Predicción de Defectos Usando Métricas Estáticas y de Cambio: un caso de estudio en un Lenguaje Dinámicamente Tipado

Mauro Gullino[1] y Gabriela Robiolo[2]

[1] Facultad Regional Haedo, Universidad Tecnológica Nacional, Buenos Aires, Argentina,
mgullino@frh.utn.edu.ar
[2] LIDTUA (CIC), Facultad de Ingeniería, Universidad Austral, Pilar, Argentina,
grobiolo@austral.edu.ar

Abstract. La utilización de métricas para la predicción de defectos es un importante área de estudio de la Ingeniería de Software. El presente trabajo selecciona un conjunto de métricas estáticas y de cambio con el objetivo de evaluar su utilidad como predictores de defectos. Se presenta un caso de estudio sobre el proyecto de código abierto MediaWiki, que soporta a Wikipedia. Este producto cuenta con 1000 clases y 365 KLOC y está desarrollado con el lenguaje de tipado dinámico PHP, extensamente usado en la industria de desarrollo web. Luego de obtener las métricas para el historial de desarrollo del producto durante un año, lo que representó analizar 2800 commits y crear herramientas específicas, se utilizó regresión logística como método de estimación. Se verifica que la cantidad y tamaño de los cambios de tipo *"Feature Introduction"* realizados a una clase son los mejores predictores de que la misma sea defectuosa para el caso de estudio. Se comprueba que utilizando métricas de cambio es posible obtener mejores resultados en la predicción que con métricas estáticas, pero las primeras representan un mayor esfuerzo de medición.

Keywords: Métricas de Software, Métricas Estáticas, Métricas de Cambio, Predicción de Defectos, Regresión Logística

1 Introducción

Hace décadas se busca a través del estudio de las métricas de software indicios de propiedades internas de éste que resulten interesantes durante el ciclo de vida. Muchas líneas de investigación han buscado la correlación entre estas métricas y la cantidad de defectos encontrados en un artefacto. Una importante *Systematic Literature Review* realizada en 2012 [1] analizó más de 200 trabajos sobre el tema solamente en la última década, lo que pone de manifiesto la vigencia y el interés que despierta la utilidad de las métricas como predictores de defectos.

Se estima [2] que la corrección de defectos en la fase de mantenimiento abarca de un 50% a un 75% del costo total de desarrollo de sistemas. Debido a su significancia económica la búsqueda de predictores confiables de defectos se ha convertido en un importante área de trabajo dentro de la Ingeniería de Software.

Hay consenso en que no existe un juego de métricas que sea un buen predictor para absolutamente todos los proyectos, sino que debe encontrarse el conjunto adecuado a cada uno. Es decir que un conjunto de métricas puede ser un predictor de defectos válido en un proyecto y sin embargo no poder aplicarse con igual éxito en otro. [3]

El objetivo de la presente investigación es definir la o las métricas de software que mejor se comporten como predictores de defectos de una clase utilizando como caso de estudio un producto de código abierto desarrollado en un lenguaje de tipado dinámico. **Pregunta de investigación: ¿Son las métricas de cambio mejores predictores de defectos que las métricas estáticas en un lenguaje de tipado dinámico?**

Se utilizó el proyecto MediaWiki como caso de estudio debido a su característica *open source*, a la disponibilidad del historial de cambios a través del repositorio de versiones y a su innegable calidad y éxito como producto. La instalación más importante de este proyecto es la enciclopedia colaborativa Wikipedia.

MediaWiki está desarrollado con el lenguaje de programación PHP, lo que representa la oportunidad de trabajar con un sistema dinámicamente tipado. La investigación sobre métricas de software y predicción de defectos se ha centrado históricamente en lenguajes de tipado estático (principalmente Java y C++) dejando de lado, de este modo, lenguajes muy extendidos en la industria. Esto plantea la cuestión de si las métricas desarrolladas y estudiadas para lenguajes de tipado estático son útiles en lenguajes de tipado dinámico.

Se utilizó regresión logística siguiendo a los autores principales del área [4] [5] [6]. Este método de estimación permite realizar una clasificación binaria de un artefacto como defectuoso/no-defectuoso a partir de variables independientes cuantitativas que lo describen (métricas).

Este trabajo se estructura de la siguiente forma: en el punto 2 se enumeran las métricas seleccionadas para caracterizar el producto bajo estudio y su proceso de desarrollo, en el punto 3 se analiza el caso de estudio MediaWiki y se presentan los resultados de los modelos de regresión obtenidos, en el punto 4 se enumeran los trabajos relacionados más importantes y en el punto 5 se presentan las conclusiones.

2 Métricas seleccionadas

Las métricas estáticas utilizadas en este trabajo surgen del framework desarrollado por Robiolo [7] y de la suite Chidamber & Kemerer (C&K) [8], muy extendida en el área. Las métricas de cambio siguen el trabajo de Moser [5] y las definiciones de Hassan [9]. Las métricas que no tienen un nombre de uso común han sido numeradas según aparecen en el trabajo original de los autores para facilitar su tratamiento.

2.1 Métricas de sistema

Las métricas de sistema (Tabla 1) se utilizan para caracterizar al producto como un todo y no forman parte de los modelos predictivos. Para éstos se utilizan métricas de clase ya que los predictores buscados apuntan a dicha granularidad.

Tabla 1. Métricas de sistema

Métrica	Descripción	Autor
LOC	*Lines of Code* (líneas de código lógicas totales)	N/D
Rob_1	Cantidad de clases totales	Robiolo
Rob_2	Cantidad de interfaces totales	Robiolo
Rob_3	Cantidad de clases externas extendidas	Robiolo
Rob_4	Cantidad de interfaces externas extendidas	Robiolo
Rob_5	Cantidad de clases que implementan interfaces externas	Robiolo
Rob_6	Cantidad de clases raíz que poseen al menos una subclase	Robiolo
Rob_7	Cantidad de interfaces raíz que son extendidas por otra interfaz	Robiolo
Rob_8	Nivel máximo de profundidad de herencia desde las clases raíz	Robiolo
Rob_9	Nivel máximo de profundidad de herencia desde las interfaces raíz	Robiolo
Rob_10	Cantidad de clases concretas en el primer nivel de jerarquía (raíz)	Robiolo
Rob_11	Cantidad de clases concretas en el primer nivel de jerarquía (raíz) que implementan alguna interfaz	Robiolo

2.2 Métricas estáticas

Las métricas estáticas (Tabla 2) miden aspectos deducibles directamente del código fuente de cada clase. Por esta razón también son llamadas métricas de producto. Además de las métricas obtenidas de los autores de referencia se proponen métricas originales para cuantificar los aspectos de tipado dinámico del lenguaje.

Tabla 2. Definición de métricas estáticas

Métrica	Descripción	Autor
LOC	*Lines of Code* (líneas de código lógicas de la clase)	N/D
CK_WMC	*Weigthed Methods per Class* (cantidad de métodos, sin contar los heredados)	C&K
CK_CBO	*Coupling Between Object Classes* (cantidad de clases a las cuales la clase está acoplada[1])	C&K
CK_RFC	*Response For a Class* (cantidad de métodos más la cantidad de métodos que a su vez son llamados en cada uno de ellos)	C&K
Rob_12	Porcentaje de los métodos concretos que sobreescriben un método concreto heredado, sobre el total de métodos concretos	Robiolo
Rob_13	Cantidad de mensajes polimórficos enviados[2]	Robiolo
Rob_14	Promedio de sentencias que tienen los métodos de la clase	Robiolo

[1] El acoplamiento puede deberse a varios factores, entre ellos: llamadas a métodos, instanciación, recepción como parámetro, etc.

[2] Se considera "mensaje polimórfico" a una llamada a método realizada sobre objetos con tipo declarado como interfaz o clase abstracta.

Rob_15	Cantidad de métodos públicos de una clase, incluidos el constructor, los métodos abstractos y los heredados[3]	Robiolo
Rob_16	Misma métrica que Rob_15 pero aplicada en interfaces	Robiolo
Gul_1	Cantidad de métodos que no declaran un tipo de retorno ni tampoco lo documentan[4]	Gullino
Gul_2	Cantidad de métodos que no declaran un tipo de retorno pero sí lo documentan	Gullino
Gul_3	Cantidad de métodos en los que el tipo de retorno está explícitamente declarado	Gullino
Gul_4	Cantidad de parámetros de todos los métodos de la clase, incluido el constructor, que no declaran un tipo ni lo documentan	Gullino
Gul_5	Cantidad de parámetros de todos los métodos de la clase que no declaran el tipo pero sí se encuentra documentado	Gullino
Gul_6	Cantidad de parámetros de todos los métodos de la clase que declaran el tipo de manera explícita	Gullino

Se han omitido algunas métricas tanto de la suite original C&K como del framework Robiolo debido a su dificultad de medición a consecuencia del sistema de tipos del lenguaje. Dado que la declaración de tipos es opcional, por ejemplo en los parámetros de los métodos, las métricas de colaboración entre clases requerirían inspección manual o un desarrollo de inferencia de tipos que queda fuera del alcance de este trabajo. De las métricas C&K se omitieron las reportadas [4] [6] como peores predictores.

2.3 Métricas de cambio

Las métricas de cambio seleccionadas (Tabla 3) provienen de la importante investigación dentro del área realizada por Moser en 2008 [5]. A diferencia de las definiciones originales las métricas de cambio se calcularon a nivel de clase, cuando el autor las define en base a archivos. Se ha descartado la métrica Refactorings por no encontrarse un formato sistemático dentro de los mensajes de commit que pueda utilizarse para su deducción informatizada de manera confiable.

Un "cambio" se introduce a una clase por medio de un commit al sistema de control de versiones (Git). Estos se clasificaron en tres tipos, siguiendo a Hassan [9]:

- *Feature Introduction* (FI): cambio que agrega nueva funcionalidad al sistema. Abarca todo cambio que no se produce a consecuencia de un bugfix. En términos generales, es todo cambio en el código fuente que no pueda ser atribuido a las siguientes dos categorías.
- *Fault Repairing* (FR): cambio que se produce para corregir un defecto[5]. Es decir que se trata de los bugfixes que tuvo una clase.

---

[3] Es similar a CK_WMC pero considerando la herencia

[4] La documentación de un método se realiza por medio de un *docblock* o bloque de comentario, con un formato específico, que precede al código fuente del método.

[5] Se define "defecto" siguiendo a la norma IEEE 1044 [10].

- *General Maintenance* (GM): actualización de números de versión, tabulación, co-rrección de estilo, corrección de documentación. En general, cambio no funcional.
- Descartados: cambios que no se realizan sobre clases sino sobre archivos auxilia-res, por ejemplo archivos de configuración o de internacionalización.

Debido a que las métricas de cambio cuantifican las modificaciones hechas al código fuente en lugar del código mismo, este cálculo siempre tiene en cuenta un período de tiempo. Por esto también se llaman métricas de proceso. Las métricas estáticas, en cambio, representan una especie de "fotografía", a través de las métricas, del estado del código fuente en un punto temporal específico del repositorio.

Tabla 3. Definición de métricas de cambio

Métrica	Descripción	Autor
Revisions	Cantidad de commits de tipo FI que modificaron la clase	Moser
Bugfixes	Cantidad de commits de tipo FR que modificaron la clase	Moser
Authors	Cantidad de autores distintos que realizaron al menos un commit de tipo FI a la clase	Moser
LOC_Added	Cantidad de líneas agregadas a una clase con commits de tipo FI	Moser
Max_LOC_Added	Cantidad de líneas agregadas por el commit de tipo FI que más líneas agregó a la clase	Moser
Ave_LOC_Added	Promedio de líneas agregadas considerando todos los commits de tipo FI que modifican la clase	Moser
LOC_Deleted	Cantidad de líneas eliminadas de una clase con commits de tipo FI	Moser
Max_LOC_ Deleted	Cantidad de líneas eliminadas por el commit de tipo FI que más líneas eliminó de la clase	Moser
Ave_LOC_ Deleted	Promedio de líneas eliminadas considerando todos los commit de tipo FI que modifican la clase	Moser
CodeChurn	Es el resultado de restar LOC_Added - LOC_Deleted para una clase	Moser
Max_CodeChurn	Valor absoluto mayor que tuvo la métrica CodeChurn para todos los commits de tipo FI realizados a la clase	Moser
Ave_CodeChurn	Promedio de las líneas agregadas menos las eliminadas en los commits de tipo FI que modifican la clase	Moser
Ave_ChangeSet	En promedio, cuántas otras clases se modifican junto a ésta en el mismo commit, considerando los commits FI	Moser
Max_ChangeSet	La mayor cantidad de otras clases que se modificaron junto a cierta clase mediante el mismo commit de tipo FI	Moser
Age	Edad de la clase en semanas al momento de finalizar el período, contando desde su aparición en el repositorio	Moser
Weighted_Age	Fórmula que pondera la cantidad de líneas agregadas con la edad de la clase al momento de agregar esas líneas.	Moser

3 Caso de Estudio

MediaWiki es un sistema wiki de código abierto desarrollado por la Wikimedia Foundation cuya principal instalación, dentro de decenas de miles, es Wikipedia, posicionado dentro de los diez sitios web más visitados a nivel global[6]. El producto cuenta con 1000 clases y 365 KLOC dentro del período analizado. Se trata de un producto maduro, con un historial de 15 años desde su primera versión.

El desarrollo sigue un modelo de integración continua, donde los cambios en el código fuente se despliegan directamente en los sitios de forma regular. El código enviado al repositorio de versiones por los desarrolladores es revisado por un par (*peer review*), aprobado, y luego integrado en el producto, que a su vez se despliega en los servidores productivos rápidamente, de manera continua.

3.1 Obtención de las métricas

En la Tabla 4 se presenta una muestra de las métricas estáticas obtenidas, que totalizan 1968 filas. Las columnas seleccionadas corresponden a las métricas que forman parte del modelo de regresión que mejores resultados arrojó en la predicción posterior.

Para obtener las métricas estáticas del código fuente se construyó un programa que analiza el código fuente de cada una de las clases a considerar. Esta herramienta se realizó en lenguaje PHP aprovechando sus potentes capacidades de reflexión, esto es, la posibilidad de un programa de acceder a la propia estructura de alto nivel de sí mismo (por ejemplo: variables, clases, métodos, etc.).

Tabla 4. Selección de algunas métricas estáticas (variables independientes) y métrica Bugfixes (variable dependiente)

Semestre	Clase	LOC	CK_RFC	Rob_14	Gul_5	Bugfixes
1	Title	2644	373	5,82	60	2
1	Parser	3639	408	14,73	108	4
1	File	855	197	2,87	3	4
1	WikiPage	1887	289	8,0	54	6
2	Title	2650	377	5,6	0	2
2	Parser	3674	410	14,79	0	6

Al abrir con esta herramienta un archivo que contiene una declaración de clase se logra que el intérprete la cargue en memoria, lo que provee de una interfaz orientada a objetos para consultar, por ejemplo: si es o no una clase concreta, cuántos métodos tiene y cómo se llaman, si existe herencia, los parámetros de los métodos, etc.

Haciendo uso del análisis con expresiones regulares línea por línea de cada método de cada clase y de las características de reflexión del lenguaje se han podido obtener las métricas estáticas requeridas como datos primarios. A los efectos de calcular las

[6] Métrica disponible online en www.alexa.com/siteinfo/wikipedia.org

métricas estáticas con esta herramienta, se posicionó el repositorio de versiones en dos puntos: primer cambio realizado en enero de 2014 y primer cambio realizado en julio de 2014. Esto permite tener las métricas al inicio del primer y segundo semestre del año bajo análisis, respectivamente. Como se menciona en 2.3, la métrica Bugfixes es la cantidad de commits de tipo FR que recibió la clase en el período.

Las métricas de cambio fueron mucho más trabajosas de obtener (Tabla 5, muestra del total de 1060 filas). Esto se debe, naturalmente, a que son métricas del proceso de desarrollo y, como tales, se debieron extraer del registro del trabajo de los desarrolladores que se encuentra en el repositorio de versiones, de alguna forma "desorganizado" para su procesamiento inmediato. En este repositorio se encuentran cada uno de los commits que realizó cada programador, indicando un "mensaje de commit" redactado por él y los cambios, línea por línea, que introduce a los archivos del sistema. Cada commit puede modificar varias líneas de varios archivos, simultáneamente.

Tabla 5. Selección de algunas métricas de cambio (variables independientes) y métrica Bugfixes_prox_trim (variable dependiente)

Trimestre	Clase	Revisions	Ave_LOC_Added	Codechurn	Bugfixes_prox_trim
1	Title	13	19,7	30	0
1	Parser	8	2,5	6	4
2	Title	10	7,6	15	2
2	Parser	14	20,43	182	2
3	Title	19	6,9	-340	0
3	Parser	7	13,71	44	4

En primer lugar se obtuvo el registro total de los cambios, línea por línea, a los archivos del sistema para el período temporal bajo análisis (todo el año 2014). El archivo final conteniendo todos estos commits posee aproximadamente 500 mil líneas.

En segundo lugar se construyó un programa para realizar el *parsing* de estos cambios, de manera de organizar en una base de datos relacional todos los commits y los cambios que se realizaron a cada clase con cada uno de estos commits.

En tercer lugar se creó un programa para catalogar manualmente cada uno de los commits según las tres categorías de cambios descriptas en 2.3. En esencia, se trata de diferenciar los cambios que introducen nuevas características en el sistema (FI) de los cambios que son correcciones de defectos (FR). Debido a la ausencia de información confiable para realizar esta clasificación de manera automática, se optó por realizarla de forma manual. Para esto se construyó un programa a través del cual un operador calificado pueda examinar cada commit y decidir a qué tipo corresponde, analizando si fuera necesario los cambios en las clases involucradas línea por línea. De esta forma, un operador humano decidió a qué tipo corresponde cada commit haciendo una deducción de la intención de los cambios introducidos por ese commit y con la guía (a veces, ciertamente, escueta) del mensaje introducido por el autor. Esta tarea cognitiva sobre los 2871 commits analizados ocupó unas 60 horas reloj a lo largo de 15 días de

trabajo y debe considerarse un alto costo dentro del proceso debido al conocimiento de la estructura del código fuente, la tecnología y la historia del producto que esta tarea requiere. Considerando todos los commits realizados en el proyecto durante el año 2014, se encuentran estos resultados:

- Commits descartados: 263
- Commits no descartados: 2608, de los cuales resultan
  - Feature Introduction: 1874 (72%)
  - Fault Repairing: 369 (14%)
  - General Maintenance: 365 (14%)

3.2 Método de predicción

El objetivo de los modelos de regresión logística es predecir si una clase deberá o no ser corregida (variable binaria) en función de las métricas (variables cuantitativas).

Para generar estos modelos se agregó una columna booleana "Fixed" al conjunto de datos que indica si una clase tuvo o no bugfixes, esto es, la variable dependiente a predecir. Para el análisis con métricas estáticas, realizado por semestres, una clase se considera corregida si tuvo al menos un bugfix en el semestre (cfr. Tabla 4). Para el análisis con métricas de cambio, una clase se considera corregida si tuvo al menos un bugfix en el trimestre siguiente (cfr. Tabla 5).

Los modelos de regresión logística estiman la probabilidad (entre 0 y 1) de cada clase de pertenecer a uno de los grupos (defectuosa o no-defectuosa). La asignación dicotómica final se hace en función de las probabilidades predichas. Por ejemplo, si una clase tiene un 20% de probabilidades de ser defectuosa, ¿se clasifica como defectuosa o como no-defectuosa? Luego de crear el modelo, se debe establecer un umbral de probabilidad a partir del cual se considerará que una clase es defectuosa.

		predicción	
		no-defectuosa	defectuosa
observación	no-defectuosa	811 ($n_{1,1}$)	12 ($n_{1,2}$)
	defectuosa	118 ($n_{2,1}$)	27 ($n_{2,2}$)

Fig. 1. Matriz de confusión obtenida para el modelo Fixed ~ LOC + CK_RFC + Rob_14 + Gul_5 aplicando un umbral de probabilidad de 0,5 en los datos del primer semestre

En la Tabla 6 se definen los indicadores utilizados para analizar los resultados. Estos resultados se computan a partir de la matriz de confusión obtenida con cada modelo. La Figura 1 presenta un ejemplo de matriz de confusión.

Tabla 6. Indicadores a utilizar en el análisis de los modelos de regresión

Indicador	Forma de cálculo
True Positive Rate (TP), también llamado *Recall* o *Sensitivity*	$n_{2,2} / (n_{2,2} + n_{2,1}) * 100\%$
False Positive Rate (FP)	$n_{1,2} / (n_{1,2} + n_{1,1}) * 100\%$
True Negative Rate (TN), también llamado *Specificity*	$n_{1,1} / (n_{1,2} + n_{1,1}) * 100\%$
False Negative Rate (FN)	$n_{2,1} / (n_{2,2} + n_{2,1}) * 100\%$
Porcentaje de Predicción Correcta (PC), también llamado *Accuracy*	$(n_{1,1} + n_{2,2}) / (n_{1,1} + n_{1,2} + n_{2,1} + n_{2,2}) * 100\%$
Precisión Balanceada (PB)	(TP + TN) / 2

3.3 Análisis de los resultados

Las métricas estáticas fueron calculadas sobre el código fuente correspondiente a los días 1 de enero y 1 de julio. Esto provee de dos "fotografías" del código fuente al inicio de cada semestre del año.

Todas las métricas de cambio fueron calculadas para los primeros tres trimestres del año excepto por Bugfixes (cfr. Tabla 3) que se calculó para los cuatro trimestres del año. De esta forma puede trabajarse con las métricas de un trimestre con el objetivo de predecir los bugfixes del siguiente. Por esta razón no se realizaron predicciones para el cuarto trimestre, ya que esto implica obtener los bugfixes del primer trimestre del año 2015, algo que queda fuera del alcance de este trabajo.

La Tabla 7 presenta las métricas de sistema definidas en el framework Robiolo [7] y algunos conteos relevantes, a los efectos de dimensionar el producto bajo análisis.

En líneas generales se verifica que sólo un 15% de las clases han precisado correcciones de defectos durante el período analizado. Las clases modificadas por commits de tipo FI, sin embargo, son aproximadamente el 50% y abarcan pocas líneas.

Respecto de un semestre al siguiente se evidencia un leve crecimiento en la cantidad de clases del sistema (3%) en seis meses, comparado con un cambio mínimo en las LOC (0,4 por mil). También crece en este período la cantidad de interfaces.

Tabla 7. Métricas de sistema para MediaWiki, año 2014

Métrica	Resumen	1/enero	1/julio
LOC	Líneas lógicas de código totales	364.224	365.696
Rob_1	Cantidad de clases internas	922	941
Rob_2	Cantidad de interfaces internas	24	28
---	Cantidad de clases externas	22	31
---	Cantidad de clases e interfaces totales	968	1000
Rob_3	Clases externas especializadas	1	1
Rob_4	Interfaces externas extendidas	0	0
Rob_5	Clases que implementan interfaces externas	0	0

Rob_6	Jerarquías de clase	80	82
Rob_7	Jerarquías de interfaz	0	0
Rob_8	Niveles por jerarquía de clases	5	5
Rob_9	Niveles por jerarquía de interfaces	2	2
Rob_10	Clases raíz concretas	245	245
Rob_11	Clases raíz concretas que implementan interfaces	45	46
Clases modificadas	Cantidad de clases que fueron modificadas con al menos un commit de tipo FI en el semestre	459	566
Clases no modificadas	Cantidad de clases que no fueron modificadas con ningún commit de tipo FI en el semestre	509	434
Clases sin defectos	Cantidad de clases que no fueron modificadas por ningún commit de tipo FR en el semestre	823	866
Clases con defectos	Cantidad de clases que fueron modificadas con al menos un commit de tipo FR en el semestre	145	134

A continuación se elaboran modelos de regresión utilizando las métricas obtenidas (estáticas y de cambio, para cada clase) como variables independientes.

Métricas estáticas

Se utilizaron las métricas del primer semestre para crear el modelo (datos de entrenamiento) y posteriormente se aplicó este modelo para predecir las observaciones del segundo semestre. Se construyó un modelo con las métricas LOC, CK_RFC, Rob_14 y Gul_5 ya que no están correlacionadas[7] y presentan un p-value menor a 0,05. Si se utiliza un umbral de 0,5 para la clasificación (probabilidad > 50% = clase defectuosa), se consigue la matriz de confusión de la Figura 1.

En la Tabla 8 se encuentra un resumen de los indicadores para distintos umbrales. Siguiendo a Moser [5] se considera que es más costoso corregir posteriormente en el ciclo de vida un defecto no detectado (falso negativo) que inspeccionar una clase clasificada como defectuosa cuando realmente no lo es (falso positivo). Por lo tanto, es deseable mantener bajo el indicador FN.

Como ya lo observa Moser, a medida que se modifica el modelo para reducir los falsos negativos (que resultan muy costosos) también se incrementa el esfuerzo de inspección de clases que realmente no tienen defectos, es decir, se obtendrán más falsos positivos. Esto es porque la disminución en el umbral traslada más casos “hacia la derecha” de la matriz de confusión (al clasificar más elementos como defectuosos), donde se encuentran el falso positivo y el verdadero positivo. Con otras palabras, al enviar casos hacia la columna derecha de la matriz se está reduciendo el falso negativo (en la columna izquierda de la matriz), hecho que ya se describió como deseable.

[7] Correlación de Pearson menor a 0,3

Tabla 8. Indicadores resultantes del modelo Fixed ~ LOC + CK_RFC + Rob_14 + Gul_5 para datos de entrenamiento (primer semestre) y varios umbrales de probabilidad (valores en %)

Umbral	TP	FP	TN	FN	PC	PB
0,5	19	1	99	81	87	59
0,2	48	13	87	52	82	67,5
0,15	**57**	**23**	**77**	**43**	**74**	**67**
0,1	78	45	55	22	58	66,5

En la Tabla 9 puede observarse que el desempeño del modelo con datos nuevos (segundo semestre) es prácticamente igual a los datos de entrenamiento (primer semestre). Podemos concluir que el modelo entrenado con métricas estáticas puede aplicarse a nuevos datos manteniendo su consistencia, siendo el umbral de 0,15 el más óptimo con una PB de 68,5%, TP de 55% y FN de 45%.

Tabla 9. Indicadores resultantes del modelo Fixed ~ LOC + CK_RFC + Rob_14 + Gul_5 para datos nuevos (segundo semestre) y varios umbrales de probabilidad (valores en %)

Umbral	TP	FP	TN	FN	PC	PB
0,2	40	10	90	60	84	65
0,15	**55**	**18**	**82**	**45**	**78**	**68,5**
0,1	71	40	60	29	61	65,5

Métricas de cambio

Se construyó un modelo de regresión logística con las métricas de cambio y se procedió a evaluar su desempeño de manera similar a como se hizo con las métricas estáticas. Se toman los datos del primer trimestre como entrenamiento, y se realiza la predicción para los trimestres segundo y tercero, en función de comparar su poder predictivo.

En la Tabla 10 se hallan los indicadores para un modelo de regresión construido con las métricas de cambio Revisions y Ave_LOC_Added, ya que no están correlacionadas. Puede observarse que para un umbral de 0,15 se obtiene una PB de 70%, TP de 69% y FN de 31%, lo que resulta una mejora (mayor PB, mayor TP, menor FN) sobre el modelo con métricas estáticas.

En la Tabla 11 se encuentran los indicadores para el mismo modelo, aplicado a los datos del segundo y tercer trimestre. Al igual que con las métricas estáticas, con el modelo generado con métricas de cambio se verifica un comportamiento consistente respecto de los datos de entrenamiento cuando se utiliza el modelo con datos nuevos.

Tabla 10. Indicadores resultantes del modelo Fixed ~ Revisions + Ave_LOC_Added para datos de entrenamiento (primer trimestre) y varios umbrales de probabilidad (valores en %)

Umbral	TP	FP	TN	FN	PC	PB
0,2	55	16	84	45	79	69,5
0,15	**69**	**29**	**71**	**31**	**71**	**70**
0,1	100	93	7	0	24	53,5

Tabla 11. Indicadores resultantes del modelo Fixed ~ Revisions + Ave_LOC_Added para datos nuevos (segundo y tercer trimestre) y varios umbrales (valores en %)

Datos	Umbral	TP	FP	TN	FN	PC	PB
Segundo trimestre	0,3	29	5	95	71	81	62
	0,2	43	18	82	57	74	62,5
	0,15	**57**	**31**	**69**	**43**	**66**	**63**
	0,1	99	95	5	1	24	52
Tercer trimestre	0,3	42	8	92	58	86	67
	0,2	55	18	82	45	79	68,5
	0,15	**75**	**31**	**69**	**25**	**70**	**72**
	0,1	94	89	11	6	22	52,5

Combinando métricas no correlacionadas se pueden conformar modelos con diversas variables predictoras. Sin embargo, los modelos precedentes representan las mejores combinaciones encontradas en este trabajo para métricas estáticas y de cambio.

3.4 Amenazas a la validez

Por un lado existe una amenaza a la validez interna debido a las herramientas utilizadas para la obtención de las métricas estáticas. Estas herramientas debieron ser desarrolladas para esta investigación y, si bien existe confianza suficiente en los resultados debido a las pruebas realizadas, pueden existir combinaciones sintácticas en el código fuente de rara aparición que no hayan sido observadas.

Por otro lado, el procedimiento de categorización manual necesario para calcular las métricas de cambio también puede presentar inconsistencias debido a su carga cognitiva y a la presencia de casos límite. Una decena de commits (~1%) podrían categorizarse tanto FI como FR debido a la naturaleza mixta de los cambios que introduce el desarrollador con ellos, lo que llevó a tomar una decisión que introduce un error menor.

En cuanto a la validez externa de los resultados debe observarse que, según se describió, el producto bajo estudio es de código abierto y se desarrolla mediante un proceso comunitario de integración continua, revisión de pares y testing automatizado. Esto puede limitar el alcance de los resultados si se consideran las heterogéneas

metodologías utilizadas en la industria, incluso trabajando con el mismo lenguaje. Los resultados presentados corresponden a un caso de estudio individual.

4 Trabajos relacionados

Gyimothy et al. [6] analizaron las métricas C&K como predictoras de defectos. Como caso de estudio trabajaron con el código fuente del cliente de correo Mozilla, desarrollado en C++. Aplicando distintos métodos de estimación (regresión lineal, logística y árboles de decisión) logran una precisión menor a 70%, hallando en CBO y LOC los mejores predictores.

Basili [4] comparó la suite C&K con otras métricas estáticas del código fuente, encontrando que aquellas son mejores predictores (excepto LCOM). Utilizó proyectos universitarios desarrollados en C++ y aplicó regresión logística.

Moser [5], en su trabajo seminal, introduce una suite de métricas de cambio y aplica varias técnicas de estimación (regresión logística, Naive Bayes, árboles de decisión) utilizando un proyecto en lenguaje Java. Al igual que la presente investigación, encuentra que las métricas de cambio son mejores predictores que las métricas estáticas del código fuente, logrando modelos con un TP de 80%. Kamei [11], Madeyski [12] y Zhang [2] arriban a resultados similares utilizando casos de estudio comparables, con varios métodos de estimación.

Giger [13] trabajó a nivel de método y también confirmó que las métricas de cambio son mejores predictores que las métricas estáticas. Logra una precisión superior al 80% con varias técnicas de *machine learning* (Random Forest, Bayesian Network, árboles de decisión). Los casos de estudio incluyen proyectos *open source* en Java.

5 Conclusiones

La presente investigación analizó el código fuente, y las modificaciones hechas a éste, del proyecto MediaWiki durante el período de un año. Para ello se construyeron herramientas que computan las métricas y se inspeccionaron manualmente 2871 commits al repositorio de versiones. Utilizando estas métricas se construyeron modelos de regresión logística con el objetivo de obtener el mejor conjunto de métricas que predigan defectos futuros de una clase.

Los modelos definidos con métricas de cambio son los que lograron mejores resultados en la predicción. Las métricas Revisions y Ave_LOC_Added constituyen un modelo que predice una clase defectuosa con un TP del 75%, falso negativo del 25% y precisión balanceada de 72% (cfr. Tabla 11). Esto significa que la cantidad y tamaño de las modificaciones FI hechas a una clase son los mejores predictores de defectos de dicha clase.

Los modelos con métricas estáticas constituyen buenos predictores, aunque por debajo de las métricas de cambio, como ya observan otros autores (cfr. 4). Con la combinación de métricas estáticas LOC + CK_RFC + Rob_14 + Gul_5 se puede construir un modelo que predice defectos con un TP del 55%, FN 45% y PB de 69%.

Nótese que las métricas de cambio son más costosas de obtener, como se describe en 3.1.

La predicción de defectos en lenguajes de tipado dinámico es de interés para la industria si se consideran lenguajes muy extendidos como PHP, Python o JavaScript.

Referencias

1. T. Hall, S. Beecham, D. Bowes, D. Gray, y S. Counsell, «A Systematic Literature Review on Fault Prediction Performance in Software Engineering», *IEEE Transactions on Software Engineering*, vol. 38, n.º 6, pp. 1276-1304, nov. 2012.
2. F. Zhang, A. Mockus, I. Keivanloo, y Y. Zou, «Towards building a universal defect prediction model», 2014, pp. 182-191.
3. N. Nagappan, T. Ball, y A. Zeller, «Mining metrics to predict component failures», 2006, p. 452.
4. V. R. Basili, L. C. Briand, y W. L. Melo, «A validation of object-oriented design metrics as quality indicators», *IEEE Transactions on Software Engineering*, vol. 22, n.º 10, pp. 751-761, oct. 1996.
5. R. Moser, W. Pedrycz, y G. Succi, «A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction», 2008, p. 181.
6. T. Gyimothy, R. Ferenc, y I. Siket, «Empirical validation of object-oriented metrics on open source software for fault prediction», *IEEE Transactions on Software Engineering*, vol. 31, n.º 10, pp. 897-910, oct. 2005.
7. Robiolo, Gabriela, «Métricas de Diseño Orientado a Objetos aplicadas en Java», tesis de Maestría, Universidad Nacional de La Plata, Argentina, 2004.
8. S. R. Chidamber y C. F. Kemerer, «A metrics suite for object oriented design», *IEEE Transactions on Software Engineering*, vol. 20, n.º 6, pp. 476-493, jun. 1994.
9. A. E. Hassan, «Predicting faults using the complexity of code changes», 2009, pp. 78-88.
10. Institute of Electrical and Electronics Engineers, IEEE Computer Society, Software Engineering Standards Committee, y IEEE-SA Standards Board, *IEEE standard classification for software anomalies*. New York: Institute of Electrical and Electronics Engineers, 2010.
11. Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, y A. E. Hassan, «Revisiting common bug prediction findings using effort-aware models», 2010, pp. 1-10.
12. L. Madeyski y M. Jureczko, «Which process metrics can significantly improve defect prediction models? An empirical study», *Software Quality Journal*, vol. 23, n.º 3, pp. 393-422, sep. 2015.
13. E. Giger, M. D'Ambros, M. Pinzger, y H. C. Gall, «Method-level bug prediction», 2012, p. 171.