados para JDOM (parcial).

	SAPCyTI	JDOM
Algoritmos utilizados	60	60
Subconjuntos seleccionados	14	12
Modelos generados	840	720

Tabla 4.14 Resumen de la generación de modelos candidatos (parcial).

Modelo basado en	Exactitud	Precisión	Sensib.
k-NN	61.1%	64.6%	66.1%
Clasificación Vía Regresión	57.5%	58.3%	81.1%
Regresión Logística Lineal	55.3%	55.5%	93.6%
Árbol de Hoeffding	55.2%	55.5%	91.8%
Regresión Log. de Kernel	55.2%	55.6%	90.5%
Umbral de Punto Medio	55.2%	55.7%	87.9%
Regresión Log. Multinomial	55.1%	55.7%	87.5%
Clasificador Multiclase	55.1%	55.7%	87.5%
Árbol LM	54.8%	55.3%	92.0%
Tablas de Decisión	54.7%	56.7%	73.2%
Árbol Aleatorio	54.6%	56.9%	70.2%
Percep. Mult. Optimizado	52.2%	53.5%	89.4%
Promedio	**55.5%**	**56.6%**	**84.2%**

Tabla 4.15 Desempeño de los 12 mejores modelos externos a JDOM.

Modelo basado en	Exactitud	Precisión	Sensib.
Empaquetado	67.6%	63.8%	81.8%
Clasificación Vía Regresión	66.8%	76.1%	60.1%
Bosques Aleatorios	66.7%	72.3%	62.8%
Árbol REP	66.6%	63.4%	78.3%
Comité Aleatorio	66.1%	71.2%	58.2%
Árbol Aleatorio	65.0%	62.7%	78.4%
Árbol de Hoeffding	64.7%	61.6%	80.7%
Clasificador Multiclase	64.4%	61.4%	78.4%
Umbral de Punto Medio	64.3%	61.7%	76.7%
k-NN	63.3%	65.9%	58.0%
Regresión Logística Lineal	61.7%	59.3%	74.7%
Percep. Mult. Optimizado	54.2%	53.7%	78.7%
Promedio	**64.3%**	**64.4%**	**72.2%**

Tabla 4.16 Desempeño de los 12 mejores modelos externos a SAPCyTI.

5. ANÁLISIS DE RESULTADOS

A lo largo de los capítulos 3 y 4, nos enfocamos en diseñar y evaluar una nueva Técnica de Identificación de Alertas Accionables (AAIT, *Actionable Alert Identification Technique*) con el fin de generar un conjunto de modelos de clasificación de nuevas alertas (ya sean internos o externos al proyecto) para dos proyectos de software actualmente en uso (SAPCyTI [47] y JDOM [34]). Con base en los resultados obtenidos en cada una de las configuraciones propuestas en nuestro experimento, durante este capítulo utilizaremos dichos resultados para analizar sus propiedades y sus relaciones, además de analizar las posibles implicaciones (ventajas y desventajas) que conllevaría ir por uno u otro camino. Dividiremos este capítulo en dos secciones, las cuales son:

1. **Análisis de modelos.** Aquí, haremos un breve análisis sobre las características de los proyectos estudiados y de los conjuntos de Vectores de Características de Alerta (VCA) generados con el fin de profundizar en los resultados obtenidos durante la etapa de generación de modelos (subconjuntos de características y mejores modelos seleccionados).
2. **Análisis de propuesta**. En ella, analizaremos el grado de fiabilidad que tendría la incorporación de nuestra AAIT dentro del Proceso de Desarrollo de Software (PDS) a través de su impacto en el número de defectos descubiertos y su costo de reparación una vez que ha concluido la fase de codificación.

5.1. Análisis de modelos

Como ya hemos mencionado antes (véase sección 4.1), elegimos a JDOM para la construcción de nuestros modelos externos debido a que cuenta con características similares a SAPCyTI, al menos de manera cuantitativa (por ejemplo, el tamaño, el número de archivos y el número de clases). Lo anterior, nos hace creer en la posibilidad de exportar un modelo entrenado con las alertas de JDOM para aplicarlo al conjunto de alertas generadas en SAPCyTI.

Y hablando acerca de las alertas generadas para cada uno de los proyectos, ambos conjuntos de VCA también lucen prometedores en cuanto a números se refiere, tratándose de conjuntos de entrenamiento formados por 304 alertas (50% accionables contra 50% no accionables) para SAPCyTI y 402 alertas (55% accionables contra 45% no accionables) para JDOM (véase tabla 4.4); más adelante veremos que, a pesar de que dichos conjuntos se encuentran parcialmente equilibrados (mismo número

de alertas accionables y no accionables), esto no precisamente garantiza una similitud en el desempeño logrado por el conjunto de modelos internos y el conjunto de modelos externos.

Antes de comenzar el análisis de los resultados obtenidos por nuestros modelos internos, queremos mencionar algunas particularidades encontradas en los 13 tipos de alertas reetiquetados. Además de estar agrupados en 6 categorías, algunos de ellos realmente describen defectos relevantes dentro del código fuente (véanse tipos de alerta 1, 2, 3, 5, 7, 8, 9, 10 y 13 de la tabla 4.3) y algunos resultan ser de mediana trascendencia para la funcionalidad del sistema (véanse los tipos 4, 6, 11 y 12) desde la perspectiva del desarrollador, situación que nos coloca en dos caminos: o la heurística de reetiquetado de alertas puede ser mejorada para reflejar un alto porcentaje de defectos o alertas relevantes a los ojos del desarrollador, o bien, la visión del desarrollador se ve afectada por su nivel de experiencia (cuestión que lo lleva a omitir posibles alertas relevantes a la correcta funcionalidad del sistema).

5.1.1. Modelos internos

Durante la selección de características (véase sección 4.3) y de acuerdo a la experiencia de diferentes investigadores [7, 6, 25, 23, 24, 11, 19, 10, 9], hemos omitido aquellos subconjuntos con menos de 3 y más de 16 Características de Alerta (CA) con el fin de lograr el mejor desempeño posible del modelo generado. Nos apoyaremos en la figura 5.1 para analizar el comportamiento de cada una de las características seleccionadas dentro de los subconjuntos generados, dividiremos nuestro análisis en 3 tópicos distintos: CA sin relevancia o que nunca fueron seleccionadas, CA propias del proyecto y CA comunes a ambos proyectos.

Dentro de las características que no fueron incluidas en los subconjuntos generados, resulta interesante observar que la complejidad ciclomática no tuvo la relevancia esperada en la clasificación de alertas. Esto sugiere un resultado similar al obtenido por Heckman y Williams [6, 7], pero a diferencia de ellas, creemos que no se trata de una cuestión de una característica a nivel de método pues, en el caso de SAPCyTI, algunas características sí fueron elegidas (por ejemplo, el número de alertas en el método). Por el contrario, decimos que simplemente se trata de una cuestión de relevancia, la cual se ve disminuida al compararse con otras CA que cuentan con un mayor número de apariciones, caso similar al número de versión de creación del archivo, de donde son obtenidas características como la edad y la ranciedad del archivo. El total de alertas accionables en la versión, el número de paquetes en el proyecto, el número de métodos en el archivo y el tamaño del archivo tampoco formaron parte de los subconjuntos de CA.

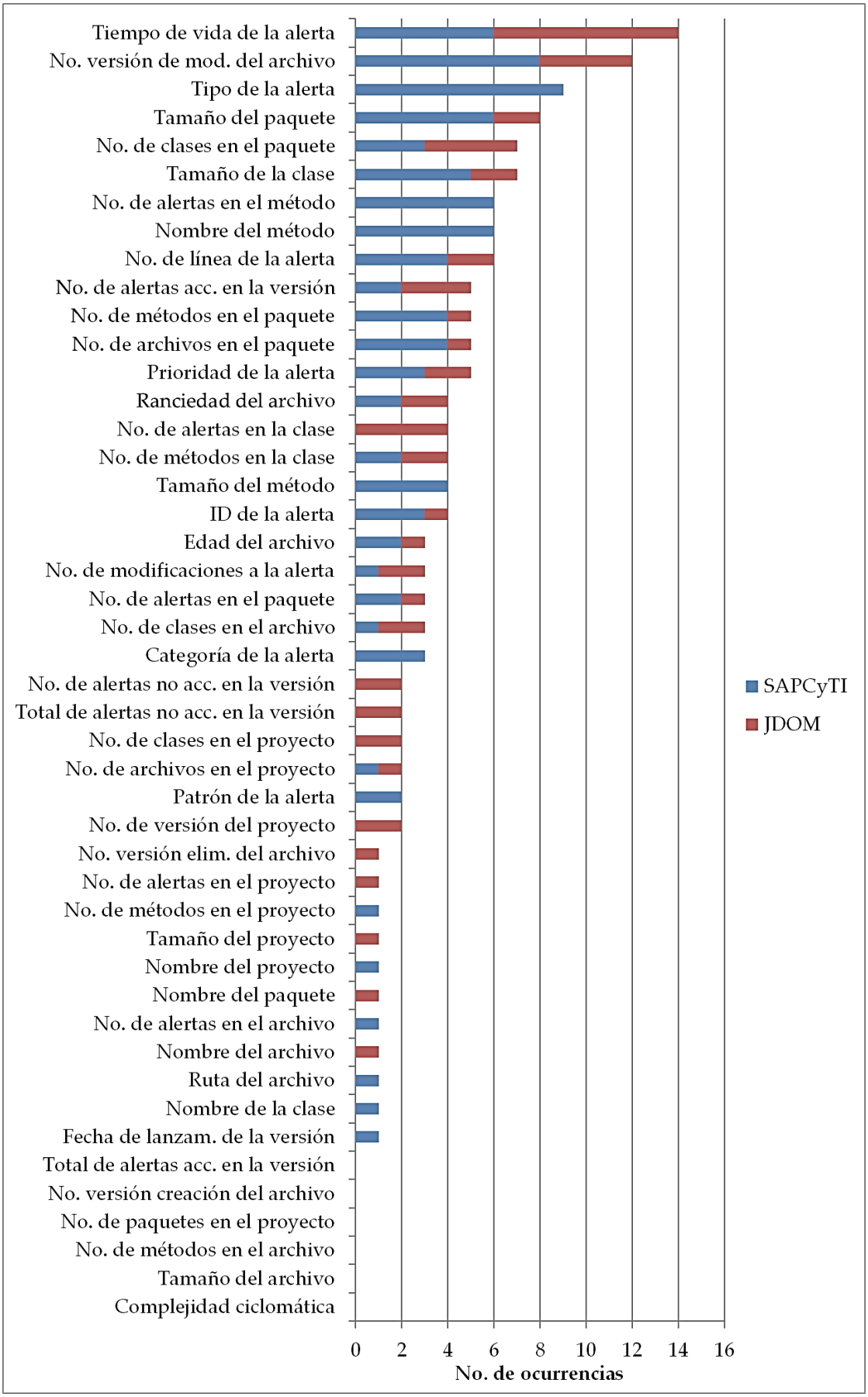


Figura 5.1 Número de ocurrencias de CA seleccionadas para modelos internos.

Por otro lado, CA que tienen que ver con la información extraída del repositorio de código fuente (por ejemplo, la fecha de lanzamiento y el número de versión del proyecto) y el nombre del artefacto (en todos sus niveles), son el ejemplo más claro de que existen características ajenas entre sí o propias de cada proyecto, lo cual guarda cierta congruencia al pensar que dos proyectos no nombrarán de igual forma a cada uno de sus artefactos y que no coincidirán en sus fechas de lanzamiento y sus números de versiones. CA como el tipo, patrón y categoría de la alerta, podrían sugerir que las alertas generadas también son exclusivas de cada proyecto, situación que no necesariamente es cierta, pues hemos observado que sí existen alertas del mismo tipo en los conjuntos de VCA de SAPCyTI y de JDOM. Algunas otras CA, principalmente a nivel de versión o de proyecto (por ejemplo, número de alertas no accionables en la versión, tamaño del proyecto, número de clases, métodos y alertas en el proyecto), presentan un nivel de relevancia muy bajo o casi nulo (es decir, un bajo número de ocurrencias).

Entre las 18 CA comunes a ambos proyectos se encuentra el tiempo de vida de la alerta, la cual es seleccionada de manera constante dentro de los subconjuntos y, en común acuerdo con distintos autores [6, 7, 11, 19], reafirmamos que esta característica es altamente relevante para la clasificación de alertas independientemente de su valor, es decir, tiempos de vida cortos no necesariamente describen a una alerta accionable. También, ambos proyectos comparten varias características que tienen que ver con el historial de archivos (número de versión de modificación, edad y ranciedad del archivo), suponemos que este tipo de CA son seleccionadas debido a que los cambios efectuados sobre un archivo se encuentran relacionados con la transición de una alerta no accionable a una accionable. Un par de características que han llamado nuestra atención son el número de línea y la prioridad de la alerta, las cuales reflejan la profundidad dentro del archivo en la que pudiera manifestarse un defecto y el impacto que esta tiene dentro del sistema respectivamente, cuestiones que parecen coherentes al momento de considerar a una alerta como relevante. Algunas otras características con mayor número de ocurrencias son seleccionadas a nivel de paquete, como el tamaño, el número de métodos y el número de alertas (véase figura 5.1).

En general, podemos destacar algunos puntos importantes de la selección de características: (1) el número de CA seleccionadas varió de 3 a 16 para SAPCyTI y de 3 a 13 para JDOM, (2) el número de subconjuntos de CA válidos fue de 11 para SAPCyTI y de 9 para JDOM, de los cuales (3) la estrategia BestFirst [32, 22], cuya descripción puede ser consultada en la tabla A.3 (véase apéndice A), logró combinarse con todos los métodos y (4) el método ClassifierSubsetEval (véase tabla A.2) [32, 22] logró combinarse con todas las estrategias para generar subconjuntos de CA válidos.

Por otro lado, analizamos y discutimos las propiedades y el comportamiento de los modelos seleccionados para cada proyecto en cuestión (véase figura 5.2 y figura 5.3). En ambos casos, los modelos se encuentran ordenados de manera ascendente de acuerdo a su valor de exactitud, además de incluir el promedio del desempeño logrado por dichos modelos en conjunto.

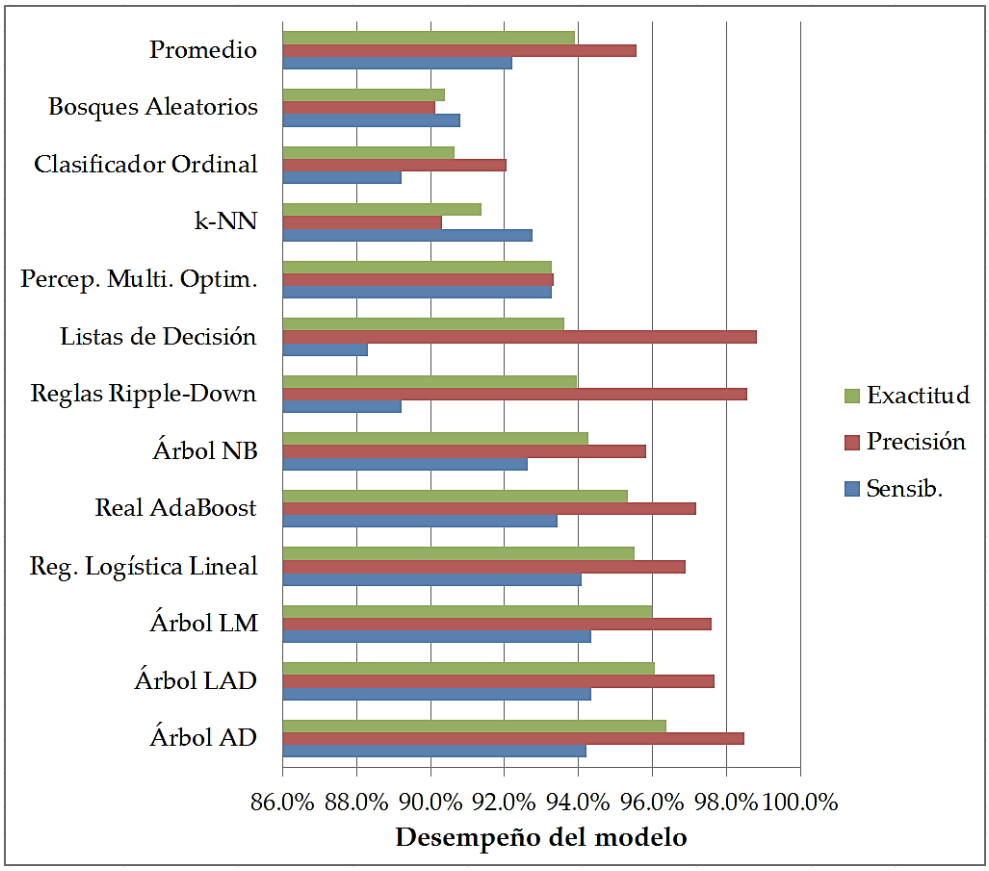


Figura 5.2 Desempeño de los mejores modelos internos para SAPCyTI.

Como se puede ver en la figura 5.2, los mejores modelos internos generados para SAPCyTI alcanzan exactitudes mayores al 90%, los cuales son valores aceptables e incluso superiores a los reportados en la literatura [6, 7, 8, 11, 19]. En particular, 5 de ellos están por encima del 95% y los 3 mejores están basados en algoritmos de árboles de decisión (Árbol AD, Árbol LAD y Árbol LM [32, 22] con 96.4%, 96.1% y 96.0% respectivamente), seguidos de los algoritmos Regresión Logística Lineal (95%) y Real AdaBoost (95.3%) [32, 22].

Respecto a la relación entre las medidas de desempeño de cada modelo, generalmente la precisión es mayor a la exactitud y esta a su vez es mayor a la sensibilidad (excepto con k-NN [32, 22]); dicho comportamiento es el que buscamos dentro de nuestro estudio debido a que deseamos disminuir la mayor cantidad posible de falsos positivos dentro de las alertas predichas como accionables, recordemos que, a mayor precisión menor número de falsos positivos. Sabemos que a menor

sensibilidad mayor número de falsos negativos y que esto implicaría la omisión de alertas realmente accionables, sin embargo, recordemos que uno de los objetivos de este proyecto es disminuir el número de defectos irrelevantes por reparar dentro del código fuente y, por este motivo, deseamos fijarnos en mejores precisiones en lugar de mejores sensibilidades.

Una de las cosas que más ha llamado nuestra atención es que la precisión y la exactitud se encuentran íntimamente relacionadas, es decir, conforme los valores de ambas decrecen, se alcanza un modelo completamente equilibrado, tal es el caso del modelo generado por el algoritmo Perceptrón Multicapa Optimizado [32, 22] con un desempeño promedio del 93.3%. Por último, el mejor de nuestros modelos para SAPCyTI puede ser logrado a través del subconjunto de CA generado por la combinación WrapperSubsetEval y BestFirst (véase tabla 4.6) [32, 22], dicho modelo se basa en el algoritmo Árbol LM [32, 22] y logra un desempeño del 98.0%, 98.7% y 97.4% para la exactitud, precisión y sensibilidad respectivamente (véase figura 5.2).

Al igual que el desempeño logrado por los modelos de SAPCyTI, los mejores modelos de JDOM también alcanzan exactitudes superiores al 90% (véase figura 5.3). Dentro de los 5 modelos situados en las primeras posiciones, se encuentran 2 modelos basados en árboles de decisión (Bosques Aleatorios y Árbol Best-First [32, 22] con el 93.9% y 92.4% respectivamente) y 3 modelos basados en los algoritmos Comité Aleatorio (93.1%), K* (92.5%) y Clasificador Ordinal (92.3%) [32, 22], de los cuales el mayor desempeño ha sido obtenido por el algoritmo Bosques Aleatorios [32, 22].

Aquí, la precisión siempre es mayor a la exactitud y, la exactitud siempre es mayor a la sensibilidad, esto quiere decir que (a diferencia de SAPCyTI) ningún modelo generado para JDOM alcanza un equilibrio en sus 3 medidas de desempeño. Sin embargo, este resultado se encuentra lejos de ser un comportamiento negativo y sin problema alguno conduciría a disminuir el número de falsos positivos.

Adicionalmente, el mejor modelo para JDOM se obtiene de combinar ClassifierSubsetEval y RankSearch [32, 22] con el algoritmo Árbol AD [32, 22], logrando un desempeño del 96.3%, 97.2% y 95.9% para la exactitud, precisión y sensibilidad. Curiosamente, dicho modelo cuenta con el menor promedio de desempeño de todo el conjunto, lo cual plantea la idea de que existen modelos que necesitan una gran cantidad de información para lograr un alto desempeño y, si observamos la combinación utilizada (véase tabla 4.7), Árbol AD [32, 22] logra dicho desempeño apoyado de un subconjunto de 13 CA.

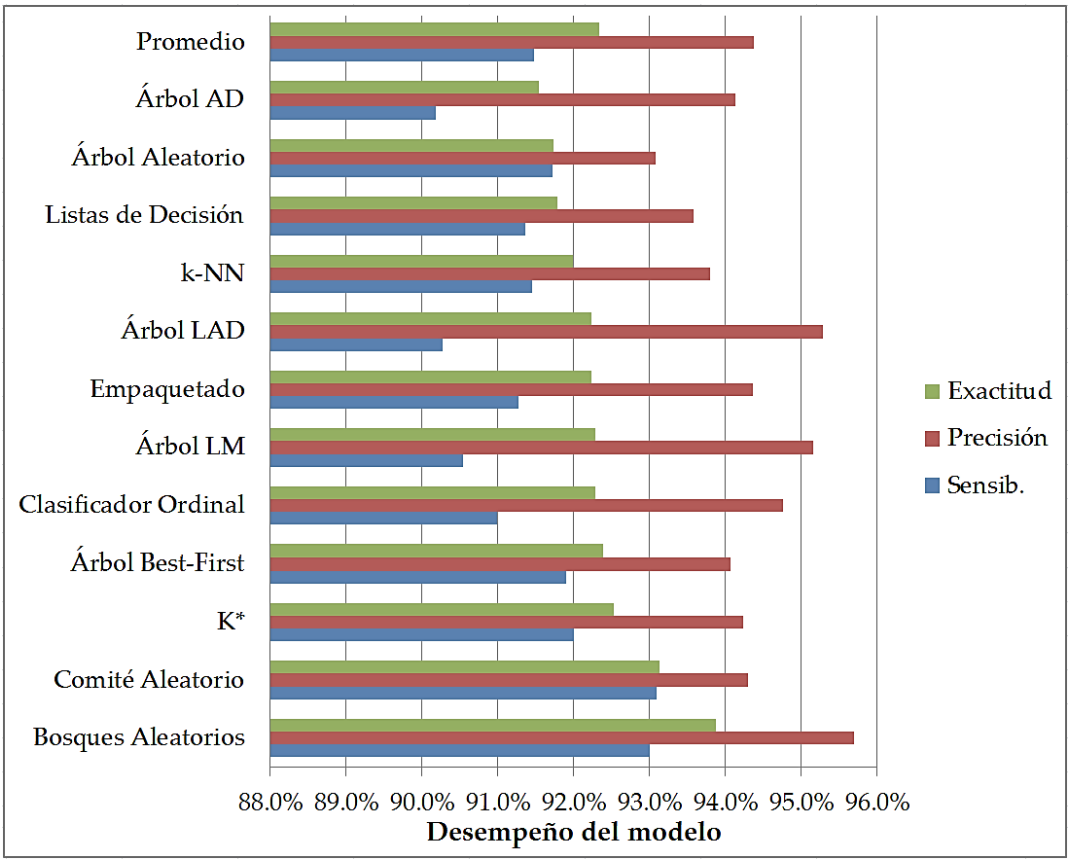


Figura 5.3 Desempeño de los mejores modelos internos para JDOM.

Queremos finalizar esta sección mencionando un par de cuestiones muy importantes observadas a lo largo de este análisis:

1. El desempeño logrado por cada conjunto de mejores modelos ha favorecido a SAPCyTI por encima de JDOM, aunque no de manera significativa. 93.9% contra 92.4% (exactitud), 95.6% contra 94.4% (precisión) y 92.2% contra 91.5% (sensibilidad), lo cual sugiere estabilidad en el uso de nuestra AAIT en diversos proyectos de software.
2. Es cierto que algunos de los 12 mejores modelos seleccionados difieren entre proyectos, e inclusive, que los 5 mejores modelos de SAPCyTI no coinciden con los 5 mejores de JDOM. Sin embargo, también es cierto que 7 de ellos sí coinciden independientemente de su posición dentro del listado (Árbol AD, k-NN, Árbol LAD, Árbol LM, Listas de Decisión, Clasificador Ordinal y Bosques Aleatorios [32, 22]) y mejor aún, las combinaciones elegidas para generar los subconjuntos de CA coinciden totalmente (CfsSubsetEval con BestFirst y GeneticSearch, ClassifierSubsetEval con RaceSearch y RankSearch, WrapperSubsetEval con BestFirst [32, 22]). Lo anterior, nos hace suponer que los modelos generados para cada proyecto tienen propiedades en común, las cuales pueden ser utilizadas para llevar un modelo de un proyecto a otro, situación que será discutida en la siguiente sección.

5.1.2. Modelos externos

De manera similar a los modelos internos, nos apoyamos de la figura 5.4 para analizar las características seleccionadas a partir de los conjuntos de VCA generados para cada proyecto y que no incorporan características de temporalidad (por ejemplo, el tiempo de vida de la alerta, la edad y la ranciedad del archivo), comenzado con las CA sin relevancia (aquellas que nunca fueron elegidas), siguiendo con las CA propias del proyecto y finalizando con las CA comunes a ambos proyectos.

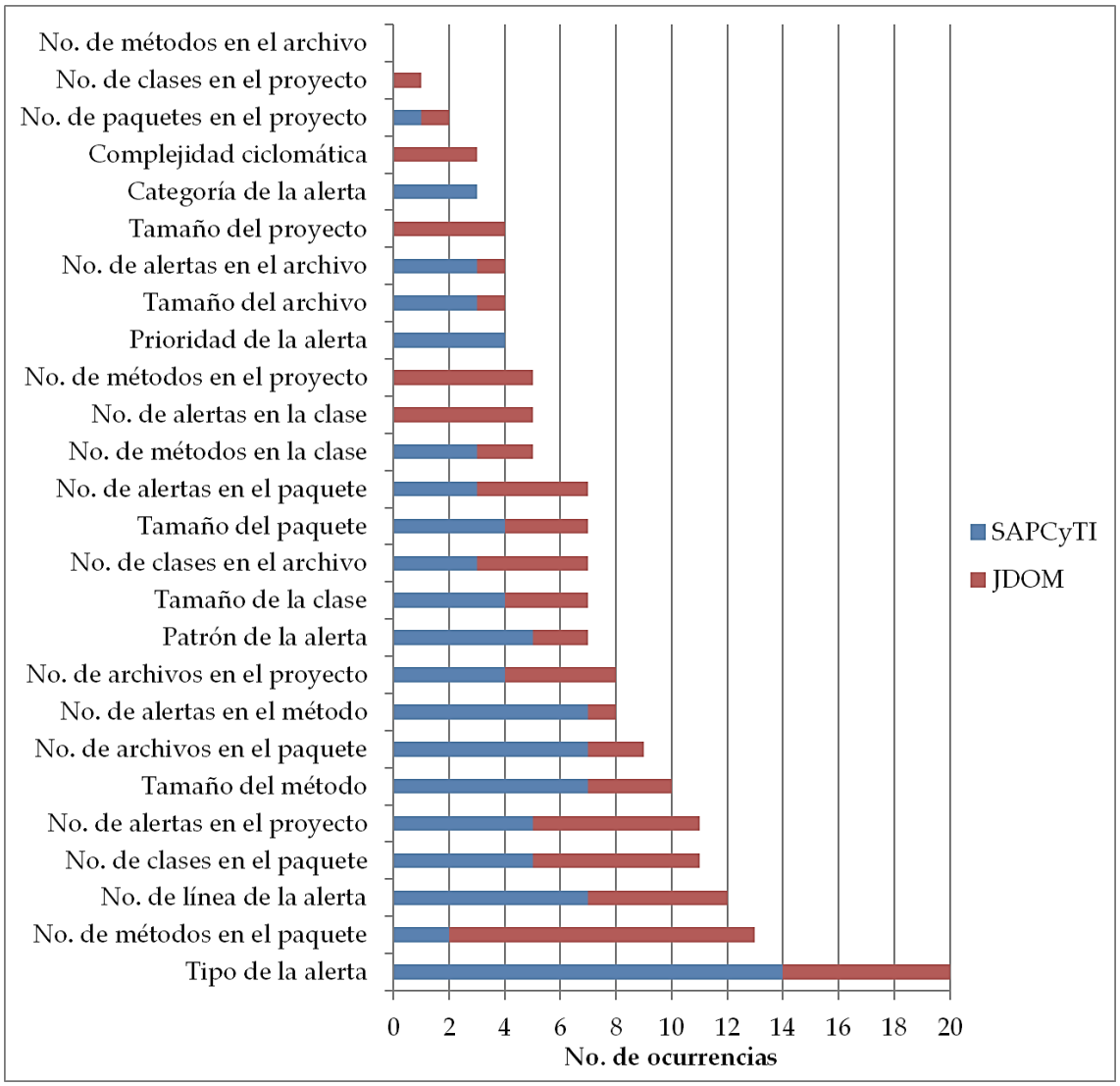


Figura 5.4 Número de ocurrencias de CA seleccionadas para modelos externos.

Como se puede ver en la figura 5.4, el número de métodos en el archivo no tiene relevancia alguna dentro de los subconjuntos y, a diferencia de la selección de CA en modelos internos, la complejidad ciclomática manifiesta tener relevancia cuando menos para el proyecto JDOM. Como mencionamos antes, atribuimos dicho contraste a la ausencia de características de temporalidad en el conjunto de VCA, las cuales demostraron ser altamente relevantes (alto número de ocurrencias) en la

accionabilidad de una alerta. Una situación similar ocurre con el tamaño y el número de métodos en el archivo, además del número de paquetes en el proyecto.

Una de las características propias del proyecto SAPCyTI es la categoría de la alerta, situación que nos lleva a pensar lo siguiente: si dicha CA es seleccionada sólo para SAPCyTI y nunca para JDOM, entonces el equipo de desarrollo de SAPCyTI tiende a introducir demasiados defectos en categorías que no se presentan con regularidad dentro de los defectos introducidos en JDOM. Algunas de las características seleccionadas para JDOM que no son seleccionadas para SAPCyTI ocurren a nivel de proyecto (tamaño y número de clases), creemos que este comportamiento sucede debido a las propiedades específicas de cada proyecto (las cuales fueron en la sección 4.1 -véase tabla 4.1-).

Sin embargo, lo realmente interesante se encuentra en las 18 CA comunes a ambos proyectos, lo cual demuestra que pueden existir características similares entre dos proyectos de software ajenos entre sí, situación que sustenta nuestra idea de utilizar modelos externos para la clasificación de nuevas alertas. Como era de esperarse y a consecuencia de nuestra heurística de reetiquetado de alertas, el tipo de alerta dejó de ser una características propia de SAPCyTI (véase figura 5.1) para convertirse en una característica común a ambos proyectos y de alta relevancia en la clasificación. La misma lógica es aplicada para el patrón de alerta, el cual está íntimamente ligado al tipo de la alerta.

Algunas de las características que, bajo la perspectiva de modelos internos, son ajenas entre sí y que ahora son comunes a ambos proyectos son el número de alertas en el proyecto, el número de alertas en el archivo, el tamaño y el número de alertas en el método; por otro lado, algunas características cuya relevancia (o número de ocurrencias) aumentó son el número de línea de la alerta y el número de archivos y de alertas en el proyecto. Suponemos que este comportamiento se debe a la ausencia de características de temporalidad, las cuales son muy relevantes en la clasificación y tienden a disminuir la aparición de características que ahora sí forman parte de los subconjuntos generados.

De lo anterior, podemos destacar lo siguiente: (1) el número de CA seleccionadas varió de 3 a 11 para ambos proyectos, (2) el número de subconjuntos de CA válidos fue de 14 para SAPCyTI y de 12 para JDOM, de los cuales (3) BestFirst y GeneticSearch [32, 22] lograron combinarse con todos los métodos y (4) nuevamente ClassifierSubsetEval [32, 22] se combinó con todas las estrategias para generar subconjuntos de CA válidos.

A continuación, analizamos el comportamiento de los 12 mejores modelos externos con base en su desempeño, la figura 5.5 y la figura 5.6 muestran la exactitud, la precisión y la sensibilidad promedio lograda por cada modelo externo a JDOM y a SAPCyTI respectivamente.

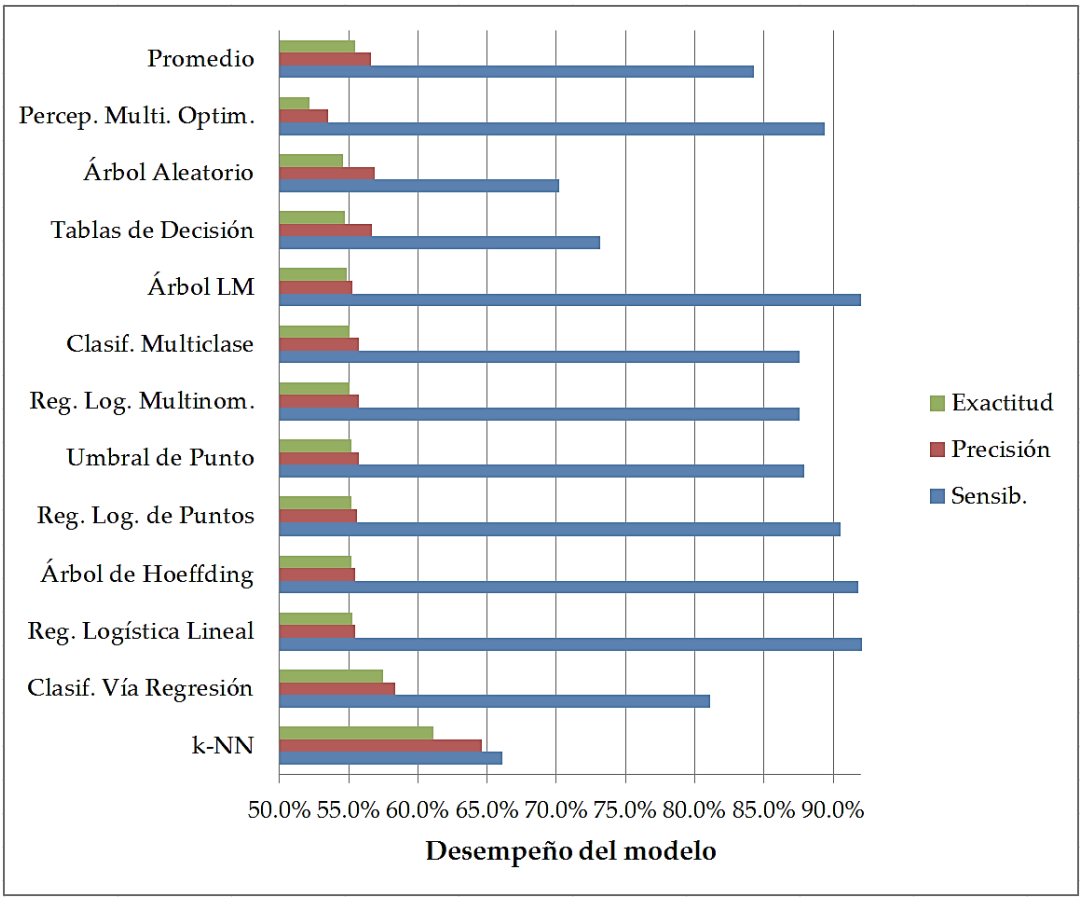


Figura 5.5 Desempeño de los mejores modelos externos a JDOM.

Para los modelos que son entrenados mediante las alertas de SAPCyTI y evaluados sobre las alertas de JDOM (véase figura 5.5) se tienen valores de exactitud que oscilan entre el 52.2% y el 61.1%, donde las 5 primeras posiciones son ocupadas por modelos basados en los algoritmos k-NN (61.1%), Clasificación Vía Regresión (57.5%), Regresión Logística Lineal (55.3%), Árbol de Hoeffding (55.2%) y Regresión Logística de Kernel (55.2%) [32, 22].

Como se puede ver en la figura 5.5, la sensibilidad de cada modelo siempre es mayor a la precisión y a la exactitud, con una superioridad de hasta un 38.4% para el caso donde alcanza su más alto nivel (modelo basado en el algoritmo Regresión Logística Lineal [32, 22] con el 93.6%). Dicho comportamiento nos ayuda cumplir el objetivo de aumentar el número de defectos relevantes descubiertos al disminuir las mayor cantidad posible de falsos negativos, sin embargo, presenta una desventaja que radica en la precisión; recordemos que a menor precisión, mayor será el número de alertas que el desarrollador deberá reparar para descubrir aquellas realmente relevantes. Por lo anterior, deseamos fijarnos en aquellos modelos cuyo desempeño se encuentre equilibrado en cuanto a exactitud, precisión y sensibilidad.

Y hablando de equilibrio entre las 3 medidas de desempeño, también se puede observar que conforme la sensibilidad decrece, la precisión y la exactitud aumenta. Desafortunadamente, en este tipo de modelos no se alcanza un modelo cuyo desempeño sea equilibrado tal como ocurrió con el algoritmo Perceptrón Multicapa Optimizado [32, 22] (véase figura 5.2); sin embargo, k-NN [32, 22] logra un equilibrio aceptable dentro del conjunto (véase figura 5.5). Por último, el mejor modelo externo a JDOM se obtiene mediante el subconjunto de CA generado por la combinación de ClassifierSubsetEval y GreedyStepwise (véase tabla 4.12) [32, 22] y la ejecución del algoritmo k-NN [32, 22], logrando un desempeño del 68.9%, 73.9% y 66.8% para la exactitud, precisión y sensibilidad respectivamente.

Por otra parte, el desempeño conseguido por los modelos externos a SAPCyTI (véase figura 5.6) es relativamente mayor al de JDOM, oscilando entre el 54.2% y el 67.6% y logrando un aumento del 8.8% sobre el promedio del conjunto. De acuerdo a su exactitud, dentro de las 5 primeras posiciones de nuestro conjunto de mejores modelos, se encuentran 3 metaclasificadores y 2 algoritmos de árboles de decisión, los cuales son: Empaquetado, Clasificación Vía Regresión, Bosques Aleatorios, Árbol REP y Comité Aleatorio [32, 22] con el 67.6%, 66.8%, 66.7%, 66.6% y 66.1% respectivamente.

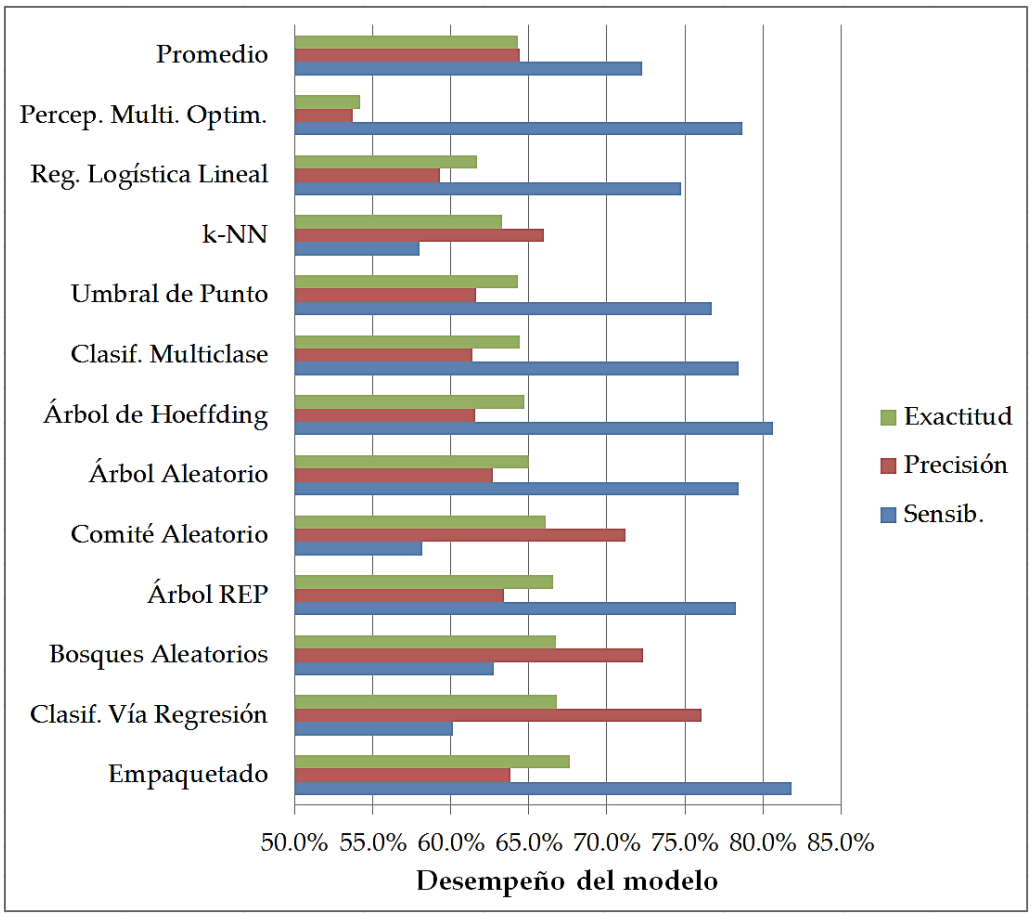


Figura 5.6 Desempeño de los mejores modelos externos a SAPCyTI.

A diferencia del conjunto de mejores modelos externos a JDOM, dentro de los modelos generados para SAPCyTI no es posible visualizar un patrón de comportamiento en el desempeño de cada uno de ellos; sin embargo, se pueden observar un par de particularidades dentro de dicho conjunto, las cuales son:

1. Como se puede ver en la figura 5.6, hay 8 modelos cuya exactitud es mayor a la precisión y la exactitud menor a la sensibilidad (por ejemplo, Empaquetado, Árbol de Hoeffding y Perceptrón Multicapa Optimizado [32, 22]), lo cual no resulta del todo conveniente y explicamos el porqué de esta afirmación: una sensibilidad superior en combinación con una exactitud mayor a la precisión, indica que el mayor número de alertas correctamente clasificadas se encuentran en la clase no accionable y, en relación a la precisión, existen un número elevado de falsos positivos, lo cual implica la reparación innecesaria de alertas y por tanto, un comportamiento no deseado dentro de nuestros modelos.
2. Hay 4 casos en que la precisión es mayor a la exactitud y la exactitud mayor a la sensibilidad (por ejemplo, k-NN y Bosques Aleatorios [32, 22]). De no ocurrir un equilibrio entre las 3 medidas de desempeño, este es el comportamiento esperado (cuyo porqué ya ha sido discutido en la sección anterior) y qué mejor que lograrlo dentro de las 5 primeras posiciones del conjunto, tal es el caso de Comité Aleatorio [32, 22], el cual construye al mejor modelo externo para SAPCyTI con un 70.7%, 69.3% y 74.3% para la exactitud, precisión y sensibilidad respectivamente mediante el subconjunto generado por la combinación de ClassifierSubsetEval y GreedyStepwise [32, 22].

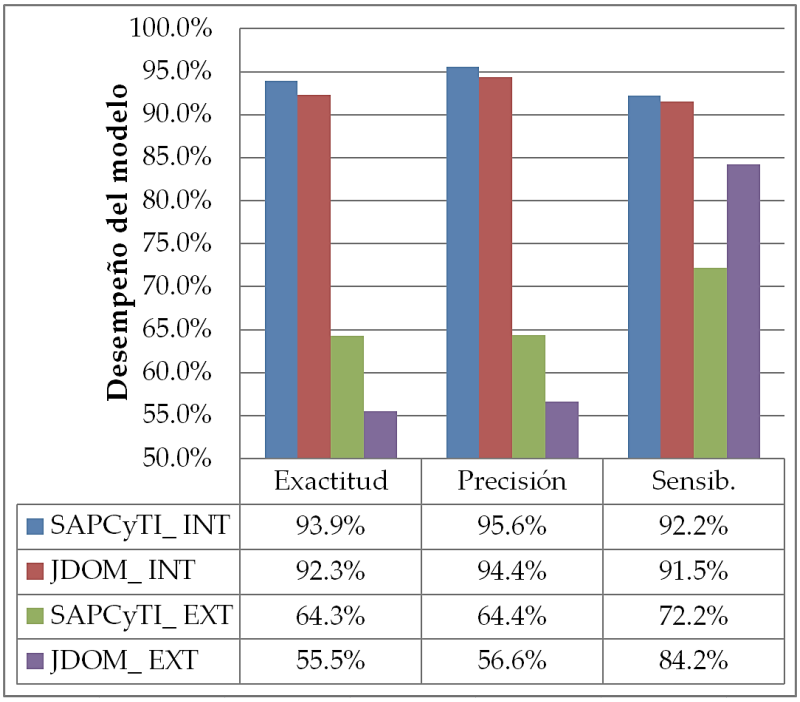
A continuación, queremos mencionar dos cuestiones importantes observadas a lo largo de este análisis, las cuales son:

1. El desempeño logrado por cada conjunto de modelos externos ha favorecido a SAPCyTI por encima de JDOM, pero no de manera significativa: 64.3% contra 55.5% (exactitud), 64.4% contra 56.6% (precisión) y 72.2% contra 84.2% (sensibilidad), lo cual sugiere una mayor estabilidad en los modelos aplicados a SAPCyTI.
2. 8 de los 12 mejores modelos seleccionados son comunes a ambos proyectos: 2 basados en algoritmos lineales o funciones (Perceptrón Multicapa Optimizado y Regresión Logística Lineal [32, 22]), 2 basados en árboles de decisión (Árbol de Hoeffding y Árbol Aleatorio [32, 22]), 3 basados en metaclasificadores (Clasificación Vía Regresión, Clasificador Multiclase y Umbral de Punto [32, 22]) y 1 algoritmo de vecinos más cercanos (k-NN [32, 22]).

Los dos puntos anteriores, nos permiten observar que existen algoritmos que aprenden mejor con atributos numéricos, tal es el caso de los algoritmos lineales (funciones) y de los metaclasificadores, otros que se desempeñan mejor con atributos nominales como en el caso de los algoritmos de árboles de decisión y basados en reglas (los cuales fueron seleccionados para los modelos internos) y, en especial, un algoritmo capaz de desempeñarse satisfactoriamente en ambos casos (k-NN [32, 22]).

5.1.3. Evaluación comparativa

A lo largo de esta sección, analizaremos el desempeño obtenido por los modelos generados mediante nuestra AAIT y los confrontaremos contra el desempeño de diferentes modelos propuestos por diferentes investigadores [7, 11, 9]. Para conseguirlo, calculamos la exactitud, la precisión y la sensibilidad promedio de todos los modelos generados a partir de las 5 mejores combinaciones elegidas para cada proyecto (véase sección 4.3) y seleccionamos aquellos que se basan en los algoritmos utilizados por distintos autores, los cuales son: Heckman y Williams [7], Yüksel y Sözer [11] y Hanam *et al.* [9, 10]. Los primeros dos estudios utilizan un conjunto de VCA que incluyen características de temporalidad (por ejemplo, el tiempo de vida de la alerta, la edad y la ranciedad del archivo), contrario a lo realizado por el último de ellos que, con base en el trabajo de Heckman y Williams [7], propone una AAIT que no utiliza dichas características.



	Exactitud	Precisión	Sensib.
SAPCyTI_ INT	93.9%	95.6%	92.2%
JDOM_ INT	92.3%	94.4%	91.5%
SAPCyTI_ EXT	64.3%	64.4%	72.2%
JDOM_ EXT	55.5%	56.6%	84.2%

Figura 5.7 Comparativa entre modelos internos y modelos externos.

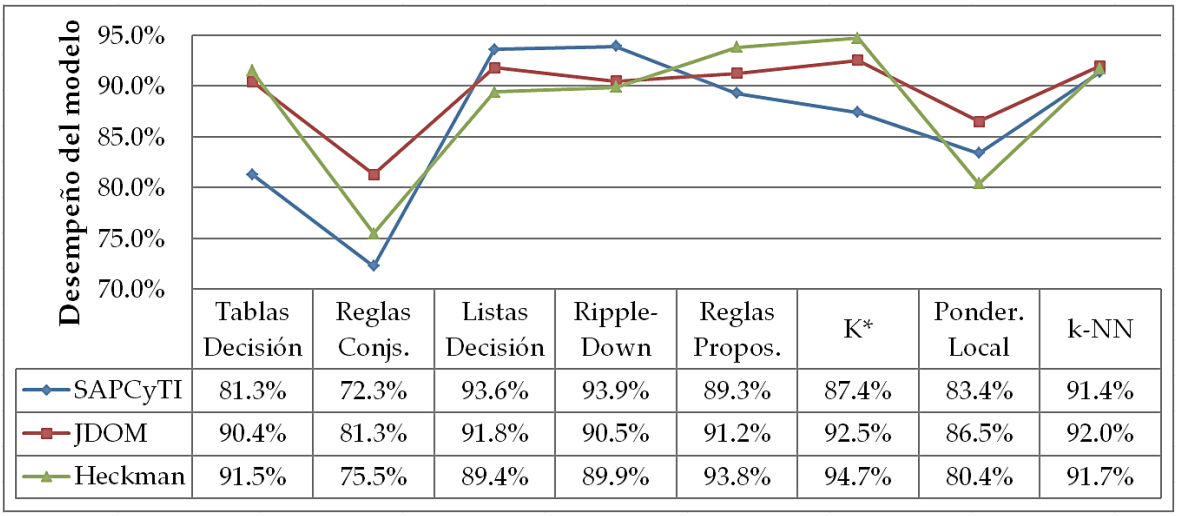
Como se puede ver en la figura 5.7, existe un evidente contraste entre utilizar modelos internos y modelos externos al proyecto, diferencias del 29.6% y 36.8% (exactitud), 31.2% y 37.8% (precisión) y

20.0% y 7.3% (sensibilidad) para SAPCyTI y JDOM respectivamente. Indudablemente, dichas mermas en el desempeño de los modelos se deben en gran medida a la omisión de aquellas características cuya aportación a la correcta clasificación de alertas es sobresaliente, tal es el caso del tiempo de vida de la alerta, el número de versión de modificación del archivo, la ranciedad del archivo, entre otras (véase sección 5.1.1). Sin embargo, inferimos que no sólo se trata del número de CA utilizadas o del grado de aportación de cada característica, sino también de otros factores que provocan una disminución en el desempeño al utilizar modelos externos para la clasificación de nuevas alertas, los cuales son:

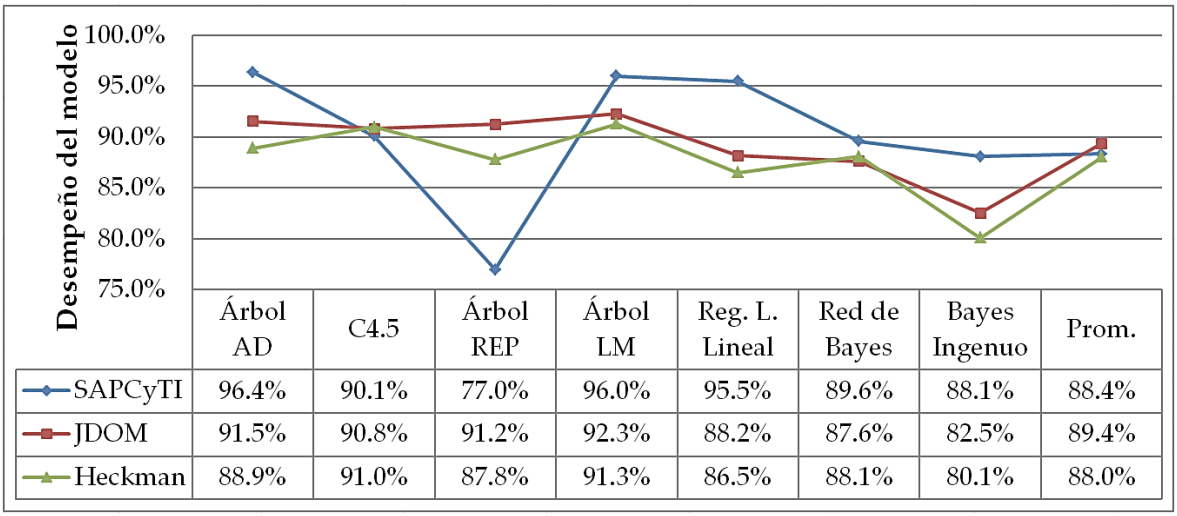
1. Las principales características de ambos proyectos son similares entre sí en el aspecto cuantitativo, sin embargo, podemos suponer que existen otros elementos (de tipo cualitativo) que disminuyen el desempeño de un modelo externo, los cuales provienen de la naturaleza misma del proyecto de software y de las personas involucradas en el desarrollo de dicho sistema. Algunos ejemplos de estos elementos son: el dominio de aplicación, las reglas de negocio y políticas de desarrollo, la arquitectura de software y los atributos de calidad que se persiguen, la(s) metodología(s) y la(s) herramienta(s) utilizadas, el estilo de programación y el nivel de conocimiento sobre el lenguaje de los desarrolladores. Todos estos componentes seguramente inciden en el tipo de defectos introducidos por los desarrolladores y hacen únicas a algunas de las características que rodean a cada alerta o defecto.
2. Contrario a la afirmación hecha por Heckman y Williams [6, 7], la cual sostiene que es posible utilizar versiones tipo *major* o *minor* en lugar de revisiones de manera indistinta para la generación de modelos, deseamos argumentar aquellas cuestiones que nos llevan a contradecir dicha aseveración. Sabemos que surge una nueva revisión cuando una serie de modificaciones al código fuente es dada de alta en el repositorio donde este es almacenado [25, 26] y cuyo objetivo es, o la corrección de pequeños defectos o la refactorización del código existente. Por otro lado, una versión tipo *major* o *minor* es un componente que cubre funcionalidades concretas y que en consecuencia, cumple directamente con los requerimientos funcionales del sistema en mayor o menor medida una vez que ha descubierto y corregido el mayor número posible de grandes defectos [26]. Con esto en mente, mientras que las revisiones podrían introducir una gran variedad de defectos relevantes como consecuencia de las múltiples modificaciones al código fuente, las versiones tendrían un número reducido de defectos relevantes y posiblemente una gran cantidad de defectos irrelevantes a la correcta funcionalidad del sistema, lo cual podría ocasionar efectos no deseados dentro de nuestra AAIT.

3. El hecho de que los tipos de defectos y sus CA asociadas difieran entre proyectos impacta sobre la efectividad de nuestra heurística de reetiquetado, haciendo que los subconjuntos de CA y los modelos seleccionados varíen entre modelos internos y modelos externos. Recordemos que dicha heurística únicamente tiene en cuenta el tipo de la alerta y evade las demás características, dando lugar a posibles perturbaciones en el aprendizaje de cada algoritmo al etiquetar una alerta como accionable cuando la mayoría de las características respaldan la no accionabilidad de dicha alerta.

Continuando con el análisis comparativo, Heckman y Williams [7] utilizaron 51 CA, 3 métodos de evaluación y 3 estrategias de búsqueda para construir modelos basados en 15 algoritmos de aprendizaje (5 basados en reglas, 4 basados en árboles de decisión, 3 de vecinos más cercanos, 2 bayesianos y 1 lineal) y evaluar su exactitud en 5 subconjuntos de CA generados para el proyecto JDOM. La figura 5.8 muestra el desempeño promedio obtenido por dichas autoras en cada uno de sus modelos y los resultados obtenidos por nosotros una vez que llevamos nuestro experimento a las condiciones establecidas en el experimento de Heckman y Williams [7].



	Tablas Decisión	Reglas Conjs.	Listas Decisión	Ripple-Down	Reglas Propos.	K*	Ponder. Local	k-NN
SAPCyTI	81.3%	72.3%	93.6%	93.9%	89.3%	87.4%	83.4%	91.4%
JDOM	90.4%	81.3%	91.8%	90.5%	91.2%	92.5%	86.5%	92.0%
Heckman	91.5%	75.5%	89.4%	89.9%	93.8%	94.7%	80.4%	91.7%



	Árbol AD	C4.5	Árbol REP	Árbol LM	Reg. L. Lineal	Red de Bayes	Bayes Ingenuo	Prom.
SAPCyTI	96.4%	90.1%	77.0%	96.0%	95.5%	89.6%	88.1%	88.4%
JDOM	91.5%	90.8%	91.2%	92.3%	88.2%	87.6%	82.5%	89.4%
Heckman	88.9%	91.0%	87.8%	91.3%	86.5%	88.1%	80.1%	88.0%

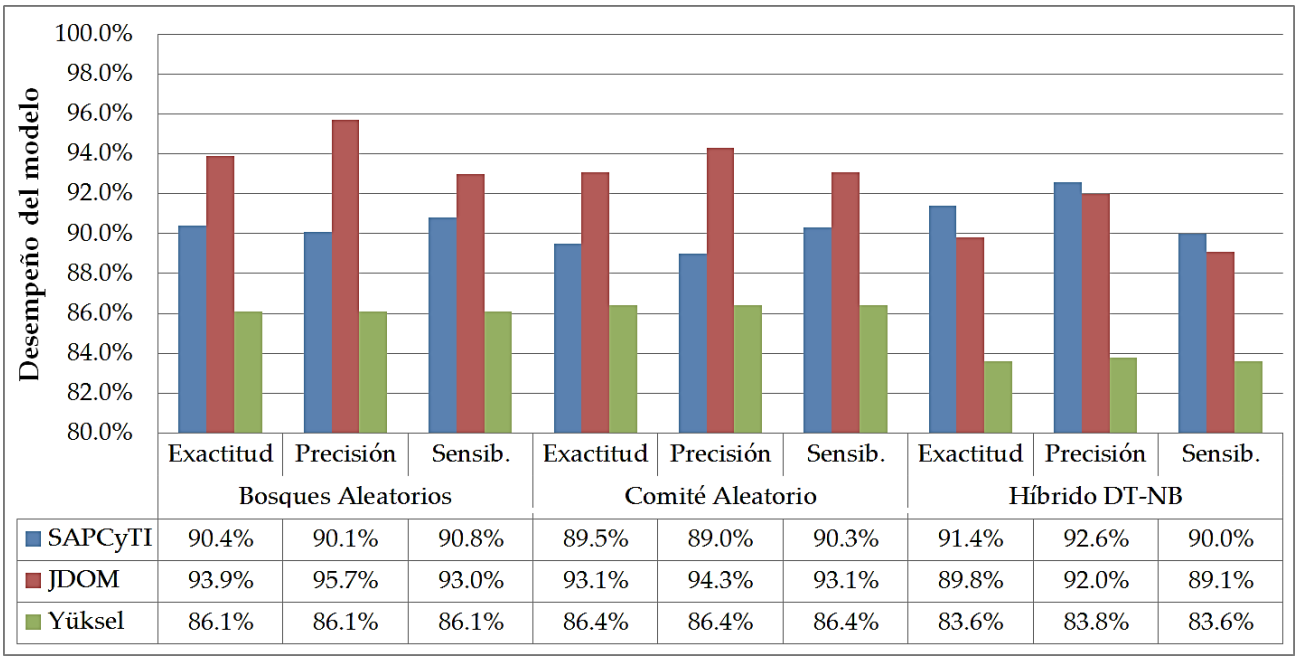
Figura 5.8 Comparativa entre modelos de Heckman y modelos internos.

Al observar la figura 5.8, reafirmamos el hecho de que nuestra técnica está inspirada en el trabajo realizado por estas autoras, basta con mirar el desempeño logrado por el conjunto de modelos generados: 88.4% para SAPCyTI, 89.4% para JDOM bajo nuestra AAIT y 88.0% para JDOM bajo la AAIT de Heckman y Williams [7]. Cabe mencionar que estos valores de exactitud se encuentran por debajo de nuestros conjuntos de mejores modelos con un 3.3% adicional para JDOM y un 5.9% para SAPCyTI (véase figura 5.7).

Y hablando del desempeño promedio logrado (exactitud), consideramos que esta mejora del 1.4% para el caso de JDOM (véase figura 5.8) sucede por la siguiente razón: la heurística de etiquetado de alertas utilizada en [6, 7] sólo describe la manera en que debe realizarse dicho etiquetado, sin embargo, no es muy clara al definir lo que debe suceder con los valores asignados a cada CA. Supongamos que una alerta es reportada en una versión temprana del proyecto y su estado permanece como no accionable a lo largo de varias versiones para después desaparecer. Si dicha alerta contiene una característica llamada tamaño del proyecto, formulamos dos preguntas: ¿el valor que se asigna a esta CA debe ser igual al valor asignado durante la aparición de la alerta o debe ser actualizado a medida que se avanza a través de las distintas versiones? y ¿el valor de esta CA debe congelarse o debe actualizarse una vez que la alerta ha sido etiquetada como accionable? La respuesta a estas dos preguntas hace que la diferencia sea notable, ya que si se elige actualizarlo, el tamaño del proyecto actuaría como una característica acumulativa y es posible que a mayor tamaño del proyecto, mayor probabilidad de accionabilidad y en consecuencia, el algoritmo de aprendizaje estaría basando su clasificación en el tamaño del proyecto y no sería un comportamiento deseable. Cabe mencionar que el caso descrito para la característica llamada tamaño del proyecto es aplicable a las demás características cuyos valores sean numéricos. En conclusión, la forma en que se asignan los valores a cada CA influye en el desempeño de cada modelo.

Finalmente, en la figura 5.8 se puede observar que nuestros modelos basados en los algoritmos Listas de Decisión, Reglas Ripple-Down, Ponderación Local, Árbol AD, Árbol LM, Regresión Logística Lineal y Bayes Ingenuo [32, 22] superan a los modelos construidos por estas autoras con diferencias relativamente pequeñas, las cuales oscilan entre el 0.6% y el 9.0%. Los algoritmos Reglas Conjuntivas, Ponderación Local y Bayes Ingenuo [32, 22] obtienen las peores exactitudes dentro del conjunto (76.4%, 83.4% y 83.6% respectivamente) y en contraste, Árbol LM, k-NN y Árbol AD (93.2%, 92.3% y 91.7% respectivamente) [32, 22] son aquellos que logran desempeñarse de mejor manera independientemente del proyecto.

Por su parte, Yüksel y Sözer [11] recopilaron y analizaron 1,147 alertas (cada una descrita por 10 CA) para un sistema de software de televisión digital con el fin generar modelos de clasificación mediante 10 algoritmos de selección de atributos y 34 algoritmos de aprendizaje. La figura 5.9 muestra el desempeño de los 3 mejores modelos logrados por dichos autores (Bosques Aleatorios, Comité Aleatorio e Híbrido DT-NB [32, 22]) en comparación al desempeño obtenido por nosotros luego de evaluar estos 3 modelos de acuerdo a su exactitud, precisión y sensibilidad.



	Bosques Aleatorios			Comité Aleatorio			Híbrido DT-NB		
	Exactitud	Precisión	Sensib.	Exactitud	Precisión	Sensib.	Exactitud	Precisión	Sensib.
SAPCyTI	90.4%	90.1%	90.8%	89.5%	89.0%	90.3%	91.4%	92.6%	90.0%
JDOM	93.9%	95.7%	93.0%	93.1%	94.3%	93.1%	89.8%	92.0%	89.1%
Yüksel	86.1%	86.1%	86.1%	86.4%	86.4%	86.4%	83.6%	83.8%	83.6%

Figura 5.9 Comparativa entre modelos de Yüksel y modelos internos.

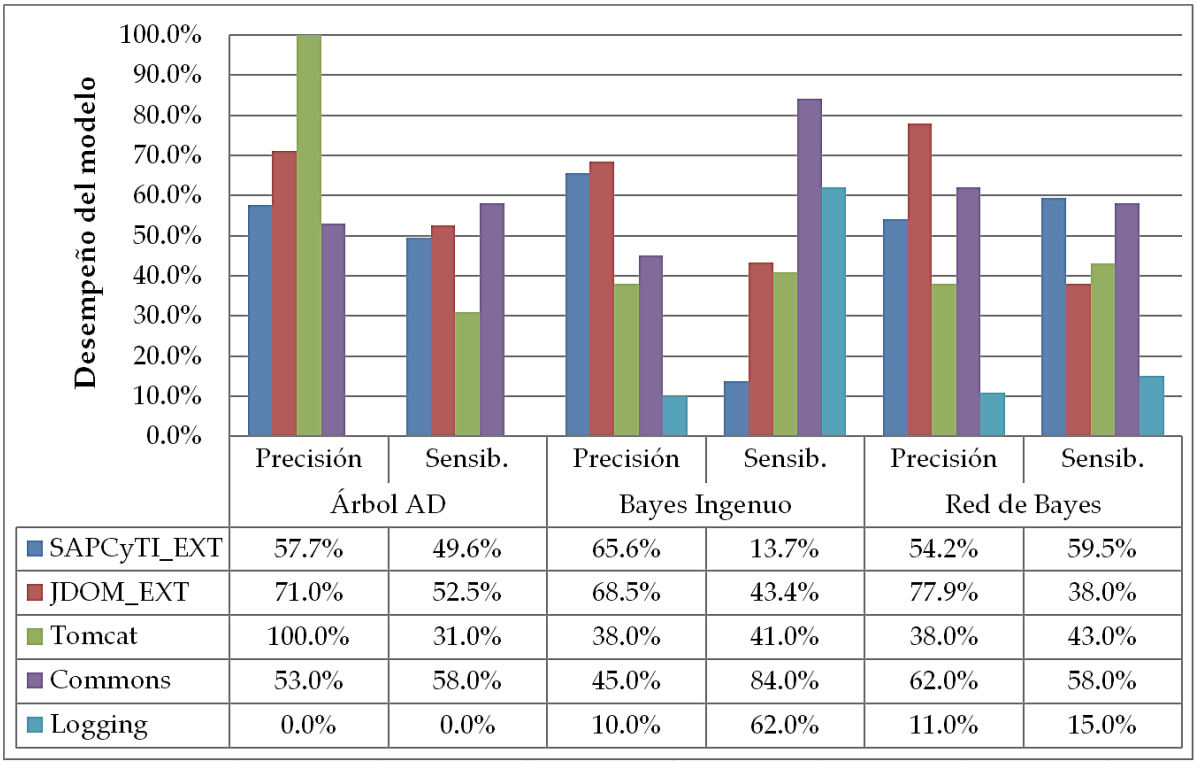
Como se puede ver en la figura 5.9, una de las características más sobresalientes del trabajo realizado por estos investigadores es el total equilibrio que logran en el desempeño de sus modelos (mismos valores de exactitud, precisión y sensibilidad), aun cuando ellos no indican si estos valores corresponden a un valor promedio o a un valor en particular de los 30 modelos generados por cada algoritmo [11]. Suponemos que dicho equilibrio fue el criterio para la selección de sus mejores modelos, a pesar de esto, su desempeño resulta ser menor con un 5.1% respecto a SAPCyTI y con un 7.3% respecto a JDOM (tomando en cuenta que el desempeño promedio de los conjuntos es de 90.5%, 92.7% y 85.4% para SAPCyTI, JDOM y Yüksel respectivamente).

Una de las posibles causas que seguramente disminuyen la exactitud, precisión y sensibilidad de esta AAIT es el nivel de relevancia de cada característica en la accionabilidad de cada alerta. De acuerdo a nuestra experiencia, tan sólo 2 CA son de alta relevancia (el tiempo de vida de la alerta y la idea del desarrollador) y el resto son de mediana relevancia (por ejemplo, el tipo de la alerta y el número de

alertas en el método). Creemos que estas características pueden ser complementadas con otras para lograr un mejor desempeño, por ejemplo, la ranciedad y el tiempo de vida del archivo.

Ya hemos hablado acerca del impacto que tiene dar demasiado peso a una sola CA (en nuestro caso, el tipo de la alerta) y justo eso sucede con la idea del desarrollador, la cual juega un papel tan importante en la clasificación que deja de lado la posible relevancia que pudieran tener las 9 características restantes, perturbando al algoritmo de aprendizaje y disminuyendo su desempeño. Este efecto no impacta tanto en nuestros modelos debido a que describimos una alerta mediante un mayor número de características, además de ser muy pocas las alertas reetiquetadas como accionables.

Por último, Hanam *et al.* [9, 10] evaluaron la precisión y la sensibilidad de 3 modelos basados en los algoritmos Árbol AD, Bayes Ingenuo y Red de Bayes [32, 22], excluyendo características de temporalidad y recopilando tan sólo 19 CA para 3 proyectos de software. La figura 5.10 muestra el desempeño de sus modelos en contraste con el desempeño de los nuestros bajo dichos algoritmos.



	Árbol AD		Bayes Ingenuo		Red de Bayes	
	Precisión	Sensib.	Precisión	Sensib.	Precisión	Sensib.
SAPCyTI_EXT	57.7%	49.6%	65.6%	13.7%	54.2%	59.5%
JDOM_EXT	71.0%	52.5%	68.5%	43.4%	77.9%	38.0%
Tomcat	100.0%	31.0%	38.0%	41.0%	38.0%	43.0%
Commons	53.0%	58.0%	45.0%	84.0%	62.0%	58.0%
Logging	0.0%	0.0%	10.0%	62.0%	11.0%	15.0%

Figura 5.10 Comparativa entre modelos de Hanam y modelos externos.

A diferencia del equilibrio que puede lograrse en la exactitud, precisión y sensibilidad de modelos que incorporan características de temporalidad, aquellos modelos que no lo hagan carecerán de dicha estabilidad. Esta es una observación que puede hacerse al mirar nuestros modelos externos (véase figura 5.5 y figura 5.6) y los modelos generados por Hanam *et al.* [9, 10]. Como se puede ver en la figura 5.10, a mayor precisión, menor sensibilidad y a mayor sensibilidad, menor precisión; ejemplos de

esto es el desempeño obtenido por los algoritmos Árbol AD [32, 22] en el proyecto Tomcat (cuya precisión es del 100.0% y cuya sensibilidad es del 31.0%) y por Bayes Ingenuo [32, 22] en SAPCyTI (con una exactitud del 65.6% y una sensibilidad del 13.7%).

Lejos de redundar en las causas que originan esta falta de equilibrio entre la precisión y la sensibilidad (discutida en la sección anterior), resulta más interesante analizar el bajo desempeño de los modelos generados para Logging, en particular la mala clasificación realizada por el algoritmo Árbol AD [32, 22]. Estamos convencidos de que la principal causa de este comportamiento radica en la proporción de alertas etiquetadas como accionables (tan sólo el 13% [9]), lo cual dificulta el aprendizaje de cualquier algoritmo al contar con poca información acerca de la accionabilidad de alertas dentro del conjunto de VCA, inclinando su aprendizaje a la clasificación de alertas no accionables. Y justamente los resultados validan esta afirmación: el algoritmo Árbol AD [32, 22] clasifica todas las alertas del conjunto como no accionables, por su parte Bayes Ingenuo y Red de Bayes [32, 22] son completamente propensos a clasificar la mayor parte de las alertas como no accionables. En este sentido, el conjunto de alertas generado para Commons se encuentra equilibrado de mejor manera con un 32.9% de alertas accionables [9], situación que se ve reflejada en el desempeño de sus modelos (véase figura 5.10).

En general, el desempeño de los modelos generados por Hanam *et al.* [9, 10] y de nuestros modelos se comporta de manera similar, excepto para Logging, cuyos modelos se desempeñan 33.7% y 42.2% debajo de SAPCyTI y de JDOM respectivamente. Cabe mencionar que nosotros hemos producido mejores modelos de clasificación que no incorporan características de temporalidad, sin embargo, quisimos adaptarnos a las condiciones particulares de estos autores para efectos de comparación.

5.2. Análisis de propuesta

Una de las recomendaciones más importantes hecha en el libro de Jones [2] menciona que, para lograr una eficiencia de eliminación de defectos dentro del código fuente cercana al 99%, debe utilizarse la combinación de 3 tipos de prácticas de ingeniería de software, las cuales son: revisiones o inspecciones, análisis estático y las distintas etapas de la fase de pruebas. La primera de ellas es llevada a cabo de forma manual, la segunda y la tercera pueden realizarse de forma automatizada.

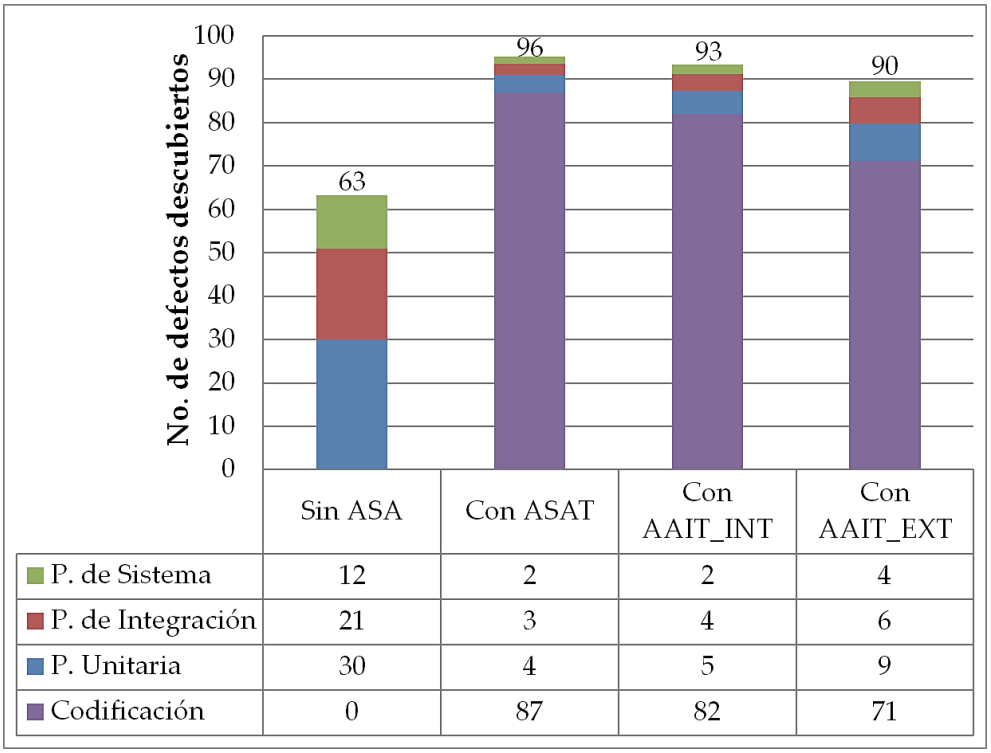
Como vimos en la sección 1.2, el 71.3% de los desarrolladores no implementan análisis estático dentro de sus PDS [14, 15], situación que provocaría la omisión de posibles defectos que más tarde habrán de afectar la correcta funcionalidad del sistema. Por lo anterior, nuestra propuesta plantea la

utilización de una AAIT luego de haber concluido la fase de codificación y previo a las distintas etapas de la fase de pruebas con el objetivo de descubrir el mayor número posible de defectos relevantes dentro del código fuente. Recordemos que dicha AAIT se basa en la utilización de modelos internos o externos al proyecto para la clasificación de nuevas alertas (véase capítulo 3).

Dicho lo anterior, procederemos a realizar un análisis cuantitativo sobre el impacto que produciría nuestra AAIT una vez incorporada al PDS con el fin de establecer la diferencia entre utilizarla o no. Realizaremos este análisis desde 3 perspectivas diferentes, las cuales son: (1) el número de defectos descubiertos y el número de defectos presentes en cada una de las fases del PDS, (2) el número de alertas que deben ser reparadas antes de lograr descubrir cada uno de los defectos relevantes al sistema y (3) el tiempo total que se emplea en la corrección de dichos defectos.

Como vimos en la sección 3.1, el resultado de la codificación es un conjunto de archivos que almacenan el código fuente del proyecto en cuestión. Ahora, supongamos que dentro de este conjunto de archivos existen 100 defectos relevantes (los cuales afectan a la correcta funcionalidad del sistema) y que, de acuerdo a Jones [2], cada práctica de ingeniería de software tiene asociada una eficiencia de eliminación de defectos, es decir, el Análisis Estático Automatizado (ASA, *Automated Static Analysis*) puede descubrir hasta un 87% del total de estos defectos, las pruebas unitarias encuentran hasta un 30%, las pruebas de integración un 30% y las de sistema un 25%. En particular, la eficiencia del ASA se verá afectada por la sensibilidad del modelo de clasificación utilizado (ya sea interno o externo), por lo que su nueva eficiencia será la multiplicación de la sensibilidad del modelo por la eficiencia definida en el libro de Jones [2]. Cabe mencionar que, hemos utilizado el valor de sensibilidad de nuestro mejor modelo interno (Árbol AD [32, 22] con el 94.2%) y el de nuestro mejor modelo externo (Empaquetado [32, 22] con el 81.8%), ambos evaluados sobre el proyecto SAPCyTI.

Para calcular el número de defectos descubiertos después de cada fase del PDS, bastaría con multiplicar el número de defectos aún presentes en el sistema (comenzando con 100) por la eficiencia de la práctica de ingeniería de software asociada a cada fase. Por ejemplo, nuestra AAIT basada en un modelo externo a SAPCyTI, logrará descubrir 100 * 0.712 = 71 defectos después de la fase de codificación, 29 * 0.3 = 9 defectos luego de las pruebas unitarias, seguido de 20 * 0.3 = 6 defectos para las pruebas de integración y de 14 * 0.25 = 4 para las pruebas de sistema. La figura 5.11 muestra los cálculos realizados para un PDS que no utiliza el ASA, otro que sí lo hace pero de manera directa mediante el uso de una ASAT, uno más que utiliza nuestra AAIT basada en un modelo interno (AAIT_INT) y por último, utilizando nuestra AAIT basada en un modelo externo (AAIT_EXT).



	Sin ASA	Con ASAT	Con AAIT_INT	Con AAIT_EXT
P. de Sistema	12	2	2	4
P. de Integración	21	3	4	6
P. Unitaria	30	4	5	9
Codificación	0	87	82	71

Figura 5.11 No. de defectos descubiertos en cada fase del PDS.

Como se puede ver en la figura 5.11, un PDS que sólo se apoya en las distintas etapas de la fase de pruebas para descubrir la mayor cantidad posible de defectos dentro del sistema, siempre será menos eficaz que aquel que sí incorpore el uso del análisis estático. La omisión del ASA causaría que 37 defectos lleguen a ser descubiertos por el usuario final del sistema, en contraste a la utilización de una Herramienta de Análisis Estático Automatizado (ASAT, *Automated Static Analysis Tool*), de un modelo interno y de un modelo externo, los cuales sólo dejarían pasar 4, 7 y 10 defectos respectivamente. Decimos entonces que, no utilizar análisis estático en el desarrollo de software provocaría una disminución en la eficiencia de eliminación de defectos de entre 27.0% y 33.0% respecto a procesos que sí lo hagan.

Aunque parecería que utilizar únicamente una ASAT para encontrar defectos dentro del código fuente es una mejor alternativa debido a que podrían descubrirse entre 3 y 6 defectos adicionales respecto al uso de nuestra AAIT, no debemos olvidar que en promedio, el 64% de las alertas reportadas por una ASAT son alertas no accionables (incluyendo a FindBugs) [6, 4, 16, 8, 17] y que esto impacta directamente en el número de alertas que el desarrollador debe tratar antes de lograr descubrir cada defecto relevante del sistema. Basados en el 100.0% - 64.0 % = 36.0% de alertas accionables que una ASAT es capaz de descubrir y la precisión obtenida por nuestra AAIT aplicada de manera interna y externa al proyecto SAPCyTI (98.5% y 63.8% respectivamente), calculamos el número de alertas a

reparar por cada 5 defectos relevantes descubiertos, dividiendo el número de defectos entre el porcentaje de alertas accionables detectadas por la ASAT o por nuestra AAIT. Por ejemplo, para descubrir 50 defectos relevantes, habrán de repararse 50 / 36.0% = 139, 50 / 98.5% = 51 y 50 / 63.8% = 78 alertas reportadas por la ASAT, por nuestro modelo interno (AAIT_INT) y por nuestro modelo externo (AAIT_EXT) respectivamente. La figura 5.12 muestra el resultado de dichos cálculos para los primeros 50 defectos relevantes descubiertos.

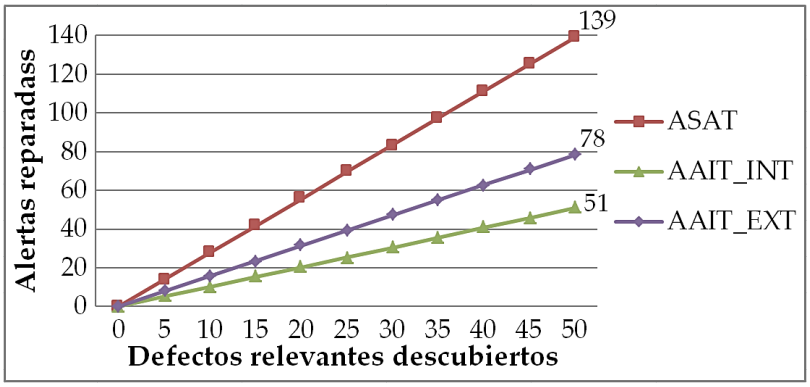


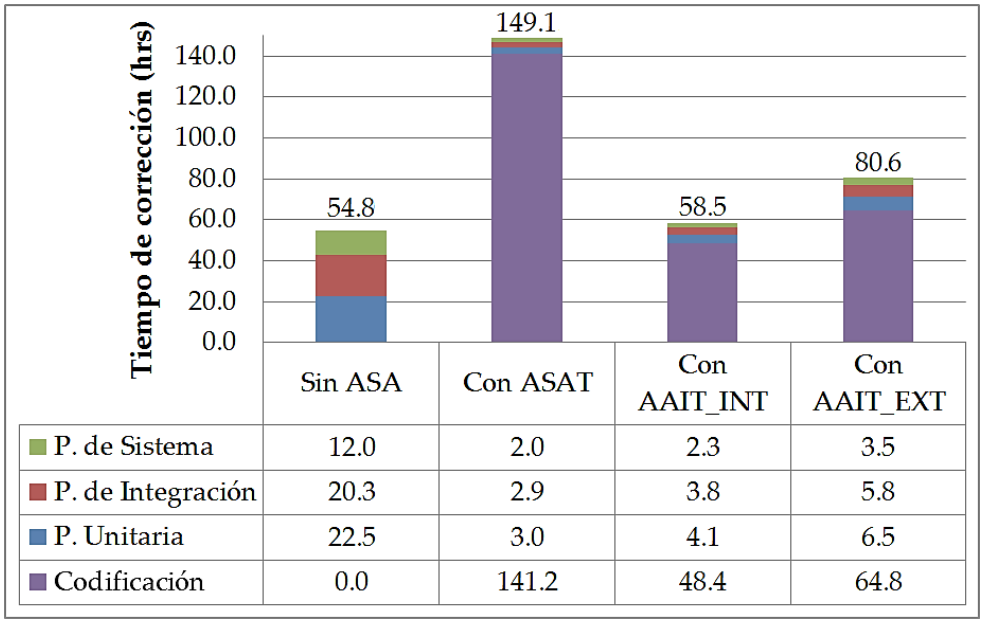
Figura 5.12 No. de defectos descubiertos contra no. de alertas reparadas.

Justo hasta este momento, es cuando se puede observar la diferencia entre realizar análisis estático a través de una ASAT o mediante la implementación de nuestra AAIT (ya sea interna o externa). Si bien es cierto que con el uso de una ASAT podrían descubrirse entre 3 y 6 defectos adicionales respecto a nuestra AAIT, la relación entre reparar 3 alertas para descubrir 1 defecto relevante (en el caso de la ASAT) contra reparar 1 o 2 alertas (para el caso de nuestra AAIT), haría que cualquier equipo de desarrollo eligiera utilizar nuestra técnica al observar que, para descubrir 50 defectos relevantes en el sistema habrían de repararse al menos 139 alertas (utilizando la ASAT) contra 78 o 51 (utilizando nuestra AAIT), sacrificando esta aparente ventaja de descubrir entre 3 y 6 defectos relevantes por encima de nuestra AAIT.

Dicho lo anterior, queremos asociar el número de defectos descubiertos y el número de defectos reparados con el tiempo que tomaría su corrección. Recordemos que el tiempo requerido para reparar un defecto crece de manera exponencial conforme transcurren las fases del PDS [3] y con base en el trabajo de Menzies *et al.* [57], asumimos que el tiempo para reparar un defecto al concluir la fase de codificación será de 35 minutos, después de las pruebas unitarias será de 45 minutos, luego de las pruebas de integración serán 58 minutos y para las pruebas de sistema se invertirán 60 minutos. Por tanto, para obtener el tiempo total de reparación de los defectos descubiertos después de cada fase, bastaría con multiplicar el tiempo requerido por el número de defectos descubiertos en cada una de ellas (véase figura 5.11) y, para el caso de la codificación, agregar el tiempo invertido en la corrección de

alertas que resultaron ser no relevantes a la funcionalidad del sistema (véase figura 5.12) ya que, a pesar de no reflejar un defecto relevante, estas alertas también requirieron de la inspección y corrección por parte del desarrollador.

Aunque el tiempo de corrección de defectos después de la fase de codificación es mayor en aquellos procesos que sí incorporan análisis estático en su desarrollo, como se puede ver en la figura 5.13, la diferencia ocurre una vez que se llega a las distintas etapas de la fase de pruebas al observar que el tiempo disminuye de tal manera que se termina invirtiendo un tiempo similar para corregir un mayor número de defectos respecto a un PDS que no utiliza el ASA, es decir, si se invierten 54.8 horas para corregir tan sólo 63 defectos, esto representa (1) 6.8% menos de esfuerzo pero con un 47.6% menos de defectos descubiertos en relación a nuestra AAIT_INT y (2) 47.1% menos de esfuerzo pero con 42.9% menos de defectos descubiertos en relación a nuestra AAIT_EXT.



	Sin ASA	Con ASAT	Con AAIT_INT	Con AAIT_EXT
P. de Sistema	12.0	2.0	2.3	3.5
P. de Integración	20.3	2.9	3.8	5.8
P. Unitaria	22.5	3.0	4.1	6.5
Codificación	0.0	141.2	48.4	64.8

Figura 5.13 Tiempo de corrección de defectos en cada fase del PDS.

Un último detalle a mencionar es la repercusión que tiene el utilizar únicamente una ASAT para descubrir defectos relevantes dentro del sistema, la cual radica en el tiempo total que se invierte en la corrección de dichos defectos, siendo este 172.1% mayor al tiempo requerido por un proceso sin ASA, 154.9% mayor a nuestra AAIT_INT y 85.0% mayor a nuestra AAIT_EXT. Creemos que este hecho haría que los equipos de desarrollo se inclinen por un menor tiempo de corrección de defectos a costa de un mayor número de defectos presentes en el sistema, sin embargo, el análisis hecho a lo largo de esta sección sugiere que nuestra AAIT podría atacar ambos tópicos, lo cual es: lograr descubrir un mayor número de defectos relevantes a la funcionalidad del sistema utilizando un tiempo relativamente similar para la corrección de los mismos.

6. CONCLUSIONES Y TRABAJO A FUTURO

Uno de los aspectos más importantes que se deben considerar para la producción de sistemas de software de alta calidad es la utilización de herramientas o técnicas que permitan descubrir el mayor número posible de defectos dentro del código fuente con el fin de corregirlos antes de llegar a fases posteriores del Proceso de Desarrollo de Software (PDS), siendo un ejemplo de esto las Herramientas de Análisis Estático Automatizado (ASAT, *Automated Static Analysis Tool*) y las Técnicas de Identificación de Alertas Accionables (AAIT, *Alert Actionable Identification Technique*). Es por esta razón que, mediante el presente trabajo de investigación, desarrollamos una propuesta para minimizar el número de falsos positivos por reparar y maximizar el número de defectos relevantes descubiertos utilizando un mínimo de esfuerzo, es decir, utilizando la menor cantidad de tiempo posible para su corrección.

Presentamos el diseño y la evaluación de una AAIT que utiliza un conjunto de algoritmos de aprendizaje maquinal para generar múltiples modelos internos y externos de clasificación de nuevas alertas a través de una metodología de 3 etapas, las cuales son: (1) generación de versiones, (2) generación de características de alerta y (3) generación de modelos. Posteriormente, realizamos una evaluación comparativa del desempeño logrado por nuestros modelos contra otros modelos ya existentes. Por último, planteamos la utilización de estos modelos dentro del PDS mediante un análisis de su impacto en el número de defectos relevantes descubiertos y en el número de alertas reparadas, además de su tiempo de corrección. Todas estas aportaciones contribuyen al conocimiento acerca de cómo complementar el análisis estático con una AAIT, además de alentar a los equipos de desarrollo a la utilización de este tipo de técnicas en la producción de sus sistemas de software.

Para cada uno de los proyectos de software seleccionados, utilizamos el código fuente compilado (proveniente de sus distintas versiones) para construir un conjunto de 304 y 402 Vectores de Características de Alerta (VCA) para SAPCyTI [47] y JDOM [34] respectivamente, dichos vectores representan alertas de análisis estático (generadas por FindBugs [27]) previamente etiquetadas como accionables o no accionables mediante una heurística de etiquetado [6, 7] y nuestra heurística de reetiquetado basada en tipos. Asimismo, utilizamos la herramienta WEKA [30] para ejecutar 5 métodos de evaluación y 5 estrategias de búsqueda sobre dichos conjuntos para definir entre 9 y 14 subconjuntos de Características de Alerta (CA) relevantes a la clasificación y, con base en ellos, entrenar 60 algoritmos

de aprendizaje maquinal con el fin de evaluar su exactitud, precisión y sensibilidad y de esta manera, seleccionar a los 12 mejores modelos de clasificación.

Derivado de las dos configuraciones de nuestro experimento, dividiremos nuestras conclusiones en 4 partes, las cuales son: (1) aquellas que muestran la capacidad máxima que nuestra técnica ofrece al construir y evaluar modelos internos de clasificación de nuevas alertas utilizando conjuntos de vectores compuestos por 46 características y (2) aquellas que muestran la construcción y evaluación de modelos externos utilizando conjuntos de VCA que no incorporan características de temporalidad, (3) aquellas que generalizan las observaciones realizadas en las primeras dos y (4) aquellas que muestran el impacto de utilizar dichos modelos para el descubrimiento de nuevos defectos una vez que nuestra técnica es incorporada al PDS.

Modelos internos.

Dentro de los 11 y 9 subconjuntos de CA seleccionados para SAPCyTI y JDOM respectivamente, la estrategia BestFirst [32, 22] logró combinarse con todos los métodos de evaluación y el método ClassifierSubsetEval [32, 22] logró combinarse con todas las estrategias de búsqueda. Sobre las características seleccionadas, 6 no tuvieron presencia alguna en los subconjuntos (por ejemplo, la complejidad ciclomática, el número total de alertas accionables en la versión y el tamaño del archivo), 24 demostraron ser propias del proyecto (tal es el caso del nombre del artefacto a nivel de método, clase, archivo, paquete y proyecto) y 18 resultaron ser comunes a ambos proyectos, donde el tiempo de vida de la alerta se posicionó como la CA más relevante en la accionabilidad de una alerta, resultado que concuerda con trabajos anteriores [6, 7, 11, 19]. Dichas observaciones nos permiten sustentar que existen CA (ya sean obtenidas del análisis estático o del código que la rodea) que son en mayor o menor medida predictivas de la accionabilidad de una alerta.

Los resultados obtenidos por nuestra AAIT han demostrado la capacidad máxima que nuestros modelos pueden ofrecer en la clasificación de nuevas alertas, logrando un desempeño mayor al 90% en términos de exactitud, precisión y sensibilidad para todos los casos y un desempeño promedio del 93.9% y 92.7% para los conjuntos de mejores modelos de SAPCyTI y JDOM respectivamente. Dichos resultados son valores aceptables e incluso superiores a los reportados en la literatura [6, 7, 8, 11, 19].

Para el caso de SAPCyTI, el desempeño promedio de los 5 primeros modelos es superior al 95.3%, siendo el mejor de ellos generado por el algoritmo Árbol AD [32, 22] con una exactitud, precisión y sensibilidad del 96.4%, 98.5% y 94.2% respectivamente, Árbol LAD y Árbol LM [32, 22]

también son capaces de lograr buenos resultados. Para JDOM, tan sólo dos modelos alcanzaron un desempeño promedio superior al 93.5% mediante los algoritmos Bosques Aleatorios y Comité Aleatorio [32, 22], donde el primero de ellos obtuvo el 93.9%, 95.7% y 93.0% en exactitud, precisión y sensibilidad respectivamente. En general, 7 modelos coincidieron en ambos proyectos (Árbol AD, k-NN, Árbol LAD, Árbol LM, Listas de Decisión, Clasificador Ordinal y Bosques Aleatorios [32, 22]) y, a diferencia de Heckman y Williams [6, 7], esto nos permite decir que dichos modelos podrían compartir propiedades que pueden ser llevadas de un proyecto a otro.

Adicionalmente, hemos comparado el desempeño logrado por nuestra AAIT contra las técnicas desarrolladas en dos trabajos similares, de lo cual hemos observado una mejora del 1.4% de nuestra técnica aplicada a JDOM bajo las condiciones del experimento de Heckman y Williams [7] y una mejora del 5.5% sobre la exactitud promedio de nuestro conjunto de mejores modelos. Respecto al trabajo de Yüksel y Sözer [11], la diferencia se vuelve más notoria al observar la precisión y sensibilidad de sus modelos en comparación a la de los nuestros, registrando un aumento del 7.3% bajo las condiciones de estos autores y del 8.5% respecto a nuestro conjunto de mejores modelos. Por lo anterior, podemos concluir que la incorporación de una heurística de reetiquetado de alertas incrementa el desempeño de los modelos generados mediante nuestra técnica respecto al desempeño logrado por los modelos de técnicas ya existentes, en particular la de Heckman y Williams [7].

Modelos externos.

Dentro de los 14 y 12 subconjuntos de CA seleccionados para SAPCyTI y JDOM, las estrategias BestFirst y GeneticSearch [32, 22] lograron combinarse con todos los métodos de evaluación y el método ClassifierSubsetEval [32, 22] se combinó con todas las estrategias de búsqueda. Todas las características, excepto el número de métodos en el archivo, fueron seleccionadas al menos una vez, de las cuales 18 son comunes a ambos proyectos (por ejemplo, el tipo de alerta, el número de línea de la alerta, el tamaño del método y el número de alertas en el método) y 7 son características propias del proyecto (como la complejidad ciclomática, el tamaño del proyecto y la prioridad de la alerta). De lo anterior, observamos que el 69.2% de las CA son comunes a ambos proyectos, situación que sustenta la construcción de modelos basados en la información de JDOM y su evaluación bajo las alertas de SAPCyTI con el fin de comprobar la hipótesis planteada en este estudio.

Para este enfoque, los resultados obtenidos pueden ofrecer una clasificación de alertas con un desempeño promedio de hasta un 71.1% para modelos externos a SAPCyTI y del 65.4% para aquellos que sean externos a JDOM. El mejor modelo para SAPCyTI alcanzó un desempeño

promedio superior al 70.0% (67.6% de exactitud, 63.8% de precisión y 81.8% de sensibilidad) mediante el algoritmo Empaquetado [32, 22]. Por otra parte, el mejor modelo externo a JDOM fue generado mediante el algoritmo k-NN [32, 22], el cual logró un desempeño promedio del 63.9% (61.1% de exactitud, 64.6% de precisión y 66.1% de sensibilidad).

En general, 8 modelos coincidieron en ambos proyectos (Clasificación Vía Regresión, Árbol de Hoeffding, k-NN, Perceptrón Multicapa Optimizados, Árbol Aleatorio, Regresión Logística Lineal, Clasificador Multiclase y Umbral de Punto Medio [32, 22]). De lo anterior, observamos que el 66.7% de los algoritmos utilizados para construir estos modelos coinciden entre sí, lo cual indica que existen modelos que clasifican alertas con base en CA semejantes e independientes al proyecto.

De manera similar a los modelos internos, hemos comparado el desempeño logrado por nuestros modelos externos contra la técnica propuesta por Hanam *et al.* [9, 10], en la cual se reporta un desempeño promedio (precisión y sensibilidad) del 48.5%, 60.0% y 16.3% para 3 proyectos de software. Nuestros modelos (bajo las condiciones de dichos autores) logran un desempeño promedio del 50.1% y 58.6% para SAPCyTI y JDOM respectivamente, contrario a lo que logramos mediante nuestro conjunto de mejores modelos (68.3% y 70.4% en SAPCyTI y JDOM). Es así como evidenciamos la competitividad de nuestra AAIT basada en modelos externos dentro de esta área del conocimiento, a pesar de no contar con desempeños tan buenos como los obtenidos por nuestra técnica basada en modelos internos.

Conclusiones generales.

Es evidente que existe una diferencia entre utilizar modelos internos y externos al proyecto para la clasificación de nuevas alertas en cuanto a desempeño promedio se refiere, causando disminuciones de hasta un 25.3% y 30.3% para los mejores modelos de SAPCyTI y JDOM respectivamente. A pesar de que es claro que dicho comportamiento es provocado por la omisión de aquellas CA con mayor grado de aportación a la clasificación de alertas (por ejemplo, el tiempo de vida de la alerta, la edad y la ranciedad del archivo), podemos concluir que no solamente se trata de esto, sino también de otros posibles factores que perjudican el desempeño de nuestros modelos, las cuales son:

1. El estilo de programación del equipo de desarrollo, el dominio de aplicación, las reglas de negocio, los atributos de calidad, la(s) metodología(s) y la(s) herramienta(s) utilizadas son aspectos que seguramente condicionan el tipo de defectos introducidos, situación que hace únicas a algunas características de cada uno de los VCA.
2. Nuestra heurística de reetiquetado de alertas únicamente tiene en cuenta el tipo de la alerta y evade las demás características, otorgándole total relevancia y dando lugar a posibles

perturbaciones en el aprendizaje de cada algoritmo al etiquetar una alerta como accionable cuando la mayoría de las características respaldan la no accionabilidad de dicha alerta.

3. Existen algoritmos que aprenden mejor con atributos numéricos, tal es el caso de los metaclasificadores y de los algoritmos lineales (funciones), y otros que se desempeñan mejor con atributos nominales, como los algoritmos de árboles de decisión y los algoritmos basados en reglas. En principio, esta situación hace difícil la generación de un modelo que se adapte a todo tipo de atributos y que además logre un alto desempeño al momento de llevarlo de un proyecto a otro, sin embargo, hemos descubierto que el algoritmo k-NN [32, 22] es capaz de adaptarse y desempeñarse satisfactoriamente en ambos escenarios
4. Contrario a la afirmación hecha por Heckman y Williams [6, 7], concluimos que las versiones tipo *major* o *minor* y las revisiones de un proyecto no pueden ser utilizadas de manera indistinta. Mientras que una revisión describe aquellas modificaciones al código fuente encargadas de corregir una gran cantidad de defectos, una versión es un componente que ya ha eliminado gran parte de los defectos más relevantes del sistema. Entonces, mientras que las revisiones serían capaces de proporcionar una gran variedad de defectos relevantes, las versiones seguramente sólo proporcionarán una gran cantidad de defectos irrelevantes a la correcta funcionalidad del sistema.

Con base en los resultados obtenidos por nuestros conjuntos de mejores modelos (ya sean internos o externos), podemos concluir que el uso de las AAIT puede ser un enfoque viable para la clasificación precisa de nuevas alertas, además de que la incorporación de nuestra heurística de reetiquetado de alertas, permite lograr un incremento en el desempeño de cada modelo de clasificación, superando a los resultados obtenidos en trabajos anteriores (como el de Heckman y Williams [7]).

Conclusiones sobre el impacto de las AAIT sobre el PDS.

Utilizamos el mejor modelo interno (basado en el algoritmo Árbol AD [32, 22]) y el mejor modelo externo (Empaquetado [32, 22]) generados para el proyecto SAPCyTI con el fin de analizar el impacto que producirían al ser incorporados al PDS. Para esto, planteamos la existencia (dentro del código fuente) de 100 defectos relevantes que perjudican la correcta funcionalidad del sistema y, con base en los valores de precisión y sensibilidad de nuestros modelos, podemos concluir lo siguiente:

1. Un PDS que sólo se apoye en las distintas etapas de la fase de pruebas (es decir, que no incorpora análisis estático dentro del desarrollo) para descubrir tan sólo el 63.0% de los defectos introducidos en el código fuente, será menos eficaz que un proceso que sí lo haga, en cuyo caso

una ASAT permitiría descubrir el 96.0%, una AAIT basada en un modelo interno lograría el 93.0% y una AAIT externa el 90.0%. Concluimos que no utilizar análisis estático en el PDS provocaría una disminución en la eficiencia de eliminación de defectos de hasta un 33.0%.

2. Es innegable que una ASAT es capaz de descubrir una mayor cantidad de defectos respecto a una AAIT, sin embargo, existen otros factores a considerar dentro de esta comparativa. Hemos definido y analizado la relación que existe entre el número de alertas que el desarrollador debe reparar antes de lograr descubrir cada defecto relevante del sistema. Los resultados muestran que, cuando se utiliza únicamente una ASAT, el desarrollador debería reparar al menos 139 alertas antes de lograr descubrir 50 defectos relevantes en contraste a nuestra AAIT interna o externa, en las cuales sólo tendrían que repararse 51 y 78 alertas respectivamente.
3. Derivado de lo anterior, el tiempo total invertido en la corrección de los defectos relevantes luego de las fases de codificación y pruebas sería de: 54.8 horas en un PDS sin análisis estático, 149.1 horas utilizando una ASAT, 58.5 y 80.6 horas a través de nuestra AAIT interna y externa respectivamente. Al respecto podemos decir que, la inversión de un tiempo similar para corregir un mayor número de defectos podría motivar a que los equipos de desarrollo eligieran utilizar nuestra técnica por encima de utilizar únicamente una ASAT.

Los 3 puntos anteriores nos hacen concluir que: es posible aumentar el número de defectos relevantes descubiertos y disminuir el número de alertas irrelevantes por reparar antes de llegar a la fase de pruebas mediante la utilización de un modelo externo para la clasificación de nuevas alertas, lo cual valida la hipótesis planteada en la sección 1.5.

Hasta ahora, los estudios anteriores sólo han abordado las AAIT bajo la perspectiva de modelos internos al proyecto, utilizando características de temporalidad y sin tomar en cuenta el tipo de alerta inspeccionada. Por ello, creemos que la propuesta presentada a lo largo de este estudio contribuye al estado del conocimiento en cuanto a AAIT se refiere y que, a través de nuestro análisis cuantitativo, es posible motivar a los equipos de desarrollo en la utilización de este tipo de técnicas para la producción de sistemas de software.

6.1. Trabajo a futuro

A lo largo de este estudio, hemos identificado algunos puntos de mejora que podrían perfeccionar nuestra AAIT, los cuales están enfocados en intentar aumentar y equilibrar la exactitud, precisión y sensibilidad de nuestros modelos externos de clasificación. Como vimos en la sección 5.1.3,

una versión (*major* o *minor*) es un componente que ya ha descubierto y corregido la mayor cantidad posible de los defectos más relevantes del sistema y por tanto, podría contener una gran cantidad de defectos irrelevantes. Por ello, planeamos extender nuestra técnica al uso de únicamente revisiones del proyecto porque creemos que dichas revisiones permitirían contar con una mayor variedad de defectos relevantes dentro del conjunto de entrenamiento.

Uno de los resultados más importantes de este trabajo fue el decreciente desempeño obtenido por nuestros modelos externos respecto a los modelos internos una vez que se omitieron las diferentes características de temporalidad, además de observar una gran variabilidad en las características seleccionadas. Con el fin de atender esta problemática, nos hemos planteado la posibilidad de incrementar nuestro conjunto de CA y enriquecerlo con la selección de un mayor número de métricas de software, ya no sólo la complejidad ciclomática o el tamaño del artefacto, sino también métricas orientadas a objetos (cohesión y acoplamiento), orientadas a clases (profundidad del árbol de herencia y número de hijos) y orientadas a operaciones (el número de operadores y el número de operandos) [38], por mencionar algunas. Es mencionar que, gran parte de estas métricas también pueden ser extraídas desde la herramienta Metrics 1.6.0 [42] utilizada a lo largo de este estudio (véase sección 4.2) o bien, desde JavaNCSS 21.41 [56] (utilizada en el trabajo de Heckman y Williams [6, 7]) y en ambos casos, con un mínimo de tiempo y esfuerzo; sin embargo, la investigación de nuevas herramientas capaces de calcular la mayor cantidad posible de dichas métricas (por ejemplo, JHawk 6.1.3 [58]) plantea uno de los tópicos a resolver para enriquecer nuestro conjunto de CA.

Continuando con la idea anterior, sabemos que el objetivo principal de cada una de las métricas antes mencionadas es evaluar cuándo un método del sistema es propenso a fallas y cuándo no lo es [38]. Con esto en mente, proponemos utilizar algunos de los conjuntos de VCA almacenados en el repositorio de ingeniería de software PROMISE [59] para retomar el trabajo de [60] y adaptarlo a nuestra AAIT con el fin de verificar la siguiente idea: si un método es susceptible a causar fallas dentro de un sistema, entonces todas las alertas contenidas en dicho método deben ser etiquetadas como alertas accionables. Este hecho constituye un precedente hacia una nueva heurística de etiquetado de alertas y cuyo impacto a nuestra técnica podría ser un conjunto de VCA cuyo desempeño sea mayor al logrado por nuestros modelos hasta este momento. Cabe mencionar que, los conjuntos de datos antes propuestos caracterizan y clasifican (como defectuosos o no defectuosos) a cada uno de los métodos que componen a distintos proyectos de software de la NASA (*National Aeronautics and Space Administraion*).

También, debemos mencionar que nuestra técnica y el trabajo a futuro antes sugerido deben ser ampliados a la exploración del código fuente de otros proyectos bajo diferentes dominios de aplicación, arquitecturas de software, metodologías y herramientas de desarrollo, esto con el fin de estudiar las posibles implicaciones que nuestra AAIT causaría dentro de diversos contextos de desarrollo. Desafortunadamente, un estudio como este presentaría una serie de desafíos, tal es el caso del tiempo que tomaría la recopilación de la información necesaria para realizar el análisis. Sabemos que la gran mayoría de los proyectos de software desarrollados a nivel profesional son grandes y evolucionan de manera constante a través del tiempo, esto hace que las correcciones a las alertas de análisis estático causen un impacto hasta unos meses o incluso unos años después de haberlas realizado. Con base en esto, proponemos realizar dicho estudio sobre un proyecto de software pequeño, lo cual podría proporcionar una idea clara acerca del impacto de nuestra AAIT sobre el PDS.

Por último, una de las cuestiones que deben ser abordadas con el fin de comprobar la eficacia de nuestros modelos en la clasificación de nuevas alertas dentro de un caso real (ya no sólo hipotético) es la validación de nuestras clasificaciones con el equipo de desarrollo. Teniendo en cuenta que ya se ha seleccionado un conjunto de mejores modelos externos de clasificación y que el equipo de desarrollo cuenta con el código fuente compilado de la última versión de SAPCyTI, dicha validación podría ser realizada de la siguiente manera:

1. Ejecutar la ASAT sobre la última versión de SAPCyTI y aplicar el proceso de generación de CA (véase la figura 3.4 de la sección 3.2), esto con el objetivo de obtener el conjunto de VCA que describa a cada una de las alertas reportadas por la herramienta.
2. Presentar este conjunto de alertas al equipo de desarrolladores, quienes deben inspeccionar cada una de ellas y, de acuerdo a su experiencia, deben determinar la accionabilidad de cada alerta; aclaramos, este proceso debe realizarse de manera manual y podría ser que la recolección de la información tome demasiado tiempo, sin embargo, al final se tendrá un nuevo conjunto de VCA cuyas alertas se encuentren etiquetadas como accionables o no accionables según sea el caso.
3. Ejecutar nuestros modelos externos sobre este nuevo conjunto de VCA mediante la herramienta WEKA y con base en las predicciones realizadas, determinar el desempeño real de cada uno de ellos en términos de exactitud, precisión y sensibilidad.

Lo anterior, permitiría conocer aquellos modelos que mejor se adecúen a las necesidades del equipo de desarrollo, además de verificar el grado de confiabilidad de nuestra técnica en la clasificación de nuevas alertas para un caso real.

A. APÉNDICE

FindBugs [27] es una herramienta que utiliza análisis estático para realizar la búsqueda de posibles defectos dentro del código fuente Java [20] a través de más de 426 tipos de alertas, los cuales describen patrones de código bien definidos que con frecuencia reflejan fallos en el sistema. Dichos tipos se encuentran clasificados en 9 categorías y priorizados de acuerdo a su nivel de gravedad; como podrá verse, en la tabla A.1 se describen los tipos de alerta utilizados durante este estudio

Por otro lado, los métodos de evaluación valoran una serie de subconjuntos de características mediante el cálculo de un valor numérico, el cual representa qué tan predictivo es dicho subconjunto en la clasificación de nuevas instancias [32, 22]; en general, estos métodos nos ayudan a seleccionar subconjuntos de características relevantes en la accionabilidad de una alerta. La tabla A.2 describe cada uno de los métodos de evaluación de atributos utilizados a lo largo de este estudio.

Para que lo anterior se cumpla, WEKA [30] proporciona un conjunto de estrategias de búsqueda que se encargan de realizar una exploración de las características más relevantes dado un conjunto de datos con el fin de proponer una serie de subconjuntos, los cuales (como ya hemos dicho) habrán de ser valorados mediante un método de evaluación. La tabla A.3 describe cada una de las estrategias de búsqueda utilizadas en este estudio.

Por último, decimos que un modelo es creado cuando se aplica un algoritmo de aprendizaje a un conjunto de datos con el fin de identificar posibles patrones y obtener nuevo conocimiento a partir de dicha información [22]. Por ello, en la tabla A.4 describimos de manera breve cada uno de los algoritmos de aprendizaje utilizados en la etapa de generación de modelos (descrita en la sección 3.2).

Categoría	Tipo	Descripción
Malas prácticas	**ES_COMPARING_ STRINGS_WITH_ EQ**	Compara dos objetos *String* mediante los operadores "==" o "!=". A menos que dichos objetos sean constantes, una misma cadena puede ser representada por dos objetos distintos. Es mejor idea utilizar el método *equals* del objeto.
	OS_OPEN_STREAM_ EXCEPTION_PATH	Crea un flujo de E/S de datos y no es asignado a una variable, enviado o recibido desde un método o bien, cerrado en alguna excepción. Se recomienda definir un bloque final para asegurar el cierre de dicho flujo.
	RV_RETURN_ VALUE_IGNORED_ BAD_PRACTICE	No se verifica el valor devuelto por el método invocado. Este debe comprobarse para evitar un comportamiento inesperado. De no ocurrir, es posible que se obtenga un valor de retorno atípico.
	DM_EXIT	La instrucción apaga la máquina virtual de Java, esto hace imposible llamar a dicho código desde otro. La alternativa es lanzar una *RuntimeException*.
Exactitud	**DMI_BAD_MONTH**	Envía un valor de mes fuera de rango, el valor enviado debe estar entre 0 y 11.
	NP_NULL_PARAM_ DEREF	Llama a un método cuyo parámetro no debe ser nulo o bien, dicho parámetro nunca debe ser desreferenciado.
	RCN_REDUNDANT_ NULLCHECK_ WOULD_HAVE_…	Esta instrucción comprueba si un valor es nulo, sin embargo, el análisis arroja que dicho valor nunca podría ser nulo, haciendo que esta comprobación sea redundante.
Experimental	**OBL_UNSATISFIED_ OBLIGATION_ EXCEPTION_EDGE**	Este método puede fallar al limpiar una secuencia, objeto o recurso que necesite ser limpiado de manera explícita. Se recomienda definir un bloque final para asegurar dicha limpieza.
Código malicioso	**MS_PKGPROTECT**	El modificador de acceso hace vulnerable a la variable. Para protegerla, su modificador de ser cambiado a nivel de paquete.
Concurrencia	**LI_LAZY_INIT_ STATIC**	Contiene una inicialización perezosa no sincronizada, lo cual no garantiza que todos los hilos accedan a un objeto completamente inicializado. Una posible solución es trabajar sobre un campo volátil.
Código desagradable	**BC_UNCONFIRMED _CAST_OF_RETURN_ VALUE**	A pesar de que la lógica del programa garantiza una correcta funcionalidad, no se realiza una verificación del valor devuelto por el método invocado y su correspondiente conversión de tipos.
	ICAST_IDIV_CAST_ TO_DOUBLE	Esta línea realiza una división de enteros para su posterior conversión a *double*. Para evitar pérdida de información, es buena idea realizar dicha conversión antes de realizar la división.
	REC_CATCH_ EXCEPTION	El método utiliza bloques *try-catch* que capturan excepciones de todo tipo. Es recomendable definir de manera explícita el tipo de excepción a capturar dentro del bloque.

Tabla A.1 Tipos de alerta reportados por FindBugs.

Estrategia	Descripción
CfsSubsetEval	Evalúa el poder predictivo de cada característica de forma individual e identifica aquellos subconjuntos con características no redundantes entre sí (o con baja intercorrelación) y altamente correlacionadas con la clase.
ClassifierSubsetEval	Utiliza un algoritmo de aprendizaje para estimar la capacidad predictiva de cada subconjunto de características dado un conjunto de entrenamiento o de prueba.
ConsistencySubsetEval	Encuentra subconjuntos de características cuyos valores de la clase sean consistentes (es decir, accionables o no accionables) dentro del conjunto de entrenamiento. Verifica la homogeneidad de dichas características en concordancia con su accionabilidad.
FilteredSubsetEval	Ejecuta un método de evaluación de forma arbitraria sobre el conjunto de datos de entrenamiento previamente filtrados.
WrapperSubsetEval	Utiliza un algoritmo de aprendizaje para evaluar subconjuntos de características y, mediante validación cruzada, estima la exactitud del modelo generado. Por defecto, el algoritmo utilizado es Árbol C4.5.

Tabla A.2 Métodos de evaluación implementados por WEKA.

Método	Descripción
BestFirst	De manera codiciosa, añade (búsqueda directa) o elimina (búsqueda hacia atrás) características a medida que estas aumentan el poder predictivo del subconjunto. Es capaz de retroceder en caso de que, luego de agregar u omitir características, el subconjunto no presente mejoras predictivas.
GeneticSearch	Utiliza un algoritmo genético simple para encontrar los subconjuntos de las características más relevantes.
GreedyStepwise	Realiza una búsqueda codiciosa similar a la realizada por BestFirst pero sin retroceso, donde la condición de paro radica en dejar de agregar o eliminar características una vez que se produce la primera disminución en el poder predictivo del subconjunto.
RaceSearch	Utiliza un método de evaluación para implementar 4 tipos de búsqueda, los cuales son: selección directa, eliminación hacia atrás, búsqueda de esquemas (mediante búsquedas iteradas se determina si una característica debe ser o no incluida) y búsqueda de priorización (en un inicio, las características son priorizadas para luego competir por los primeros lugares).
RankSearch	Clasifica y ordena las características por su nivel de relevancia (de mayor a menor) y, mediante un método de evaluación, selecciona a los subconjuntos más prometedores de manera creciente, es decir, de menor a mayor número de características (la mejor característica más la siguiente mejor característica).

Tabla A.3 Estrategias de búsqueda implementadas por WEKA.

Categoría	Nombre	Algoritmo	Descripción
Bayesianos	BayesNet	Red de Bayes	Aprende redes bayesianas utilizando los algoritmos K2 o TAN para estimar las tablas de probabilidad condicional. Sólo funciona para características nominales y sin valores faltantes.
	NaiveBayes	Bayes Ingenuo	Implementa un clasificador probabilístico, produciendo estimaciones (en lugar de predicciones) acerca de qué tan probable es que una instancia pertenezca o no a una clase.
Funciones	Kernel Logistic Regression	Regresión Logística de Kernel	Genera un modelo de regresión logística de dos clases minimizando la probabilidad negativa con una penalización cuadrática utilizando el método BFGS (Broyden-Fletcher-Goldfarb-Shanno).
	Logistic	Regresión Logística Multinomial	Construye un modelo de regresión logística multinomial con un estimador de cresta, prediciendo las probabilidades de los diferentes resultados posibles.
	MLP Classifier	Perceptrón Multicapa Optimizado	Entrena un Perceptrón Multicapa con una capa oculta optimizada por WEKA, minimizando la función de pérdida y agregando una penalización a través del método BFGS.
	Multilayer Perceptron	Perceptrón Multicapa	Implementa una red neuronal artificial de 3 capas que se entrena utilizando una función sigmoide y retropropagación.
	Simple Logistic	Regresión Logística Lineal	Genera un modelo de regresión lineal utilizando funciones de regresión simple (LogitBoost) como clasificadores base, las cuales determinan el número de iteraciones a realizar.
Perezosos	IBk	k-NN (k-Vecinos Más Cercanos)	Utiliza la distancia euclidiana normalizada para encontrar la instancia de entrenamiento más cercana a la instancia de prueba. Si varias instancias clasifican como las más cercanas, estas son priorizadas por su distancia a la instancia de prueba.
	KStar	K* (K estrella)	Similar a k-NN, la clase de una nueva instancia se basa en la clase de aquellas instancias de entrenamiento similares a ella de acuerdo a una función de similitud (función de distancia generalizada basada en entropía).
	LWL	Ponderación Local	Con base en los pesos asignados a cada una de las instancias mediante un algoritmo basado en instancias, utiliza Bayes Ingenuo para generar un clasificador.

Metaclasi-ficadores	Bagging	Empaque-tado	Empaqueta los resultados de un clasificador base con el fin de reducir la varianza. Realiza un promedio de las estimaciones de probabilidad de los miembros del conjunto
	Classification Via Regression	Clasificación Vía Regresión	Realiza la clasificación utilizando métodos de regresión. El clasificador hace binaria a la clase y construye un modelo para cada valor de dicha clase.
	MultiClass Classifier	Clasificador Multiclase	Utiliza un clasificador base de dos clases (Reg. Log. Multinomial) para trabajar sobre conjuntos de datos con varias clases utilizando uno de estos métodos: uno contra el resto, clasificación por pares mediante votación o códigos exhaustivos (o seleccionados al azar) de corrección de errores.
	Ordinal Class Classifier	Clasificador Ordinal	Aplica algoritmos de aprendizaje estándar a conjuntos de datos cuya clase sea ordinal. Utiliza a C4.5 por defecto.
	Random Committee	Comité Aleatorio	Construye un conjunto de clasificadores base, los cuales son construidos utilizando una semilla de número aleatorio diferente. La predicción final es un promedio de las predicciones de cada clasificador.
	Real AdaBoost	Real AdaBoost	Utiliza el método Real AdaBoost (derivado del algoritmo AdaBoost), para aumentar el desempeño de un clasificador base de dos clases (árboles de decisión).
	Threshold Selector	Umbral de Punto Medio	Optimiza el desempeño de un clasificador base (exactitud, precisión, sensibilidad, entre otros) mediante la definición de un umbral de probabilidad a la salida del clasificador.
Reglas	Conjunctive Rule	Reglas Conjuntivas	Implementa un clasificador de reglas conjuntivas con poda de reducción de error (retener una porción de los datos originales para estimar el error de cada regla). El antecedente serán las características y el consecuente será la clase.
	Decision Table	Tablas de Decisión	Crea un clasificador basado en tablas de decisión.
	DTNB	Híbrido DT-NB	Crea un clasificador híbrido Tablas de Decisión (DT)-Bayes Ingenuo (NB). Utiliza una búsqueda de selección directa donde las características seleccionadas son modeladas por NB y el resto por DT.
	JRip	Reglas Propoci-sionales	Construye un conjunto de reglas que luego es reducido y optimizado mediante una poda de reducción de error. Se trata de una implementación del algoritmo RIPPER.
	PART	Listas de Decisión	En cada iteración, crea un árbol parcial basado en C4.5 y convierte al mejor nodo en una regla (lista de decisiones).
	RIDOR	Reglas Ripple-Down	Define una regla y, mediante poda de reducción de error, encuentra excepciones de la regla con las menores tasas de error. Dichas excepciones establecen un conjunto de reglas.

Árboles	ADTree	Árbol AD	Genera un árbol de decisión alternativo utilizando un algoritmo de refuerzo (boosting). En cada iteración se añaden tres nodos, uno de decisión y dos de predicción (basados en las reglas producidas por el algoritmo de refuerzo).
	BFTree	Árbol Best-First	Mientras que C4.5 expande sus nodos de acuerdo a la profundidad, este clasificador expande primero al "mejor" nodo cuya división (de entre todos los nodos disponibles) conduzca a una reducción máxima del índice de Gini.
	Hoeffding Tree	Árbol de Hoeffding	Basado en el límite de Hoeffding, determina el conjunto de instancias necesarias para elegir el nodo cuya división sea óptima, las predicciones de dichos nodos son realizadas mediante Bayes Ingenuo.
	J48	Árbol C4.5	Basado en el algoritmo C4.5, construye un árbol de decisión (podado o sin podar) utilizando el concepto de entropía de información.
	LADTree	Árbol LAD	Similar a Árbol AD, construye un árbol de decisión alternativo con LogitBoost como algoritmo de refuerzo.
	LMT	Árbol LM	Emplea un mecanismo de poda de costo-complejidad mínimo para construir un árbol logístico de clasificación cuyos nodos contienen funciones de regresión logística (estas pueden ser configuradas).
	NBTree	Árbol NB	Genera un árbol híbrido en cuyos nodos se utiliza validación cruzada para decidir si estos deben ser divididos o si se deben ser clasificados mediante Bayes Ingenuo.
	Random Forest	Bosques Aleatorios	Construye un bosque de árboles aleatorios, para ello, se entrenan múltiples árboles de decisión de forma independiente y se obtiene el promedio del desempeño de los mismos. En esencia, es un clasificador de empaquetado.
	RandomTree	Árbol Aleatorio	Para construir el árbol, considera un número determinado de características elegidas de manera aleatoria en cada nodo.
	REPTree	Árbol REP	Utiliza ganancia de información y poda de reducción de error para construir un árbol de regresión.

Tabla A.4 Algoritmos de aprendizaje maquinal implementados por WEKA.

BIBLIOGRAFÍA

[1] «Systems and software engineering-vocabulary,» *ISO/IEC/IEEE 24765:2010(E),* pp. 1-418, 2010.

[2] C. Jones, Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies, 2 ed., The McGraw-Hill Companies, 2010.

[3] C. Jones, Applied Software Measurement: Global Analysis of Productivity and Quality, 3 ed., The McGraw-Hill Companies, 2008.

[4] S. Heckman y L. Williams, «A Systematic Literature Review of Actionable Alert Identification Techniques for Automated Static Code Analysis,» *Information and Software Technology,* vol. 53, nº 4, pp. 161-170, 2011.

[5] S. Heckman y L. Williams, «On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques,» *2nd International Symposium on Empirical Software Engineering and Measurement (ESEM '08),* pp. 41-50, 2008.

[6] S. S. Heckman, «A Systematic Model Building Process for Predicting Actionable Static Analysis Alerts,» *Dissertation Computer Science,* 2009.

[7] S. Heckman y L. Williams, «A Model Building Process for Identifying Actionable Static Analysis Alerts,» *2nd International Conference on Software Testing Verification and Validation (ICST '09),* pp. 161-170, 2009.

[8] S. Heckman y L. Williams, «A Comparative Evaluation of Static Analysis Actionable Alert Identification Techniques,» *9th International Conference on Predictive Models in Software Engineering (PROMISE '13),* pp. 1-10, 2013.

[9] Q. Hanam, L. Tan, R. Holmes y P. Lam, «Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking,» *11th Working Conference on Mining Software Repositories (MSR '14),* pp. 152-161, 2014.

[10] Q. Hanam, «Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking,» *M.S. thesis,* 2014.

[11] U. Yüksel y H. Sözer, «Automated Classification of Static Code Analysis Alerts: A Case Study,» *29th International Conference on Software Maintenance (ICSM '13),* pp. 532-535, 2013.

[12] A. Vetro', M. Torchiano y M. Morisio, «Assessing the Precision of FindBugs by mining Java Projects developed at a University,» *7th Working Conference on Mining Software Repositories (MSR '10),* pp. 110-113, 2010.

[13] D. Falessi y A. Voegele, «Validating and Prioritizing Quality Rules for Managing Technical Debt: An Industrial Case Study,» *7th International Workshop on Managing Technical Debt (MTD '15),* pp. 41-48, 2015.

[14] V. Research, Automated Defect Prevention for Embedded Software Quality, Parasoft, 2012.

[15] S. Wagner, E. Weilemann y J. P. Ostberg, «Does Personality Influence the Usage of Static Analysis Tools? An Explorative Experiment,» *9th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '16),* pp. 75-81, 2016.

[16] S. Allier, N. Anquetil, A. Hora y S. Ducasse, «A Framework to Compare Alert Ranking Algorithms,» *19th Working Conference on Reverse Engineering (WCRE '12),* pp. 277-285, 2012.

[17] H. Shen, J. Fang y J. Zhao, «EFindBugs: Effective Error Ranking for FindBugs,» *14th International Conference on Software Testing, Verification and Validation (ICST '11),* pp. 299-308, 2011.

[18] J. P. Ostberg, J. Ramadani y S. Wagner, «A Novel Approach for Discovering Barriers in Using Automatic Static Analysis,» *17th International Conference on Evaluation and Assessment in Software Engineering (EASE '13),* pp. 78-81, 2013.

[19] U. Yüksel, H. Sözer y M. Sensoy, «Trust-based Fusion of Classifiers for Static Code Analysis,» *17th International Conference on Information Fusion (FUSION '14),* pp. 1-6, 2014.

[20] «Java,» [En línea]. Available: https://www.java.com/es/. [Último acceso: 9 Abril 2017].

[21] «TIOBE Index - The Software Quality Company,» [En línea]. Available: http://www.tiobe.com/tiobe-index/. [Último acceso: 1 Febrero 2017].

[22] I. H. Witten y E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, 2 ed., Morgan Kaufmann Publishers, 2005.

[23] I. Moriggl, «Intelligent Code Inspection using Static Code Features: An approach for Java,» *M.S. thesis,* 2010.

[24] H. Tribus, I. Moriggl y S. Axelsson, «Using Data Mining for Static Code Analysis of C,» *8th International Conference on Advanced Data Mining and Applications (ADMA '12),* pp. 603-614, 2012.

[25] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie y H. Mei, «Automatic Construction of an Effective Training Set for Prioritizing Static Analysis Warnings,» *25th International Conference on Automated Software Engineering (ASE '10),* pp. 93-102, 2010.

[26] T. Preston-Werner, «Semantic Versioning,» [En línea]. Available: http://semver.org/. [Último acceso: 9 Abril 2017].

[27] «FindBugs - Find Bugs in Java Programs,» [En línea]. Available: http://findbugs.sourceforge.net/. [Último acceso: 9 Abril 2017].

[28] D. Hovemeyer y W. Pugh, «Finding Bugs is Easy,» *19th Conference on Object Oriented Programming Systems and Applications (OOPSLA '04),* pp. 132-136, 2004.

[29] J. Yoon, M. Jin y Y. Jung, «Reducing False Alarms from an Industrial-Strength Static Analyzer by SVM,» *21st Asia-Pacific Software Engineering Conference (APSEC '14),* pp. 3-6, 2014.

[30] «Weka 3 - Data Mining Software in Java,» [En línea]. Available: http://www.cs.waikato.ac.nz/ml/weka/index.html/. [Último acceso: 9 Abril 2017].

[31] «The University of Waikato - Te Ware Wananga o Waikato,» [En línea]. Available: http://www.waikato.ac.nz/. [Último acceso: 9 Abril 2017].

[32] E. Frank, M. A. Hall y I. H. Witten, The WEKA Workbench. Online Appendix for Data Mining: Practical Machine Learning Tools and Techniques, 4 ed., Morgan Kaufmann Publishers, 2016.

[33] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann y I. H. Witten, «The WEKA Data Mining Software: An Update,» *SIGKDD Explorations Newsletter,* vol. 11, nº 1, pp. 10-18, 2009.

[34] «The JDOM Project,» [En línea]. Available: http://www.jdom.org/. [Último acceso: 9 Abril 2017].

[35] «The Eclipse Foundation open source community website,» [En línea]. Available: http://www.eclipse.org/. [Último acceso: 9 Abril 2017].

[36] «WALA Project,» [En línea]. Available: https://sourceforge.net/projects/wala/. [Último acceso: 9 Abril 2017].

[37] H. Cervantes Maceda, P. Velasco Elizondo y L. Castro Careaga, Arquitectura de Software. Conceptos y ciclo de desarrollo, 1 ed., Cengage Learning Editores, 2016.

[38] R. S. Pressman, Software Engineering: A Practitioner's Approach, 7 ed., The McGraw-Hill Companies, 2010.

[39] I. Somerville, Ingeniería de Software, 9 ed., Pearson Educación de México, 2011.

[40] «Apache Subversion - Enterprise class centralized version control for the masses,» [En línea]. Available: https://subversion.apache.org/. [Último acceso: 9 Abril 2017].

[41] «TortoiseCVS - Enjoyable Version Control,» [En línea]. Available: http://www.tortoisecvs.org/. [Último acceso: 9 Abril 2017].

[42] «Metrics - Getting started,» [En línea]. Available: http://metrics.sourceforge.net/. [Último acceso: 9 Abril 2017].

[43] M. Sharma, P. Bedi, K. K. Chaturvedi y V. B. Singh, «Predicting the Priority of a Reported Bug Using Machine Learning Techniques and Cross Project Validation,» *12th International Conference on Intelligent Systems Design and Applications (ISDA '12)*, pp. 539-545, 2012.

[44] T. J. McCabe, «A Complexity Measure,» *IEEE Transactions on Software Engineering*, vol. 2, nº 4, pp. 308-320, 1976.

[45] R. R. Bouckaert, E. Frank, M. Hall, R. Kirkby, P. Reutemann, A. Seewald y D. Scuse, WEKA Manual, vol. 3, GNU General Public License, 2016.

[46] «CSV2ARFF - Online converter from .csv to WEKA .arff,» [En línea]. Available: http://ikuz.eu/csv2arff/. [Último acceso: 9 Abril 2017].

[47] «Sistema de Administración de Posgrado en Ciencias y Tecnologías de la Información,» Universidad Autónoma Metropolitana (Unidad Iztapalapa), [En línea]. Available: http://paoscalu.izt.uam.mx/. [Último acceso: 9 Abril 2017].

[48] «Posgrado en Ciencias y Tecnologías de la Información,» Universidad Autónoma Metropolitana (Unidad Iztapalapa), [En línea]. Available: https://paoscalu.izt.uam.mx/sapcyti/login/. [Último acceso: 9 Abril 2017].

[49] «Spring Framework,» [En línea]. Available: https://spring.io/. [Último acceso: 9 Abril 2017].

[50] «Software Engineering Research Group - FaultBench,» [En línea]. Available: http://www.realsearchgroup.org/faultbench/. [Último acceso: 9 Abril 2017].

[51] «Ubuntu - The leading operating system for PCs,» [En línea]. Available: https://www.ubuntu.com/. [Último acceso: 9 Abril 2017].

[52] «Centro de Desarrollo y de Descarga Java,» [En línea]. Available: http://www.oracle.com/technetwork/es/java/javase/overview/index.html/. [Último acceso: 9 Abril 2017].

[53] «Apache Maven Project,» [En línea]. Available: http://maven.apache.org/. [Último acceso: 9 Abril 2017].

[54] «Spring Tool Suite - STS,» [En línea]. Available: https://spring.io/tools/sts/. [Último acceso: 9 Abril 2017].

[55] «Open Source Software Engineering Tools - Subclipse,» [En línea]. Available: http://subclipse.tigris.org/. [Último acceso: 9 Abril 2017].

[56] «JavaNCSS - A Source Measurement Suite for Java,» [En línea]. Available: http://www.kclee.de/clemens/java/javancss/. [Último acceso: 9 Abril 2017].

[57] T. Menzies, W. Nichols, F. Shull y L. Layman, «Are Delayed Issues Harder to Resolve? Revisiting Cost-to-Fix of Defects throughout the Lifecycle,» *Empirical Software Engineering*, vol. 22, nº 4, pp. 1903-1935, 2016.

[58] «Virtual Machinery - JHawk,» [En línea]. Available: http://www.virtualmachinery.com/jhdownload.htm/. [Último acceso: 9 Abril 2017].

[59] «PROMISE Software Engineering Repository,» [En línea]. Available: http://promise.site.uottawa.ca/SERepository/. [Último acceso: 9 Abril 2017].

[60] S. Aleem, L. F. Capretz y F. Ahmed, «Benchmarking Machine Learning Techniques for Software Defect Detection,» *International Journal of Software Engineering & Applications (IJSEA '15)*, vol. 6, nº 3, pp. 11-23, 2015.

Casa abierta al tiempo

PC y TI

UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

ANÁLISIS ESTÁTICO DE SOFTWARE

Idónea comunicación de resultados que
para obtener el grado de:
Maestro en Ciencias y Tecnologías de la Información

Presenta:

José Miguel Ortiz Roque

Asesores:

Dr. René Mac Kinney Romero
Ing. Luis Fernando Castro Careaga

Jurado calificador:

Presidente:	Dra. Perla Velasco-Elizondo
Secretario:	Dr. Humberto Cervantes Maceda
Vocal:	Dr. René Mac Kinney Romero

Ciudad de México, Septiembre 2017

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

ACTA DE EXAMEN DE GRADO

No. 00064
Matrícula: 2153805355

ANÁLISIS ESTÁTICO DE SOFTWARE

JOSE MIGUEL ORTIZ ROQUE
ALUMNO

REVISÓ

LIC. JULIO CÉSAR DE LARA ISASSI
DIRECTOR DE SISTEMAS ESCOLARES

En la Ciudad de México, se presentaron a las 12:00 horas del día 8 del mes de septiembre del año 2017 en la Unidad Iztapalapa de la Universidad Autónoma Metropolitana, los suscritos miembros del jurado:

DRA. PERLA INES VELASCO ELIZONDO
DR. RENE MAC KINNEY ROMERO
DR. HUMBERTO GUSTAVO CERVANTES MACEDA

Bajo la Presidencia de la primera 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 TECNOLOGIAS DE LA INFORMACION)

DE: JOSE MIGUEL ORTIZ ROQUE

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

DIRECTOR DE LA DIVISIÓN DE CBI

DR. JOSE GILBERTO CORDOBA HERRERA

PRESIDENTA

DRA. PERLA INES VELASCO ELIZONDO

VOCAL

DR. RENE MAC KINNEY ROMERO

SECRETARIO

DR. HUMBERTO GUSTAVO CERVANTES MACEDA