

Predicting failure-proneness in an evolving software product line



Sandeep Krishnan^{a,*}, Chris Strasburg^{a,b}, Robyn R. Lutz^a, Katerina Goseva-Popstojanova^c, Karin S. Dorman^d

^a Department of Computer Science, Iowa State University, Ames, IA 50011-1041, United States

^b Ames Laboratory, US DOE, Iowa State University, Ames, IA 50011-3020, United States

^c Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV 26506-6109, United States

^d Department of Statistics, Iowa State University, Ames, IA 50011-1210, United States

ARTICLE INFO

Article history:

Received 14 February 2012

Received in revised form 29 September 2012

Accepted 28 November 2012

Available online 12 December 2012

Keywords:

Software product lines

Change metrics

Reuse

Prediction

Post-release defects

Failure-prone files

ABSTRACT

Context: Previous work by researchers on 3 years of early data for an Eclipse product has identified some predictors of failure-prone files that work well. Eclipse has also been used previously by researchers to study characteristics of product line software.

Objective: The work reported here investigates whether classification-based prediction of failure-prone files improves as the product line evolves.

Method: This investigation first repeats, to the extent possible, the previous study and then extends it by including four more recent years of data, comparing the prominent predictors with the previous results. The research then looks at the data for three additional Eclipse products as they evolve over time. The analysis compares results from three different types of datasets with alternative data collection and prediction periods.

Results: Our experiments with a variety of learners show that the difference between the performance of J48, used in this work, and the other top learners is not statistically significant. Furthermore, new results show that the effectiveness of classification significantly depends on the data collection period and prediction period. The study identifies change metrics that are prominent predictors across all four releases of all four products in the product line for the three different types of datasets. From the product line perspective, prediction of failure-prone files for the four products studied in the Eclipse product line shows statistically significant improvement in accuracy but not in recall across releases.

Conclusion: As the product line matures, the learner performance improves significantly for two of the three datasets, but not for prediction of post-release failure-prone files using only pre-release change data. This suggests that it may be difficult to detect failure-prone files in the evolving product line. At least in part, this may be due to the continuous change, even for commonalities and high-reuse variation components, which we previously have shown to exist.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

A software product line displays a high degree of commonality among the products that comprise it. The products differ one from another via a set of allowed variations. The commonalities are implemented in files reused in every product, while the variations are implemented in files available for reuse in the subset of products requiring those options or alternatives.

The high degree of commonality and low degree of variations lead us to investigate whether we can learn something about predicting failure-prone files in the product line from information

about changes and failures experienced previously by the same or other products in the product line.

We perform classification of files as failure-prone and not failure-prone (two-class classification) using supervised learning methods. We define a *failure-prone file* to be a file with one or more non-trivial post-release bugs recorded. File-level predictions are then grouped at the component level to examine whether the level of reuse has an impact on the prediction of failure-proneness at the component level. For the Eclipse product line studied in this work, we classify the components based on their level of reuse: *Common* components reused in all products, *High-reuse variation* components reused in more than two products and *Low-reuse Variation* components reused in at most two products.

File-level predictions are also grouped at the product level to investigate whether the classification capability improves for different products in the product line. Data at the product level is an aggregation of data at the component level, i.e., the files in a

* Corresponding author. Tel.: +1 515 451 2338.

E-mail addresses: sandeepk@iastate.edu (S. Krishnan), cstras@iastate.edu (C. Strasburg), rlutz@iastate.edu (R.R. Lutz), Katerina.Goseva@mail.wvu.edu (K. Goseva-Popstojanova), kdorman@iastate.edu (K.S. Dorman).

product are the files of the components that belong to that particular product. Each file is in one and only one component, but may be in multiple products.

Ongoing change is typical in product lines, including the one studied here. Change proceeds along two main dimensions. The first dimension is evolution of the product line in which, as the product line matures, more products are built. These additional products typically include new features (e.g., units of functionality [1]). The changes also may propagate to other, previously built products [2]. When the changes are incorporated into the product line, the product line asset repository is updated so that future products can reuse them.

The second dimension of product line evolution is change in an individual product from one of its releases to another. This is similar to the evolution and maintenance of a single system, except that it may happen to each system in the product line.

In previous work [3], we found that even files implementing commonalities experience change on an on-going basis and that, as the product line evolves, fewer serious failures occur in components implementing commonalities than in components implementing variations. We also found that the common components exhibit less change than the variation components over time. This led us to explore, beginning in [4], whether the stabilizing behavior of the commonalities as the product line evolves supports prediction of failure-prone files.

The following research questions motivate the work reported in this paper:

- Are there any change metrics that serve as good predictors for which files are failure-prone as a product matures over releases?
- Do any of these change metrics also serve as good predictors across all the products and components in a product line over time?
- Does our ability to predict the failure-prone files improve over time across products as the product line matures?
- Does the ability to predict failure-prone files differ across components belonging to different categories of reuse?
- How do datasets with different data collection and prediction periods affect prediction performance?
- Do datasets with incrementally increasing data collection periods yield better results?

To investigate these questions, we explore here whether accurate and meaningful predictions of failure-prone files can be made, both across the sequential releases of a single product and across the various products in a product line, taking into consideration the periods of data collection and prediction. We study whether there are good predictors of failure-prone files for individual products in the product line, whether there are good predictors across the product line, and how they are related. We study whether predicting failure-prone files over shorter time gaps is easier as compared to the standard prediction of failure-prone files six months after release.

The results reported in this paper extend our previous work to evaluate failure prediction for the Eclipse product line at the product level to also consider the component level. In brief, the new contributions first reported here include: (1) results from an investigation into whether any specific learner performed significantly better than the J48 learner we previously used for classifying failure-prone files using change data on Eclipse, (2) a quantitative evaluation of differences in defect prediction performance with respect to alternative time periods for change data collection and prediction, (3) findings from analysis of defect prediction for the three categories of reuse levels described above (commonalities, high-reuse variations, and low-reuse variations) across these peri-

ods, and (4) results from experiments using incrementally increasing data collection periods.

Our data-driven investigation uses the large open-source project Eclipse. Following Chastek et al. [5], we consider Eclipse to be a product line. We distinguish evolution of a single Eclipse product from evolution of the Eclipse product line and the evolution of its components. We also build on previous work by Zimmermann et al. [6] and by Moser et al. [7]. The authors in [6] studied defects from the bug database of three early releases of an Eclipse product at both the file and package level. They built logistic regression models to predict post-release defects. At the file level, the models had mixed results, with low recall values less than 0.4 and precision values mostly above 0.6. The authors in [7] found that change metrics performed better than code metrics on a selected subset of the same Eclipse dataset, and that the performance of the J48 decision tree learner surpassed the performance of logistic regression and Naïve Bayes learners.

Following [7], we use 17 change metrics collected over different periods of Eclipse's release cycle. Existing studies have used different types of metrics for predicting failure-prone files, including code metrics [8–12], change metrics [7,13–15] and previous defects [16]. Such metrics are used either to classify files as defective or not (binary), or to predict the number of defects per file. In general, it is easier to perform classification than to predict the number of defects. In this study, we seek to classify files as failure-prone or not with the goal being to predict whether files have one or more post-release failures.

From a product line perspective, we are most interested in observing whether predictive ability improves as the product line evolves and whether the set of prominent predictors, identified by a feature selection method based on gain ratio, changes both between products and as the product line evolves over time. In the work described in this paper, we first replicate the decision tree portion of the study presented in [7] to validate previous results and then extend it by including four more recent years of data.

In our previous work [4], we used the J48 tree-based learning algorithm for prediction. Our effort in this paper is not to identify the most optimal machine learner; rather it is to investigate improvement in prediction ability in an evolving product line. However, to validate if the J48 learner is a good choice, we perform a preliminary comparison analysis of the performance of 17 machine learners. Consistent with Menzies et al. [9,17] and Lessmann et al. [18], we observe that there is no statistically significant difference between the performance of most machine learners. As a result, in this work we continue our analysis with the J48 machine learner as implemented in Weka [19].

We look at the evolution of one particular product, Eclipse Classic, over a period of 9 years. We observe the classification results during its early evolution (versions 2.0, 2.1, and 3.0), as in [7], but also look at its more recent evolution (versions 3.3, 3.4, 3.5, and 3.6). We find some overlaps and some differences between the most prominent predictors (identified based on gain ratio) over the shorter and longer time periods for these components.

We then repeat the effort for three additional products in the Eclipse product line, Eclipse Java, Eclipse JavaEE and Eclipse C/C++, across the last four years of their evolution. We perform this analysis for three types of datasets, distinguished by their data collection and prediction periods. This is new work that has not been reported previously. We observe mixed results, with very high recall and low false-positive rates when no distinction is made between pre-release and post-release defects. However, we find that the recall rates drop significantly, if we use pre-release change data to predict post-release defects. We also observe that classifying failure-prone files using incrementally increasing data collection periods does not give better results even for commonality components. All our data and results are available at [20].

Several interesting findings resulting from the investigation are described in the rest of the paper. The main observations of the work are:

- *Product evolution.* As each product evolves, there is a set of change metrics that are consistently prominent predictors of failure-prone files across its releases.
- *Product line evolution.* There is some consistency among the prominent predictors for early vs. late releases for all the considered products in the product line. For predicting post-release failure-prone files using pre-release change data, the subset of change metrics, *Bugfixes*, *Revisions* and *Age* are among the prominent predictors for all the products across most of the releases.
- *Component evolution.* Looking at the evolution of components in the different categories of reuse in the product line (i.e., commonalities, high-reuse variations and low-reuse variations), we find that there is consistency among the prominent predictors for some categories, but not among all of them. For predicting post-release failure-prone files using pre-release change data, the change metric *Bugfixes* appears to be prominent in all three categories, although not across all releases. The change metric *Age* is prominent for both high and low reuse variations but not for commonalities.
- *Prediction trends.* As the product line matures, prediction of post-release failure-prone files using pre-release change data for four products in the Eclipse product line shows statistically significant improvement in accuracy across releases, but not in recall. Similarly, components in the three categories of reuse show significant improvement in accuracy and false-positive rate but not in recall. Further, there is no statistically significant difference in performance improvement across releases among the three categories of reuse.

The rest of the paper is organized as follows. Section 2 describes Eclipse and gives the reasons for considering it as a software product line. The approach to data collection and analysis is presented in Section 3. Section 4 lists the research questions studied for this work. Section 5 discusses the evaluation of 17 machine learners to select a suitable learner for this study. Section 6 describes findings for the evolution of single products. Section 7 reports findings as the product line evolves and gives results of statistical tests to support the observations. Section 8 discusses results at the component level across the three categories of reuse. Section 9 reports the performance of prediction using incrementally increasing data collection periods. Section 10 considers threats to validity. Additional related work is discussed in Section 11. Section 12 provides a summary and discussion of broader impact in the context of software product lines.

2. Eclipse product line

A product line is “a family of products designed to take advantage of their common aspects and predicted variabilities” [21]. The systematic reuse and maintenance of code and other artifacts in the product line repository has been shown to support faster development of new products and lower-cost maintenance of existing products in many industries [22–24,21]. As the common and variation code files are reused across products, they go through iterative cycles of testing, operation and maintenance that over time identify and remove many of the bugs that can lead to failures. There is thus some reason to anticipate that the quality and reliability of both the existing products and the new products may improve over time.

The lack of available product line data, however, makes it hard to investigate such and similar claims. The availability of Eclipse

data is a noteworthy exception. The Eclipse project, described on its website as an ecosystem, documents and makes available bug reports, change reports, and source code that span the evolution of the Eclipse products.

Chastek et al. [5] were the first that we know of to consider Eclipse from a product line perspective. Eclipse provides a set of different products tailored to the needs of different user communities. Each product has a set of common features, yet each product differs from other products based on some variation features. The features are developed in a systematic manner with planned reuse for the future. The features are implemented in Eclipse as plug-ins and integrated to form products. The products in the Eclipse product line are thus the multiple package distributions provided by Eclipse for different user communities.

2.1. Products

Each year, Eclipse provides more products based on the needs of its user communities. For Java developers, the Eclipse Java package is available; for C/C++ developers, Eclipse provides the C/C++ distribution package, etc. In 2007, five package distributions were available: Eclipse Classic, Eclipse Java, Eclipse JavaEE, Eclipse C/C++, and Eclipse RCP. In 2008, two more products became available: Eclipse Modeling and Eclipse Reporting. Year 2009 saw the introduction of Eclipse PHP and Eclipse Pulsar. In 2010, Eclipse had twelve products, including three new ones: Eclipse C/C++ Linux, Eclipse SOA and Eclipse Javascript. Fig. 1's columns list the 2010 products. New products are introduced by reusing the common components and existing variation components, and by implementing any required new variations in new component files.

In this study we observe four products (Eclipse-Classic, Eclipse-C/C++, Eclipse-Java, and Eclipse-JavaEE). Each product has a release during the years 2007–2010, with Eclipse-Classic also having releases for years 2002–2004. The yearly releases of Eclipse products are given release names in addition to the release numbers: *Europa* for year 2007, *Ganymede* for 2008, *Galileo* for 2009 and *Helios* for 2010. The release numbers corresponding to each release are 3.3 for Europa, 3.4 for Ganymede, 3.5 for Galileo, and 3.6 for Helios. In the rest of the paper, to refer to a particular release of a product, we mention the release name along with the release number, i.e., Classic-3.3 (Europa), Java-3.4 (Ganymede), etc. For the older releases from 2002 to 2004 we refer to them using their release numbers, namely 2.0, 2.1 and 3.0, respectively.

2.2. Components

The products are composed of components which are implemented as plugins. For the 2010 release, the components in the Eclipse product line are shown in the first column in Fig. 1. The individual cells indicate which components are used/reused in each product.

Based on the level of reuse we observe three categories of components: commonalities, high-reuse variations and low-reuse variations. Table 1 lists the components studied in this paper, grouped by level of reuse.

The first category contains the common components reused in all products. The large component RCP/Platform is the only common component reused across all products. Henceforth in the paper, we abbreviate the RCP/Platform component to Platform.

The second category is the set of variation components with high reuse, which are reused in more than two products but not in all products. The number of products in which these components are reused increases with each subsequent release from 2007 to 2010. The components in this category are EMF, GEF, JDT, Mylyn, Webtools, XMLtools, and PDE.













	 Java	 Java EE	 C/C++	 C/C++	 RCP/Plugin	 Modeling	 Reporting	 PHP	 Pulsar	 SOA	 Javascript	 Classic
RCP/Platform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CVS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EMF	✓	✓				✓	✓	✓				
GEF	✓	✓				✓	✓					
JDT	✓	✓			✓	✓	✓		✓			✓
Mylyn	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Web Tools		✓					✓	✓		✓	✓	
Linux Tools				✓								
Java EE Tools		✓					✓					
XML Tools	✓	✓			✓		✓	✓	✓			
RSE		✓					✓					
EclipseLink		✓					✓					
PDE		✓			✓	✓	✓					✓
Datatools		✓					✓					
CDT			✓	✓								
BIRT							✓					
ECF					✓							
GMF						✓						
PDT								✓				
MDT						✓						
MTJ									✓			
Swordfish										✓		

Fig. 1. Eclipse product line for the year 2010 [<http://www.eclipse.org/downloads/compare.php>].

Table 1

List of components.

Category	Component
Common	Platform
High-reuse variation	EMF
	GEF
	JDT
	Mylyn
	Webtools
	XMLtools
	PDE
Low-reuse variation	CDT
	Datatools
	JEEtools

The third category is the set of variation components with low reuse. This category includes components that are reused only in two products, and the number of products in which they are reused does not increase with each release. The components in this category are CDT, Datatools and Java EE Tools (called JEEtools here).

3. Approach

3.1. Data collection and integration

In order to both replicate and extend the work conducted by Moser et al. [7], we collected CVS log data and bug tracking database entries from May 2001 to May 2011 for the Eclipse-Classic product. This data was partitioned into time periods corresponding with 6 months before and after the release of Eclipse 2.0, Eclipse 2.1, Eclipse 3.0, Eclipse 3.3 (Europa), Eclipse 3.4 (Ganymede), Eclipse 3.5 (Galileo), and Eclipse 3.6 (Helios). Fig. 2 shows the time periods for each release.

We extracted the same set of 17 change metrics as in [7], including identifying bug-fixes, refactorings, and changeset size as listed in Table 2. A detailed description of these metrics is given in [7]. For pre-Europa releases, i.e. releases 2.0, 2.1, and 3.0, as in [6], we mined the CVS log data by looking for four and five digit strings matching the bug IDs. For Europa and later releases, we matched six-digit strings to bug IDs. A manual review of data in-

stances showed that no entries containing the word “bug” existed which were not caught by this pattern match. Extracting the metric *Refactorings* followed Moser’s approach, namely tagging all log entries with the word “refactor” in them. Refactoring the code involves restructuring parts of the code to improve code quality while preserving its internal structure. The Age metric was calculated by reviewing all CVS log data from 2001 onward and noting the timestamp of the first occurrence of each file name.

To determine changeset size, we used the CVSPS tool [25]. This tool identifies files which were committed together and presents them as a changeset. Slight modifications to the tool were required to ensure that the file names produced in the changesets included the path information to match the file names produced by our rlog processing script.

We wrote custom scripts to parse the CVS logs, converting the log entries into an SQL database. This data, along with changesets, bugs, and refactorings, were used to compute the metric values for each file. Finally, Weka-formatted files (ARFF) were produced. We also found and corrected an error in the script we had used to extract the change data from the database into ARFF files in [4]. This error had caused the data to be extracted beyond the stated end date (beyond 6 months pre-release) for 13 of the 17 metrics. Fig. 3 provides an overview of this process.

To ensure that the data resulting from the various input sources all contained matching filenames (the key by which the data were combined), and covered the same time periods, a few on-the-fly modifications were necessary. In cases where a file has been marked “dead”, it is often moved to the Attic in CVS. This results in an alteration of the file path, which we adjusted by removing all instances of the pattern “/Attic/” from all file paths.

An artifact of using the CVS rlog tool with date filtering is that files which contain no changes during the filter period will be listed as having zero revisions, with no date, author, or other revision-specific information. This is true even if the file was previously marked “dead” on a branch. Thus, rather than examining only the date range required for each specific release, we obtained the rlog for the entire file history and determined the files which were alive and the revisions which applied to each release.

To validate our approach, we compared our resulting file set for the pre-Europa releases (2.0, 2.1 and 3.0) with the file sets

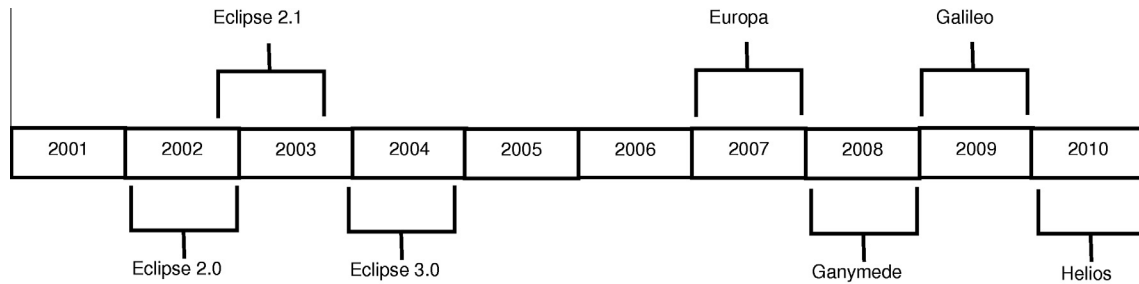


Fig. 2. Data timeline of Eclipse classic.

Table 2

List of change metrics [7].

Metric name	Description
REVISIONS	Number of revisions made to a file
REFACTORINGS	Number of times a file has been refactored
BUGFIXES	Number of times a file was involved in bug-fixing (pre-release bugs)
AUTHORS	Number of distinct authors that made revisions to the file
LOC_ADDED	Sum over all revisions of the number of lines of code added to the file
MAX_LOC_ADDED	Maximum number of lines of code added for all revisions
AVE_LOC_ADDED	Average lines of code added per revision
LOC_DELETED	Sum over all revisions of the number of lines of code deleted from the file
MAX_LOC_DELETED	Maximum number of lines of code deleted for all revisions
AVE_LOC_DELETED	Average lines of code deleted per revision
CODECHURN	Sum of (added lines of code – deleted lines of code) over all revisions
MAX_CODECHURN	Maximum CODECHURN for all revisions
AVE_CODECHURN	Average CODECHURN per revision
MAX_CHANGESET	Maximum number of files committed together to the repository
AVE_CHANGESET	Average number of files committed together to the repository
AGE	Age of a file in weeks (counting backwards from a specific release to its first appearance in the code repository)
WEIGHTED_AGE	$\frac{\sum_{i=1}^N \text{Age}(i) \times \text{LOC_ADDED}(i)}{\sum_{i=1}^N \text{LOC_ADDED}(i)}$, where $\text{Age}(i)$ is the number of weeks starting from the release date for revision i and $\text{LOC_ADDED}(i)$ is the number of lines of code added at revision i

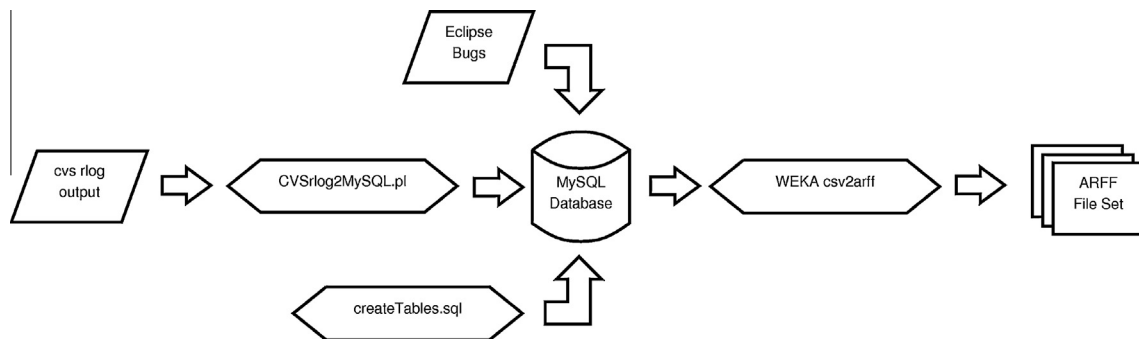


Fig. 3. Data collection process.

available from Zimmermann's work [6]. We found that there were a few differences in the two datasets due to the independent data collection processes. While most of the files were common to both datasets, there was a small subset of files which were unique to each of them. For the three components, Platform, JDT and PDE, in the 2.0 release, we included 6893 files as compared to their 6730 files. In the 2.1 release, we had 7942 files while they had 7888, and in the 3.0 release, we had 10,822 files as compared to 10,593 in theirs. Further inspection showed that there were some differences in the list of plugins included in both studies. We also observed that some files which were not present in the earlier dataset did have revisions during the development and production lifetime of the respective releases, and hence should have been included in the analysis. We thus included those in our dataset.

Moser et al. [7] use a subset of the dataset used in [6] (57% of Classic-2.0 files, 68% of Classic-2.1 files and 81% of Classic-3.0 files) and annotate it with change metrics. Since this dataset is not publicly available, we cannot compare our dataset with theirs. As discussed earlier, our dataset is comparable in size with the Zimmermann dataset in [6] and hence larger than the Moser dataset in [7].

3.2. Types of datasets

Based on the research that has been done in this area, it appears that there are different types of datasets used in previous classification studies. Some previous defect prediction studies have used datasets that divide the time period into pre-release and post-re-

lease [6,7,26–28]. In these studies, metrics are collected for a specified period before the release of the software (typically 6 months) and these metrics are used to predict the post-release defects six months after release. Other studies have used datasets which do not have such division of data into time periods. These include datasets from the NASA MDP repository [29] and the PROMISE repository [30]. MDP and PROMISE datasets provide static metrics at file (or class) level but do not distinguish between pre-release and post-release defects [33,34].

Studies using the NASA MDP and PROMISE datasets have shown good prediction performance (e.g., [9,31,32]), applying cross-validation to predict the defective files. However, the high recall rates in experiments carried out on these datasets may not be achievable in our goal of a product line project predicting future failure-prone files from past data.

Studies which have divided their data into pre-release and post-release periods have observed mixed results in terms of prediction performance. For studies on open-source systems, Zimmermann et al. [6] report that for three releases of the Eclipse system, classifying files as failure-prone or not gave low recall rates (the best being 37.9% for Eclipse 3.0) when static metrics were used. Moser et al. [7] reported much better results for the same releases of Eclipse when change metrics were used with recall rates greater than 60%. However, this dataset is not publicly available and hence the reproducibility of the results is not certain. Recently, D'Ambros et al. performed a study to provide a benchmark for existing defect prediction strategies [28]. They report high AUC values (greater than 0.85) for five open-source systems, when change metrics were used. Studies from Microsoft by Nagappan et al. [14] report very high recall and precision rates (both greater than 90%) when using change burst metrics for predicting defect-prone binaries. However, they also report that the same change burst metrics perform poorly for some open-source projects like Eclipse (recall rate of only 51%).

To check the consistency of results across datasets with different data collection and prediction periods, we experiment with three existing approaches to classifying our datasets, each involving a different time period for collecting change and defect data. For every release of the Eclipse products (i.e., 2.0, 2.1, 3.0, 3.3, 3.4, 3.5 and 3.6), we collected change and defect data for 6 months before and after release. Except for release 2.1, which was released in March, 2003 the other releases were in June of their respective years. We partition this collected change and defect data in three different ways to form the three types of datasets. We then compare results among the three types of datasets as we investigate the research questions.

- *UseAll_PredictAll*: This dataset uses the same approach as the NASA MDP and PROMISE datasets [29,30]. For this type of dataset, change data is collected for the entire twelve months (Jan-Dec) of each release. Pre-release and post-release defects are grouped into a single field. If a file has any defects associated with it, we tag the file as defective; otherwise, the file is tagged as non-defective. In this type of dataset we do not distinguish between pre-release and post-release defects. Therefore, the metric BUGFIXES is not included in the feature set, i.e., only the other 16 change metrics are included.
- *UseAll_PredictPost*: This dataset is a variant of the approach used in our earlier paper [4]. As with the previous dataset, the change data is collected for the twelve months (Jan-Dec) of each release. Pre-release defects are distinguished from post-release defects. The number of pre-release defects (defects in Jan-June) are counted and recorded in the BUGFIXES metric. If a file has any post-release defects (defects in Jul-Dec), it is tagged as defective; otherwise, the file is tagged as non-defective.

- *UsePre_PredictPost*: This dataset uses the same approach as that used by Zimmermann et al. [6] and others [7,26–28]. For this dataset, change data is collected for six months (Jan-Jun) pre-release, including the BUGFIXES metric. Again, pre-release defects are distinguished from post-release defects. If a file has any post-release defects (defects in Jul-Dec), it is tagged as defective; otherwise, the file is tagged as non-defective.

One reason for wanting to distinguish pre and post-release defects is that, since post-release defects are encountered and reported by customers, they may have a higher impact on the quality of the software as perceived by the customer. Additionally, in terms of the practical utility of prediction, projects may seek to use metrics collected from the pre-release period to predict post-release defects. Using pre-release data to predict pre-release defects or post-release data to predict post-release defects may have limited practical value.

3.3. Data analysis

The base probabilities (proportion of defective files) for all releases of all four products for the three datasets are given in Table 3. The total number of files for each release of each product is given in the third column. For the *UseAll_PredictAll* datasets, the percentage of defective files is shown in the fourth. For both the *UseAll_PredictPost* and *UsePre_PredictPost* datasets, the percentage of defective files is the same as shown in the last column. The percentage of defective files in the *UseAll_PredictAll* dataset, which includes both pre-release and post-release defects, are two to three times larger than in *UseAll_PredictPost* and *UsePre_PredictPost* datasets, for all products and releases.

In our previous work [4], the prediction was done at the product level, for each product in the product line. In this work, we perform prediction and analysis at the component level as well. Data at product level is an aggregation of data at component level, i.e., the total number of files in a product is an aggregation of the files of all the components that belong to that particular product. For example, Eclipse-Classic is composed of three components, Platform, JDT and PDE. As such, the total files for any release of Eclipse-Classic is an aggregation of all the files of Platform, JDT and PDE for that release.

Table 3
Base probability for all releases for multiple products of Eclipse.

Product	Release	Total files	<i>UseAll_PredictAll</i> (%)	<i>UseAll_PredictPost</i> and <i>UsePre_PredictPost</i> (%)
Classic	2.0	6893	54.6	26.2
	2.1	7942	45.9	23.3
	3.0	10822	47.6	23.5
	3.3	15661	32.1	16.7
	3.4	17066	32.1	16.6
	3.5	16663	24.0	11.9
	3.6	17035	18.6	8.3
C/C++	3.3	14303	36.7	18.3
	3.4	15689	37.6	21.3
	3.5	16489	32.6	16.6
	3.6	16992	30.4	10.5
Java	3.3	18972	40.4	18.1
	3.4	20492	32.4	17.8
	3.5	20836	25.8	13.7
	3.6	21178	21.2	8.6
JavaEE	3.3	35311	48.7	24.2
	3.4	39033	34.8	16.5
	3.5	39980	26.3	11.5
	3.6	41274	19.1	6.6

Table 4
Confusion matrix.

		Predicted class	
		Not failure-prone	Failure-prone
True class	Not failure-prone	$n_{11}(TN)$	$n_{12}(FP)$
	Failure-prone	$n_{21}(FN)$	$n_{22}(TP)$

We perform an initial exploration using seventeen different learners including Bayesian methods, decision tree methods, support vector techniques, neural network techniques and nearest neighbor methods. Based on the results reported in Section 5, we choose the J48 decision tree learner for the subsequent work. The prediction results are obtained using 10-fold cross validation (CV). We divide the dataset into 10 folds and use 9 folds for training and 1 fold for testing. This is done for each fold and the results of the 10 folds are averaged. For some statistical tests, we repeat the 10-fold CV multiple times as indicated in the text.

Based on the confusion matrix shown in Table 4, we use the following metrics of learner performance, consistent with [6,7].

$$PC = \frac{(n_{11} + n_{22})}{(n_{11} + n_{12} + n_{21} + n_{22})} * 100\% \quad (1)$$

$$TPR = \frac{n_{22}}{(n_{21} + n_{22})} * 100\% \quad (2)$$

$$FPR = \frac{n_{12}}{(n_{11} + n_{12})} * 100\% \quad (3)$$

$$Precision = \frac{n_{22}}{(n_{12} + n_{22})} * 100\% \quad (4)$$

The metric PC, also known as *Accuracy*, relates the number of correct classifications to the total number of files. The metric TPR, also known as *Recall*, relates the number of files predicted and observed to be failure-prone to the total number of failure-prone files. It is also known as the probability of detection. The metric *Precision* gives the number of files that are actually failure-prone within the files that are predicted as failure-prone. The measure *False Positive Rate (FPR)* relates the files incorrectly classified as failure-prone to the total number of non-failure-prone files. We use these metrics to compare our results with those by Moser et al. [7] and Zimmermann et al. [6]. In addition to these metrics, we also use the *Area Under the ROC Curve (AUC)* as a performance metric.

In addition to the prediction results obtained from 10-fold cross-validation, we identify the metrics which are most prominent. We find the Gain Ratio (GR) for each metric. GR has been found to be an effective method for feature selection [35]. Information Gain (IG) favors features with a larger number of values, although they actually have less information [36]. GR improves upon IG by normalizing it with the actual intrinsic value of the feature. Gain Ratio is calculated as

$$GR(C, a) = (H(C) - H(C|a))/H(a) \quad (5)$$

where H is the entropy function, C is the dependent variable (CLASS) and a is the feature being evaluated. We modified the J48 code in Weka to output the gain ratio weights assigned to the nodes of the tree based on the number of correctly classified files from the total number of files.

Based on the GR of the features we perform a step-wise greedy feature selection approach. We first select the feature with the highest GR to perform classification. We then add the feature with second-highest GR to the dataset and repeat the classification. If there is significant improvement in classification performance, this feature is added to the prominent predictor list. Features are added in decreasing GR order until no additional feature significantly improves classification performance. We repeat the procedure for each release of each product (or component). Note that the significance levels reported by this procedure are not literal (since pre-

dictors are pre-screened by GR and the t -test is not valid because the 10-fold CV values are not independent). As a result, this feature selection procedure neither guarantees the best set of predictors nor that each predictor actually significantly improves prediction, but it is a reasonable procedure to identify likely important predictors in a standard way.

Finally, we investigate an incremental prediction approach that uses increasing amount of change data (instead of the usual 6 months) to predict the failure-prone files in the remaining post-release months. We increment the change data period from 6 months to 11 months, in steps of 1 month, while simultaneously reducing the post-release failure-prone file data from 6 months to 1 month.

Note that in order to control the family-wise error rate (FWER) at the 0.05 level due to multiple statistical tests performed in this paper (Sections 6.3, 7.2, and 8.2 with 12 tests in each section), we use a cut-off significance value of $0.05/36 = 0.001$.

4. Research questions

This paper explores the following research questions for each of the three types of datasets described above: *UseAll_PredictAll*, *UseAll_PredictPost* and *UsePre_PredictPost*.

RQ1. Classifier selection

- (i) Is there a specific machine learner that is significantly better than other learners for classifying failure-prone files using change data?

RQ2. Single product evolution

- (i) How do our results related to learner performance compare with previously published results?
- (ii) Does learner performance improve as a single product evolves?
- (iii) Is the set of prominent predictors consistent across releases of a single product?

RQ3. Product line evolution

- (i) Does learner performance improve as the product line evolves?
- (ii) Is the set of prominent predictors consistent across products as the product line evolves?

RQ4. Evolution of components at different levels of reuse

- (i) Does the learner performance improve for components in each category of reuse (commonalities, high-reuse variation and low-reuse variation)? Does performance differ across categories of reuse?
- (ii) Is there a common set of best predictors across all categories of reuse?

RQ5. Incremental prediction

- (i) Does performing incremental prediction (increasing the period of change data collection) improve the prediction performance?

The next five sections address these five sets of research questions in turn.

5. Classifier selection

In this section, we explore RQ1 from the list of research questions. In our previous work [4], we used the J48 machine learner

to perform classification of failure-prone files. In the past, researchers have shown that prediction performance is not crucially dependent on type of classification technique used. Menzies et al. [9,17] and Lessmann et al. [18] observed that there is no statistical difference between the performance of most learners. However, there were a few learners that performed significantly worse than others.

We wanted to check whether J48 performs well enough when compared to other learners. Hence, we performed analysis similar to that of Lessmann et al. [18]. We evaluated a total of 17 classifiers over the 11 distinct component datasets identified in Table 1. The goal of this research is not to find the best classification algorithm. Hence, we do not delve into the details of each classifier. All the 17 chosen classifiers are implemented in the Weka machine learning software [19]. The classifiers used are listed in Table 5.

We evaluated the performance of the 17 classifiers over the 11 components for the 2007 Europa release. As this was part of a pilot study and as we were interested in observing the general trends, we did not consider all the releases. We measured the AUC and

the recall (TPR) values for each learner-component combination. To test whether the differences in AUC or TPR are significant, we carried out the Friedman test. A p -value $< 2.2 \times 10^{-16}$ suggested that the hypothesis of equal performances among the classifiers was unlikely to be true. This shows that there is a statistically significant difference between some pairs of learners. This was true when comparing AUC as well as TPR values. We then conducted the post hoc Nemenyi tests to find where was the difference, and represented the results with Demsar's Critical Difference (CD) diagram [37]. For 11 datasets and 17 classifiers the CD value was 7.45 at a significance level of 0.05.

The results of Nemenyi tests for the AUC and TPR values are shown in Fig. 4. When using AUC as the performance measure, we find that there is no statistical difference between the top 10 classification algorithms. Furthermore, we observe that there is no significant difference between the performance of the J48 learner and the observed best performer, RandomForest, both in terms of AUC and TPR. Since our focus is not on analysis of classifier performances, we do not present the details of the ranking of the different classifiers.

Fig. 4 shows the results for the *UseAll_PredictAll* dataset (i.e., for each component, the change metrics and defect data encompass the entire 12 months). Similar results are observed for the *UseAll_PredictPost* and *UsePre_PredictPost* data. Due to space limitations, we do not show all the results here. In all, there are 6 cases (three types of datasets and two performance metrics, AUC and TPR). Although the individual rankings differ for each case, there is no statistical difference between the performance of J48 and the best learner in 5 out of 6 cases. Only for one case is there a statistically significant difference, (AUC ranking for *UsePre_PredictPost* dataset). Since J48's performance was good overall, we continued our analysis in this paper with J48.

Table 5

List of classifiers.

Type	Classifier
Statistical	Naive Bayes
	Bayesian networks
	Logistic regression
	Bayesian logistic regression
Decision tree methods	J48
	ADTree
	LADTree
	RandomForest
Support vector methods	Voted perceptron
	SPegasos
	SMO
	RBF network
Neural network methods	IBk
Nearest neighbor methods	DecisionTable
Others	OneR
	Bagging with J48
	RandomSubSpace with J48

6. Single product evolution

In this section we discuss the performance of the J48 machine learner and the sets of prominent predictors for a single product, Classic, in the Eclipse product line. We look at each of the questions listed in RQ2 in Section 4.

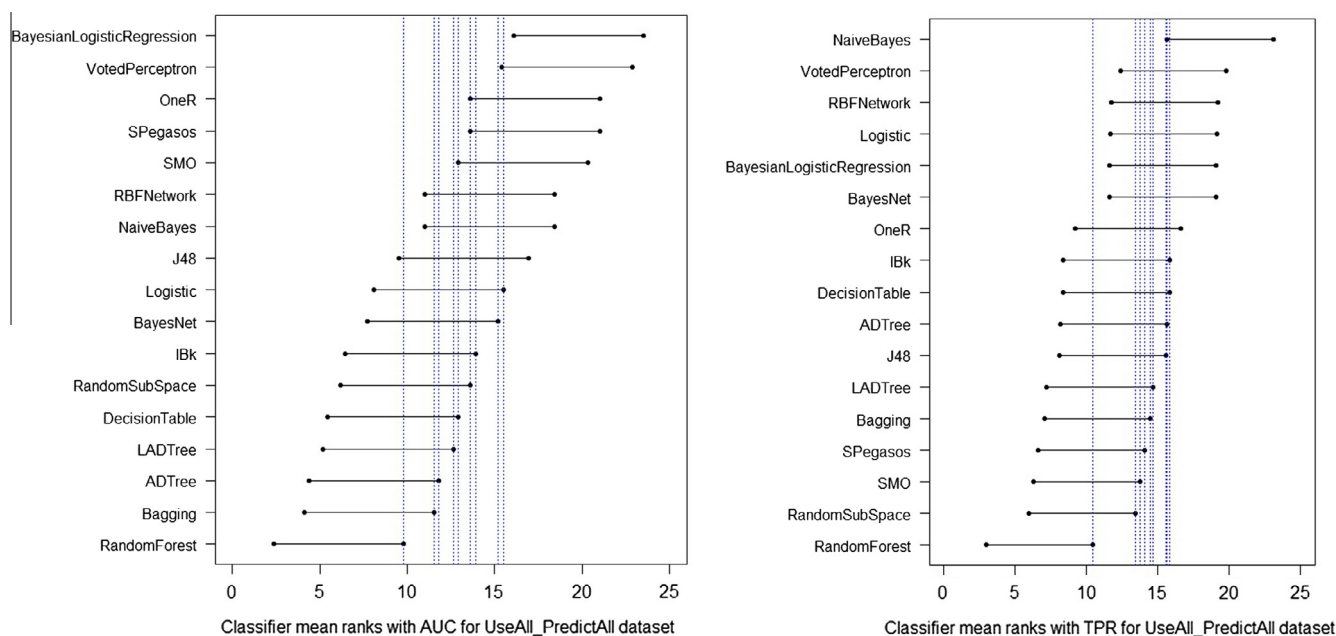


Fig. 4. Diagram for AUC and TPR Ranks of UseAll_PredictAll Dataset.

6.1. How do our results related to learner's performance compare with previously published results?

Older releases of Eclipse did not have many components. Platform, JDT and PDE were the important components, and the combination of these three components was distributed as Eclipse SDK. This combination of components is now one product called Eclipse Classic in the Eclipse product line. Moser et al. in [7] looked at three releases, 2.0, 2.1 and 3.0 of this product. We performed classification on the same three releases for this product using the J48 learner.

Table 6 compares our results with the results by Zimmermann et al. [6] and Moser et al. [7]. The authors in [6,7] used pre-release data to predict post-release defects. Hence we compare their results with our results for the *UsePre_PredictPost* dataset of Eclipse-Classic for releases 2.0, 2.1 and 3.0.

We see that our results using change data are better than the results of Zimmermann et al. [6] which are based on using static data. The values of PC and Precision are similar to theirs, while the TPR and FPR values are much better. The TPR values reported by Moser et al. [7] are higher than the TPR values we observed. It should be noted that the dataset used in [7] is significantly smaller. Because that dataset is not publicly available, we are unable to further investigate the discrepancy of the results.

A reason for the difference in results may be the different number of files used by Moser et al. and us. The datasets used in [7] consisted of significantly smaller subsets of the files in [6], i.e., 57% of the 2.0 files, 68% of the 2.1 files, and 81% of the 3.0 files, which was mentioned to be due to incomplete CVS history. Instead, we use the dataset used in [6] as a reference point. As described in Section 3.1, our datasets are comparable in size to the datasets in [6], with few differences between them.

6.2. Does learner performance improve as a single product evolves?

We next add to the analysis four additional releases of the same product, Eclipse Classic, for the three types of datasets. The results in Table 7 show values for PC, TPR, FPR and AUC over the seven years for the three datasets, *UseAll_PredictAll*, *UseAll_PredictPost* and *UsePre_PredictPost*. The comparison over the three datasets reveals that results that may look promising when using a

particular type of dataset need not hold for other types of datasets. In our case, the results are promising for the *UseAll_PredictAll* and *UseAll_PredictPost* datasets. However, when we look at more practical datasets like the *UsePre_PredictPost*, the results are much worse. PC, TPR, FPR and AUC values for *UseAll_PredictAll* and *UseAll_PredictPost* datasets are improving with time. For the later releases, the PC and TPR values are above 85% which is very promising. Similarly the FPR values are as low as 2%. Quite opposite to the other two datasets, for *UsePre_PredictPost* the TPR values for the later releases of Eclipse-Classic are worse than for the older releases. The highest TPR value for the later releases is 40% for the Ganymede release.

We used statistical methods to test for differences in learner performance in time and then estimate the magnitude of the change in performance over time for each dataset. For each release of the Classic product, we computed the average PC, TPR, and FPR of the J48 learner over a 10-fold cross-validation. To reduce the variance in these estimated statistics, we repeated the ten-fold cross-validation 1,000 times.

First, we used one-way analysis of variance (ANOVA) to test for constant mean PC, TPR, FPR and AUC across all releases. For all three datasets, this hypothesis was resoundingly rejected (p -value $< 5 \times 10^{-16}$) for all four responses. The ANOVA assumption of normality was largely satisfied, except for response TPR on the Europa release (p -value 4×10^{-4}) for the *UseAll_PredictAll* dataset, for response PC on the Ganymede release (p -value 2×10^{-3}) for the *UsePre_PredictPost* dataset, and for response PC on the Galileo release (p -value 8×10^{-3}) for the *UseAll_PredictPost* dataset. The equal variance assumption was violated for all responses of all datasets (based on Figner-Killeen test p -values $< 5 \times 10^{-16}$). As a precaution against these violated assumptions, we carried out the non-parametric Kruskal–Wallis test which does not have assumptions about distributions. The hypothesis of equal distributions was resoundingly rejected (p -value $< 5 \times 10^{-16}$) for all four responses (PC, TPR, FPR and AUC) in all three datasets.

Given that there was change in PC, TPR, and FPR across releases, we next sought to characterize the size and direction of the trend over time. Our interest is in detecting possible trends in time and since there are only seven releases (and only four in later sections), we restrict our attention to linear trends. If the temporal trend is in

Table 6
Comparison of classification performance for 2.0, 2.1, and 3.0 releases of Eclipse classic for *UsePre_PredictPost* Dataset.

Release	Moser et al. [7]				Zimmermann et al. [6]				This study			
	PC	TPR	FPR	Precision	PC	TPR	FPR	Precision	PC	TPR	FPR	Precision
Classic-2.0	82	69	11	71	77	24	27	66	79	52	11	63
Classic-2.1	83	60	10	65	79	22	24	65	81	46	8	63
Classic-3.0	80	65	13	71	71	38	34	66	80	38	7	63

Table 7
Comparison of results for newer releases (3.3–3.6) with older releases (2.0, 2.1, 3.0) of Eclipse classic.

Release	UseAll_PredictAll				UseAll_PredictPost				UsePre_PredictPost			
	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC
Classic-2.0	83.5	85.7	19.1	86.5	88.3	77.0	7.6	90.2	79.3	52.0	11.0	73.7
Classic-2.1	84.8	84.1	14.6	86.8	90.2	79.0	6.4	91.7	81.1	46.0	8.2	72.8
Classic-3.0	83.6	84.1	16.7	87.1	89.7	78.6	6.9	91.9	80.2	37.9	6.8	70.5
Classic-3.3	94.4	94.7	5.7	97.1	95.7	87.3	2.6	96.3	84.4	25.2	3.7	65.1
Classic-3.4	94.8	92.2	4.0	97	95.6	86.5	2.6	96.0	87.9	39.8	2.5	75.0
Classic-3.5	97.2	96.3	2.5	98.7	96.4	85.7	2.2	96.6	89.1	23.1	1.9	65.8
Classic-3.6	97.8	94.5	1.9	99	96.9	85.9	2.1	95.4	92.0	19.4	1.4	68.0
Estimated slope of improvement (in%)	2.0**	1.7	−2.3**	1.9**	1.2**	1.3	−0.8**	0.8**	1.5**	−3.6	−1.1**	−0.5
p -value	0.0004	0.003	0.0001	0.0002	0.0001	0.007	0.0002	0.001	0.0002	0.004	8.6×10^{-05}	0.22

fact linear, then the estimated slopes are a more parsimonious and precise summary of the trend than pairwise post hoc tests. To estimate the linear trend in PC, TPR, and FPR over time, we fit a linear mixed model to the 1000 repeated measures for each release using R package nlme [38]. We estimated a separate variance for each release. The slopes and associated p -values for testing the null hypothesis of no temporal trend are shown in the last row of Table 7. Cells marked with ** denote values that are statistically significant at the 0.001 level. For the *UseAll_PredictAll* dataset, PC increased 2.0% per year (p -value 0.0004); TPR increased 1.7% per year (p -value 0.003); FPR decreased 2.3% per year (p -value 0.0001) and AUC increased 1.9% per year (p -value 0.0002). However, for the *UsePre_PredictPost* dataset, only PC and FPR have an improving trend, whereas TPR and AUC have a worsening, but not significant, trend. PC increased 1.5% per year (p -value 0.0002); TPR decreased 3.6% per year (p -value 0.004); FPR decreased 1.1% per year (p -value 8.6×10^{-05}) and AUC decreased 0.5% per year (p -value 0.22).

For the *UsePre_PredictPost* dataset it is difficult to assess whether performance is increasing or decreasing over time. However, there is a clear reduction in the TPR and AUC for the *UsePre_PredictPost* dataset as compared to the others. Thus, training only on pre-release data makes it very difficult to successfully find post-release failures. One likely reason for the high recall rates and improving performance for the *UseAll_PredictAll* and *UseAll_PredictPost* datasets is that the changes made to correct the post-release defects are included in the change data collection period. Another possible reason for the worse performance of the *UsePre_PredictPost* dataset is the lower percentage of defects, i.e., it is a less balanced dataset. Looking back at Table 3, we can see that the percentage of defective files for the *UsePre_PredictPost* datasets is between 6–27%, almost half of the percentage for the *UseAll_PredictAll* datasets. However, since the *UseAll_PredictPost* datasets also have high recall rates, class imbalance does not appear to be as important as the period of collection of change data and prediction data here. It appears that the continuous change observed [3] even in the components that implement commonalities and high-reuse variabilities makes classification more difficult.

6.3. Is the set of prominent predictors consistent across releases of a single product?

We next explore whether the set of prominent predictors remains stable across releases for a single product in the product line, namely Eclipse Classic. To identify the prominent predictors,

we order the 17 change metrics with decreasing Gain Ratio (GR) weights, and perform a step-wise feature selection approach followed by classification of each feature selected subset using the J48 machine learner. We run the following algorithm to perform the step-wise feature selection:

1. Let m be the set of all metrics for the dataset.
2. Select $m' = \max GR(m)$.
3. Add m' to the prominent predictor list.
4. Add m' to temporary dataset d' .
5. Perform J48 classification on d' . Store result in R_1 .
6. Delete m' from m .
7. While $m \neq \phi$, repeat steps 8–12.
8. Select $m'' = \max GR(m)$.
9. Add m'' to d' .
10. Perform J48 classification on d' . Store result in R_2 .
11. If (R_2 is statistically significantly better than R_1) then {Add m'' to prominent predictor list; $R_1 = R_2$ }.
12. Delete m'' from m .
13. Output prominent predictors.

We performed the above steps for all releases of Eclipse-Classic product. For each feature-selected dataset, we performed 10-fold cross validation. To test whether a metric should be included in the prominent predictor list, we compared the performance when a new feature is added with the previous feature selected dataset (that resulted in a prominent predictor) using the t -test. The feature with the highest GR is considered as prominent by default. For example, to test if the feature with second highest GR should be included in the prominent predictor set, we do a t -test between 10 outputs of 10-fold CV for the second dataset (when the highest and second highest GR features are selected) and the 10 outputs of 10-fold CV of the dataset with only the highest GR feature. If the improvement is significant, we add the feature with second highest GR to the prominent predictor set. As multiple t -tests had to be performed, we applied a Bonferroni correction to the p -value. Since the number of t -tests to be performed were not known a priori (due to all metrics not contributing towards GR), we took a conservative approach for Bonferroni correction. A maximum of 16 t -tests would be performed if all features contribute towards GR, and each being a one-sided test to check for increase in the AUC value, we compared the p -value returned by t -tests with $0.05/16 = 0.003125$.

Results of the feature selection approach for the different releases of Eclipse Classic across the three types of datasets are shown in Tables 8 and 9. Table 8 gives the prominent predictors

Table 8
Comparison of prominent predictors for older releases of Eclipse classic.

Release	Top 3 predictors from [7]	Top predictors from this study		
		<i>UseAll_PredictAll</i>	<i>UseAll_PredictPost</i>	<i>UsePre_PredictPost</i>
Classic-2.0	Max_Changeset, Revisions , Bugfixes	Revisions , Age, Authors	Revisions , Weighted_Age	Revisions, Loc_Deleted
Classic-2.1	Bugfixes, Max_Changeset, Revisions	Revisions , Ave_Changeset	Revisions , Weighted_Age	Bugfixes, Max_Changeset
Classic-3.0	Revisions , Max_Changeset, Bugfixes	Revisions , Max_Changeset, Age	Revisions , CodeChurn	Bugfixes, Revisions

Table 9
Prominent Predictors for Newer Releases of Eclipse Classic.

Release	Top predictors		
	<i>UseAll_PredictAll</i>	<i>UseAll_PredictPost</i>	<i>UsePre_PredictPost</i>
Classic-3.3 (Europa)	Max_CodeChurn, Age, Loc_Added, Authors	Revisions , Max_Changeset, Max_Loc_Added	Revisions
Classic-3.4 (Ganymede)	Authors , Revisions, Age, Ave_Changeset	Revisions , Age, Ave_Changeset	Age, Bugfixes, Ave_Loc_Added
Classic-3.5 (Galileo)	Ave_CodeChurn, Age, Ave_Changeset, Authors	Revisions , Max_Changeset, Loc_Added, Authors	Revisions, Bugfixes
Classic-3.6 (Helios)	Authors , Ave_Changeset	Revisions , Authors, Bugfixes	Loc_Added, Age

for the older releases of Eclipse-Classic, while Table 9 gives the results for the newer releases. We find that in both tables the *UseAll_PredictAll* and *UseAll_PredictPost* datasets have a prominent predictor that is common across the respective sets of releases (*Revisions* for older releases of *UseAll_PredictAll*, *Authors* for newer releases of *UseAll_PredictAll* and *Revisions* for older and newer releases of *UseAll_PredictPost*). However the *UsePre_PredictPost* dataset does not have a prominent predictor that is common across all the considered releases. The previous study by Moser et al. [7] identified *Bugfixes*, *Revisions* and *Max_Changeset* as the most common predictors. Although it did not mention using any statistical test to check for prominence, we find that there is some overlap between those results and our results for the *UsePre_PredictPost* dataset. We also find that *Bugfixes* and *Revisions* appear as prominent in more than one release. For the newer releases, in addition to *Bugfixes* and *Revisions*, we find that *Age* also appears in more than one release.

7. Product line evolution

In this section we discuss how the performance of the machine learner and the sets of prominent predictors change as the product line evolves, looking at both of the questions in RQ3 given in Section 4. In addition to the Eclipse Classic product studied in Section 6, we applied the learning algorithm to three other products in the Eclipse product line, Eclipse Java, Eclipse JavaEE, and Eclipse C/C++.

7.1. Does learner performance improve as the product line evolves?

Figs. 5–7 show the results for PC, TPR and FPR across four years 2007–2010, for the four products in Eclipse's product line, for the three types of datasets. The X-axis shows the four products and the Y-axis shows the PC, TPR and FPR values.

As in the case with the Eclipse-Classic product, we observe that across the product line, results show an improving trend for all products in the *UseAll_PredictAll* and *UseAll_PredictPost* datasets. In terms of correctly classified instances, all products have PC rates above 94%. The true positive rates are almost all above 85% for both these datasets. False positives show very low values, less than 6% with the 2010 Helios release of the JavaEE product having the lowest FPR for both datasets. For the *UsePre_PredictPost* dataset, we see similar results as in Section 6, i.e., although the PC and FPR values are improving with time, the recall values are low and do not show improvement. The highest recall value is of 60% for the 2007 Europa release of the JavaEE product.

The plots of Figs. 5–7 appear to show some trends over time. Specifically, PC appears to increase, FPR appears to decrease, while TPR increases for two of the three datasets. To test whether this tendency is a global and significant trend across products, we regress each of these responses separately on time (release). We used a linear mixed model with random intercept to account for covariance due to repeated measures on the same product. The slope values along with the corresponding *p*-values are shown in Table 10. The estimated trends from these four years of data are similar to the results obtained from the Classic product over seven

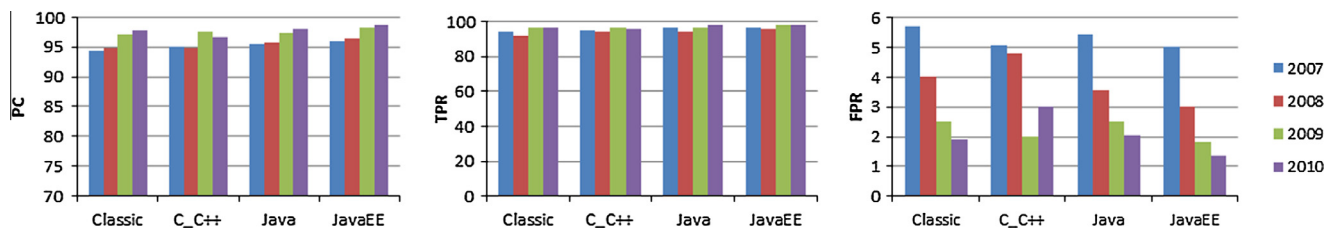


Fig. 5. PC, TPR and FPR comparison of Eclipse products across releases for *UseAll_PredictAll* dataset.

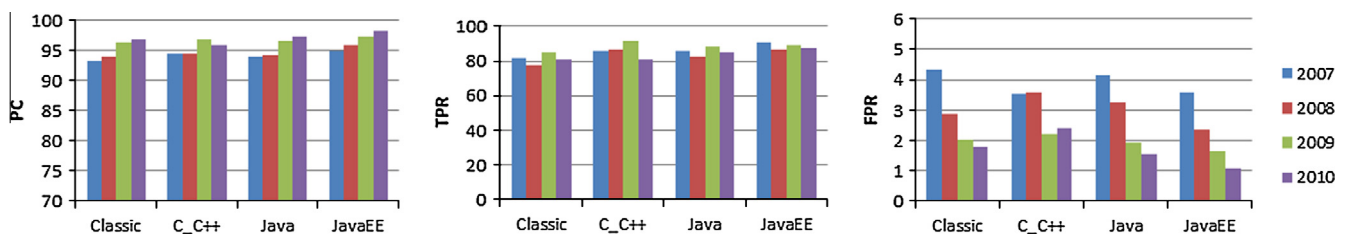


Fig. 6. PC, TPR and FPR comparison of Eclipse products across releases for *UseAll_PredictPost* dataset.

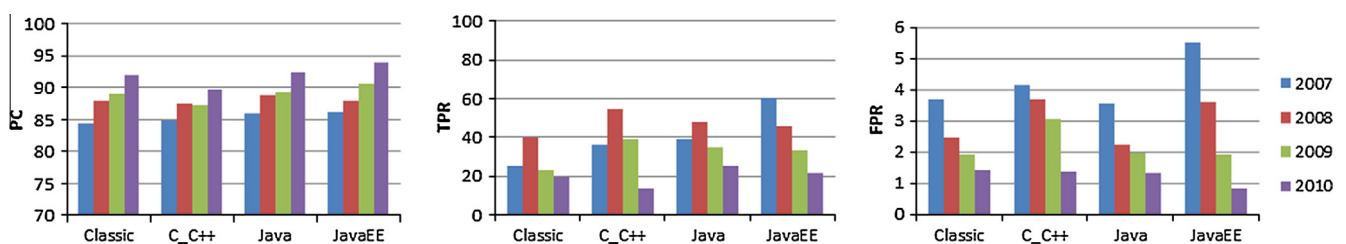


Fig. 7. PC, TPR and FPR comparison of Eclipse products across releases for *UsePre_PredictPost* dataset.

Table 10

Performance trends for all products.

Release	UseAll_PredictAll				UseAll_PredictPost				UsePre_PredictPost			
	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC
Estimated slope of improvement (in %)	0.9**	0.7	−1.1**	0.6**	1.1**	−0.2	−0.8**	0.5	1.4	−2.3	−0.6	−0.4
p-value	3.9×10^{-05}	0.05	5.6×10^{-06}	9.6×10^{-05}	2.4×10^{-05}	0.81	3.1×10^{-06}	0.04	0.009	0.39	0.03	0.79

years (Table 7), however none of the slopes estimated for the *UsePre_PredictPost* dataset are significant, that is, the predictions do not show a recognizable trend as the product line evolves.

In order to remove any bias in the results due to the changing balance of the data across releases we repeated our experiments with balanced data (equal number of positive and negative instances). All the results from the statistical tests hold when balanced data are used. The estimated trends are only marginally different from the values using unbalanced data.

Similar to Section 6.3, there is no evidence to conclude any performance trend in time for the *UsePre_PredictPost* dataset, but there is an obvious reduced TPR for the *UsePre_PredictPost* dataset at all releases relative to the others. Why this is so is a topic of current research, but it seems that ongoing change [3] is altering the patterns associated with failure as the products evolve in time. Products are made of both commonalities and variations, and it is reasonable to suspect that failure patterns are more stable in commonalities. In Section 8, we check to see if files from commonalities are easier to predict than files from variations.

7.2. Is the set of prominent predictors consistent across products as the product line evolves?

In Section 6.3 we discussed the prominent predictors of failure-prone files over time for the three types of datasets for the Eclipse-Classic product. Here we investigate whether the set of prominent predictors differs for different products in the product line. We use the algorithm explained in Section 6.3 to identify the prominent predictors.

Table 11 compares multiple products across the 2007–2010 (Europa, Ganymede, Galileo and Helios) releases. Each cell gives a summary of the prominent predictors for that particular product and in how many of the four releases they appeared as prominent. We find that for the *UseAll_PredictAll* dataset, the *Authors* metric is common across all releases of all products, followed by *Ave_Changeset* which is prominent in three releases of each product. For the *UseAll_PredictPost* dataset, *Revisions* is common appearing in 15 of 16 releases across the four products. *Authors* and *Max_Changeset* are the next most common. For the *UsePre_PredictPost* dataset, however, there is no common predictor across each product and each release. *Age* is the most common predictor, appearing in 11 of 16 releases across the four products. *Bugfixes*

and *Revisions* are the next most common prominent predictors for the product line, appearing in 9 releases and 8 releases respectively across four products.

The observations suggest that while there are predictors which are common across all releases for the *UseAll_PredictAll* and *UseAll_PredictPost* datasets, for the *UsePre_PredictPost* dataset no common predictor exists across all releases.

8. Evolution of components at different levels of reuse

We explore the learner performance and consistency of predictors for components grouped by level of reuse (Commonalities, High-reuse variations and Low-reuse variations) considering both questions listed in RQ4 in Section 4.

8.1. Does the learner performance improve for components in each category of reuse? Does performance differ across categories of reuse?

Failure prediction at the product level showed that the prediction performance is improving across time only for PC and FPR, but not for recall. Products are an aggregation of components, so we wanted to observe whether there is an improvement in prediction for components in the different reuse categories. Intuitively, we expect that the learner performance would improve for each category of reuse. Since commonalities are reused in every product, change less and have fewer defects [3], we expect the J48 learner to show better performance for higher reuse, i.e., performance improvement for commonalities to be better than high-reuse variations which in turn would be better than low-reuse variations. To explore this, we performed 10-fold cross validation using the J48 learner for the individual components.

We used a linear mixed effects model with random intercept to estimate the slope of improvement and considered the main and interaction effects of “time (year)” and “Type of reuse”. The overall increase/decrease rates for PC, TPR, FPR and AUC averaged across all components for the three types of datasets are shown in Table 12. The results are similar to the previous results obtained for products. For *UseAll_PredictAll* and *UseAll_PredictPost* datasets, we observe significant improvement trends for all the responses (with the exception of FPR and AUC for *UseAll_PredictAll*). For the *UsePre_PredictPost* dataset we see similar patterns as before, although PC is significantly improving for components.

Table 11

Prominent predictors at product level.

Dataset type	Classic	Java	JavaEE	C/C++
UseAll_PredictAll	Authors:4 Ave_Changeset:3 Age:3 Loc_Added:1 Max_CodeChurn:1 Ave_CodeChurn:1	Authors:3 Revisions:3 Ave_Changeset:2 Loc_Added:1 Age:1 Weighted_Age:1 Max_Changeset:1 CodeChurn:1	Authors:4 Age:3 Ave_Changeset:3 Revisions:1 Loc_Deleted:1 Max_Changeset:1	Authors:4 Revisions:3 Ave_Changeset:3 Age:1 Max_Changeset:1
UseAll_PredictPost	Revisions:4 Max_Changeset:2 Authors:2 Ave_Changeset:1 Age:1 Max_Loc_Added:1 Loc_Added:1 Bugfixes:1	Revisions:3 Bugfixes:3 Authors:3 Max_Changeset:2 CodeChurn:1 Age:1 Ave_Loc_Added:1	Revisions:4 Authors:4 Max_Changeset:3 Age:1 Loc_Added:1 Refactorings:1 Max_CodeChurn:1	Revisions:4 Max_Changeset:3 Authors:2 Age:2 CodeChurn:1 Ave_Loc_Added:1 Max_CodeChurn:1 Max_Loc_Added:1 Ave_Changeset:1
UsePre_PredictPost	Bugfixes:2 Revisions:2 Age:2 Ave_Loc_Added:1 Loc_Added:1	Revisions:3 Age:3 Bugfixes:2 Max_Loc_Added:1	Bugfixes:3 Age:3 Authors:1 Revisions:1 Ave_Code_Churn:1	Age:3 Revisions:2 Bugfixes:2 Authors:1 Ave_Loc_Added:1

Table 12

Performance trends for components at different levels of reuse.

Release	UseAll_PredictAll				UseAll_PredictPost				UsePre_PredictPost			
	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC
Estimated Slope of Improvement (in%)	1.9**	2.8**	−1.1	1.5	1.6**	4.0**	−0.9**	2.1**	1.4**	−2.5	−1.0	−0.7
p-value	2.1×10^{-15}	2.0×10^{-08}	0.225	0.0017	5.5×10^{-12}	1.9×10^{-07}	2.5×10^{-07}	1.4×10^{-10}	5.9×10^{-05}	0.08	0.0055	0.275

Table 13

Prominent predictors for components at different levels of reuse.

Dataset type	Commonalities	High-reuse variations	Low-reuse variations
UseAll_PredictAll	Authors:3 Ave_Changeset:2 Max_Changeset:2 Revisions:2 Max_CodeChurn:1 Age:1 Max_Loc_Added:1	Revisions:3 Authors:2 Ave_Changeset:2 Age:2 Max_Changeset:1 Ave_CodeChurn:1 Max_CodeChurn:1 Weighted_Age:1	Authors:4 Age:3 Max_Changeset:2 Revisions:1 Ave_Changeset:1
UseAll_PredictPost	Revisions:3 Max_Changeset:3 Authors:3 Weighted_Age:1 Max_CodeChurn:1 Max_Loc_Added:1 Loc_Added:1	Revisions:4 Authors:3 Max_Changeset:2 Bugfixes:2 Age:2 Code_Churn:1 Ave_Changeset:1 Loc_Added:1	Max_Changeset:3 Loc_Added:3 Weighted_Age:2 Age:1 Authors:1 Bugfixes:1 Revisions:1 Max_CodeChurn:1
UsePre_PredictPost	Bugfixes:2 Authors:2 Loc_Added:2 Age:1 Max_Changeset:1	Age:4 Bugfixes:2 Ave_Changeset:1 Weighted_Age:1 Max_Code_Churn:1	Age:3 Bugfixes:2 Weighted_Age:1 Max_Loc_Added:1 Revisions:1

We found that with time, there is an improvement in learner performance for each category of reuse for the *UseAll_PredictAll* and the *UseAll_PredictPost* datasets. Similar to the results in Table 10, most of the results for the *UsePre_PredictPost* dataset are not statistically significant. For each dataset, when comparing the different categories of reuse, we found that no category has a performance increase that is significantly less (or more) than the overall improvement rate. Hence, the values in Table 12 indicate the overall improvement rates for all three categories of reuse. In some cases, as expected, commonalities seem to be classified better than the other two categories, while for others, commonalities are classified worse, which does not confirm our intuition.

It should be noted that except for three components (Platform, JDT and PDE), other components had change data for only four releases (2007–2010). Due to limited data we are not able to conclusively say whether one category of reuse performs better than the others. In addition, the components are much smaller in size compared to products and hence we expect more noise in the data at the component level.

8.2. Is there a common set of best predictors across all categories of reuse?

Next we explore whether the set of prominent predictors differ across component categories. We use the algorithm described in Section 6.3 for feature selection.

Table 13 lists the prominent predictors for the three reuse categories, for the three types of datasets. Each cell gives a summary of the prominent predictors for that particular reuse category and in how many of the four releases they appeared as prominent. We observe that there is some overlap among the prominent predictors for the three reuse categories. For the *UseAll_PredictAll* dataset, the metric *Authors* is prominent and common across all three reuse categories. Similarly for the *UseAll_PredictPost* dataset, *Bugfixes* and *Max_Changeset* are common and prominent across all reuse categories. For the *UsePre_PredictPost* dataset, the metric *Bugfixes* is common across all reuse categories, although it appears as prominent in only two of the four releases (2007–2010). Additionally, the metric *Age* is also common between the two types of variations (high-reuse and low-reuse) and appears in three or more releases. *Age* is prominent for Commonalities in only a single release. This indicates that while there are some metrics that are prominent

across all reuse categories, there are also differences among the prominent predictors for the different reuse categories.

9. Prediction with incrementally increasing data collection periods

In this section we explore RQ5. Results in Sections 6–8 showed that predicting post-release failure-prone files using pre-release change data gives low recall values. In this section we investigate whether increasing the period of collecting change data improves the prediction of failure-prone files. The *UsePre_PredictPost* type of datasets use 6 months pre-release data to predict failure-prone files 6 months post-release. We would like to investigate whether using post-release change data in monthly increments, combined with pre-release change data helps to better classify post-release failure-prone files in the remaining months. In our incremental approach we begin from the *UsePre_PredictPost* dataset (i.e. using 6 months pre-release change data to predict 6 months post-release failure-prone files). We increment the change data period from 6 months to 11 months in increments of 1 month, while simultaneously reducing the post-release failure-prone file data from 6 months to 1 month, i.e. our final dataset will have 11 months of change data to predict failure-prone files in the 12th month.

Fig. 8 shows the results of incremental prediction for the four products in the product line. We find that increasing the period of change data does not improve recall values. One possible reason is that as the period of change data increases (from 6 months to 11 months), the number of files that are failure-prone in the remaining months reduces. As a result the J48 learner may not have a sufficient number of defective files to learn from. We find that for the last two iterations the recall values drop as compared to the first four iterations.

Similar results are observed for the three reuse categories, as shown in Fig. 9. Even commonalities, which should change less and hence have a good classification performance, show low recall values. In fact, the recall values for commonalities are in some cases lower than for the other two reuse categories. High-reuse variations have the highest recall values.

Results from Sections 6.3 and 7.2 indicated that using only pre-release change data to predict post-release failure-prone files is difficult. The results presented in this section indicate that even when post-release change data are added to pre-release change data predictions do not improve.

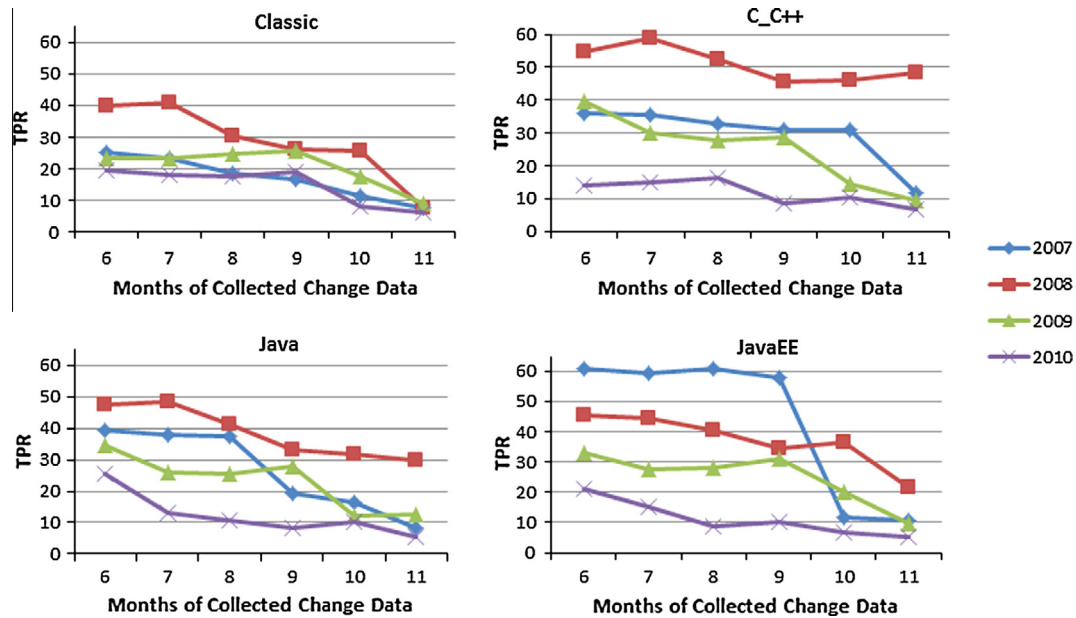


Fig. 8. Incremental prediction for four Eclipse products.

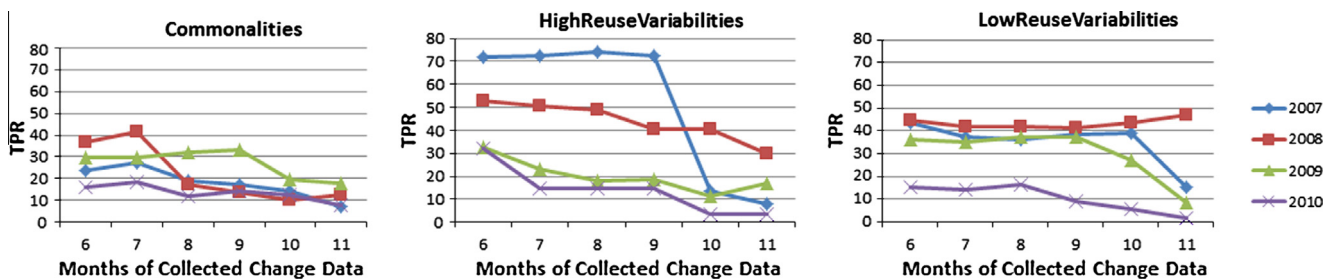


Fig. 9. Incremental prediction for three reuse categories.

10. Threats to validity

This section discusses the threats to validity of the study.

10.1. Construct validity

A threat to the construct validity is the limited number of releases in the study. While analyzing more releases might give additional insight into the trends, the 2007–2010 releases provide a representative picture of the current product line situation. We did not include the minor quarterly releases into our analysis because there were fewer users downloading them and because the entries in the bug database for these minor releases were missing data for several components. Furthermore, some of the minor releases reported higher numbers of failures while others did not report any. We plan to observe future releases as they become available and incorporate the new data for analysis.

As mentioned by Moser et al. [7], a possible threat to construct validity could be the choice of metrics used in this study. We followed [7] in using a particular set of change metrics. In general, there could be other change metrics that give different results. We believe that our results are comparable to results from previous studies which evaluate the performance of different metric sets in terms of classification of failure-prone files. Arisholm et al. [39] observe process metrics to be the best metric set. However, they also report low recall and precision values (in the range of 36–62%) when using process metrics.

10.2. Internal validity

Inaccuracies in our data collection process at one or more steps could be one of the possible threats to internal validity of this study. We performed manual and automated inspections on our dataset to verify and validate its accuracy, including comparison with data provided by Zimmermann et al. [6].

10.3. Conclusion validity

A threat to the conclusion validity of this study may be that we performed analysis using only one machine learning algorithm, namely J48. Moser et al. [7] additionally used Naïve Bayes and logistic regression learners but found J48 to give the best results. In this paper we also analyzed the performance of 17 machine learners, including J48 and found that there is no statistically significant difference between the performance of J48 and learners with higher mean rank (e.g., Random Forest).

Another possible threat to the conclusion validity is the class-imbalance problem. The datasets used in this study are imbalanced, i.e. the proportion of defective files is smaller than the percentage of non-defective files. Several studies have identified that the learner performance improves when trained on balanced data, using techniques such as over-sampling and under-sampling [17,40]. A point to note in this case, is that our emphasis in this work is on the trends in prediction performance as the product line evolves. We carried out the tests to check performance trends (Sec-

tions 6.3, 7.2 and 8.2) on both balanced and imbalanced datasets and found that the slopes of the trends (either improving trend or worsening trend) hold for both datasets. While the performance may be improved using balancing methods, it appears that the trends in defect prediction as the product line evolves do not depend on the balancing of datasets.

A typical threat to conclusion validity relates to the validity of the assumptions of the statistical tests and errors in statistical conclusions. As much as possible, we analyzed the validity of the statistical assumptions. Given the small number of releases, the linear mixed effects models parsimoniously account for some correlation among responses due to repeated temporal measures on the same product or component, but far more complex correlation is likely. Releases close in time are more likely to share common characteristics. Further, many files, especially high reuse files, are included in multiple products and hence contribute to multiple performance responses. Improper modeling of the covariance can have a large impact on estimated significance levels. The component datasets do not suffer from the potentially unaccounted covariance due to shared files because each file exists in only one component, so the component p -values are likely the most reliable.

Finally, we included releases from products spanning 2002–2010, but only the Classic product and its components were available prior to 2005, and the years 2005 and 2006 were not sampled. In the future, we aim to collect data for more products and releases and check whether the results still hold true.

10.4. External validity

An external validity threat to this study is the extent to which these observations can be generalized to other product lines. Eclipse is a large product line with many developers in an open-source, geographically distributed effort. This may mean that the development of the Eclipse product line is more varied in terms of the people involved and the development techniques used than in commercial product lines. Chastek, McGregor and Northrop consider the open-source development to be largely beneficial in terms of quality [5]. We hope to study other open-source software product lines and have studied an industrial software product line [41] to learn more about reuse, change and reliability in product lines. We have made our dataset public so that other researchers can validate the results of this study and/or use it to conduct other types of analysis.

11. Related work

There have been few studies that consider defects in software product lines or mine their bug/change tracking databases. As noted earlier, the lack of available datasets for product lines has seriously hampered investigation of the relationships between product evolution and product quality, including the ability to predict failure-proneness. Inaccessibility of datasets is a pervasive problem in many areas. For example, Catal and Diri recently reported that only 31% of the 74 papers they reviewed used public datasets, making it difficult to reproduce or extend results [42].

With regard to product lines, Mohagheghi and Conradi [43,44], compared the fault density and stability (change) of the reused and non-reused components in a system developed using a product family approach. They found that reused components have lower fault density and less modified code as compared to non-reused components.

Recently we have also studied pre-release software faults in an industrial software product line [41]. Our results showed that in a software product line setting, faults are more highly correlated to change metrics than to static code metrics. Also, variation components unique to individual products had the highest fault density

and were most prone to change. We also showed that development and testing of previous products benefited the new products in the software product line.

Besides the work of [6] and [7] described previously, several different approaches for defect prediction also have used Eclipse as the case study, giving additional insights into the role of various product and process metrics in the Eclipse product line. D'Ambros, Lanza and Robbes analyzed three large Java software systems, including Eclipse JDT Core 3.3, using regression modeling, and found correlations between change coupling (files that change together) and defects [45]. They found that Eclipse classes have, on average, many more changes and more shared transactions than classes in the other two systems studied. Kim et al. recently found that the number of bug fixes in three large open-source systems, one of them Eclipse JDT, increases after refactorings [46]. Schroter et al. found that their predictive models (regression models and support vector machines) trained in one version can be used to predict failure-prone components in later versions (here, from version 2.0 to 2.1 of Eclipse) [47]. Eaddy et al. found a moderate to strong correlation between scattering (where the implementation of a cross-cutting concern is scattered across files) and defects for three case studies, one of which was an Eclipse component [48]. Shihab et al. reported work to minimize the number of metrics in their multivariate logistic regression model [15]. In a case study on the Eclipse dataset in [12], they identified four code and change metrics. One change metric, i.e., total prior changes in the 6 months before the release, was in their short list.

Studies reported in [6,11,14,12,49] have used bug reports and bug repositories such as Bugzilla for predicting defects and failures. Jiang, Menzies, Cukic and others [50,10] have used machine learning algorithms successfully to perform defect prediction. Ostrand et al. were able with high accuracy to predict the number of faults in files in two large industrial systems [51]. Menzies et al. found that a lower number of training instances provided as much information as a higher number for predicting faulty code modules [17]. Zhang predicted the number of future component-level defects reasonably well using a polynomial regression-based model built from historical defect data [52].

There has been a significant amount of work in the area of fault-proneness and/or failure-proneness prediction (often referred to as defect prediction) for both open-source and commercial software. With regard to open-source systems, Mockus et al. [53] investigated the effectiveness of open-source software development methods on Apache in terms of defect density, developer participation and other factors. They showed that for some measures of defects and changes, open-source systems appear to perform better while for other measures, the commercial systems perform better. Paulson et al. [54] compared the growth pattern of open-source systems with that for commercial systems. They found no significant difference between the two in terms of software growth, simplicity and modularity of code. They found, however, that in terms of defect fixes, open-source systems have more frequent fixes to defects. Rahmani, Azadmanesh and Najjar compared the prediction capability of three reliability models on failure reports for five open source software systems, finding that the failure patterns for open-source software follow a Weibull distribution [55].

With regard to commercial systems, Fenton and Ohlsson [56] related the distribution of faults to failures and the predictive accuracy of some widely used metrics. They found that pre-release faults are an order of magnitude greater than the operational failures in the first twelve months. Lutz and Mikulski [57] analyzed safety-critical anomalies in seven spacecraft and found that serious failures continue to occur with some frequency during extended operations. Recently, Hamill and Goseva-Popstojanova [58] conducted a study of two large systems to identify the distribution of different types of software faults and whether they are

localized or distributed across the system. They analyzed different categories of faults and their contribution to the total number of faults in the system. Borretzen and Conradi [59] performed a study of four business-critical systems to investigate their fault profiles. They classified the faults into multiple categories and analyzed the distribution of different types of faults.

Finally, Nagappan, Ball and Zeller have shown that predictors obtained from one project are applicable only to similar projects [11]. Products in a product line are certainly similar (i.e., share commonalities), but further investigation is needed to understand under what circumstances predictors from one product in a product line are relevant to other products in the product line.

12. Conclusion

The work reported in this paper considers Eclipse as an evolving product line and distinguishes evolution of a single Eclipse product (Eclipse Classic) from evolution of the Eclipse product line and the evolution of its components. We study the performance of the J48 learner across these three evolution dimensions for a large set of change metrics extracted from Eclipse. A comparison is also made between the results for three types of datasets that differ in the data collection and prediction periods.

The research questions addressed are: (1) Whether there is a difference between the performance of different machine learners in classifying failure-prone files using change data, and whether any particular learner is better than others, (2) whether learner performance improves, i.e., whether the ability to predict failure-prone files improves as the products and components mature over time, (3) whether change metrics serve as consistent predictors for individual products as they mature over time, (4) whether any of these change metrics also serve as consistent predictors across all the products as the product line matures over time, (5) whether any of these change metrics serve as consistent predictors across the components in different categories of reuse, and (6) whether using change data that encompasses incrementally a larger time period improves prediction of failure-prone files.

The highlights of the observations from the study are summarized as follows:

In previous work, we used the J48 decision tree learner to classify failure-prone files. In experiments with other learners, in this paper, we found that there is no statistically significant difference between the performance of J48 and learners which perform slightly better (e.g., Random Forest).

A replication study, comparing our results with results from previous studies for the same releases of Eclipse-Classic showed that while change metrics were better predictors than static metrics, predicting post-release failure-prone files using pre-release data led to low recall rates. Although accuracy and false-positive rates were impressive, the low recall rates suggest that it was difficult to classify failure-prone files effectively based on pre-release change data.

A comparison between different types of datasets distinguished by the data collection and prediction period showed that datasets that do not distinguish pre-release period with post-release period (similar to MDP) have better performance with respect to accuracy, recall and false-positive rate.

From the product line perspective, prediction of failure-prone files for four products in the Eclipse product line based on pre-release data did not show a recognizable trend across releases (i.e., the estimated trends were not statistically significant).

When comparing the prediction trends among the three categories of reuse (i.e., commonalities, high-reuse variations and low-reuse variations), the results showed statistically significant improvement in accuracy, but not statistically significant trends for the other performance metrics.

As each product evolved, there was a set of change metrics that were consistently prominent predictors of failure-prone files across its releases. This set was different for the different types of datasets (with respect to change and defect data collection period) considered in this study.

There was some consistency among the prominent predictors for early vs. late releases for all the considered products in the product line. This set was different for the different types of datasets considered here. For predicting post-release failure-prone files using pre-release change data, the subset of change metrics, *Bugfixes*, *Revisions* and *Age* was among the prominent predictors for all the products across most of the releases.

Looking at the evolution of the different categories of components in the product line (i.e., commonalities, high-reuse variations and low-reuse variations), we found that there was consistency among the prominent predictors for some categories, but not among all categories. For predicting post-release failure-prone files using pre-release change data, the change metric *Bugfixes* appeared to be prominent in all three categories, although not across all releases. Metrics such as *Age* were prominent across more than one category but not across all three of them.

It is still unclear whether it will become possible to detect post-release failure-prone files across the products in an evolving product line based on pre-release data. The high level of reuse in product lines which encourages that hope is offset by the on-going change and failures seen even in files implementing commonalities. The results of the current study suggest that further investigation of failure prediction in both open-source and proprietary product lines may yield a better understanding of how evolution of individual products affects the prediction of failure-prone files within product lines.

Acknowledgments

We thank the reviewers for several helpful suggestions that improved this work. This work was supported by National Science Foundation grants 0916275 and 0916284 with funds from the American Recovery and Reinvestment Act of 2009. Part of this work was performed while the third author was visiting the California Institute of Technology and the Open University, UK.

References

- [1] D.S. Batory, D. Benavides, A.R. Cortés, Automated analysis of feature models: challenges ahead, *Commun. ACM* 49 (12) (2006) 45–47.
- [2] Z. Stephenson, Change Management in Families of Safety-Critical Embedded Systems, Ph.D. thesis, University of York, 2002.
- [3] S. Krishnan, R. Lutz, K. Goševa-Popstojanova, Empirical evaluation of reliability improvement in an evolving software product line, in: *Mining Software Repositories*, MSR, 2011, pp. 103–112.
- [4] S. Krishnan, C. Strasburg, R.R. Lutz, K. Goseva-Popstojanova, Are change metrics good predictors for an evolving software product line? in: *PROMISE*, vol. 7, 2011b.
- [5] G. Chastek, J. McGregor, L. Northrop, Observations from viewing eclipse as a product line, in: *Proceedings on the Third International Workshop on Open Source Software and Product Lines*, 2007, pp. 1–6.
- [6] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007.
- [7] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: *International Conference on Software Engineering*, 2008a, pp. 181–190.
- [8] N. Nagappan, T. Ball, Static analysis tools as early indicators of pre-release defect density, in: *Proceedings of the 27th International Conference on Software Engineering*, ICSE'05, ACM, New York, NY, USA, 2005, pp. 580–586. ISBN 1-58113-963-2.
- [9] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, *IEEE Trans. Softw. Eng.* 33 (1) (2007) 2–13. ISSN 0098-5589.
- [10] T. Menzies, Z. Milton, B. Turhan, B. Cucic, Y. Jiang, A. Bener, Defect prediction from static code features: current results, limitations, new approaches, *Autom. Softw. Eng.* 17 (2010) 375–407. ISSN 0928-8910.

- [11] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, ACM, New York, NY, USA, 2006, pp. 452–461. ISBN 1-59593-375-1.
- [12] T. Zimmermann, N. Nagappan, A. Zeller, Predicting bugs from history, in: *Predicting Bugs from History*, Springer, 2008, pp. 69–88. ISBN 9783540764397.
- [13] R. Moser, W. Pedrycz, G. Succi, Analysis of the reliability of a subset of change metrics for defect prediction, in: *ESEM, 2008b*, pp. 309–311.
- [14] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, B. Murphy, Change Bursts as Defect Predictors, in: *ISSRE, 2010*, pp. 309–318.
- [15] E. Shihab, Z.M. Jiang, W.M. Ibrahim, B. Adams, A.E. Hassan, Understanding the impact of code and process metrics on post-release defects: a case study on the Eclipse project, in: *ESEM, 2010*.
- [16] S. Kim, T. Zimmermann, J. Whitehead, A. Zeller, Predicting faults from cached history, in: *ICSE, 2007*, pp. 489–498.
- [17] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, Y. Jiang, Implications of ceiling effects in defect predictors, in: *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering, PROMISE '08*, ACM, New York, NY, USA, 2008, pp. 47–54. ISBN 978-1-60558-036-4.
- [18] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, *IEEE Trans. Softw. Eng.* 34 (4) (2008) 485–496.
- [19] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update, *SIGKDD Explor. Newsl.* 11 (2009) 10–18. ISSN 1931-0144.
- [20] Data used for this study, 2011. <<http://www.cs.iastate.edu/~ss/EclipsePLPredictionData.tar.gz>>.
- [21] D.M. Weiss, C.T.R. Lai, *Software Product-Line Engineering: A Family-based Software Development Process*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-69438-7.
- [22] Software Engineering Institute, *Software Product Lines*. <<http://www.sei.cmu.edu/productlines/>>.
- [23] H. Goma, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004. ISBN 0201775956.
- [24] K. Pohl, G. Böckle, F.J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 3540243720.
- [25] D. Mansfield, CVSps-Patchsets for CVS. <<http://www.cobite.com/cvpsps/>>.
- [26] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy, Cross-project defect prediction, in: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM, 2009.
- [27] N. Nagappan, T. Ball, B. Murphy, Using historical in-process and product metrics for early estimation of software failures, in: *ISSRE, 2006b*, pp. 62–74.
- [28] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, *Empirical Softw. Eng.* 17 (4–5) (2012) 531–577.
- [29] NASA IV&V Metrics Data Program. <<http://mdp.ivv.nasa.gov>>.
- [30] PROMISE repository. <<http://promisedata.org/>>.
- [31] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, D.R. Cok, Local vs. global models for effort estimation and defect prediction, in: *ASE, 2011*, pp. 343–351.
- [32] N. Bettenburg, M. Nagappan, A.E. Hassan, Think locally, act globally: improving defect and effort prediction models, in: *MSR, 2012*, pp. 60–69.
- [33] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic review of fault prediction performance in software engineering, in: *IEEE Transactions on Software Engineering 99 (PrePrints)*, ISSN 0098-5589.
- [34] D. Gray, D. Bowes, N. Davey, Y. Sun, B. Christianson, The misuse of nasa metrics data program data sets for automated software defect prediction, in: *EASE, 2011*.
- [35] S. Shivaji, J. Whitehead, R. Akella, S. Kim, Reducing features to improve bug prediction, in: *ASE, 2009*, pp. 600–604.
- [36] H. Wang, T.M. Khoshgoftaar, R. Wald, Measuring robustness of feature selection techniques on software engineering datasets, in: *IRI, 2011*, pp. 309–314.
- [37] J. Demšar, Statistical comparisons of classifiers over multiple data sets, *J. Mach. Learn. Res.* 7 (2006) 1–30. ISSN 1532-4435. <<http://dl.acm.org/citation.cfm?id=1248547.1248548>>.
- [38] J. Pinheiro, D. Bates, S. DebRoy, D. Sarkar, R Development Core Team, NLME: linear and nonlinear mixed effects models, R package version 3.1-103, 2001.
- [39] E. Arisholm, L.C. Briand, E.B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, *J. Syst. Softw.* 83 (1) (2010) 2–17.
- [40] C. Drummond, R. Holte, C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling, in: *Workshop on Learning from Imbalanced Datasets, 2003*.
- [41] T.R. Devine, K. Goseva-Popstajanova, S. Krishnan, R.R. Lutz, J.J. Li, An empirical study of pre-release software faults in an industrial product line, in: *International Conference on Software Testing, Verification and Validation, 2012*.
- [42] C. Catal, B. Diri, A systematic review of software fault prediction studies, *Expert Syst. Appl.* 36 (4) (2009) 7346–7354.
- [43] P. Mohagheghi, R. Conradi, An empirical investigation of software reuse benefits in a large telecom product, *ACM Trans. Softw. Eng. Methodol.* 17 (2008) 13:1–13:31.
- [44] P. Mohagheghi, R. Conradi, O.M. Killi, H. Schwarz, An empirical study of software reuse vs. defect-density and stability, in: *Proceedings of the 26th International Conference on Software Engineering, ICSE '04*, IEEE Computer Society, Washington, DC, USA, 2004, pp. 282–292. ISBN 0-7695-2163-0.
- [45] M. D'Ambros, M. Lanza, R. Robbes, On the relationship between change coupling and software defects, in: *Proceedings of the 2009 16th Working Conference on Reverse Engineering, WCRE '09*, IEEE Computer Society, Washington, DC, USA, 2009, pp. 135–144. ISBN 978-0-7695-3867-9.
- [46] M. Kim, D. Cai, S. Kim, An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution, in: *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, 2011, pp. 151–160.
- [47] A. Schröter, T. Zimmermann, A. Zeller, Predicting component failures at design time, in: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE '06*, ACM, New York, NY, USA, 2006, pp. 18–27. ISBN 1-59593-218-6.
- [48] M. Eaddy, T. Zimmermann, K.D. Sherwood, V. Garg, G.C. Murphy, N. Nagappan, A.V. Aho, Do crosscutting concerns cause defects?, *IEEE Trans Softw. Eng.* 34 (4) (2008) 497–515.
- [49] P.J. Guo, T. Zimmermann, N. Nagappan, B. Murphy, Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows, in: *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE'10*, vol. 1, ACM, New York, NY, USA, 2010, pp. 495–504.
- [50] Y. Jiang, B. Cukic, T. Menzies, Can data transformation help in the detection of fault-prone modules?, in: *Proc of the 2008 Workshop on Defects in Large Software Systems, DEFECTS '08*, ACM, New York, NY, USA, 2008, pp. 16–20. ISBN 978-1-60558-051-7.
- [51] T.J. Ostrand, E.J. Weyuker, R.M. Bell, Predicting the location and number of faults in large software systems, *IEEE Trans. Softw. Eng.* 31 (2005) 340–355. ISSN 0098-5589.
- [52] H. Zhang, An initial study of the growth of Eclipse defects, in: *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08*, ACM, New York, NY, USA, 2008, pp. 141–144. ISBN 978-1-60558-024-1.
- [53] A. Mockus, R.T. Fielding, J. Herbsleb, A case study of open source software development: the Apache server, in: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, ACM Press, 2000, pp. 263–272.
- [54] J.W. Paulson, G. Succi, A. Eberlein, An empirical study of open-source and closed-source software products, *IEEE Trans. Softw. Eng.* 30 (2004) 246–256. ISSN 0098-5589.
- [55] C. Rahmani, A. Azadmanesh, L. Najjar, A comparative analysis of open source software reliability, *J. Softw.* 5 (2010) 1384–1394.
- [56] N.E. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, *IEEE Trans. Softw. Eng.* 26 (2000) 797–814.
- [57] R.R. Lutz, I.C. Mikulski, Empirical analysis of safety-critical anomalies during operations, *IEEE Trans. Softw. Eng.* 30 (2004) 172–180.
- [58] M. Hamill, K. Goševa-Popstojanova, Common trends in software fault and failure data, *IEEE Trans. Softw. Eng.* 35 (2009) 484–496. ISSN 0098-5589.
- [59] J.A. Borretzen, R. Conradi, Results and experiences from an empirical study of fault reports in industrial projects, in: *PROFES 2006, LNCS*, Springer, 2006, pp. 389–394.