Using Smell-based Metrics to Predict Structural Changes: A Technical Report

Huihui Liu*, Yijun Yu† Bixin Li*, Yibiao Yang‡, and Ru Jia§
*School of Computer Science and Engineering, Southeast University, Nanjing, China
†Centre for Research in Computing, The Open University, UK
‡State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
§School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia
lhhshuxue@126.com, y.yu@open.ac.uk, bx.li@seu.edu.cn, yangyibiao@nju.edu.cn, rjia@swin.edu.au

***Abstract*—Bad code smells (also named as code smells) are symptoms of poor design choices in implementation. Existing studies confirmed empirically that the presence of code smells increases the likelihood of subsequent changes (i.e., change-proness). However, to the best of our knowledge, no prior studies leverage smell-based metrics to predict particular change type (i.e., structural changes). Moreover, when evaluating the effectiveness of smell-based metrics in structural change-proneness prediction, none of existing studies take into account the effort inspecting those change-prone source code . In this paper, we consider five smell-based metrics for effort-aware structural change-proneness prediction and compare these metrics with a baseline of well-known CK metrics in predicting particular categories of change types. Specifically, we first employ univariate logistic regression to analyze the correlation between each smell-based metric and structural change-proneness. Then, we build multivariate prediction models to examine the effectiveness of smell-based metrics in effort-aware structural change-proneness prediction when used alone and used together with the baseline metrics, respectively. Our experiments are conducted on six Java open-source projects with up to 60 versions and results indicate that: (1) all smell-based metrics are significantly related to structural change-proneness, except metric ANS in hive and SCM in camel after removing confounding effect of file size; (2) in most cases, smell-based metrics outperform the baseline metrics in predicting structural change-proneness; and (3) when used together with the baseline metrics, the smell-based metrics are more effective to predict change-prone files with being aware of inspection effort.**

I. Introduction

Flower [1] coined the term "code smells" and defined the concept as symptoms of bad design and implementation choices in source code. Since then, many empirical studies have investigated the impact of code smells on software quality properties, such as program understandability [2], change-/fault-proneness [3]–[6] and code maintainability [7]–[9]. These studies indicated that, when appeared in source code alone or together, code smells could significantly increase change-proneness.

Furthermore, the empirical results in the study of Romano et al. [10] have shown that some smells could even result in certain types of structural code changes (instead of changing arbitrary lines in the code), e.g., *API* changes (a category of change types, see Table IV) are more likely to occur in the classes affected by *Swiss Army Knife* smell.

To the best of our knowledge, these results only indicated qualitatively that certain code smells could lead to textual changes or a particular type of structural changes. Software practitioners were not readily instructed on how to predict these change-prone modules (such as files or classes) based on the code smells, although it is obviously beneficial to be able to do so. For example, if developers were aware of likely API changes due to code smells they could plan ahead of time to allocate resources for system-wide integration tests on the dependent modules. Moreover, they might set aside additional time to update the API and design documents so that source code and API can be ensured to co-change timely. On the contrary, if only minor changes to statements inside a method's body were predicted, unit testing would be sufficient because no further change impact can be expected. Differentiating the effort of such predicted would-be changes would add to the effective planning to be more prepared.

To achieve these goals, in this work, we consider mining the structural change history of files to build an effort-aware structural change-proneness prediction model based on five code smell metrics. Specifically, we employ univariate logistic regression approach to analyze the correlations between each smell-based metric and structural change-proneness (without discriminating particular change types); then we build multivariate prediction models to examine the effectiveness of smell-based metrics, in comparison or in combination to with baseline metrics (i.e., CK metrics), in predicting effort-aware structural change-proneness.

The experiments are conducted on six typical Java open-source projects with up to 60 versions. Our results show that: (1) smell-based metrics are significantly correlated to structural change-proneness; (2) in most cases, smell-based metrics outperform the baseline metrics in predicting structural change-proneness; and (3) when combined with the baseline metrics, smell-based metrics are more effective in predicting change-prone files with large LOC to inspect.

Our study provides valuable insight for researchers and practitioners to better understand the usefulness of five smell-based metrics in predicting structural change-prone files under the scenarios of ranking and classification, being aware of the effort to inspect large LOC in the files-to-change.

The rest of the paper is organized as follows. Section II

discusses the related work. Section III introduces the study design including research questions, subject selection, data collection and prediction model construction, as well as model evaluation. Section IV reports the experimental results, answers our research questions. Section V examines the threats to validity of our study. Section VI concludes.

II. Related Work

In this section, we introduces three aspects that mostly related to our research topic which consist of smell detection techniques, smell's impact on quality and change prediction models.

A. Smells Detection

Code smells are symptoms of poor design and implementation choice. Fowler [1] gave a brief description to 22 code smells and presented some refactoring techniques for their removal. After that, many approaches have been proposed for automatic smell detection, e.g., metrics-based techniques, visualization-based techniques and search-based techniques, as stated in recent literature reviews [11]–[14]. These literatures reveal that a notable portion (around 37%) of detection approaches are metrics-based. Although this approach has its limitation, e.g., no consensus on the standard threshold, it is still widely adopted by many researchers due to its maturity and effectiveness. Based on aforementioned approaches, many smell detection tools (up to 84) have been developed [11], [12]. These tools include open-source tools, e.g., `PMD`[1], `iPlasma` [15], `JDeodrant`[2], `CBSDetector` [16] and `AJCSD`[3], and closed-source tools, e.g., `Borland Together`[4] and `InCode`[5]. In contrast to open-source tools, closed-source tools have the limitation that it is usually hard to know what smell detection strategies are employed behind them. However, none of these tools can detect all 22 code smells defined by Fowler [1]. In average, four smell types can be detected per tool.

B. Impact of Smells on Quality

Due to our topic focusing on change-proneness prediction, we only introduce studies of how smells affect code change-proneness. Khomh et al [5] investigated more than 12 smells on several projects and their results showed that the classes affected by code smells have higher change-proneness than other smell-free classes. Particularly, recent studies by Palomba et al. [6] further confirmed this result. Moreover, they also highlighted that the change-proneness of smelly classes would significantly decrease after removing corresponding smells contained in these classes. Olbrich et al [3], [4] conducted similar experiment focusing on God Class, Shotgun Surgery and Brain Class, and concluded that these smells did had higher change-proneness than smell-free components. However, this conclusion did not hold when the change size (sum of lines modified, added and deleted) was normalized by source lines of code. Besides, several researchers paid more attention on the effect of combined smells on the quality. Abbes et al. [17] conducted three experiments on 24 subjects and investigated whether the occurrence of antipatterns (alias to code smells) affected code understandability during the period of code comprehension and maintenance tasks. They concluded that although the occurrence of one antipattern did not significantly decrease developers' performance, a combination of two antipatterns significantly hindered developers' performance during code comprehension and maintenance tasks. Furthermore, Yamashita et al. [7], [9] reported that code smells appearing together in the same file (i.e., collocated-smells) can interact with each other and lead to various types of maintenance issues. They also found that smells interactions occurred across coupled files (i.e., coupled smells) could result in comparable negative effects as collocated smells. In particular, some studies also investigated the relationship between code smells and structural change-proneness, rather than textual change-proneness [5], [10], [18]. Their empirical results indicated that code smells also had significant correlation with structural changes and even some smells could be more structurally change-prone as compared with other smells for a particular change category (see Table IV).

C. Change Prediction Models

Tollin et al. [19] used coding rules violations to predict code changes through a set of machine learning models. Their results indicated that these classification models achieved satisfactory performance, especially when predicting changes in the next version. Lu et al. [20] used statistical meta-analysis to investigate the predictive capability of 62 OO metrics by performing empirical validation on 102 Java systems. Their study indicated that size metrics showed moderate predictive capability, inheritance-based metrics had the least predictive capability for change-prone class prediction. The predictive ability of coupling and cohesion metrics was lower than size metrics for change prediction tasks. In contrast to textual change-proneness prediction, Romano and Pinzger [21] used code metrics to predict change-prone Java interfaces. Soon after that, Giger et al. [22] employed code metrics (i.e., CK metrics) and network measures (based on code dependency graph) to predict particular type of changes (similar to change categories in Table IV). The classifiers they chosen were Bayesian Networks and Neural Networks. Their results showed that the model built on CK metrics and network measures in combination achieved good prediction performance in terms of AUC, precision and recall indicators.

III. Study Design

The goal of this experimental study is to construct a change-proneness prediction model using smell-based metrics and to evaluate the contribution of these metrics with respect to effort estimatations such as LOC of changed files. In the following subsections, we propose three research questions and criteria

[1]http://pmd.sourceforge.net
[2]http://www.jdeodorant.com
[3]https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector
[4]http://borland.com/products/together
[5]http://intooitus.com/products/incode

to choose subject projects, illustrate the procedure of data collection, and construct the effort-aware change-proneness prediction models using logistic regression and evaluate their performance under the scenarios of ranking and classification.

A. Research Questions and Subjects Selection

To investigate the effectiveness of smell-based metrics in effort-aware change-proneness prediction, we set up three research questions as follows:

RQ1: How are smell-based metrics correlated with structural change-proneness?

RQ2: Are smell-based metrics more or less effective than the commonly used code metrics in predicting change-prone files regarding each change category?

RQ3: Are the combination of smell-based metrics and commonly-used metrics more effective in predicting change-prone files regarding each change category?

The purpose of RQ1 is to determine whether or not each of five smell-based metrics is potentially a useful predictor for change-proneness prediction. If so, those useful metrics can be employed alone or together with commonly used code metrics. As did in Giger et al's work [22], we choose the Chidamber and Kemerer metrics suite [23] as the commonly used code metrics (i.e., baseline metrics, here after namely CK metrics, listed in Table I) since existing studies have shown that they are powerful to distinguish which files are more change-prone [20], [21]. RQ2 and RQ3 are set to assess the across-version predictive ability in the scenarios of ranking and classification. Our empirical study is conducted on six typical Java open

TABLE I: Description of CK metrics [23]

Abbr.	Description
WMC	*Weighted methods per class* is the sum of the cyclomatic complexity of all methods of a class.
CBO	*Coupling between object classes* counts the coupling to other classes.
LCOM	*Lack of cohesion in methods* counts the number of pairwise methods without any shared instance variables, minus the number of pairwise methods that share at least one instance variable.
DIT	*Depth of inheritance tree* denotes the maximum depth of the inheritance tree of a class.
NOC	*Number of children* is the number of direct subclasses of a class
RFC	*Response for class* counts the number of local methods (including inherited methods) of a class.

source projects with up to 60 releases hosted in GitHub.

We use the following criteria to select subject projects: (1) long software version history with more than 2 years; (2) sizes ranging from 100 KLOC to 1MLOC; (3) the availability of mature releases from GitHub; (4) different domains that projects belong to, e.g., DataBase system, search engine, and data analyzer. Some basic information of these open source projects is described in Table II. More data information for replication including release IDs and download URLs refer to our online appendix [24].

B. Data Collection

This subsection mainly introduces some techniques including how code metrics, code changes and smells are extracted from source code of a measured project with multiple releases. Fig. 1 shows the procedure of collecting these data.

TABLE II: The description of studied projects

Project	Period	Description	#Releases	KLOC[‡]
camel	2009-2017	Integration framework	8	377
derby	2005-2016	Relational DB Management System	6	496
elasticsearch	2010-2017	Distributed Search and Analytics Engine	9	336
hive	2010-2016	Data Warehouse Software Facilitates	7	592
pig	2010-2017	Big Data Set Analyzer	10	592
lucene	2011-2017	Text Search Engine	20	763

[‡] KLOC represents kilo lines of code in average

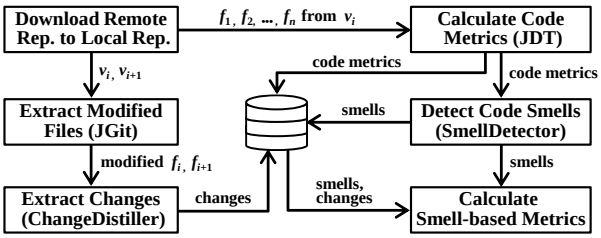

Fig. 1: Process of data collection

Specifically, all remote repositories of studied subjects are needed to be downloaded from GitHub and stored as local repositories, because accessing local repository not only promotes code parsing speed for calculating code metrics, but also is the requirement to extract structural changes by the tool `ChangeDistiller` [25].

(1) *Code Metrics Extraction and Smells Detection*

The CK metrics values mentioned in section III-A can be obtained using the tool `ck`[6] that are developed based on `JDT`[7] and is reliable by our manual testing on several large projects. In this paper, we consider 14 code smells listed in Table III,

TABLE III: Fourteen kinds of code smells and their description

Smell Type	Short Smell Description
Public Field	A class contains more than one public field.
Data Class	A class contains data but not behavior related to the data.
Large Class	A class has grown too large with more than specified LOC.
Middle Man	A class delegating to other classes most of their methods.
Refused Parent Bequest	A subclass not using the protected methods of its superclass.
Speculative Generality	An unused class, method, field or parameter.
Data Clump	Some fields together appeared in a couple of classes.
Divergent Change	Different parts occur changes in class for different reasons.
Feature Envy	A method is more interested in other class than its own.
Long Method	A long method exists if it has extreme LOC.
Long Parameter List	A method having a long list of parameters.
Message Chains	A long chain of method call to implement a complex role.
Shotgun Surgery	A single change in a method needs many scattered changes.
Switch Case	A switch-case was misused to implement polymorphism.

where the first column describes smell types and the second column shows their short descriptions. Noting that some smells have their aliases, e.g., *Large Class* is also named as *God Class* in some literatures [15]. We choose these smells for two reasons. First, they can be well measured based on the combination of code metrics and hence can be automatically identified by tools. Second, several systematic review papers [11]–[14] revealed that these smells were widely investigated by researchers in the past decade. In this paper, two open-source smell detection tools, namely `BSDectector` [16] and

[6]https://github.com/mauricioaniche/ck

[7]http://www.eclipse.org

AJCSD[8] are employed to identify 14 kinds of code smells for Java programs. The smell detection strategies behind these two tools are described in [15], [16].

(2) *Structural Changes Extraction*

The procedure of fine-grained change extraction is divided into two steps. First, JGit[9] tool is employed to extract each pair of modified files having identical name from any two successive versions v_i and v_{i+1} of a given project (denoted by f_i and f_{i+1} in Fig. 1, respectively). Second, ChangeDistiller tool [25] is employed to extract fine-grained changes from file f_i and f_{i+1}. We choose ChangeDistiller to detect fine-grained structural changes due to its powerful capability of identifying up to 48 types of fine-grained changes for Java code. It is worth noting that ChangeDistiller tool only processes file pairs tagged by MODIFY in Git. As did in [5] and [6], file-level change type DELETE, ADD and COPY defined in JGit are not considered in this work, because we mainly use the presence of smells in version v_i to predict potentially change-prone files as the system evolves from version v_i to v_{i+1}.

(3) *Smell-based Metrics Calculation*

In this paper, we use smell-based metrics as predictors in prediction model. A point to note that, these smell-based metrics are all derived from the evolutionary history of smelly files, rather than only one version snapshot. In previous study, Taba et al had proposed [26] four smell-based metrics (originally namely antipattern-based metrics), i.e., *Average Number of Smells (ANS)*, *Smell Complexity Metric (SCM)*, *Smell Recurrence Length (SRL)* and *Smell Cumulative Pairwise Differences (SCPD)*. In the context of our study, the first three smell-based metrics are selected and *SCPD* metric is not considered due to its simplicity. In addition, we newly defined two smell-based metrics which leverage the structural change information during the period of file evolution.

(3.1) Average Number of Smells (ANS)

Let's $\{S_1, S_2, ..., S_n\}$ be the list of consecutive versions of a system S, where, S_1 is the first version, and $S_n = S$. For each file $f \in S$, we denote f_i as the corresponding snapshot of f in S_i, i.e., $f_i \in S_i, i \in \{1, 2, ...n\}$ and $f = f_n$. Before defining ANS, we first give an indicator function χ as follows:

$$\chi(f_i) = \begin{cases} 1, & \text{if } DIFF(f_i, f_{i+1}) > 0 \text{ and } 1 \leq i < n \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Noting that, "$DIFF(f_i, f_{i+1}) > 0$" in the condition of above equation refers to f_i undergoing structural (not textual) changes when evolving from version S_i to S_{i+1}. Next, let's $TNOS(f_i)$ be the total number of smells in $f_i, i \in \{1, 2, ...n\}$. To capture the distribution of smells in the past modified versions of a system for a file f_i, we now introduce the function $NOCS(f_i)$ (Number Of Change-prone Smells) as follows:

$$NOCS(f_i) = \begin{cases} \chi(f_i) \times TNOS(f_i), & \text{if } 1 \leq i < n \\ TNOS(f_i), & \text{if } i = n \end{cases} \tag{2}$$

[8]https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector

[9]http://www.eclipse.org/jgit/

Based on aforementioned equations (1) and (2), we define the Average Number Of Smells (ANS) as follows:

$$ANS(f) = \frac{1}{n} \sum_{k=1}^{n} NOCS(f_i) \tag{3}$$

where n is the total number of evolutionary versions of file f and $f = f_n$.

(3.2) Smell Complexity Metric (SCM)

Before defining SCM, we first define the following Shannon entropy of smells in version $S_i, i \in \{1, 2, ..., n\}$ for capturing the distribution of smells in different source files.

$$H_i = -\sum_{k=1}^{m} p(f_i^k) \times \log_2 p(f_i^k) \tag{4}$$

where m is the total number of files in version S_i and $p(f_i^k)$ is the percent of smells in the k-th file f_i^k of S_i, and $p(f_i^k)$ is computed using following Equation (5).

$$p(f_i^k) = \frac{TNOS(f_i^k)}{\sum_{l=1}^{m} TNOS(f_i^l)} \tag{5}$$

Using the entropy of smells in versions S_i, we introduce Smell Complexity Metric (SCM) by Equation (6).

$$ACM(f) = \sum_{i=1}^{n} p(f_i) \times H_i \tag{6}$$

where $p(f_i)$ is the percent of smells for the file f_i in version S_i, n is the total number of versions in the history of f and $f = f_n$.

(3.3) Smell Recurrence Length (SRL)

To mathematically define the SRL metric, we first give an indicator function of smelly file ψ similar to Equation (1) as follows:

$$\psi(f_i) = \begin{cases} 1, & \text{if } TNOS(f_i) > 0 \text{ and } 1 \leq i \leq n \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

We also use Equation (8) to quantify the number of modified versions (NOMV) in the history of file f, where each corresponding snapshot of f has at least one smell.

$$NOMV(f) = \sum_{i=1}^{n-1} \chi(f_i) \times \psi(f_i) \tag{8}$$

Besides, we also consider the smell lifespan during software evolution. Equation (9) is proposed to measure the maximal length of consecutive streams of smells (MLCSS) in the history of file f.

$$MLCSS(f) = \begin{cases} \max\limits_{1 \leq i \leq k \leq j \leq n} \text{len}(f_i, f_j), & \text{if } NOCS(f_k) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

For a particular file f, there may exist more than one instances of longest consecutive smell streams. To locate the nearest version place of the longest consecutive smell streams (the latest one), we use the ending index of the longest consecutive stream of smells (EILCSS) in modified versions of file f.

Based on the calculation of MLCSS and EILCSS, we now use following Equation (10) to define the Smell Recurrence

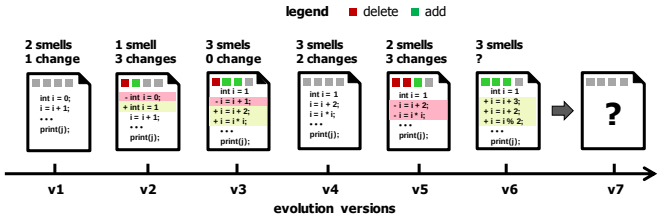

Fig. 2: example of evolutionary files

Length (SRL) for capturing the consecutive occurrence of smells in a file:

$$SRL(f) = MLCSS(f) \times e^{\frac{1}{n}(NOMV(f)+EILCSS(f))} \quad (10)$$

where n is total number of evolutionary versions of file f. Equation (10) seems to be hard to understand, so an example in Figure 2. is taken to show how we calculate the metric SRL of a file f with evolving from version v_1 to v_6 (v_7 will be released soon). It is readily to find the file f was all affected by at least one smell and was modified at least one developer as the file f evolves from version v_1 to v_6, while an except case was that file f was not changed between version v_3 and v_4. Therefore, we conclude that $NOMV(f) = 4$, $MLCSS(f) = 3$ (because the longest consecutive changed versions is $v_4 \rightarrow v_5 \rightarrow v_6$) and $EILCSS(f) = 6$, then we get $SRL(f) = 3 \times e^{\frac{1}{6}(4+6)}$.

(3.4) Average Structural Changes (ASC)

For a file, the average structural changes (ASC) represents the average changes over the file's evolutionary history where the changes occurred in a certain version would be filtered out if the file was not affected by any smells considered in this paper. More details about ASC refer to our online appendix [24].

(3.5) Number Of Distinctive Contributors (NDC)

The number of distinctive contributors (NDC) describes the number of distinct contributors who made their changes in a given file in the period of overall version evolution, where this file was affected by at least one smell in a certain version.

From the definition of these five smell-based metrics, the common characteristic they shared is to exploit the information of code smells and structural changes contained in the history of software evolution. While their history information is stored in the database, as showed in Fig. 1, therefore, these smell-based metrics values can be easily obtained.

C. Model Construction and Evaluation

(1) *Variable Description*

The goal of this study is to assess the actual predictive power of smell-based metrics for effort-aware change-proneness prediction towards each fine-grained change type. However, the number of fine-grained change types in ChangeDistiller [25] reached up to 48 in total and our initial investigation reveals that some change types have very lower occurrences in our dataset e.g., *adding attribute modifiability* (i.e., removing the keyword `final` from an attribute declaration). As a result, in order to have higher frequencies of some change types in our experiments we combine several change types into one change type category according to their semantics, specifically, all fine-grained change types are classified into four change (type) categories: *api*, *state*, *functionality* and *statement*, showed in Table IV. In this paper, the modeling technique we choose is logistic regression where the dependent variable is binary. While each of four change categories is continuous type as it is aggregated from some fine-grained change types. Therefore, prior to model application, each change category Y needed to be transformed into a binary variable $sign(Y)$, i.e., $sign(Y) = 1$ if $Y > \alpha$, otherwise $sign(Y) = 0$; where α represents the median value of all files of that studied project for a change category Y, as did in [21], [22].

TABLE IV: Categories of fine-grained source code changes

Category	Ingredients Description
api	Changes that involve the declaration of a class (e.g., class renaming and class API changes) and signature of method (e.g., modifier changes, method renaming, return type changes, changes of the parameter list).
state	Changes that affect object states of a class (e.g., fields addition and deletion).
functionality	Changes that affect the functionality of a class (e.g., methods addition and deletion).
statement	Changes that modify executable statements (e.g., statements insertion and deletion) amd alter condition expressions in control structures and the modification of else-parts.

The independent variables in this study are composed of two categories of metrics: CK metrics (see Table I) and five smell-based metrics (i.e., ANS, SCM, SRL, ASC and NDC). All these metrics are collected at the file level. Hence, when a file contains more than one top-level classes, CK metrics need to be aggregated at the file level. Noting that LOC of file is also taken into account as an independent variable, due to its moderate predictive ability in discriminating between change-prone and not change-prone file in previous studies [20]. With these variables, we can build a corresponding change prediction model.

(2) *Modeling Technique*

We choose logistic regression technique mainly for the following reasons. First, logistic regression does not make any assumption about normality, linearity, or homogeneity of variance for the independent variables [27]. Therefore, the distribution of the independent variables will not influence the regression model, and consequently not affect the prediction results. Second, the results derived from logistic regression is easy to interpret to what extend each independent variable (metric) affects the dependent variable (change-proneness). Third, logistic regression is a well-known strategy, widely used in software engineering for fault-/change-proneness prediction [28]–[30].

There are two types of logistic regression, i.e., univariate logistic regression and multivariate logistic regression. Univariate analysis is used to find the individual effect of the independent variable on the dependent variable, whilst multivariate analysis is used to find the combined effect of the independent variables on the dependent variable. The multivariate logistic regression model is defined as follows:

$$Pr(Y = 1|X_1, X_2, ..., X_n) = \frac{e^{\beta_0+\beta_1 X_1+\beta_2 X_2+...+\beta_n X_n}}{1 + e^{\beta_0+\beta_1 X_1+\beta_2 X_2+...+\beta_n X_n}} \quad (11)$$

where independent variables $X_1, X_2, \ldots, X_n$ represent the metrics considered in this paper; while dependent variable Y only takes binary value of 1 or 0. β_i $(i = 0, 1, ..., n)$ is the regression coefficient and can be estimated through the maximization of log-likelihood. $Pr(Y = 1|X_1, X_2, ..., X_n)$ stands for the probability of $Y = 1$, which indicates a file (class) being structural change-prone with respect to one of four change categories in Table IV. As regard to Formula (11), *Odds Ratio* (OR) is the most commonly used measure to assess the impact of each independent variable X_i on dependent variable Y (actually, $P(Y = 1)$). Once the logistic regression is performed, the OR will be estimated by the exponential function of regression coefficient for a given independent variable X_i. Similar to work in [29], [30], we use the adjusted OR, the odds ratio associated with one standard deviation increase, to provide an intuitive insight into the impact of the dependent variable X_i and is formally defined in mathematics as follows:

$$Odds(Y = 1|X_i = x) = \frac{Pr(Y = 1|X_i = x)}{1 - Pr(Y = 1|X_i = x)} \tag{12}$$

$$OR(X_i) = \frac{Odds(Y = 1|X_i = a + \sigma_i)}{Odds(Y = 1|X_i = a)} = e^{\beta_i \sigma_i} \tag{13}$$

where β_i and σ_i are respectively the regression coefficient and the standard deviation of the variable X_i, and $Odds$ is the ratio of probability that an event (e.g., a file being change-prone for a change category) occurring to the probability that an event not occurring. $OR(X_i)$ can be used to compare the relative magnitude of the effects of different independent variables, as the same unit is increased. For a given independent variable X_i (i.e., one metric in this paper), an expression $OR > 1$ means that the independent variable X_i is positively associated with the dependent variable Y; $OR = 1$ means that independent variable does not affect the dependent variable, while $OR < 1$ indicates that independent variable is negatively associated with the dependent variable. The univariate logistic regression model is a special case of the multivariate logistic regression model, where there is only one independent variable.

(3) *Analysis Method*

In this section, univariate logistic regression was used to evaluate the correlation between individual smell-based metrics and change-proneness. Meanwhile, multivariate logistic regression was employed to build three kinds of models based on only CK metrics, only smell-based metrics and their metrics in combination, respectively, when the particular dependent variable was chosen.

(3.1) *Univariate Logistic Regression Analysis for RQ1*

To avoid lower proportion of change types in the files, we did not discriminate the specific change types and particular versions, that is, all kinds of fine-grained change types (48 in total) were accumulated in a file and the data from multiple versions of a considered project are combined as a unique dataset. After all, our goal is preliminarily to find which smell-based metrics are potentially useful variables in change-proneness prediction models. In this study, a smell-based metric is considered to be significantly related with change-proneness at the significance level α of 0.05, as suggested in [27] . When performing univariate logistic regression analysis for RQ1, we use Cook's distance to identify the influential observations [31]. Cook's distance is a measure of how much the residual of all observations would change if a particular observation was removed from the calculation of the regression coefficients. If the Cook's distance of an observation is equal to or more than 1, such observation is considered as influential and is recommended to removed from dataset [31]. Moreover, for each smell-based metric, we use OR, defined in Formula (13), to quantify its effect on the change-proneness. This adjusted OR allows us to compare the relative magnitude of the effects of individual metrics on the change-proneness. In previous study, module size (i.e., file's LOC in this study) might have a potential effect on the relationships between smell-based metrics and change-proneness [4], [8]. In other words, module size may falsely obscure or accentuate the true correlations between smell-based and change-proneness. Therefore, there is a need to remove the potentially confounding effect of module size in order to understand their actual relationships, as suggested by Zhou et al [32]. For each smell-based metric, if corresponding p-value is less than 0.05 and OR is larger than 1, such metric is regarded as having significant positive correlation with change-proneness.

(3.2) *Multivariate Logistic Regression Analysis for RQ2 and RQ3*

In order to answer RQ2 and RQ3, we perform a forward stepwise variable selection procedure to build three kinds of multivariate logistic regression models, i.e., "B" model built on CK metrics, "S" model built on smell-based metrics and "B+S" model built on their combination. One common problem in multivariate logistic regression analysis is multicollinearity in which two or more independent variables are highly linearly related. These highly correlated predictors may lead to inaccurate coefficient estimation in logistic regression model. Variance inflation factor (VIF) is a widely used indicator of multicollinearity. In this study, cut-off 10 is recommended to deal with multicollinearity in a regression model [33]. Furthermore, similar to the analysis in univariate logistic regression, all raw data are checked for the presence of influential observations using Cooks distance [31]. After building the above models, we compare the prediction effectiveness for two pairs of models, i.e., "S" vs "B" and "B+S" vs "B". To conduct a realistic and comprehensive comparison, the prediction effectiveness data are often generated by three methods, i.e., cross-validation, across-version prediction and inter-project prediction. In this paper, we only use across-version prediction to generate effectiveness data, because such method is supposed to be more suitable and practical for industrial software practitioners. Specifically, across-version prediction uses a model trained on earlier versions to predict change types in follow-up versions within the same project. i.e., building a prediction model on a version v_i and then applying the model to predict change types in any follow-up version v_j $(j > i)$ of the same studied project. If a project has

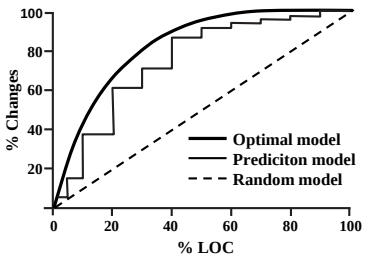

Fig. 3: LOC-based Alberg diagram

n versions, this method will produce $n \times (n-1)/2$ prediction effectiveness values for each model.

Ranking and classification are two typical scenarios of applying change-proneness prediction. In both scenarios, we evaluate the effectiveness of smell-based metrics in the context of effort-aware change-proneness prediction where we use the source lines of code (LOC) in a file to measure the effort required to inspect it. The intuition is that a larger file takes a longer time to review than a smaller one, hence smaller files should be inspected earlier if the possibility of being change-prone is the same. In practice, such inspection strategy would greatly promote work performance during software development and maintenance. Therefore, in effort-aware change-proneness prediction, the files are ordered according to their *relative change risk*, which is defined as the ratio of the predicted possibility of change-proneness by logistic regression model to its LOC. In the ranking scenario, files are ranked in order from the most to the least relative change risk. Software practitioners could simply select from the list as many potential high-risk files as available resources are allowed for software quality enhancement, e.g., integration test and unit test. In the classification scenario, files are first classified into two categories: high-risk and low-risk. After that, those files classified as high-risk are targeted for software quality enhancement. More details about these two performance indicators are described as follows.

• *CE in Ranking Scenario*

We borrowed the cost-effectiveness measure CE from Arisholm et al.'s work [34] to evaluate the effort-aware ranking effectiveness of a change-proneness prediction model. CE measure is based on the concept of the LOC-based Alberg diagram. In this diagram, the x-axis is the cumulative percentage of LOC of the files selected from the file ranking list and the y-axis is the cumulative percentage of changes about selected files. Consequently, each change-proneness prediction model corresponds to a curve in the diagram. Fig. 3 is an example LOC-based Alberg diagram showing the ranking performance of a prediction model *m* (in our context, the prediction model *m* could be the "B" model, the "S" model, and the "B+S" model). To compute CE, we also consider two additional curves, which respectively correspond to "random" model and "optimal" model. In the "random" model, files are randomly selected to inspect. In the "optimal" model, files are sorted in decreasing order according to their actual change densities (the ratio of overall changes about a particular change category occurred in the file to the LOC of that file). Based on this diagram, the effort-aware ranking effectiveness of the prediction model *m* is defined as follows [34]:

$$CE_\pi(m) = \frac{Area_\pi(m) - Area_\pi(random\,model)}{Area_\pi(optimal\,model) - Area_\pi(random\,model)} \quad (14)$$

where, $Area_\pi(x)$ is the area under the curve corresponding to model x (*m*, *random model*, *optimal model*) for a given π percentage of LOC. The range of $CE_\pi(m)$ is $[-1, 1]$ and larger value means a better ranking effectiveness while a negative $CE_\pi(m)$ indicates that the model is inferior to the random model. The cut-off value π varies between 0 and 1, depending on the amount of available resource for inspecting files. In practice, practitioners are more interested in the ranking performance of a prediction model at the top fraction. Therefore, we chose the $CE_\pi(m)$ at $\pi = 0.1$ and $\pi = 0.2$ to evaluate the performance of each model.

• *ER in Classification Scenario*

We use the effort reduction (ER) measure, originally called LOC inspection reduction (LIR) [35], to evaluate the effort-aware classification effectiveness of a change-proneness prediction model. The ER measure represents the ratio of the reduced source lines of code to inspect by using a classification model compared with a random selection to achieve the same recall of changes. In the classification scenario, only those files predicted as high-risk will be inspected for software quality enhancement.

To simplify the presentation, We assume that the system under analysis consists of n files. Let l_i be the LOC in file f_i and c_i be the number of changes about a change category in it, $1 \leq i \leq n$. For a given prediction model *m*, let p_i be 1 if the file f_i is predicted as high-risk by *m*, and 0 otherwise, $1 \leq i \leq n$. In the classification scenario, only those files predicted to be high-risk will be inspected for software quality enhancement. In this context, the effort-aware classification effectiveness of the prediction model *m* can be formally defined as follows:

$$ER(m) = \frac{Effort(random\,model) - Effort(m)}{Effort(random\,model)} \quad (15)$$

where, *Effort*(*m*) is the ratio of the total LOC in those predicted high-risk files to the total LOC in the system, i.e., $Effort(m) = (\sum_{i=1}^{n}(l_i \times p_i))/\sum_{i=1}^{n} l_i$; while *Effort*(*random model*) is the proportion of LOC to test or inspect to the total LOC in the system that a random selection model needs to achieve the same recall of changes as the prediction model *m*, i.e., $Effort(random\,model) = (\sum_{i=1}^{n}(c_i \times p_i))/\sum_{i=1}^{n} c_i$.

Noting that our aim in classification scenario is to classify each file into two categories, while logistic regression model returns a prediction probability. Thus, we need to choose a threshold for the prediction model *m*. In the literature, there are two popular methods to determine the classification threshold on a training set. The first method is called balanced-pf-pd (BPP) method. This method employs the ROC curve corresponding to model *m* to determine the classification threshold. In the ROC curve, probability of detection (*pd*) is plotted against probability of false alarm (*pf*) [40]. Intuitively, the

closer a point in the ROC curve is to the perfect classification point (0, 1), the better the model predicts. In this context, the "balance" metric is $1 - \sqrt{((0-pf)^2 + (1-pd)^2)/2}$ that can be used to evaluate the degree of balance between pf and pd [36]. For a given training data set, BPP chooses the threshold having the maximum "balance".

The second method is balanced-classification-error (BCE) method. Unlike BPP, BCE chooses the threshold to roughly equalize two classification error rates: false positive rate and false negative rate. As stated by Schein et al. [37], such an approach has the effect of giving more weight to errors from false positives in imbalanced data sets. This is especially important for change data, as they are typically imbalanced (i.e. most files do not occur any structural changes). For the simplicity of presentation, the effort reduction metrics under the BPP and BCE thresholds are respectively called "ER-BPP" and "ER-BCE". We will use "ER-BPP" and "ER-BCE" to evaluate the effort-aware classification effectiveness of a prediction model. However, it may be not adequate to use only these two thresholds for model evaluation, as there are many other possible thresholds. In practice, it is possible that a model is good under the BPP and BCE thresholds but is poor under the other thresholds. Consequently, for software practitioners, it may be misleading to use the "ER-BPP" and "ER-BCE" metrics to select the best classification model from a number of alternatives. In this paper, we use an additional metric "ER-AVG" proposed by Zhou et al. [32] to alleviate this problem. For a given model, the "ER-AVG" metric is the average effort reduction of the model over all possible thresholds on the test data set.

IV. Results and Analysis

In this section, we try to answer three research questions proposed in section III-A based on experimental results. Due to limited space, we only report evaluation results of CE_π at $\pi = 0.2$ for effort-aware *api* prediction model in ranking scenario and *ER-AVG* in classification scenario. Results from other prediction models where dependent variable respectively takes on remainder change categories (i.e., *state*, *functionality*, *statement*) can refer to our online appendix [24].

RQ1: How are the smell-based metrics correlated with structural change-proneness?

We use results from univariate regression analysis to answer RQ1. For each smell-based metric, we test its correlation with structural change-proneness for each analyzed project. Noting that we only carry out correlation analysis at project level, i.e., do not discriminate specific version data. In Table VIII, the column "Metric" represents the name of smell-based metrics, the column "Coeff.", "p-value" and "OR" represent the estimated regression coefficient, the statistical significance of the coefficient from Z test and the odds ratio associated with one standard deviation increase, respectively. From Table VIII, we can see that all smell-based metrics have significant correlation with structural change-proneness because their p-values are all less that 0.05 and OR values are more than 1. This indicates that files with a higher value of smell-based metric tend to be change-prone. Particularly, the OR values from metrics ANS and SRL are mostly larger than 2 in studies projects.

A representative example is the class *MulticastProcessor* from project `camel`. This class implements the multicast pattern to send a message exchange to a number of endpoints in which each endpoint receives a copy of the message exchange. In the starting version `2.0M2`, the LOC of this class is 292 and it is affected by two smell instances, i.e., *Speculative Generality*(1) and *Shotgun Surgery*(1). When it evolves from version `2.0M2` to `2.3.0` through `2.1.0`, the number of smells affecting it increases to 6 by adding 4 instances of *Message Chains*, accordingly, the structural changes underwent during this period are 7, 35 and 75, respectively. In subsequent version `2.6.0`, the magnitude of LOC quickly reaches to 1016 (two times as many as prior one) and 7 new smell instances are newly added, i.e., *Large Class*(1), *Speculative Generality*(1), *Message Chains*(1), *Divergent Change*(2), *Long Parameter List*(2), which lead to average 45 structural changes in follow-up versions (i.e., `2.6.0`→`2.7.0`→`2.10.5`→`2.14.0`). From the evolutionary history of this file, we can find that, the more smells a file contains, the more likely it will be subject to structural changes, which confirmed previous findings in [10], [18]. Therefore, the metric of Average Number of Smells (ANS) is indeed a suitable indicator for potentially change-prone files. On the other hand, we also find that once the smells are introduced, the developers may be aware of code smells, but are not willing to remove these smells by refactoring operations, which is consistent to findings by Peters and Zaidman [38] as well as Tufano et al. [39]. Such long lifespan of smells (because of delayed refactoring behaviour) may also explain the significant relation between metric Smell Recurrence Length (SRL) and structural changes with OR ranging from 1.6 to 2.3, indicated in Table VIII and IX.

However, the results shown in Table VIII may not reflect the true correlations of the investigated metrics with structural change-proneness, as the potentially confounding effect of file size is not taken into account [32]. Table 10 reports the univariate analysis results after removing the confounding effect of file size. As can be seen, most smell-metrics except ANS and SCM in camel and hive, are still significantly related to structural change-proneness. For example, the smell metric ASC (Average Structural Changes) is significantly correlated to structural changes due to $OR > 1$. This result is expected and also confirms the basic assumption towards "modules changing often in the past will be most likely change-prone in the future as well". However, for smell metric ANS (*Average Number of Smells*) and SCM (*Smell Complexity Metric*), their significant correlations with structural changes disappear in project hive after removing confounding effect of file size. We do not find extreme biased distribution about these two metrics and LOC of files in all versions of hive after manually inspecting their source code. The probable reason is due to other factors driving structural changes, rather than code quality enhancement, for example, user's requirement for new

features. These results suggest that the characteristics captured by smell-metrics are mostly different from file size and can be used as change-proneness indicators.

Overall, our univariate logistic regression analysis results indicate that smell-based metrics are, in most cases, significantly related to structural changes occurred in files regardless of whether the confounding effect of file size is removed or not.

RQ2: Are the smell-based metrics more effective to predict change-prone files regarding each change category than CK metrics?

In order to answer RQ2, we first build corresponding models "B" and "S" as described in section III-C. Then we compare the prediction effectiveness of the "B" and "S" models in ranking and classification scenarios under across-version prediction. Fig. 4 shows a series of boxplots which describe the distribution of the CE values at cutoff π=0.2 and ER-AVG values generated from the across-version *api* change-proneness predictions for models "B" and "S". From Fig. 4, the model built on smell-based metrics tends to perform generally better than the model built using CK metrics, specifically, we can obtain the following observations:

• *ranking performance*

For derby, hive and pig, their "S" models gain significant advantage over "B" models towards $CE_{0.2}$ with p-value $<$ 0.05 when predicting *api* change-proneness. The effect sizes are all large in terms of Cliff's δ ($0.51 < \delta < 1.68$). For camel, the median CE of "S" model is larger than "B" model, but not significantly due to the p-value more than 0.05 in Wilcoxon signed-rank test and trivial Cliff's δ ($\delta = 0.19$). In particular, for elasticsearch and lucene, we can observe that the "S" model has a comparable median CE than "B" model and their median values are fluctuating around zero, indicating the ranking performance of these two models is similar to random model.

• *classification performance*

For camel, derby, hive and pig, the "S" model has a higher median *ER-AVG* as compared with "B" model. The p-values in Wilcoxon signed-rank test are all less than 0.05 and their effect sizes are all large in terms of Cliff's δ ($0.56 < \delta < 1.23$). For elasticsearch, the median *ER-AVG* of "S" model is slightly lower than "B" model with p-value more than 0.05 and trivial Cliff's effect size ($\delta = 0.11$). For lucene, although the model "S" model has significantly *ER-AVG* than the one of "B" model (p-value $<$ 0.05), the effect size in terms of Cliff's δ is moderate ($\delta = 0.36$). After deeply investigating the causes behind the improvement of performances, we did found that the model built by smell-based metrics was able to classify more than 75% of smell instances in studied projects. These results highlight that these smell-based metrics are actually useful when predicting the change-proneness (particularly, the *api* change-proneness) of a smelly file. An interesting example regarding this characteristic is represented by class *MapReduceOper* (from version `0.11.0` of `pig`), *Reply* and *BinaryOperatorNode* (from version `10.10.1.1` of `derby`), as showed in Table V. These three classes are all affected by code smells. As expected, they all suffer from *Large Class* and *Shotgun Surgery*, particularly, class Reply also contains smells *Data Clump*(1), *Long Method*(1) and *Switch Case*(2), while class BinaryOperatorNode contains smells *Message Chains*(1) and *Divergent Change*(1). Noting that, the *api* changes occurred in these classes account for more than 13%, 50% and 51%, respectively. However, the basic model classifies these three classes as not change-prone based on CK metrics, showed in Table V, but the model "S" using smell-based metrics can classify these classes as change-prone. More interestingly, we also find that the model "S" even can correctly classify 76% of non-smelly files from predicted version as change-prone, shifting by 8% as compared with model "B". This classification ability could be explained by the fact that smell-based metrics leverage information from evolutionary history of files, especially regarding structural changes occurred in the past. However, we also observe that, for elasticsearch, their median *ER-AVG* values are around zero, which indicates their classification performance is similar to random model.

In general, smell-based metrics provide an improvement over CK metrics under ranking and classification scenarios for effort-aware *api* change-proneness prediction.

TABLE V: Comparison among class MapReduceOper (from pig), Reply (derby) and BinaryOperatorNode (from derby) in terms of CK and smell-based metrics

Class†	LOC	WMC	DIT	NOC	CBO	RFC	LCOM	ANOS	SCM	SRL	ASC	NODC
MRO	542	74	2	1	10	80	1942	2.13	0.07	66.50	11.00	9
REPLY	1441	116	1	1	5	103	1419	7.43	0.10	46.04	10.43	6
BON	835	34	4	5	17	78	190	5.50	0.09	53.25	24.50	9

†org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceOper
†org.apache.derby.client.net.Reply
†org.apache.derby.impl.sql.compile.BinaryOperatorNode

RQ3: Are the combination of smell-based metrics and commonly used code metrics more effective to predict change-prone files regarding each change category than CK metrics alone?

In order to answer to RQ3, we first use the method described in section III-C to build the models "B" and "B+S" on each version data set. Then, we use the similar way did for RQ2 to compare the prediction effectiveness of the "B" and "B+S" models in ranking and classification scenarios under across-version change-proneness prediction. Fig. 5 shows paralleled boxplots which describe the distribution of the *CEs* at cutoff π=0.2 and *ER-AVGs* generated from the across-version *api* change-proneness predictions for "B" and "B+S" models. Table VI quantitatively describes the performance comparison results in ranking scenario. In Table VI, the second and third columns represent the median $CE_{0.2}$ for the "B" model and the "B+S" model, respectively. The fourth and the fifth columns are respectively the percentage of the improvement for the "B+S" model over the "B" model and the effect sizes in terms of the Cliff's δ. The sixth column represents the p-value in the Wilcoxon tests. The meanings of columns in Table VII

are similar to the ones in Table VI. From Fig. 5, Table VI and Table VII, we have the following observations:

• *ranking performance*

It is obvious that the "B+S" model significantly outperforms the "B" model for all subject projects due to their p-values less than 0.05. Specifically, for all studied projects, the "B+S" model has larger median $CE_{0.2}$ than the "B" model. For camel, derby, hive and pig, the effect sizes are large in terms of Cliff's δ is moderate ($0.47 < \delta < 1.65$), while for elasticsearch and lucene, their effect sizes are small ($|\delta| < 0.2$). On average, the "B+S" model leads to about 90% improvement over the "B" model. Noting that, in Table VI, the entries in column of "improve" with underline means that the corresponding baseline "B" model has negative median $CE_{0.2}$, indicating inferior to random model, and hence its median $CE_{0.2}$ is replaced with the median $CE_{0.2}$ of "S" model.

• *classification performance*

From Fig. 5 and Table VII, we find that, for all studied projects, the "B+S" model has a larger *ER-AVG* as compared with "B" model with p-values less than 0.05. Their effect sizes are large ($0.49 < \delta < 1.16$) except in elasticsearch ($\delta = 0.37$). Although the performance of model "B+S" for elasticsearch is not very satisfactory, it is still better than the ones of model "B" and "S". The class *TransportIndexAction* from package `org.elasticsearch.action.index` belongs to such case. The performance achieved by model "B+S" is shifted by +44% (from 0.45 to 0.65, while 0.27 achieved by model "B"). This indicates that smell-based metrics can provide supplementary information in comparison to traditional CK metrics. Besides, on average, the "B+S" model provides an improvement by 193.84%. Again, for camel, lucene and pig, their median *ER-AVGs* in Table VII are not from "B" model but from "S" model because their "B" models are inferior to random model (*ER-AVG*<0).

Overall, the above observations indicate that the "B+S" model outperforms the "B" model in effort-aware *api* change-proneness prediction under ranking and classification scenarios. This demonstrates that the smell-based metrics considered in this paper is actually useful predictors in effort-aware change-proneness prediction.

Finally, it is worth noting that, we only report results and analysis of *api* change-proneness prediction in ranking and classification scenarios. For other change categories, we generally achieve similar conclusions. More details about their prediction results can refer to our online appendix [24].

TABLE VI: Comparison results of $CE_{\pi=0.2}$ (ranking performance) between B and B+S api change-proneness prediction models

project	B	B+S	improve	$\|\delta\|$	p-value	
camel	0.0213	0.0277	29.57%	0.4735	0.0001	√
derby	0.0367	0.0885	141.19%	0.5613	0.0008	√
elasticsearch	0.0181	0.0421	133.07%	0.2851	0.0094	√
hive	0.0392	0.0992	152.63%	1.6523	< 0.001	√
lucene	0.0100	0.0156	56.59%	0.2338	< 0.001	√
pig	0.0406	0.0520	27.97%	1.2596	< 0.001	√

TABLE VII: Comparison results of *ER-AVG* (classification performance) between B and B+S api change-proneness prediction models

project	B	B+S	improve	$\|\delta\|$	p-value	
camel	0.1198	0.1334	11.40%	0.7717	< 0.001	√
derby	0.0684	0.1197	74.92%	0.7820	0.0054	√
elasticsearch	0.0575	0.0927	61.13%	0.3722	0.0100	√
hive	0.1158	0.3442	197.20%	1.0354	0.0001	√
lucene	0.0100	0.0325	224.96%	0.4914	< 0.001	√
pig	0.0100	0.0693	593.44%	1.1675	< 0.001	√

V. Threats to Validity

This section discusses the threats to validity that can affect the results of our empirical study.

Threats to *construct validity* concern the relationship between theory and observation. In our study, this threat involves the data collection of independent and dependent variables. In our prediction model, independent variables represent CK metrics and smell-based metrics. CK metrics are extracted from tool `ck`, while code smells are detected using tool `CBSDetector` [16] and `AJCSD`. The tool `ck` and `AJCSD` are open-source tools that are developed based on JDT and they are adequately tested by our lab members. Thus, the data from these tools are reliable. As for fine-grained changes detection, we employ a state of the art tool `ChangeDistiller` [25] utilizing AST-based matching algorithm with higher detection accuracy. This tool has been used by many researchers for empirical studies [10], [18], [22]. Therefore, the construct validity of data in prediction model is satisfactory.

Threats to *internal validity* concerns the factors that could influence our observations. We did not manually examine many files containing relatively high numbers of structural changes which also contained other smells beyond our consideration in this paper. Such an examination may reveal other smells that have a greater impact on structural changes, and which may be confounding our results. Yet, we filter out some test files in which changes may not have been driven by our considered code smells, but test smells [40]. The second threat to the internal validity of our study is the unknown effect of the deviation of the independent variables from the normal distribution. In our study, we used the raw data to build the logistic regression models when investigating RQ2 and RQ3. In other words, we did not take into account whether the independent variables follow a normal distribution. The reason is that, in logistic regression, there is no assumption related to normal distribution. In addition, we also consider the confounding effect of file size (LOC) in answering RQ3, as did in RQ1. After removing the potentially confounding effect of file size, however, our conclusions about *api* change-proneness prediction still hold.

Threat to *external validity* concerns the generalization of our findings. To mitigate this threat, we conducted our experiments on six open-source systems of different size and from different domains. The experimental results drawn from these subject are quite consistent. Furthermore, the data sets collected from these systems are large enough to draw statistically meaningful conclusions. We believe that our research provides a deeper

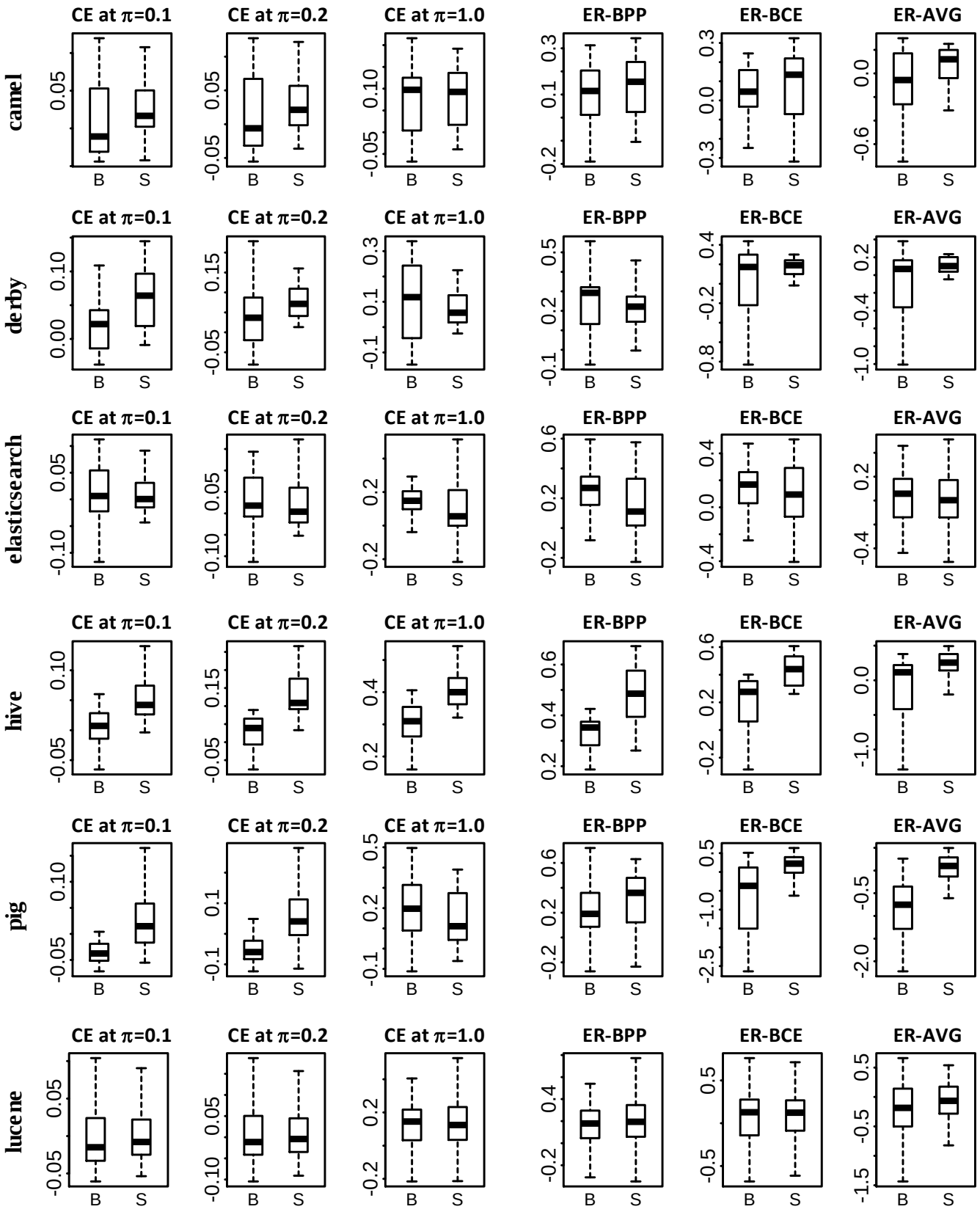

Fig. 4: B vs S evaluation under across-version change-proneness prediction with respect to api.

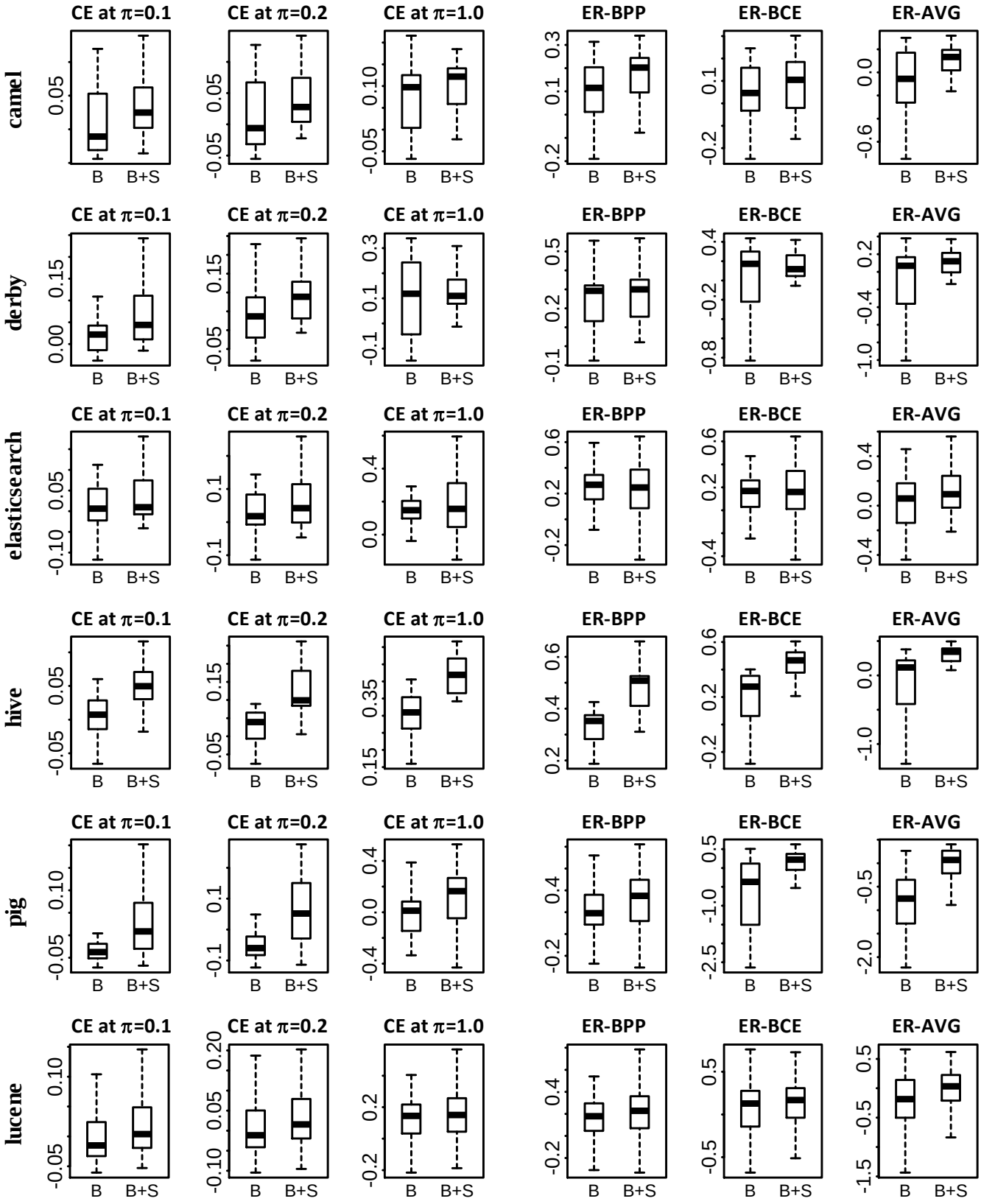

Fig. 5: B vs B+S evaluation under across-version change-proneness prediction with respect to api.

TABLE VIII: Results of Univariate Logistic Regression Analysis Before Removing the Potentially Confounding Effect of File Size (LOC)

	camel			derby			elasticsearch			hive			lucene			pig		
Metric	Coeff.	p-value	OR	Coeff.	p-value	OR	Coeff.	p-value	OR	Coeff.	p-value	OR	Coeff.	p-value	OR	Coeff.	p-value	OR
ANS	0.285	<0.001	2.847	0.108	<0.001	2.191	0.385	<0.001	2.200	0.085	<0.001	>100	0.212	<0.001	2.256	0.152	<0.001	2.463
SCM	9.620	<0.001	2.244	10.283	<0.001	1.926	6.806	<0.001	1.491	20.336	<0.001	77.196	4.188	<0.001	1.545	3.214	<0.001	1.682
SRL	0.057	<0.001	2.290	0.065	<0.001	2.366	0.053	<0.001	1.856	0.062	<0.001	2.260	0.021	<0.001	1.578	0.033	<0.001	1.760
ASC	0.056	<0.001	2.860	0.033	<0.001	2.164	0.053	<0.001	1.730	0.034	<0.001	14.401	0.030	<0.001	1.555	0.030	<0.001	1.730
NDC	0.227	<0.001	1.854	0.221	<0.001	2.004	0.816	<0.001	1.491	0.193	<0.001	2.054	0.068	<0.001	1.563	0.196	<0.001	1.656

TABLE IX: Results of Univariate Logistic Regression Analysis After Removing the Potentially Confounding Effect of File Size (LOC)

	camel			derby			elasticsearch			hive			lucene			pig		
Metric	Coeff.	p-value	OR	Coeff.	p-value	OR	Coeff.	p-value	OR	Coeff.	p-value	OR	Coeff.	p-value	OR	Coeff.	p-value	OR
ANS	0.008	0.185	1.023	0.036	<0.001	1.220	0.366	<0.001	2.028	−0.018	<0.001	0.408	0.211	<0.001	2.188	0.053	<0.001	1.254
SCM	−0.914	0.002	0.942	2.598	<0.001	1.141	5.760	<0.001	1.382	−10.398	<0.001	0.609	3.893	<0.001	1.481	0.623	0.001	1.081
SRL	0.038	<0.001	1.632	0.050	<0.001	1.828	0.050	<0.001	1.761	0.061	<0.001	2.219	0.020	<0.001	1.534	0.022	<0.001	1.415
ASC	0.016	<0.001	1.296	0.006	<0.001	1.112	0.046	<0.001	1.599	0.012	<0.001	1.503	0.028	<0.001	1.488	0.012	<0.001	1.195
NDC	0.120	<0.001	1.337	0.142	<0.001	1.464	0.739	<0.001	1.430	0.188	<0.001	2.000	0.064	<0.001	1.515	0.116	<0.001	1.296

understanding about the usefulness of smell-based metrics for effort-aware change-proneness prediction. Nonetheless, we do not claim that our findings can be generalized to all systems, as the subject systems under study might not be representative of systems in general. To mitigate this threat, we hope that other researchers will replicate our study across a wide variety of systems in the future.

VI. Conclusion

This paper reports an empirical study where five smell-based metrics have been examined for predicting effort-aware structural change-proneness. Experiments have been conducted on six typical Java open-source projects with up to 60 versions. Our findings indicate that, statistically, all the five smell-based metrics are positively correlated with structural changes (without discriminating specific change types). As expected, the smell-based metrics are better than the CK metrics in most studied projects for predicting particular change categories (i.e. *api*). Moreover, when used with the CK metrics together, smell-based metrics are more effective in predicting change-prone files under considering the effort to code inspection. In future work, we plan to validate our results in more subject projects and consider other classifiers to build more accurate change-proneness prediction models.

References

[1] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[2] Marwen Abbes, Foutse Khomh, Yann Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *European Conference on Software Maintenance and Reengineering*, pages 181–190, 2011.

[3] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *International Symposium on Empirical Software Engineering and Measurement*, pages 390–400, 2009.

[4] Steffen M Olbrich, Daniela S Cruzes, and Dag IK Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[5] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[6] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, pages 1–34, 2017.

[7] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *International Conference on Software Engineering*, pages 682–691. IEEE, 2013.

[8] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[9] Aiko Yamashita, Marco Zanoni, Francesca Arcelli Fontana, and Bartosz Walter. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *IEEE International Conference on Software Maintenance and Evolution*, pages 121–130, 2015.

[10] M. Pinzger D. Romano, P. Raila and F. Khomh. Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In *Working Conference on Reverse Engineering*, pages 437–446, 2012.

[11] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *International Conference on Evaluation and Assessment in Software Engineering*, pages 1–12. ACM, 2016.

[12] Ghulam Rasool and Zeeshan Arshad. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895, 2015.

[13] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: a review of current knowledge. *Journal of Software: Evolution and Process*, 23(3):179–202, 2011.

[14] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018.

[15] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[16] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology*, 23(4):33, 2014.

[17] Marwen Abbes, Foutse Khomh, Yann Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *European Conference on Software Maintenance and Reengineering*, pages 181–190, 2011.

[18] Huihui Liu, Bixin Li, Yibiao Yang, Wanwangying Ma, and Ru Jia. Exploring the impact of code smells on fine-grained structural change-proneness. *International Journal of Software Engineering and Knowledge Engineering*, pages 1–27, 2018 (to appear).

[19] Irene Tollin, Francesca Arcelli Fontana, Marco Zanoni, and Riccardo Roveda. Change prediction through coding rules violations. In *International Conference on Evaluation and Assessment in Software Engineeringe*, pages 61–64, 2017.

[20] Hongmin Lu, Yuming Zhou, Baowen Xu, Hareton Leung, and Lin Chen. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical software engineering*, 17(3):200–242, 2012.

[21] Daniele Romano and Martin Pinzger. Using source code metrics to predict change-prone java interfaces. In *IEEE International Conference on Software Maintenance*, pages 303–312. IEEE, 2011.

[22] Emanuel Giger, Martin Pinzger, and Harald C Gall. Can we predict types of code changes? an empirical analysis. In *IEEE Working Conference on Mining Software Repositories*, pages 217–226. IEEE, 2012.

[23] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[24] Huihui Liu, Yijun Yu, Bixin Li, Yibiao Yang, and Ru Jia. Online appendix of raw data of smell-based metrics for effort-aware structural change-proneness prediction. *URL:https://github.com/huihuiliu/change-prediction-data*, 2018.

[25] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–C743, 2007.

[26] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E Hassan, and Meiyappan Nagappan. Predicting bugs using antipatterns. In *IEEE International Conference on Software Maintenance*, pages 270–279. IEEE, 2013.

[27] Chao-Ying Joanne Peng, Kuk Lida Lee, and Gary M Ingersoll. An introduction to logistic regression analysis and reporting. *The Journal of Educational Research*, 96(1):3–14, 2002.

[28] Ruchika Malhotra and Megha Khanna. Investigation of relationship between object-oriented metrics and change proneness. *International Journal of Machine Learning and Cybernetics*, 4(4):273–286, 2013.

[29] Yibiao Yang, Yuming Zhou, Hongmin Lu, Lin Chen, Zhenyu Chen, Baowen Xu, Hareton Leung, and Zhenyu Zhang. Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? an empirical study. *IEEE Transactions on Software Engineering*, 41(4):331–357, 2015.

[30] Yibiao Yang, Mark Harman, Jens Krinke, Syed Islam, David Binkley, Yuming Zhou, and Baowen Xu. An empirical study on dependence clusters for effort-aware fault-proneness prediction. In *ACM International Conference on Automated Software Engineering*, pages 296–307. ACM, 2016.

[31] David A Belsley, Edwin Kuh, and Roy E Welsch. *Regression diagnostics: Identifying influential data and sources of collinearity*, volume 571. John Wiley & Sons, 2005.

[32] Yuming Zhou, Baowen Xu, Hareton Leung, and Lin Chen. An in-depth study of the potentially confounding effect of class size in fault prediction. *ACM Transactions on Software Engineering and Methodology*, 23(1):1–51, 2014.

[33] Michael H Kutner, Chris Nachtsheim, and John Neter. *Applied linear regression models*. McGraw-Hill/Irwin, 2004.

[34] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[35] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.

[36] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[37] Andrew I Schein, Lawrence K Saul, and Lyle H Ungar. A generalized linear model for principal component analysis of binary data. In *AISTATS*, volume 3, page 10, 2003.

[38] Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In *European Conference on Software Maintenance and Reengineering*, pages 411–416. IEEE, 2012.

[39] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of International Conference on Software Engineering*, pages 403–414. IEEE Press, 2015.

[40] Rocco Oliveto, Andrea De Lucia, Abdallah Qusef, David Binkley, and Gabriele Bavota. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *IEEE International Conference on Software Maintenance*, pages 56–65, 2012.