Empirical Software Engineering manuscript No.
(will be inserted by the editor)

The Untold Impact of Learning Approaches on Software Fault-Proneness Predictions: An Analysis of Temporal Aspects

Mohammad Jamil Ahmad · Katerina Goseva-Popstojanova · Robyn R. Lutz

Received: date / Accepted: date

Abstract This paper aims to improve software fault-proneness prediction by investigating the unexplored effects on classification performance of the temporal decisions made by practitioners and researchers regarding (i) the interval for which they will collect longitudinal features (software metrics data), and (ii) the interval for which they will predict software bugs (the target variable). We call these specifics of the data used for training and of the target variable being predicted the *learning approach*, and explore the impact of the two most common learning approaches on the performance of software fault-proneness prediction, both within a single release of a software product and across releases. The paper presents empirical results from a study based on data extracted from 64 releases of twelve open-source projects. Results show that the learning approach has a substantial, and typically unacknowledged, impact on classification performance. Specifically, we show that one learning approach leads to significantly better performance than the other, both within-release and across-releases. Furthermore, this paper uncovers that, for within-release predictions, the difference in classification performance is due to different levels of class imbalance in the two learning approaches. Our findings show that improved specification of the learning approach is essential to understanding and explaining the performance of fault-proneness prediction models, as well as to avoiding misleading comparisons among them. The paper concludes with some practical recommendations and research directions based on our findings toward improved software fault-proneness prediction.

Keywords Software fault-proneness prediction · Learning approach · Within-release prediction · Across-release prediction · Machine learning · Design of experiments

1 Introduction

The prediction of *fault-prone* software units helps software developers prioritize their efforts, reduces development costs, and leads to better quality software products (Nagappan et al., 2006, 2008). It is thus not surprising that the prediction of software fault-proneness is an active research area in software engineering.

A software bug (i.e., fault) is an accidental condition which, if encountered, may cause the software system or component to fail to perform as required (Hamill and Goseva-Popstojanova, 2009). Software bugs can lead to failures, some with serious consequences, such as private information leakage, financial loss, or loss of human life.

This research was supported in part by NASA's Software Assurance Research Program (SARP) under a grant funded in FY21 and by National Science Foundation grants 1513717, 1900716 and 2211589.

Mohammad Jamil Ahmad
Lane Department of Computer Science and Electrical Engineering. West Virginia University. Morgantown WV, 26505
Tel.: +1304-293-7939
ORCID iD: 0000-0002-4038-2379
E-mail: mohammad.ahmad@mail.wvu.edu
Present affiliation: Department of Management Information Systems, John Chambers College of Business and Economics, West Virginia University, Morgantown WV, 26505

Katerina Goseva-Popstojanova
Lane Department of Computer Science and Electrical Engineering. West Virginia University. Morgantown WV, 26506
Tel.: +1304-293-9691
ORCID iD: 0000-0003-4683-672X
E-mail: katerina.goseva@mail.wvu.edu
Corresponding author

Robyn R. Lutz
Department of Computer Science, Iowa State University. Ames Iowa, 50011
Tel.: +1515-294-3654
ORCID iD: 0000-0001-5390-7982
E-mail: rlutz@iastate.edu

A software unit (e.g., file, package, or component) is *fault-prone* if it has one or more software bugs. Note that in addition to 'fault-prone', the terms 'bug-prone', 'error-prone', and 'defective' have been used.

Over the years, researchers have built many *software fault-proneness prediction models* (Arisholm et al., 2010; Catal, 2011; Hall et al., 2012; Song et al., 2019; Hosseini et al., 2019). ('Software fault-proneness prediction' is a synonym for another widely used term 'software defect prediction'.) These models either classify software units as fault-prone and not fault-prone (e.g., (Koru and Liu, 2005; Nagappan et al., 2006; Menzies et al., 2007; Zimmermann et al., 2007, 2009; Krishnan et al., 2013; Alshehri et al., 2018; Gong et al., 2021)) or predict the number of faults (i.e., fault-count) in each software unit (e.g., (Zimmermann et al., 2007; Ostrand et al., 2005; Devine et al., 2012, 2016)). This paper focuses on classification-based predictions both because there are significantly more research works focused on classification-based prediction than on fault-count prediction, and because many of these works utilized the same publicly available datasets.

Related works have explored various factors that affect the performance of classification-based software fault-proneness prediction. These factors include the machine learning algorithms used (i.e., learner) (Lessmann et al., 2008; Krishnan et al., 2013), the choice of software metrics (i.e., features used for prediction) (Bluemke and Stepień, 2016; Wang et al., 2016; Alshehri et al., 2018; Gong et al., 2021) and the effect of data balancing techniques (Khoshgoftaar et al., 2010; Wang and Yao, 2013; Song et al., 2019; Goseva-Popstojanova et al., 2019). The features used by software fault-proneness prediction models typically include static code metrics, such as lines of code, complexity, and coupling between methods (Menzies et al., 2007; Lessmann et al., 2008; Jureczko and Madeyski, 2010; Amasaki, 2020; Kabir et al., 2021). These static code features are collected at a specific time, most often at the release date. Features used for prediction of fault-prone software units (e.g., files, packages) may also include longitudinal features that are collected over a period of time, such as change metrics (Moser et al., 2008; Nagappan et al., 2010; Krishnan et al., 2013; Goseva-Popstojanova et al., 2019) and socio-technical metrics (Bird et al., 2009).

Based on the Organization for Economic Co-operation and Development (OECD) framework for classification of AI systems (OECD, 2022), which is utilized by the NIST's AI risk management framework (NIST, 2023), there are four key dimensions of an AI system: 'Application Context', 'Data and Input', 'AI Model', and 'Task and Output'. These dimensions build a conceptual view of a generic AI system shown in the bottom part of Fig. 1. Many related works on software fault-proneness prediction utilize readily available datasets and focus mainly on the 'AI Model' dimension (which consists of 'Build and use model' and 'Verify and validate' stages), without paying close attention on the 'Data and Input' dimension. In this paper we explore another factor that is essential to understanding the classification performance of the software fault-proneness predictions, but is often omitted or not explicitly described in related works. We use the term **learning approach** to refer to this factor which encompasses the temporal aspects of (i) the interval during which the longitudinal features (such as for example change metrics) are collected and (ii) the interval for which the target variable (i.e., software bugs) is labeled. As shown in the upper part of Fig. 1, the learning approach belongs to the 'Data and Input' dimension of the OECD framework (OECD, 2022). (The 'Data and Input' dimension consists of the 'Collect and process data' stage.)

Specifically, in this paper we focus on the two most-common learning approaches, since most software fault-proneness prediction papers use one of them. As shown in Fig. 2, both learning approaches use static code metrics collected in a snapshot of time, typically at the release date. However, the learning approaches (i) use different intervals for extracting longitudinal features, such as change metrics and socio-technical metrics, and (ii) use different target variables.

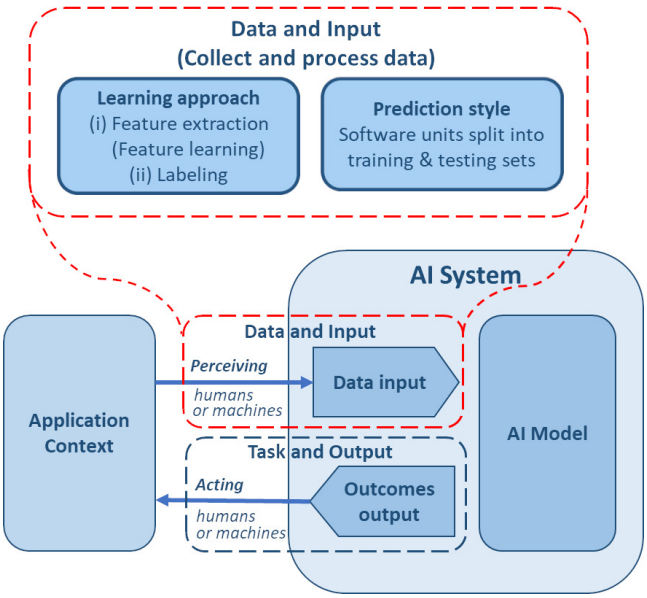

Fig. 1 Conceptual view of an AI system (per OECD AI Principles (OECD, 2022)) with an insert in the upper part detailing the factors explored in this paper

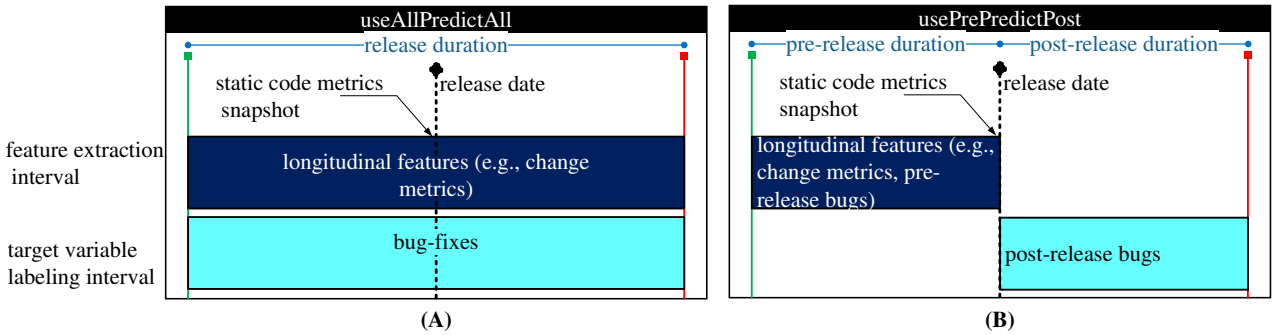

Fig. 2 Temporal aspects of the feature extraction intervals and target variable labeling intervals for the two learning approaches

- In the first case, shown in Fig. 2(A), software fault-proneness prediction models are trained using static code features collected in a snapshot of time (typically at the release date) and longitudinal features (e.g., change metrics) collected throughout the duration of the entire release to predict fault-prone software units within the same interval. (The target variable 'bug fixes' does not distinguish between pre-release and post-release bugs.) We use the shorthand label **useAllPredictAll** for this learning approach.
- In the second case, depicted in Fig. 2(B), while static code metrics are collected as in the first case, the longitudinal features (including pre-release bugs) are collected during the pre-release interval in order to predict fault-prone software units with post-release bugs. We use the shorthand label **usePrePredictPost** for this second learning approach.

In general, an AI system can be based on expert input and/or on data, both of which can be generated by humans or automated tools such as machine-learning algorithms (OECD, 2022). Note that the learning approach factor is generic and, as shown in Fig. 1, applicable both when features are designed and extracted by humans (i.e., feature extraction in traditional machine learning), and when deep learning is used to automatically learn features from raw data (i.e., feature learning). Furthermore, the same machine learning algorithm can be used for classification of software units into fault-prone and not fault prone (i.e., to build the AI model) with either learning approach (useAllPredictAll or usePrePredictPost). These classification algorithms range from traditional algorithms that have been used for decades (e.g., logistic regression, decision trees, random forest) to deep neural networks (DNN).

We show in this paper that **the learning approach has a large impact on classification performance of the software fault-proneness prediction.** Our investigation shows that several widely used datasets for software fault-proneness prediction are based on different learning approaches. Nevertheless, the impact of the learning approach is often not realized or accounted for in the literature. Additionally, we show that while the same machine learning algorithm (i.e., classification algorithm) can be used with either learning approach, it will produce significantly different classification performance. Notably, prior works apart from Krishnan et al. (2013), neither have discussed the distinctions between learning approaches when building software fault-proneness prediction models nor have explored the untold impact of the learning approach on the prediction performance.

In addition to investigating the learning approach, our analysis includes the **prediction style** that captures how software units (e.g., files) are split into training and testing sets (i.e., what files are used for training and testing of the AI models). The prediction style factor also belongs to the 'Data and Input' dimension of the OECD framework (OECD, 2022), as shown in the upper part of Fig. 1. Basically, depending on the specific learning approach used, each software unit (i.e., file) is labeled as fault-prone or not fault-prone (either for the entire release duration or post-release) and characterized with longitudinal features extracted during a specific interval (either the whole release duration or pre-release duration). Software fault-proneness prediction models that are trained and tested on data collected from the same software project are known as *within-project* prediction models. If a software project has multiple releases, the fault-proneness prediction can be done either *within-release* or *across-releases*. Prediction within-release uses training and testing data from the same release, while across-releases prediction uses training data from one or more releases and testing data from a different release. Models that use training data from one project and testing data from a separate project are known as *across-projects* prediction models. Some researchers also use the term across-company prediction for cases where prediction models are trained on data from a project developed by one company and tested on data collected from a project developed by a different company.

The goal of this paper is to study the impact of the learning approach (i.e., the temporal aspects of the extracted longitudinal features and the target variable) on the software fault-proneness prediction for within-project prediction styles (i.e., within-release and across-releases). Thus, we address the following research questions:

RQ1: Does the learning approach affect the classification performance of the software fault-proneness predictions?
RQ2: For a given learning approach, what is the difference in classification performance between using within-release and across-releases prediction styles?

The empirical results presented in this paper are based on datasets we extracted from 64 releases of 12 Apache open-source projects.

The main contributions of this study are as follows:

- We categorize the related works on software fault-proneness prediction based on the learning approach used in each work. To the best of our knowledge such categorization has not been done previously. Our identification of the learning approach(es) used in each paper enables a fuller understanding of the trends and results reported by many studies.
- We explore the effects of a hidden factor, learning approach, on the performance of software fault-proneness prediction models. For that purpose, we: (1) build software fault-proneness prediction models using the two learning approaches, *useAllPredictAll* and *usePrePredictPost*; (2) predict fault-proneness both within-release and across-releases; and (3) use a design of experiment (DoE) approach and statistical analysis to draw sound conclusions.
- We present results showing that the learning approach significantly affects the classification performance. Specifically, using the *useAllPredictAll* learning approach led to significantly higher Recall, Precision, F-Score, G-Score and similar 1 - False Positive Rate (1 - FPR) compared to using the *usePrePredictPost* learning approach. These results indicate that when comparing the performance of software fault-proneness predictions, care should be taken to compare 'apples-to-apples'. Otherwise, the performance comparisons are unfair. Furthermore, combining and reusing existing datasets that were created using different learning approaches in a new study leads to flawed statistical analysis, meta analysis, and/or machine learning experiments. Therefore, our results may call into question conclusions advanced by some prior publications that failed to recognize the use of different learning approaches and to consider their effect on the performance of the fault-proneness prediction.
- In investigating why there is a difference in classification performance when using different learning approaches, we uncover that for within-release predictions the difference is due to another hidden factor – the class imbalance. The treatment of imbalance using SMOTE improves the within-release prediction performance for each learning approach individually, as expected based on prior works. Extending the prior works, we apply class imbalance treatment on both learning approaches and compare the corresponding improvements. Furthermore, based on descriptive and inferential statistical results, we find that the imbalance treatment using SMOTE eliminates the difference in prediction performance between the learning approaches, for within-release prediction. Note however that the two learning approaches predict different target variables, i.e., fault-proneness for the whole release duration versus fault-proneness for the post-release duration.
- We discuss the implications of our findings and provide recommendations for designing, reporting, and comparing software fault-proneness prediction studies, as well as identify some open issues for future research in this area.

In brief, this paper provides evidence that the learning approach must be explicitly considered as an additional factor in the analysis of the software fault-proneness prediction performance. We show that this is due to the fact that the learning approach is intrinsic both to the way datasets are created and to the way machine learning models are trained and tested.

The remainder of the paper is organized as follows. Section 2 describes the related works, and Section 3 categorizes the related works based on the learning approach each used. Section 4 describes the datasets, data extraction process, and feature vectors used in this study. Section 5 details our machine learning approach and design of experiments approach. Analysis of our results are provided in Section 6, with threats to validity presented in Section 7. Section 8 discusses the implications of our findings and offers associated recommendations toward improved fault-proneness prediction performance. Section 9 provides concluding remarks.

2 Related work

Software fault-proneness prediction is an active area of research, as evidenced by its systematic literature review papers (Arisholm et al., 2010; Catal, 2011; Hall et al., 2012; Song et al., 2019; Hosseini et al., 2019) and the references therein. We here discuss the related studies that are the most relevant to our work.

Many different machine learning algorithms have been used in building software fault-proneness prediction models. These include J48 (Moser et al., 2008; Kamei et al., 2010; Krishnan et al., 2013), Random Forest (RF) (Guo et al., 2004; Mahmood et al., 2018; Fiore et al., 2021; Gong et al., 2021), and combinations of several machine learning algorithms, e.g., OneR, J48, and Naïve Bayes (NB) in (Menzies et al., 2007), Random Forrest (RF), NB, RPart, and SVM in (Bowes et al., 2018), J48, RF, NB, Logistic Regression (LR), PART, and G-Lasso in (Goseva-Popstojanova et al., 2019), RF, LR, NB, HyperPipes, KNN, and J48 in (Falessi et al., 2020), and Decision Tree (DT), k-Nearest Neighbor (kNN), LR, NB, and RF in (Kabir et al., 2021). With recent advances in Deep Neural Networks (DNN), some software fault-proneness prediction studies used deep learning (Wang et al., 2016; Li et al., 2017; Pang et al., 2017; Zhou et al., 2019; Zhao et al., 2021).

Software fault-proneness prediction models utilize feature vectors consisting of software metrics extracted from software source or binary code, its development history, and the associated bug tracking systems. In general, the extracted software metrics can be static code metrics, change metrics, or social metrics. Static code metrics are collected from the software source code or binary code units (Koru and Liu, 2005; Menzies et al., 2007;

Lessmann et al., 2008; Menzies et al., 2010; He et al., 2013; Ghotra et al., 2015; Bowes et al., 2018; Kabir et al., 2021). Change metrics, sometimes called process metrics, are collected from the projects' development history (i.e., commit logs) and bug tracking systems (Nagappan et al., 2010; Giger et al., 2011; Krishnan et al., 2011, 2013; Goseva-Popstojanova et al., 2019). Social metrics are extracted from the communications among developers and/or users of a software project (Bird et al., 2009). Some studies used only static code metrics (Menzies et al., 2007; Mende and Koschke, 2009; Song et al., 2011; Xu et al., 2018, 2019; Amasaki, 2020; Kabir et al., 2021), only change metrics (Giger et al., 2011; Krishnan et al., 2011, 2013; Goseva-Popstojanova et al., 2019), organizational metrics (Nagappan et al., 2008) or metrics derived from the contribution networks (Pinzger et al., 2008). There are studies that used combinations of different types of metrics (Nagappan et al., 2010; Arisholm et al., 2010; Giger et al., 2012; Alshehri et al., 2018), including studies that combined metrics extracted from contribution networks, dependency networks, and/or socio-technical networks (Bird et al., 2009; Gong et al., 2021).

Works that do prediction at the file level use features collected from units such as files (Zimmermann et al., 2007; Moser et al., 2008; Kamei et al., 2010; Krishnan et al., 2013; Goseva-Popstojanova et al., 2019), classes (Koru and Liu, 2005; Malhotra and Raje, 2015; Song et al., 2019), or methods (Giger et al., 2012; Bowes et al., 2018), while prediction at the component level is based on features aggregated at the package, module, or component level from features extracted at file, class, or method level (Zimmermann et al., 2007).

Next, we summarize related works by the prediction style they used. Note that 'prediction style' has elsewhere been referred to as 'context' (Gong et al., 2021). For within-project predictions, some works were focused only on within-release prediction style (e.g., (Nagappan et al., 2008; Krishnan et al., 2011, 2013; Alshehri et al., 2018)), while others focused only on across-releases prediction (Xu et al., 2018, 2019; Fiore et al., 2021; Kabir et al., 2021). Terms used in the literature for 'across-releases' prediction include 'cross-version' (Xu et al., 2018, 2019; Fiore et al., 2021; Gong et al., 2021) and 'inter-release' prediction (Kabir et al., 2021). Papers that included across-projects prediction models include (Nagappan et al., 2006; Zimmermann et al., 2009; He et al., 2013; Wang et al., 2016; Gong et al., 2021). Other papers specifically explored the across-company prediction style (Turhan et al., 2009). In this paper we focus on the two within-project predictions styles – within-release and across-releases – and in Section 3 categorize related works by the learning approach and prediction style used (see Table 1).

Many software fault-proneness prediction studies have derived conclusions solely from analyzing the empirical results, without applying statistical tests (Bird et al., 2009; Giger et al., 2012; Alshehri et al., 2018; Zhou et al., 2019; Fiore et al., 2021). Other studies used statistical inference to support their conclusions, typically studying one factor at a time (e.g., (Mende and Koschke, 2009; Turhan et al., 2009; Song et al., 2011; Krishnan et al., 2013; Okutan and Yıldız, 2014; Bowes et al., 2018; Gong et al., 2021)).

The design of experiment approach (DoE), which we use in our investigation here, allows one or more input factors to be systematically explored with the goal of determining their effect on the output (response) variable. DoE has been used to date only in several software fault-proneness prediction studies (Khoshgoftaar and Seliya, 2004; Lessmann et al., 2008; Gao et al., 2011; Shepperd et al., 2014; Tantithamthavorn et al., 2016; Shepperd et al., 2018)).

Khoshgoftaar and Seliya (2004) treated the machine learning algorithm as a factor and the software release as a block. The results, based on four successive releases from a large legacy telecommunication system, showed that the models' predictive performances were significantly different across the releases, implying that the predictions were influenced by the characteristics of the data. Lessmann et al. (2008) explored the effect of the machine learning algorithm on the software fault-proneness prediction performance using ten datasets from the NASA Metrics Data Program (MDP) repository. The results showed that no statistically significant difference in performance existed among the top 17 (out of 22) machine learning algorithms, i.e., the choice of machine learning algorithm appeared to be not as important as was previously assumed. Gao et al. (2011) statistically examined the effect of three factors (i.e., feature ranking method, feature subset selection method, and machine learning algorithm) on the software fault-proneness prediction performance for a large legacy telecommunication software. Shepperd et al. (2014) presented meta-analysis based on 42 primary software fault-proneness prediction studies. Note that substantial overlaps existed in dataset usage among the primary studies, with the NASA datasets family dominating with 59%, followed by the Eclipse family with 21%. This work used DoE consisting of four factors: the machine learning algorithm, dataset used, input metrics (i.e., features), and the particular researcher group (i.e., the authors). The results showed that the researcher group contributed the most to the variance of the prediction performance, followed by the dataset, and input metrics. The choice of the machine learning algorithm had the least impact on the prediction performance. Two follow-up studies (Tantithamthavorn et al., 2016; Shepperd et al., 2018) repeated the meta-analysis using subsets of the primary studies used in (Shepperd et al., 2014). The work in (Tantithamthavorn et al., 2016), which was based on the Eclipse datasets family, found that the research group had smaller impact than the input metrics. When only the NASA datasets family was used for re-analysis in (Shepperd et al., 2018), the results were more in line with the initial analysis based on all datasets families (Shepperd et al., 2014).

In summary, prior DoE studies had different goals than our study and looked at different factors. None of these studies recognized the learning approach(es) used and some mixed different learning approaches without explicit acknowledgement (Shepperd et al., 2014).

In this paper we will show how the choice of the learning approach significantly affects the prediction performance and therefore must be explicitly specified and accounted for in the analysis. Additionally, we will show how awareness of the learning approach factor provides improved understanding of the prediction results. Thus,

researchers and practitioners alike must recognize that different datasets are not only extracted from artifacts of different software systems, but also may be produced to fit different learning approaches. We believe that considering the learning approach explicitly as a factor will help disentangle the effects of the other factors, since the learning approach is intrinsic both to the datasets used and the learning process conducted.

3 Categorizing related works by learning approach used

To better understand how the selection of learning approach influences fault-proneness prediction results, we need to first identify the learning approach that was used in existing studies. However, when we set out to categorize the learning approach used in each of those studies, we found that this was a difficult task. In fact, as we will see in this section, apart from Krishnan et al. (2013), studies on software fault-proneness prediction have not explicitly specified the learning approach nor described the impact of the learning approach used on the prediction performance they reported. It appears that the characteristics of the dataset(s) used for building the software fault-proneness prediction models typically predetermined which learning approach was used. By carefully exploring the details of the datasets being used, we were able to categorize the related works according to the learning approach.

The results of our effort are shown in Table 1, which groups the related studies by: the dataset(s) used, the learning approach (i.e., *useAllPredictAll* or *usePrePredictPost*), and the prediction style (i.e., within-release or across-releases). Those studies which used prediction within-project with no releases (e.g., studies based on the NASA MDP datasets) are grouped with the studies which used prediction within-release because they were designed similarly. Studies which used multiple datasets attributed to one specific learning approach (e.g., (Kim et al., 2011; Bowes et al., 2018)) or to multiple learning approaches (e.g., (Nam et al., 2013; Shepperd et al., 2014; Tantithamthavorn et al., 2017; Song et al., 2019; Zhou et al., 2019)) are shown in more than one cell.

Next, we discuss the related works that employed only *useAllPredictAll* learning approach (subsection 3.1), only *usePrePredictPost* learning approach (subsection 3.2), as well as those related works that used datasets or results of studies that belong to both learning approaches (subsection 3.3).

3.1 Studies that used *useAllPredictAll* learning approach

Software fault-proneness prediction models employing *useAllPredictAll* learning approach are trained using longitudinal features collected from the entire duration of each release or project with the goal of predicting fault-prone files for the entire duration of the same or future release. These models do not distinguish between pre-release and post-release bugs. That is, all bugs are grouped in the *bug-fixes* metric, and a file is labeled as fault-prone if it had one or more bugs anytime during the specific release or project.

Based on our analysis of the related work, studies that used the NASA MDP datasets (e.g., those available on PROMISE (Sayyad and Menzies, 2005)) have followed the *useAllPredictAll* learning approach. Note that NASA datasets have no releases; each dataset represents an independent project for which static code metrics were extracted at a snapshot in time. Examples of studies which used the NASA MDP datasets within-project include but are not limited to (Koru and Liu, 2005; Menzies et al., 2007; Gondra, 2008; Lessmann et al., 2008; Jiang et al., 2008a,b,c; Elish and Elish, 2008; Turhan et al., 2009; Mende and Koschke, 2009; Menzies et al., 2010; Wang and Yao, 2013; Ghotra et al., 2015; Bowes et al., 2018; Zhou et al., 2019; Goyal, 2022).

Other datasets that utilized the *useAllPredictAll* learning approach were created from fifteen open-source projects (including twelve Apache projects) with a total of 48 releases and six industrial projects with a total of 27 releases (Jureczko and Spinellis, 2010; Jureczko and Madeyski, 2010) and were donated to PROMISE (Sayyad and Menzies, 2005).

These datasets have a set of static code metrics as feature vectors and were used by multiple studies for prediction both within-release (He et al., 2013; Okutan and Yıldız, 2014; Bowes et al., 2018; Song et al., 2019) and across-releases (Madeyski and Jureczko, 2015; He et al., 2015; Li et al., 2017; Xu et al., 2018, 2019; Amasaki, 2020; Kabir et al., 2021).

Different datasets obtained from Eclipse were extensively used for software fault-proneness prediction. Datasets suitable for *useAllPredictAll* learning approach were employed to predict software fault-proneness of Eclipse plugins (Giger et al., 2011, 2012), as well as of Eclipse platform (Kim et al., 2011; Krishnan et al., 2013; Shepperd et al., 2014). These works used change metrics as feature vectors.

Other open-source projects were used to extract datasets adequate for the *useAllPredictAll* learning approach, including Android (Malhotra and Raje, 2015). Some models were used for only within-release predictions (Ghotra et al., 2015) or only across-releases prediction (Fiore et al., 2021), while others were used for both within-release and across-releases predictions (Malhotra and Raje, 2015).

The *useAllPredictAll* learning approach was also used for software fault-proneness prediction of commercial software, such as within-release predictions for Microsoft products (Layman et al., 2008), and within-release (Tosun et al., 2010; Bowes et al., 2018) and across-releases (Arisholm et al., 2007) for telecommunication software.

Many works used datasets from multiple sources. Examples include NASA MDP, open-source, and telecommunication datasets (Bowes et al., 2018); Eclipse plugins with other open-source projects (Giger et al., 2012); NASA MDP and SOFTLAB telecommunication datasets (Turhan et al., 2009); NASA MDP and datasets from a Turkish

Table 1 Related work studies categorized by the learning approach and prediction style

Dataset	useAllPredictAll		usePrePredictPost	
	within-release/ project	across-releases	within-release/ project	across-releases
NASA MDP	Koru and Liu (2005); Menzies et al. (2007); Gondra (2008); Lessmann et al. (2008); Jiang et al. (2008a,b,c); Elish and Elish (2008); Turhan et al. (2009); Mende and Koschke (2009); Menzies et al. (2010); Wang and Yao (2013); Shepperd et al. (2014); Ghotra et al. (2015); Tantithamthavorn et al. (2017); Bowes et al. (2018); Shepperd et al. (2018); Zhou et al. (2019); Goyal (2022)			
Apache, other open source and industry projects Jureczko and Madeyski (2010); Jureczko and Spinellis (2010)	He et al. (2013); Okutan and Yıldız (2014); Tantithamthavorn et al. (2017); Bowes et al. (2018); Song et al. (2019)	Madeyski and Jureczko (2015); He et al. (2015); Li et al. (2017); Xu et al. (2018, 2019); Amasaki (2020); Kabir et al. (2021)		
Eclipse plugins	Giger et al. (2011, 2012)			
Eclipse platform	Kim et al. (2011); Krishnan et al. (2013); Shepperd et al. (2014); Tantithamthavorn et al. (2017)		Zimmermann et al. (2007); Moser et al. (2008); Bird et al. (2009); Kamei et al. (2010); Krishnan et al. (2013); Shepperd et al. (2014); Tantithamthavorn et al. (2016); Alshehri et al. (2018); Shepperd et al. (2018); Goseva-Popstojanova et al. (2019)	Zimmermann et al. (2007); Bird et al. (2009); Kamei et al. (2010); Shepperd et al. (2014); Tantithamthavorn et al. (2016)
Open-source	Watanabe et al. (2008); Shepperd et al. (2014); Malhotra and Raje (2015); Ghotra et al. (2015)	Malhotra and Raje (2015); Fiore et al. (2021)	Shepperd et al. (2014); Goseva-Popstojanova et al. (2019); Gong et al. (2021)	Wang et al. (2016); Gong et al. (2021)
Microsoft products	Layman et al. (2008)		Nagappan et al. (2008); Bird et al. (2009); Nagappan et al. (2010); Shepperd et al. (2014); Tantithamthavorn et al. (2017)	Nagappan et al. (2006)
Telecommunication software	Tosun et al. (2010); Shepperd et al. (2014); Bowes et al. (2018)	Arisholm et al. (2007); Shepperd et al. (2014)	Arisholm et al. (2010); Shepperd et al. (2014)	Khoshgoftaar and Seliya (2004); Arisholm et al. (2010); Gao et al. (2011); Shepperd et al. (2014)
AEEEMD'Ambros et al. (2010)			Nam et al. (2013); Tantithamthavorn et al. (2017); Song et al. (2019); Zhou et al. (2019)	
RelinkWu et al. (2011)	Wu et al. (2011); Nam et al. (2013); Zhou et al. (2019)			
Other	Turhan et al. (2009); Menzies et al. (2010); Kim et al. (2011); Giger et al. (2012); Wang and Yao (2013); Shepperd et al. (2014)	Madeyski and Jureczko (2015)	Shepperd et al. (2014)	

manufacturer (Menzies et al., 2010); and datasets obtained from Eclipse platform and other open-source programs (Kim et al., 2011).

3.2 Studies that used *usePrePredictPost* learning approach

Software fault-proneness prediction models which use the *usePrePredictPost* learning approach are trained using longitudinal features (including pre-release bugs) collected during the pre-release duration to predict files that have post-release bugs in the same or future release. Pre-release and post-release bugs are distinguished, and a file is labeled as fault-prone if it had one or more post-release bugs. This learning approach was first used for software fault-proneness prediction of three Eclipse platform releases, within and across-releases, at both file and package level (Zimmermann et al., 2007). The features consisted of static code metrics collected from each release. Since Eclipse releases were twelve months apart, the pre-release bugs were collected six months before the release date (i.e., during the development and testing phase), and the post-release bugs were collected six months after the release date (i.e., after deploying the release to users).

Eclipse datasets created for the *usePrePredictPost* learning approach were used extensively for within-release (Zimmermann et al., 2007; Moser et al., 2008; Bird et al., 2009; Kamei et al., 2010; Krishnan et al., 2013; Tantithamthavorn et al., 2016; Alshehri et al., 2018; Goseva-Popstojanova et al., 2019) and across-releases prediction (Zimmermann et al., 2007; Bird et al., 2009; Kamei et al., 2010). Some of these studies used both static code metrics and change metrics (Moser et al., 2008; Kamei et al., 2010; Alshehri et al., 2018), only static code metrics (Zimmermann et al., 2007), or only change metrics (Krishnan et al., 2013; Goseva-Popstojanova et al., 2019).

Datasets from other open-source projects were also used, sometimes in combination with one or more other datasets (e.g., Eclipse platform, Microsoft products), for within-release predictions (Goseva-Popstojanova et al., 2019), across-releases predictions (Wang et al., 2016), or both (Gong et al., 2021). The *usePrePredictPost* learning approach has also been used for software fault-proneness prediction of commercial software, including Microsoft products for within-release (Nagappan et al., 2008, 2010; Bird et al., 2009) and across-release predictions (Nagappan et al., 2006), as well as for telecommunication software for across-releases predictions (Khoshgoftaar and Seliya, 2004; Gao et al., 2011) and for both within-release and across-releases predictions (Arisholm et al., 2010).

3.3 Studies that used datasets from both learning approaches

Some related works used datasets corresponding to different learning approaches for synthesizing the findings across software fault-proneness prediction studies (Hall et al., 2012), to conduct meta-analysis (Shepperd et al., 2014, 2018), or to build and compare prediction models (Nam et al., 2013; Tantithamthavorn et al., 2017; Song et al., 2019; Zhou et al., 2019).

The systematic literature review on software fault-proneness prediction (Hall et al., 2012) reported the synthesis of results from 19 classification-based studies, some of which used *useAllPredictAll* while others used the *usePrePredictPost* learning approach. Datasets used in these studies were analyzed in (Hall et al., 2012) as a part of context factors and it was concluded that it may be more difficult to build models for some systems than for others.

The meta-analysis presented in (Shepperd et al., 2014) was based on quantitative results extracted from 42 primary software fault-proneness prediction studies which used datasets that belong to different learning approaches. Most prominently, 59% of the primary studies used the NASA datasets and 21% used the Eclipse datasets, which belong to the *useAllPredictAll* and *usePrePredictPost* learning approaches, respectively. (For other datasets and the corresponding learning approaches used by the primary studies included in (Shepperd et al., 2014) see Table 1.) The results based on using four-way ANOVA, attributed most of the variance to the researcher group and the dataset factors (with 31.0% and 11.2%, respectively). As discussed in Section 2, two follow-up studies (Tantithamthavorn et al., 2016; Shepperd et al., 2018) repeated the meta-analysis using subsets of the primary studies used in (Shepperd et al., 2014). The findings presented in (Tantithamthavorn et al., 2016) were based on using only the Eclipse datasets family, while (Shepperd et al., 2018) carried on separate analysis for the Eclipse datasets family and NASA datasets family and compared the results with those presented in (Shepperd et al., 2014; Tantithamthavorn et al., 2016).

With a goal to explore transfer learning, prediction models were built in (Nam et al., 2013) using several datasets from ReLink (Wu et al., 2011) and AEEEM (D'Ambros et al., 2010). The prediction done using ReLink followed the *useAllPredictAll* learning approach, while the AEEEM datasets were extracted following the *usePrePredictPost* learning approach. Another work (Song et al., 2019) explored the role of imbalanced learning on software fault-proneness prediction using the datasets created by Jureczko and Madeyski (2010); Jureczko and Spinellis (2010) and available in PROMISE and the AEEEM datasets (D'Ambros et al., 2010), which correspond to the *useAllPredictAll* and *usePrePredictPost* learning approaches, respectively. Several model validation techniques were compared by Tantithamthavorn et al. (2017) using the NASA MDP, Apache, Eclipse, and proprietary datasets that belong to the *useAllPredictAll* learning approach, as well as datasets from Microsoft products that followed the usePrePredictPost learning approach.

The work presented in (Zhou et al., 2019) proposed a deep forest model and compared its performance with RF, NB, LR, and SVM, using the NASA MDP, PROMISE, ReLink, and AEEEM datasets. The first three datasets belong to *useAllPredictAll* learning approach, while the forth dataset corresponds to *usePrePredictPost* learning approach.

To the best of our knowledge, none of the related works on software fault-proneness prediction has addressed the effect of the learning approach on the prediction performance, regardless of whether they used only one learning approach (see subsections 3.1 and 3.2) or both learning approaches (i.e., works discussed in this subsection). An exception is the previous work by Krishnan et al. (2013), which investigated whether the classification performance improved as the Eclipse product line evolved through seven releases. In that work, change metrics were used for within-release prediction only, and the software fault-proneness prediction performance was compared using three different learning approaches *useAllPredictAll*, *useAllPredictPost*, and *usePrePredictPost*.

One contribution of this paper is the categorization of related works by learning approach used, shown in Table 1 and discussed in this section. Knowing the learning approach used in each study often explains differences among their prediction performances and enables better understanding of how fault-proneness prediction works.

Motivated by our categorization of the related works presented here and by the initial results presented in (Krishnan et al., 2013), in the rest of this paper we explore systematically and rigorously the impact of the learning approach used on the performance of software fault-proneness prediction. We focus our analysis on *useAllPredictAll* and *usePrePredictPost* because almost all related works fit into one or the other learning approach, as shown in Table 1. Specifically, employing both static code metrics and change metrics extracted from 64 releases of 12 open-source projects, we evaluate the performance of prediction models both within-release and across-releases. To quantify and better understand the impact of the learning approach on fault-proneness prediction, we use a design of experiments approach and inferential statistical analysis. We show in this paper that the effect of the learning approach is an essential factor in understanding the prediction results. Toward better fault-proneness prediction models, we aim to encourage awareness and attention to the impact of the learning approaches going forward.

4 Data collection and building feature vectors

In this section, we first present the data extraction process used for the two learning approaches, then describe the open source projects used to extract the datasets, followed by a description of the feature vectors used for prediction.

4.1 Data extraction

We use Figs. 3(A) and 3(B), which are detailed, annotated versions of the overview Figs. 2(A) and 2(B), to illustrate the feature extraction for the *useAllPredictAll* and *usePrePredictPost* learning approaches, respectively. As shown in these figures, the release duration of any given release n is the period between the two dates $d1$ and $d2$, shown by the green and the red lines, respectively. (Following the approach introduced by Zimmermann et al. (2007), $d1$ is the middle date between the release dates of releases $n-1$ and n, while $d2$ is the middle date between the release dates of releases n and $n+1$.) In Figs. 3(A) and 3(B), the release date is annotated with the dashed line at n. It follows that the interval between $d1$ and n is pre-release, and the interval between n and $d2$ is post-release interval of release n. Table 2 provides further details on the feature collection and target variable for each learning approach.

Table 2 Details on features collection and target variable for each learning approach

		useAllPredictAll	**usePrePredictPost**
Features	Static code metrics (snapshot)	collected at a particular time (typically at release date)	collected at a particular time (typically at release date)
	Longitudinal metrics such as change metrics, socio-technical metrics	collected for the entire release duration	collected for the pre-release duration
	Fault-proneness as a feature	none	pre-release bugs
Target variable	Binary fault-proneness metric for a software file	label = 1 if a file had one or more bugs during the entire release duration; otherwise, label = 0. (No distinction between pre-release and post-release bugs)	label = 1 if a file had one or more post-release bugs; otherwise label = 0.

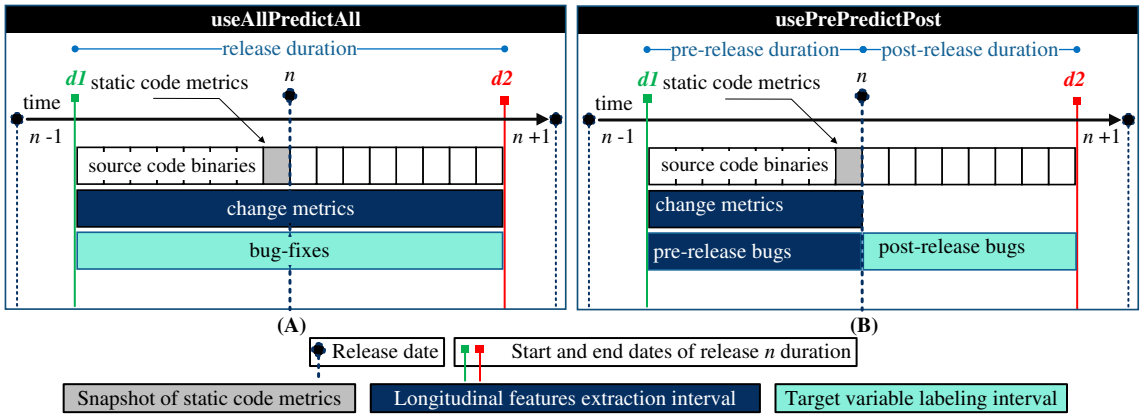

Fig. 3 Metrics extraction process for the two learning approaches

4.1.1 Data extraction for useAllPredictAll

As illustrated in Fig. 3(A), for each release we extracted the static code metrics from the latest version of the binaries available on the release date. Following the method used in (Krishnan et al., 2013), for each release we extracted the change metrics from the entire duration of that release (i.e., between the $d1$ and $d2$ dates).

When using the *useAllPredictAll* learning approach, no distinction was made between pre-release and post-release bugs; they both were grouped into one metric called *bug-fix*. In other words, a software file had a bug-fix and consequently was labeled as fault-prone if: (i) it was changed by at least one commit which was used to fix a bug, and (ii) that commit was made during the duration of that release, i.e., the period between $d1$ and $d2$ in Fig. 3(A).

Software fault-proneness prediction models using this learning approach were trained using static code metrics extracted on the release date and change metrics extracted from the entire release duration to predict the files that have bug-fixes during the entire duration of that or future release.

4.1.2 Data extraction for usePrePredictPost

Following the method illustrated in Fig. 3(B), previously used in (Nagappan et al., 2006; Zimmermann et al., 2007; Moser et al., 2008; Krishnan et al., 2013), for each release we extracted the change metrics from the pre-release duration only, that is, for the period between $d1$ and n in Fig. 3(B). The same static code metrics as for the *useAllPredictAll* learning approach were used (i.e., static code metrics extracted from the latest version of the binaries on the release date).

When using the *usePrePredictPost* learning approach we distinguished the bugs based on when they were detected and fixed. Thus, a software file had a *pre-release bug* if: (i) it was changed by at least one commit to fix a bug, and (ii) that commit was made during the pre-release duration, i.e., the period between $d1$ and n in Fig. 3(B). A software file had a *post-release bug* and consequently was labeled as fault-prone if: (i) it was changed by at least one commit to fix a bug, and (ii) that commit was made after the release date, i.e., the period between n and $d2$ in Fig. 3(B).

Software fault-proneness prediction models using the *usePrePredictPost* learning approach were trained using the static code metrics extracted at the release date, the change metrics extracted pre-release, and the pre-release bugs to predict the files that have post-release bugs in that or future release.

4.2 Open-source projects used to extract datasets

For this work we initially explored 23 projects from the Apache Software Foundation (2022a) that satisfied the reporting criteria defined by (Hall et al., 2012). Some projects, however, did not have clear release dates or a central repository for the source code, or were missing other artifacts. To be included in our study, a project had to satisfy the following six criteria: (i) Availability of the source code or binary distribution, needed to extract code metrics. (ii) Availability of the version control system, needed to generate the commit log file which is used to extract the change metrics. (iii) Availability of the bug tracking system, needed to identify the commits made to fix software bugs. (iv) At least two releases of the project, in order to build predictors across-releases (i.e., training the model on one release to predict fault-proneness for the subsequent release). (v) Release dates clearly specified, in order to be able to extract change metrics for each release as described in subsection 4.1. (vi) Release dates within the commits log file dates, to ensure that all commits made within a given release are considered.

Out of the 23 projects initially considered, eleven did not satisfy one or more of these inclusion criteria and therefore were excluded from this study. The names of these eleven projects and the specific inclusion criteria that were violated are provided in the supplemental document online (Ahmad et al., 2022).

Table 3 List of projects used in this study and their details. All projects were implemented in Java.

Project	# releases	Size (LOC)	# developers	Maturity (years)	Domain / functionality
Ant	7	376,250	47	+ 10	Command-line tool for java application building
Axis2	5	409,432	29	+ 10	Web services creating and usage
Derby	9	1,759,271	34	+ 9	Relational Database
Geronim	5	581,083	49	+ 8	Libraries for JavaEE/JakartaEE
Hadoop	6	1,500,351	72	+ 7	Distributed computing platform
Hive	6	3,255,810	43	+ 7	Data warehousing
jEdit	4	346,197	40	+ 15	Programmer text editor
MyFace	4	377,930	36	+ 9	Sub-projects for JavaServer technology
Pivot	5	215,406	7	+ 6	Platform for building install-able Internet Applications
Synapse	3	376,250	23	+ 10	Web Services
Wicket	6	398,043	21	+ 8	Web-apps developing environment
Xalan	4	398,183	32	+ 14	XSLT processor

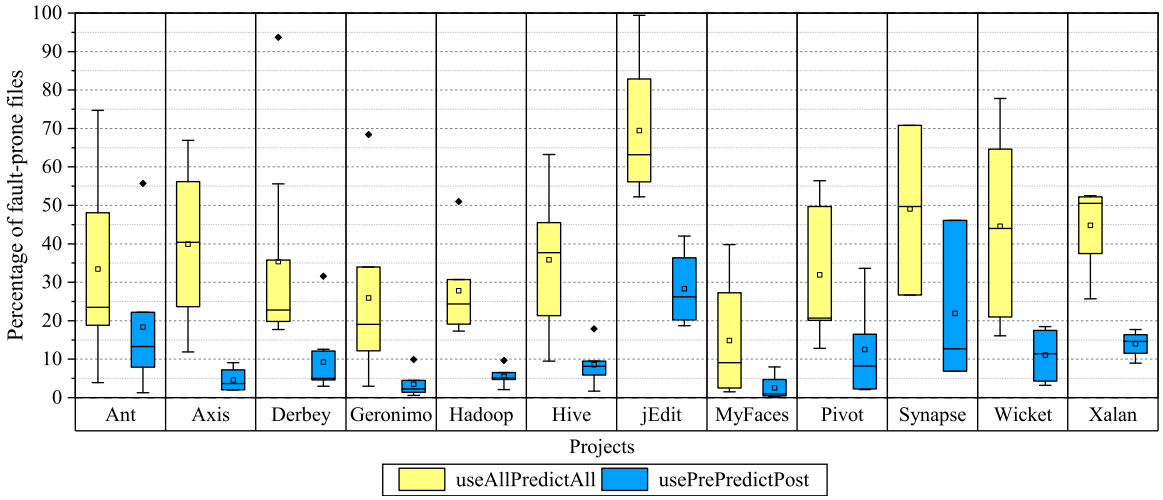

Fig. 4 Box plots of the percentages of fault-prone files per project, each with multiple releases

The datasets used in this study were extracted from the twelve open-source projects that satisfied all inclusion criteria and are listed in Table 3. Following the reporting requirements given in (Hall et al., 2012), for each project, Table 3 provides the number of releases, size in LOC, number of developers, maturity in years, and application domain/functionality. All projects were written in Java.

The total number of files, and the percentages of fault-prone files, for all releases of all twelve projects are provided in (Ahmad et al., 2022). For brevity, here we present the box plots of the percentages of fault-prone files: (1) for multiple releases of each of the twelve projects in Fig. 4, and (2) for all releases of all projects cumulatively in Fig. 5. As shown in these figures, the percentage of fault-prone files is significantly higher for the *useAllPredictAll* learning approach compared to the *usePrePredictPost* learning approach, both for the releases of each project individually (Fig. 4) and for all projects cumulatively (Fig. 5). For example, for the least fault-prone project, MyFaces, the mean percentages of fault-prone files across its four releases were 14.9% for *useAllPredictAll* and only 2.6% for *usePrePredictPost* learning approach. Similarly, the most fault-prone project, jEdit, had 69.5% and 28.3% mean percentages of fault-prone files across its four releases for the *useAllPredictAll* and *usePrePredictPost* learning approaches, respectively. These observations are explained by the facts that *useAllPredictAll* learning approach accounts for fault-prone files during the entire release duration, while the *usePrePredictPost* learning approach only considers post-release fault-prone files.

4.3 Feature vectors used for prediction

We extracted 20 static code metrics and 14 change metrics at the file level, which were then combined in feature vectors and used in our models. Note that while static code metrics represent a snapshot in time, change metrics capture the longitudinal changes made to each file over the specified duration (see section 4.1).

Static code metrics. We extracted 20 widely used static code metrics at the file level from the binary code of each release, on the release date from the Archive Server of the Apache Software Foundation (2022a). The metrics were extracted using the Chidamber and Kemerer Java Metrics (CKJM) tool (Jureczko and Spinellis, 2011). They

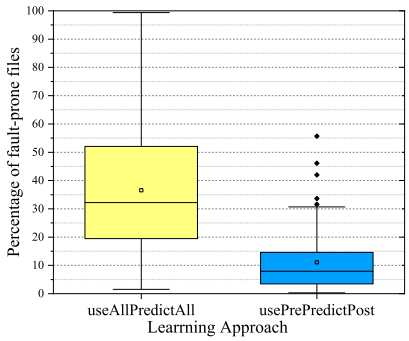

Fig. 5 Box plots of the percentages of fault-prone files for all releases of all projects

belong to six different metric suites: C&K metrics, Henderson-Sellers, Martin, QMOOD, Tang, and McCabe. The full descriptions of the static code metrics can be found in (Jureczko and Madeyski, 2010; Ahmad et al., 2022).

Change metrics. We extracted 14 change metrics at the file level following the process described in (Moser et al., 2008; Krishnan et al., 2013). Change metrics were extracted by integrating the information from the version control system (i.e., commit log) and the bug tracking system of each project. All twelve projects included in our study used the SVN version control system. Eleven of these projects used the JIRA bug tracking system (Jira, 2022), and one (i.e., Ant) used the Bugzilla bug tracking system (Apache Software Foundation, 2022b). Change metrics for each release were extracted from all commits made between $d1$ and $d2$ dates for the *useAllPredictAll* learning approach and between $d1$ and n dates for the *usePrePredictPost* learning approach, as shown in Figs. 3(A) and 3(B), respectively. The list of change metrics along with their definitions and additional details about the change metrics extraction process are provided in (Ahmad, 2021; Ahmad et al., 2022).

Feature vectors used for our machine learning experiments integrate the static code and change metrics which were extracted separately. To combine these metrics for each file, in each release and project, we created an algorithm, provided in (Ahmad et al., 2022), that matches the names of class files from the binary distribution with the names of Java files from the commit logs. Note that the number of files in the static code metrics list was higher than the number of files in the change metrics list because the binary distributions of each release also had other files from external software libraries, which were excluded from the analysis. For the files considered in this work, if the class name from the static code metrics list matched the file name in the commits logs for that release, the changes were aggregated, and the change metrics were created. Otherwise, it meant that the file did not change, and the values of all its change metrics were set to 0 for that release. The full feature vector for each file was created by concatenating the 20 static code and 14 change metrics. As described in subsection 4.1, a file was labeled as fault-prone if it had at least one bug-fix in the case of the *useAllPredictAll* learning approach or at least one post-release bug in the case of the *usePrePredictPost* learning approach.

5 Machine learning and DoE approaches

This section presents our machine learning approach, which utilizes the created feature vectors for prediction of software fault-proneness, followed by the description of the DoE approach we used to explore and quantify the effect of the learning approach on the prediction performance.

5.1 Machine learning approach

We evaluated models' prediction performance using the feature vectors consisting of both the static code metrics and change metrics (see subsection 4.3). For classification, we used several traditional machine learning algorithms widely used in software-fault proneness prediction: Linear Discriminant Analysis (LDA), kNN, LR, NB, J48, and RF. The results showed that, for almost all performance metrics, there were no statistically significant differences between J48, LDA, and LR. RF performed slightly better, while kNN and NB performed worse than these three learners. In addition to the above mentioned traditional classifiers, we experimented with one deep learner for the classification task – a Feedforward Neural Network (FNN) with an input layer, two hidden layers, and an output layer[1]. FNN led to somewhat better classification performance than J48 and RF, but nevertheless the two learning approaches had similar effect on its performance as in the cases of J48 and RF.

Since the effect of the machine learning algorithm used for classification on prediction performance has been previously explored by many works, we do not treat it as a factor in our experiments. We here only present the

[1] The input layer and hidden layers of the FNN were defined as 'dense' layers with 64 neurons and the ReLU activation function. The final output layer consisted of a single neuron that produced the prediction output using the sigmoid activation function. FNN was implemented using Keras, an open-source library that provides Python interface for artificial neural networks.

findings of using the J48 algorithm, which has been widely used for software fault-proneness prediction (Khoshgoftaar and Seliya, 2004; Guo et al., 2004; Arisholm et al., 2007; Menzies et al., 2007; Moser et al., 2008; Arisholm et al., 2010; Krishnan et al., 2013; Alshehri et al., 2018; Goseva-Popstojanova et al., 2019) and has been shown to be among the top performing algorithms (Lessmann et al., 2008; Krishnan et al., 2013; Shepperd et al., 2014). Our findings when using other learning algorithms were consistent with those presented here, including the results of using RF, which are provided in the supplemental online document (Ahmad et al., 2022).

For each learning approach, we explored both within-release and across-releases predictions. For within-release prediction, we used two data splitting techniques: 10-fold cross validation and 50/50 split. In both cases, the models were trained and tested on data collected from the same release. It is important to emphasize that, for each learning approach, the files from that release (characterized by the corresponding feature vectors and target variables, as described in Section 4) were split into a training set and testing set, which did not overlap (i.e., the prediction models were evaluated on unseen data).

For 10-fold cross validation (which we label as M1) the data were divided into ten folds using random stratified sampling; nine folds of the data were used for training and the remaining fold was used for testing. This was repeated ten times, each time using a different fold for testing. 10-fold cross validation is widely used in this field (Turhan et al., 2009; Giger et al., 2012; Bowes et al., 2018).

For the second data splitting technique, referred to as 50/50 split (and labeled as M2), stratified random sampling was used to split the data into two folds, with 50% of the data in each. One fold was used to train the model and the other to test it. To avoid bias, following (Kamei et al., 2010; Nam et al., 2013), the 50/50 split was repeated 100 times with different random stratified samples, and the averages of the performance metrics are reported. We chose this splitting technique as it is similar in design to the across-release prediction.

For the prediction across-releases (labeled as M3), the data from a given release n was used for training, and the data from release $n+1$ was used for testing (Arisholm et al., 2007; Nam et al., 2013; Song et al., 2019). Note that, as described in section 4.1, *useAllPredictAll* and *usePrePredictPost* have different intervals during which the features were collected (i.e., have different feature vectors) and different target variables (i.e., fault-prone files during entire release duration vs. fault-prone files during the post-release interval).

For evaluating the models' classification performance, we used the following metrics:

$$\text{Recall (R)} = \text{TP}/(\text{TP} + \text{FN})$$
$$\text{Precision (P)} = \text{TP}/(\text{TP} + \text{FP})$$
$$\text{False Positive Rate (FPR)} = \text{FP}/(\text{FP} + \text{TN})$$
$$\text{F-Score} = 2 \cdot \text{Recall} \cdot \text{Precision}/(\text{Recall} + \text{Precision})$$
$$\text{G-Score} = 2 \cdot \text{Recall} \cdot \text{(1-FPR)}/\,[\text{Recall} + \text{(1-FPR)}]$$

where True Positive (TP) represents the number of files that were faulty, and were predicted to be faulty; False Negative (FN) represents the number of files that were faulty, but were predicted to be not faulty; False Positive (FP) represents the number of files that were not faulty, but were predicted to be faulty; and True Negative (TN) represents the number of files that were not faulty and were predicted to be not faulty.

The values of all performance metrics are between 0 and 1. Good performance has higher Recall and Precision, and lower FPR. F-Score and G-Score are harmonic means of two metrics (i.e., Recall and Precision for F-Score and Recall and (1-FPR) for G-Score) and are high only when both metrics are high. Note that instead of FPR we report (1-FPR), which leads to higher values indicating better performance for all performance metrics.

5.2 Design of Experiments (DoE) approach

In this study we used Design of Experiment (DoE) approach to draw statistically sound findings related to the effect of factors (independent variables) on the response variables (i.e., performance metrics of the software fault-proneness prediction). Basically, we considered two factors:
Factor A: the learning approach, with two levels:
- ***useAllPredictAll***
- ***usePrePredictPost***

Factor B: the prediction style, with three levels:
- within-release using 10-fold cross validation (**M1**)
- within-release using 50/50 split (**M2**)
- across-releases (**M3**).

In selecting a DoE approach that supports our investigation, we chose nested design because the prediction style (factor B) is nested within the learning approach (factor A). Note that the widely used cross design requires each factor to be applied equivalently across each level of the other factor, which does not apply to our study because the datasets for prediction styles M1-M3 are different for different learning approaches. That is, feature vectors have different values for change metrics and the target variables are different, as described in subsection 4.1.

Furthermore, our design is unbalanced because the number of observations (i.e., instances of machine learning experiments) for different combinations of factors' levels are different. Specifically, for within-release prediction

styles (M1 and M2), the performance metrics (i.e., instances) were computed using the datasets of each of the 64 releases. However, for prediction across-releases (M3) the number of machine learning instances for each of the twelve projects (listed in Table 3) equals the number of its releases minus one, since it takes two releases to generate one prediction. In total, for the twelve projects considered in this paper, there were 52 instances for the across-releases predictions. It should be noted that several instances were excluded from the analysis because performance metrics could not be computed due to division by 0. For example, when the model classified all cases as negative (i.e., all files as not faulty) TP + FP = 0, which led to Recall = 0 and Precision that could not be computed. The predictions when TP + FN = 0 led to Precision = 0 and Recall that could not be computed. These instances, however, were rare.

For example, for within-release with 10-fold cross validation (M1) prediction style, only one model was excluded for the *useAllPredictAll* learning approach (leading to 63 instances), while six models were excluded for *usePrePredictPost* (resulting in 58 instances). The numbers of instances for all combinations of the two learning approaches and three prediction styles are shown in Table 4[2].

Table 4 The number of instances for the nested DoE

Learning approach					
useAllPredictAll			usePrePredictPost		
Prediction style			Prediction style		
M1	M2	M3	M1	M2	M3
63	64	52	58	59	46

Nested analysis of variance (ANOVA), also called a hierarchical ANOVA, is an extension of ANOVA for experiments where each group is divided into two or more subgroups. It tests if there is variation between groups, or within nested subgroups. Nested ANOVA (as ANOVA in general) is a parametric method which assumes population normality and variance homogeneity, and a balanced design is preferred. Since these assumptions do not hold in our case, we used the non-parametric alternative proposed in (Stavropoulos and Caroni, 2008) and used elsewhere (Zahalka et al., 2010). The Box-Adjusted Wald-type statistic neither assumes normality nor homogeneity of the data. It is also robust to unbalanced design. We used it to test the following null hypotheses, related to RQ1 and RQ2, respectively:

- H_0^A: There is no difference in the distributions of each performance metric (Recall, Precision, 1-FPR, F-Score and G-Score) between the two learning approaches, and
- $H_0^{B|A}$: For a given learning approach, there is no difference in the distribution of each performance metric among prediction styles M1, M2, and M3.

6 Results

In this section we report the results as they pertain to our research questions. Fig. 6 shows the box plots, and Fig. 7 depicts the means of all performance metrics for the two learning approaches and three prediction styles. The basic statistics (i.e., mean, median, standard deviation, and IQR) are provided in (Ahmad et al., 2022).

6.1 RQ1: Does the learning approach affect the classification performance of the software fault-proneness predictions?

As shown in Figs. 6 and 7, models based on *useAllPredictAll* had significantly higher Recall, Precision, F-Score and G-Score than the same models based on *usePrePredictPost*. However, 1-FPR was slightly lower when using *useAllPredictAll* than in the case of *usePrePredictPost*.

Table 5 shows the analysis of variance results for the DoE described in subsection 5.2. Since p-values were less than the significance level 0.05 for Recall, Precision, F-Score and G-Score, the null hypotheses H_0^A were rejected in favor of the alternative hypotheses H_a^A that there is a difference in the performance metrics distributions for each of these performance metrics. Furthermore, the learning approach factor contributed the most to the variance of the prediction performance, i.e., 76.4%, 72.99%, 67.57%, and 58.31% for Recall, Precision, F-Score, and G-Score, respectively. However, for the 1-FPR performance metric H_0^A cannot be rejected, i.e., unlike the other performance metrics, 1-FPR was not affected by the learning approach.

Specifically, in prediction within-release using 10-fold cross validation (M1), *useAllPredictAll* compared to *usePrePredictPost* had 52.8% higher Recall, 38.8% higher Precision, 51.7% higher F-Score, and 22.4% higher G-Score. However, 1-FPR was 2.0% lower. In prediction within-release using 50/50 split (M2) the same pattern was observed, but the differences in performance were much higher. In particular, the *useAllPredictAll* learning

[2] Note that, following many related works (e.g., (Malhotra and Raje, 2015; Bowes et al., 2018)) that used the open-source datasets created by Jureczko and Spinellis (2010); Jureczko and Madeyski (2010), we treated the releases as independent instances for the statistical analysis.

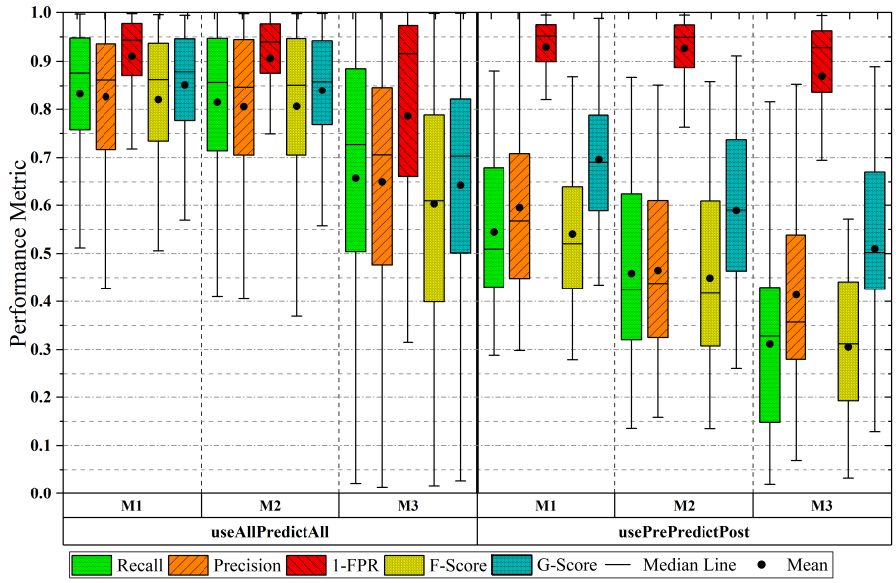

Fig. 6 Performance metrics for the two learning approaches and three prediction styles

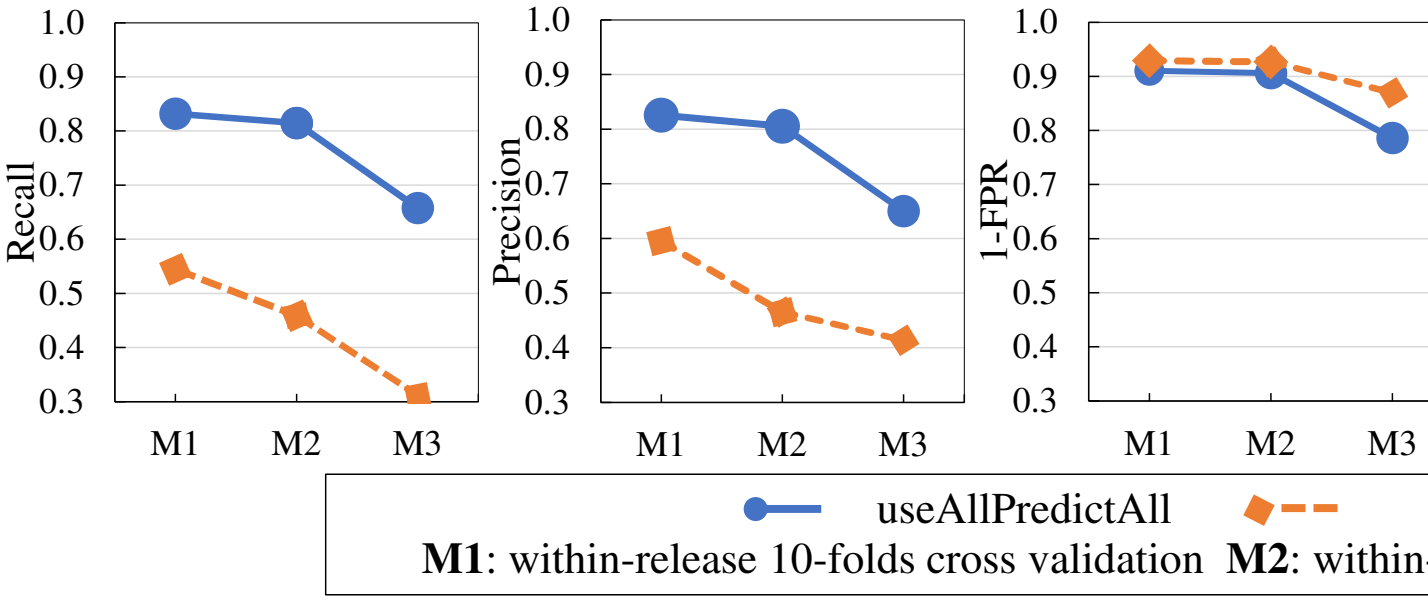

Fig. 7 Means of all performance metrics for the two learning approaches and three prediction styles

approach had 77.6% higher Recall, 73.1% higher Precision, 79.5% higher F-Score, and 42.5% higher G-Score than *usePrePredictPost*. As in the case of 10-fold cross validation, 1-FPR was 2.2% lower. Similarly, in prediction across-release (M3), compared to *usePrePredictPost* learning approach, *useAllPredictAll* had a 111.1% higher Recall, 57.1% higher Precision, 97.3% higher F-Score, and 26.1% higher G-Score. However, it had 9.6% lower 1-FPR.

Our findings thus provide a clearly affirmative answer to RQ1: the *useAllPredictAll* learning approach leads to better fault-proneness prediction performance than *usePrePredictPost*, both within-release and across-releases, for 4 of the 5 performance metrics.

6.2 RQ2: For a given learning approach, what is the difference in classification performance between using within-release and across-releases prediction styles?

To answer this research question, we again refer to Figs. 6 and 7. When the *useAllPredictAll* learning approach was used for within-release predictions, the models which used 10-fold cross validation (M1) slightly outperformed the corresponding models which used 50/50 split (M2) in terms of the mean Recall, Precision, F-Score, and G-Score, and had the same mean 1-FPR. Specifically, using 10-fold cross validation led to 2.1% higher Recall, 2.5%

Table 5 Analysis of variance results for learning approach (A) and nested prediction style ($B|A$)

Performance Metric	Factor	Wald-type Box-Adjusted Rank Statistic	p-value	H_0^A / $H_0^{B\|A}$	Cont to Var %
Recall	A	39.65865	3.02E-10	Rej	76.40
	$B \mid A$	7.86835	9.65E-02	Not Rej	23.60
Precision	A	32.51898	1.18E-08	Rej	72.99
	$B \mid A$	8.86537	6.46E-02	Not Rej	27.01
1-FPR	A	0.20742	6.49E-01	Not Rej	5.97
	$B \mid A$	3.35987	5.00E-01	Not Rej	94.03
F-Score	A	37.79243	7.87E-10	Rej	67.57
	$B \mid A$	12.64919	1.31E-02	Rej	32.43
G-Score	A	24.77202	6.45E-07	Rej	58.31
	$B \mid A$	15.80304	3.30E-03	Rej	41.69

higher Precision, 1.7% higher F-Score, and 1.5% higher G-Score compared to when 50/50 split was used. Both within-release prediction styles (M1 and M2) significantly outperformed the across-releases prediction (M3) with respect to all performance metrics. Since the two within-release styles had similar performance, we restrict the comparison to using the 50/50 split. When compared to prediction across-releases, the performance within-release using 50/50 split had 23.9% higher Recall, 23.9% higher Precision, 15.3% higher 1-FPR, 33.8% higher F-Score, and 30.5% higher G-Score.

When *usePrePredictPost* was used for within-release predictions, the models which used 10-fold cross validation (M1) outperformed the models which used 50/50 split (M2) for all performance metrics. Specifically, using 10-fold cross validation within-release led to 18.7% higher Recall, 27.9% higher Precision, 0.2% higher 1-FPR, 20.3% higher F-Score, and 18.2% higher mean G-Score compared to the same models which used the 50/50 split. Consistent with the findings for the *useAllPredictAll* learning approach, when *usePrePredictPost* was used, both within-release prediction styles (M1 and M2) outperformed the across-releases prediction (M3) with respect to all performance metrics. In particular, when using within-release with 10-fold cross validation, compared to across-releases predictions the Recall, Precision, 1-FPR, F-Score, and G-Score were 74.8%, 43.8%, 6.8%, 76.9%, and 36.5% higher, respectively. Following the same pattern, using within-release with 50/50 split compared to across-releases predictions resulted in 47.2% higher Recall, 12.4% higher Precision, 6.6% higher 1-FPR, 47.0% higher mean F-Score, and 15.5% higher G-Score.

To further investigate the effect of the prediction style on the fault-proneness predictions for a given learning approach, we refer to the rows relevant to the hypothesis $H_0^{B|A}$ in Table 5. It appears that $H_0^{B|A}$ cannot be rejected for Recall, Precision, and 1-FPR as the p-values were greater than the significance level 0.05. On the other hand, $H_0^{B|A}$ hypotheses were rejected for F-Score and G-Score in favor of the alternative hypotheses that, for a given learning approach, there is a difference in the performance due to prediction styles.

Our findings give an affirmative answer to RQ2: the classification performance is significantly better when within-release prediction is used than when across-releases prediction is used, for both learning approaches.

Our answer to RQ2 is consistent with prior works for each given learning approach. However, our findings go a step further, showing that the contribution to the variance due to prediction style is smaller than the contribution to the variance due to the learning approach, as shown in Table 5. In other words, *the learning approach matters more than the prediction style.*

6.3 On reasons behind different performance

Our results for RQ1 showed that the prediction models which used *useAllPredictAll* had significantly better performance than the models which used *usePrePredictPost* in terms of Recall, Precision, F-Score and G-Score, but had similar or slightly worse 1-FPR. To better understand and explain the reasons behind the impact of the learning approach choice on software fault-proneness prediction performance, we conducted further investigation. Specifically, we took a closer look at the datasets created for each learning approach. We hypothesize that the *useAllPredictAll* performed better than the *usePrePredictPost* learning approach is because its datasets were less imbalanced compared to the corresponding *usePrePredictPost* datasets (see Figs. 4 and 5).

In order to investigate if the different levels of class imbalance intrinsically present in the datasets used by different learning approaches cause the difference in performance, we ran experiments using three different types of imbalance treatments: Synthetic Minority Over-sampling Technique (SMOTE), Under-bagging, and Adaptive Boosting (i.e., AdaBoost).

SMOTE (Chawla et al., 2002) has been widely used in prior studies as a treatment for imbalance (e.g., (Agrawal and Menzies, 2018; Goseva-Popstojanova et al., 2019)). It works by selecting random data points from the minority class (i.e., files with bug-fixes when using *useAllPredictAll* and files with post-release bugs when using *usePrePredictPost*) and mimicking similar data points to over-represent the minority class, with the goal of enhancing the

classification performance. When using SMOTE, we oversampled the minority class (i.e., fault-prone files) in the training set to match the number of instances of the majority class. For example, when applying SMOTE for the *useAllPredictAll* learning approach with the Ant 1.3 release, the minority class was oversampled to increase the percentage of bug-fixes in the training sets from 44.7% to 50%. And when applying SMOTE for the *usePrePredict-Post* learning approach, using oversampling the percentage of post-release bugs was increased from 6.5% to 50%. (In each case, the imbalance treatment was applied only to the training set, and the testing set was not modified, that is, remained unbalanced as in each of the original datasets. This eliminates over-fitting bias and yields more reliable results.)

Under-bagging technique combines undersampling with bagging, and belongs to a broader family of ensemble learning methods (Galar et al., 2011). To use this technique, we created balanced subsets (i.e., bags) of the training dataset using undersampling (i.e., random sub-sampling with replacement) of the majority class and trained individual J48 models on 10 different, random subsets of the training dataset. The final classification outcome was aggregated using majority voting.

AdaBoost (i.e., Adaptive Boosting) combines the concepts of adaptive learning with boosting and also belongs to the family of ensemble learning methods. AdaBoost dynamically adjusts the weights of training instances in subsequent iterations of training based on the difficulty of correctly classifying the instances (i.e., increases the weights of misclassified samples and decreases the weights of correctly classified samples). The classifier used in each iteration was J48. The final prediction was determined using weighted majority vote from all the decision trees (Galar et al., 2011; Schapire, 2013).

Overall, for within-release predictions SMOTE did quite well compared to Under-bagging and AdaBoost. For example, for within-release prediction with 10-fold cross validation, SMOTE had the best performance in 41 datasets (out of 63) for useAllPredictAll and in 56 datasets (out of 58) for usePrePredictPost learning approach. Furthermore, the magnitudes of improvement were significantly higher with SMOTE. The results were similar for within-release with 50/50 split predictions. None of the data balancing techniques, however, led to improvement of the across-releases prediction performance.

Since SMOTE significantly outperformed the Under-bagging and AdaBoost, here we only present the results of applying SMOTE as an imbalance treatment. Note, however, that in this paper we use imbalance treatment to explore and (at least partially) explain why the two learning approaches lead to different prediction performance. Data imbalance is one of the most challenging problems in machine learning which, in addition to between-class imbalance addressed here, may include issues related to the separability of classes and within-class imbalance (He and Garcia, 2009), all of which are even more challenging when combined with the small sample sizes typical for software engineering. More through explorations of different issues related to data imbalance are beyond the scope of this paper.

The box plots and means of all performance metrics when SMOTE was applied are shown in Figs. 8 and 9, respectively. In Table 6 we report the differences in performance metrics when SMOTE was applied compared to when SMOTE was not applied.

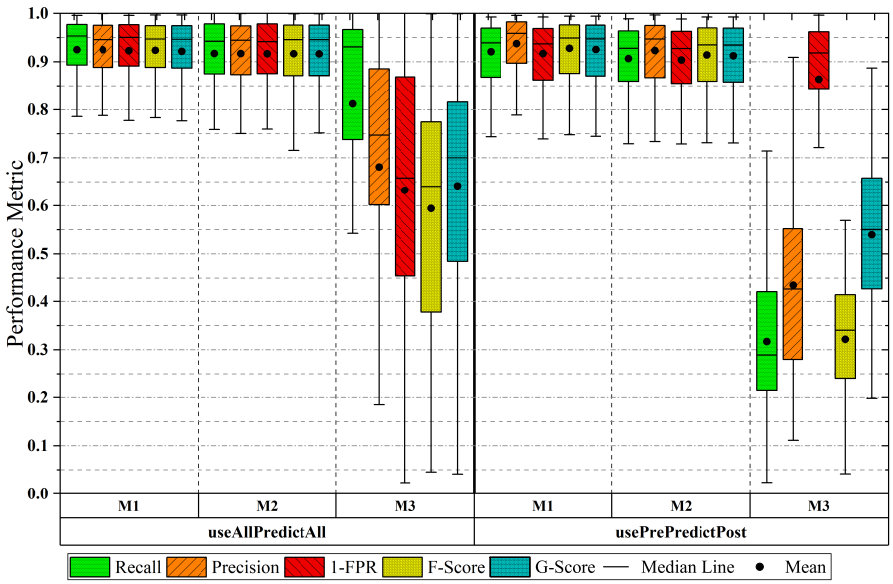

Fig. 8 Performance metrics for the two learning approaches and three prediction styles when SMOTE was applied

As can be seen from Figs. 8 and 9, and Table 6, applying SMOTE with the *useAllPredictAll* learning approach improved all performance metrics for within-release prediction styles (M1 and M2). Using SMOTE across-releases (M3) only slightly improved the Recall and 1-FPR, while it slightly decreased the Precision, F-Score and G-Score. Applying SMOTE with the *usePrePredictPost* learning approach led to a significant increase of Recall, Precision, F-

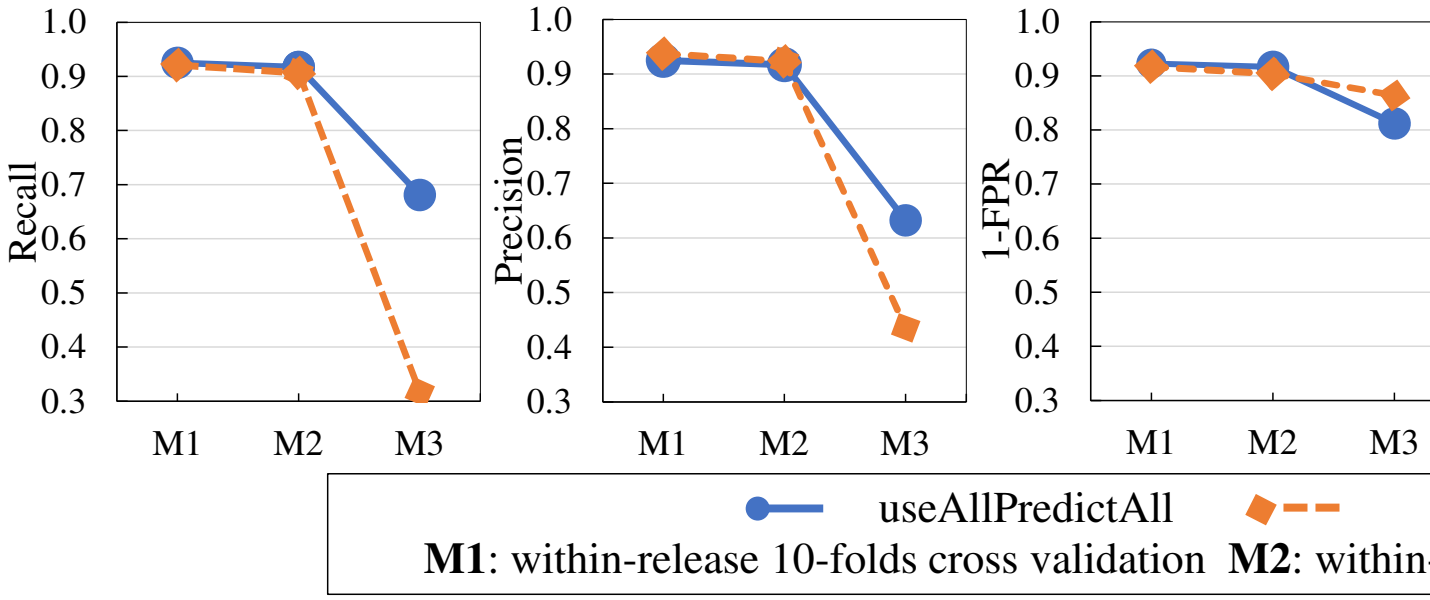

Fig. 9 Means of performance metrics for the two learning approaches and three prediction styles when SMOTE was applied

Table 6 Differences in performance metrics when SMOTE was applied compared to when no imbalance treatment was not applied

	useAllPredictAll			usePrePredictPost		
Metric	**M1**	**M2**	**M3**	**M1**	**M2**	**M3**
Recall	11.2%	12.6%	3.5%	69.1%	97.5%	1.7%
Precision	12.0%	13.9%	-2.7%	57.5%	98.5%	5.3%
1-FPR	1.4%	1.2%	3.3%	-1.3%	-2.5%	-0.7%
F-Score	12.6%	13.7%	-1.5%	71.7%	103.5%	5.3%
G-Score	8.2%	9.1%	-0.2%	33.0%	54.9%	5.8%

Score, and G-Score in within-release prediction styles (M1 and M2), while the increase in across-releases predictions (M3) was minor. In all cases, applying SMOTE resulted in slightly lower 1-FPR.

In summary, using SMOTE for within-release predictions (M1 and M2) significantly improved all performance metrics expect 1-FPR, for both learning approaches. The improvement for *usePrePredictPost* was significantly higher than for *useAllPredictAll*. This is due to the fact that the datasets used with the former were significantly more imbalanced than the datasets used with the latter, leading to a much higher positive impact of the imbalance treatment.

Table 7 shows the results of the analysis of variance for our nested design when SMOTE was applied. The statistical results confirm the observations made based on the results shown in Figs. 8 and 9 – for all performance metrics we cannot reject the null hypothesis H_0^A that there is no difference between the performance of the two learning approaches.

Table 7 Analysis of variance results for learning approach (A) and nested prediction style ($B|A$) when SMOTE is applied

Performance Metric	**Factor**	**Wald-type Box-Adjusted Rank Statistic**	**p-value**	H_0^A / $H_0^{B\|A}$	**Cont to Var %**
Recall	A	2.864469	9.06E-02	Not Rej	10.55
	$B \mid A$	38.57545	8.52E-08	Rej	89.45
Precision	A	0.1863529	6.66E-01	Not Rej	0.76
	$B \mid A$	42.73856	1.17E-08	Rej	99.24
1-FPR	A	0.7205299	3.95E-01	Not Rej	25.87
	$B \mid A$	2.089948	7.19E-01	Not Rej	74.13
F-Score	A	0.7934577	3.73E-01	Not Rej	3.15
	$B \mid A$	47.81298	1.03E-09	Rej	96.85
G-Score	A	0.518229	4.72E-01	Not Rej	2.08
	$B \mid A$	44.453	5.17E-09	Rej	97.92

It should be noted that prior works have shown that treating the class imbalance improves the software fault-proneness prediction performance. Some of these works used *useAllPredictAll* (Wang and Yao, 2013; Malhotra and Jain, 2020; Goyal, 2022), while other used *usePrePredictPost* learning approach (Goseva-Popstojanova et al., 2019). What is new here is that we apply class imbalance treatment on both learning approaches and compare the corresponding improvements. Furthermore, we use imbalance treatment with a different goal than related works –

to test our hypothesis that, for within-release predictions, the class imbalance can be used to explain the difference in prediction performance of the two learning approaches.

For within-release predictions, our findings show that the worse performance of *usePrePredictPost* compared to the *useAllPredictAll* learning approach was due to the more pronounced class imbalance of the former. When addressed, using the imbalance treatment SMOTE, the performance difference between learning approaches was eliminated.

(Note that the other two imbalance treatment we used, Under-bagging and AdaBoost, did not eliminate the performance difference between the learning approaces.)

Contrary to within-release predictions, using SMOTE for across-releases prediction (M3) had a minimal effect on prediction performance for both *useAllPredictAll* and *usePrePredictPost* learning approaches (see Table 6). As can be seen in Table 7, the null hypotheses $H_0^{B|A}$ were rejected for all performance metrics except 1-FPR in favor of the alternative hypotheses $H_a^{B|A}$ that (for a given learning approach) there is a difference in the performance metrics distributions due to the nested factor (i.e., prediction style). Moreover, when SMOTE was applied, the prediction style factor contributed most of the variance, i.e., 89.45%, 99.24%, 74.13%, 96.85%, and 97.92% in the case of Recall, Precision, 1-FPR, F-Score, and G-Score, respectively. (It should be noted that in case of across-releases prediction, Under-bagging and AdaBoost, similarly to SMOTE, had little effect on the prediction performance.)

In general, prediction across-releases is more challenging than prediction within-release because it is harder for machine learning algorithms to learn effectively when the sources of training and testing data are different, which typically results in different distributions (Xu et al., 2018, 2019). It is an open research question whether different approaches for improving the performance of across-release predictions (Xu et al., 2018, 2019; Amasaki, 2020; Kabir et al., 2021) or transfer learning approaches (Ma et al., 2012; Nam et al., 2013, 2018)) would reduce the difference in performance when using different learning approaches.

7 Threats to validity

We discuss the threats to validity grouped into four categories: construct, internal, conclusion, and external validity.

Construct validity is concerned with whether we are measuring what we intend to measure. One threat to construct validity is related to the commits made to version control systems, which were used to extract the change metrics. As expected, data were missing from some commits. Commits with no revision numbers were given auto-generated unique ID numbers and were included in the study. On the other hand, commits with no lines of code reported as added or deleted were omitted from this study, which represented less than 3% of all commits made to all projects combined. Unlike change metrics, static code metrics represent a snapshot in time. For each release, the static code metrics were extracted from the latest available binaries on the release date, and they were the same for both learning approaches. The feature vectors used for the machine learning experiments in this paper integrate the static code metrics and change metrics as described in subsection 4.3. For the statistical tests, following the common practice in this area, we treated the releases as independent instances. As the well known 'No Free Lunch Theorem' implies, there is no single best machine learning algorithm for all predictive modeling problems. Therefore, we experimented with multiple classification algorithms (i.e., J48, RF, FNN) and imbalance treatment appraches (i.e., SMOTE, Under-bagging, and AdaBoost). A brief summary of the results of machine learning experiments are presented in the paper, accompanied by detailed results of using J48 for classification and SMOTE for imbalance treatment. Note that further exploration of the classifier and the imbalance treatment effect on the prediction performance is beyond the scope of this paper, whose main goal is to explore the effect of the learning approach.

Internal validity is concerned with the effects of unknown impacts that might affect the independent and dependent variables. To ensure data quality, which is one of the major threats to internal validity, we extracted our own data from the online Apache repository. Note that we used a consistent set of static code and change metrics for all the projects and releases considered in this paper. Upon extracting the data, we implemented manual and automated sanity checks to verify the quality of the extracted metrics. In addition, for randomly selected files from each project included in this study, we manually verified and validated the values of the automatically extracted metrics.

Conclusion validity threats may impact the ability to draw correct and reliable conclusions. Some threats to conclusion validity are related to the way descriptive statistics are reported and statistical tests are being used. For descriptive statistics, we provided the box plots of performance metrics in this paper and reported their means, medians, standard deviations, and IQRs in the supplemental document online (Ahmad et al., 2022). For the inferential statistics, we used design of experiments approach and non-parametric tests suitable for our datasets. Another threat to conclusion validity is related to the data sample sizes. The work presented in this paper is based on 64 releases of 12 different open-source projects, which is comparable to or larger than the sample sizes used in related works in this area.

External validity is concerned with our ability to generalize our conclusions. The relatively large number of releases and projects that we studied provides some generalizability. However, this work was based only on open-source software written in Java and available on the Apache Projects web server. Therefore, we cannot claim that the results would be valid for software products implemented in other languages and/or from different application domains, as well as for other classifiers and imbalance treatment approaches.

8 Recommendations and research directions

In this paper we seek to bring to the attention of the research and practitioners' communities the unreported effects of the learning approach on software fault-proneness predictions. This paper presents evidence that current software fault-proneness prediction studies do not adequately take into account the learning approach used. It also shows how the effect of excluding this missing factor can inadvertently distort empirical observations.

Through empirical results and statistical tests presented in Section 6 of this paper, we show that the learning approach significantly affects the performance of software fault-proneness prediction. We also provide interpretation of our findings for research questions RQ1 and RQ2, and uncover indications that the difference in performance between the two learning approaches for within-release predictions is attributable to class imbalance. In this section, we discuss the implications of our findings and provide recommendations for designing, reporting, and comparing software fault-proneness prediction studies. We also suggest some directions where further work is needed.

Toward significantly improving future fault-proneness prediction studies, **our findings lead us to make the following five recommendations**. These recommendations all rely on existing approaches and techniques, so can readily be put into action right away.

- **Incorporate Data and Input Aspects into the Prediction Process.** Many research works on software fault-proneness prediction utilize readily available datasets and focus on the AI model dimension (i.e., the steps of building and evaluating the classification models), without thorough understanding of the data and input used by these models. However, information about the interval for which longitudinal features (software metrics data) are collected and the interval for which bugs are predicted (the target variable) needs to be specified and considered, not swept under the carpet. Omission of this information leads to overlooking the learning approach factor and not accounting for its effect on the prediction performance. For sound research and practical application of software fault-proneness prediction, all dimensions and steps of the OECD framework (OECD, 2022)) or the phases of the alternative Cross Industry Standard Process for Data Mining (CRISP-DM)[2] (Chapman et al., 2000; Wirth and Hipp, 2000) should be incorporated, regardless of whether the datasets are extracted from the software artifacts or are readily available.
- **Use the Learning Approach Appropriate for the Goal and Available Data.** Software fault-proneness prediction models using the *useAllPredictAll* approach predict fault-proneness for the entire period under consideration (e.g., release) and thus reflect mainly the *developers' viewpoint*. On the other hand, models using the *usePrePredictPost* learning approach predict software units (e.g., files) that are expected to be fault-prone after the cut-off date for prediction (e.g., post-release). They thus better reflect the *users' viewpoint* regarding the software's perceived reliability and may more readily prevent costly consequences from post-release failures.
- **Address Class Imbalance.** Since datasets used for software fault-proneness prediction are usually imbalanced, treatment of imbalance should be used. We have shown that SMOTE significantly improves the classification performance of within-release prediction style, for each learning approach. This confirms the results presented in several prior works (Wang and Yao, 2013; Agrawal and Menzies, 2018; Goseva-Popstojanova et al., 2019; Malhotra and Jain, 2020; Goyal, 2022). Furthermore, we have shown that the class imbalance contributes to the difference in the prediction performance of the two learning approaches and that when treated with SMOTE results in similar prediction performance for both learning approaches. It should be noted that more research focused on handling data imbalance for software fault-proneness prediction is needed, because it is a complex phenomenon that, in addition to the imbalanced class distributions, is affected by small sample sizes (which are typical for software fault-proneness data), separability of the classes, and the existence of within-class concepts (He and Garcia, 2009).
- **Always Specify the Learning Approach.** Our results have demonstrated that information regarding the learning approach is essential to understanding and explaining the performance of software fault-proneness prediction. However, such information is often missing or is merely implicit, in some cases requiring considerable effort to deduce. Therefore, research studies focused on software fault-proneness prediction, in addition to the existing reporting criteria (Hall et al., 2012), should explicitly specify the learning approach.
- **Compare Apples-to-Apples.** To avoid comparing 'apples to oranges,' care should be taken when comparing the performance of software fault-proneness prediction models – when comparing to the related works' results, when conducting meta-analysis, and/or when reusing existing datasets that may have been created for different learning approaches. Since useAllPredictAll and usePrePredictPost use different training sets and have different target variables, such performance comparisons are unfair, and any statistical analysis and/or

[2] CRISP-DM is a comprehensive process model for carrying out data mining efforts (Chapman et al., 2000). Similarly as the OECD framework, it is independent of both the industry sector and the technology used, and consists of the following phases: business understanding, data understanding, data preparation, modeling, evaluation, and deployment.

machine learning experiments are flawed. Therefore, our results may call into question conclusions advanced by some prior publications that failed to recognize the use of different learning approaches and to consider their effect on the performance of the fault-proneness prediction.

Additionally, **we describe limitations of the current state-of-the-art and identify some directions where future research is needed to further improve the software fault-proneness prediction.**

- **Current software fault-proneness prediction models only consider spatial information** about software faults (i.e., their location) while disregarding their temporal information (i.e., when each fault was detected). Even though software fault-proneness prediction classifies software units (the "where"), the faults within fault-prone unit(s) were detected at different times. In the case of within-release predictions, the complete lack of temporal information when using the *useAllPredictAll* learning approach leads (at least for some faults) to predicting the past from the future. This is not the case with the *usePrePredictPost* learning approach because it distinguishes between faults detected pre-release and post-release (i.e., it considers temporal information at a coarse-grained level) and predicts post-release fault-prone units. In order to follow a realistic usage scenario, it is important to ensure that training data temporally precede testing data. However, the great majority of past studies evaluated the fault-proneness prediction performance on temporally random data[3] rather than on future data (Falessi et al., 2020). Note that the temporal order of the data is preserved for across-release predictions (i.e., when the model is trained on one release and tested on a future release of the same or other software project) regardless of the learning approach used.
- **An important future research direction is thus to take into account temporal information.** Incorporating temporal information about software faults into the fault-proneness prediction will be essential to creating prediction models that are truer representations of the fault detection process. Doing so would address the fundamental drawback of the *useAllPredictAll* approach when used for within-release predictions. Our recommendation to take into account temporal information is backed by a recent study (Falessi et al., 2020) that argued in favor of preserving the order of data when validating the fault-proneness predictions. That study, however, was focused only on across-releases predictions. In order to further advance software fault-proneness prediction and increase its practical usefulness, we recommend incorporating finer-grained temporal information, beyond the current distinction between pre-release and post-release faults in the *usePrePredictPost* approach, and beyond preserving the temporal order at the coarse level of whole releases (as it is done for accross-releases predictions). Initial effort along these lines was presented in a recent study by Kabir et al. (2021) which was focused on fault-proneness prediction for *useAllPredictAll* learning approach and across-releases prediction style, and was conducted learning from data chunks that arrive in temporal order. In future work we plan to investigate incorporating both spatial and temporal information about software faults into software fault-proneness prediction.

Finally, **our findings provide software developers with some new insights into achieving accurate fault-proneness predictions**. The results of this study improve the understanding and interpretability of software fault-proneness prediction, especially with regard to how the learning approach affects it. Better fault-proneness prediction provides more accurate information to developers, helping them prioritize development and testing tasks. Ultimately, improved prediction performance results in fewer bugs and better software quality. Consistently with prior works, our results showed that imbalance treatment (here using SMOTE) improved the Recall and Precision, resulting in higher F-Scores for both learning approaches when using the within-release prediction style. Further, we discovered that addressing the imbalance by using SMOTE eliminated the inferior performance of the *usePrePredictPost* approach, which traditionally has had lower Recall (e.g., (Zimmermann et al., 2007; Krishnan et al., 2013)) than models built using the *useAllPredictAll* approach. Reducing the number of post-release false negatives (due to higher Recall) is especially important for safety-critical systems where post-release failures can have catastrophic consequences. In addition, higher Precision leads to less wasted verification and validation efforts on those software units that likely are not fault-prone.

While this paper focuses on classification-based prediction within-projects (i.e., within-release and across-releases), we anticipate that the focus on the choice of the learning approach might yield new insights when exploring across-projects classification-based prediction or prediction of fault-counts in software units for different prediction styles.

9 Conclusion

This paper focuses on one previously unexplored factor of the software fault-proneness prediction process – the learning approach. This captures the temporal aspects regarding (i) the interval for which longitudinal features (software metrics data) are collected, and (ii) the interval for which software bugs are predicted (i.e., the target variable). We first categorized related works by the learning approach each used. This helped us clarify and explain the varying performance in many of these software fault-proneness prediction studies. We then systematically and

[3] The most used technique by far was k-fold cross validation, in 61% of the studies (Falessi et al., 2020). 10-fold cross validation, which was the most used type of k-fold cross validation and the most used technique in general, was used in 51% of the studies.

rigorously explored the impact of the learning approach on the performance of software fault-proneness prediction. Our study showed empirically that the choice of learning approach significantly affects the performance of software fault-proneness prediction models. Our study also produced new insights into how those effects occur. The results presented here showed that the *useAllPredictAll* learning approach resulted in significantly better performance than the *usePrePredictPost* learning approach. Furthermore, we uncovered that the class imbalance is the reason behind this finding for within-release predictions. Addressing the imbalance by using SMOTE significantly enhanced the within-release prediction performance for both learning approaches and eliminated the difference in their performance. Finally, we described the implications of our findings; provided recommendations for designing, reporting, and comparing software fault-proneness prediction studies; and suggested some directions where further work is needed.

Going forward, we encourage fault-proneness prediction studies to always state explicitly which learning approach is being used. We also encourage increased awareness and further study of the consequences of the learning approach choice in practice.

Data Availability Statement

The datasets used in this paper, which were extracted from 64 releases of 12 open source projects, can be requested at https://forms.gle/d7yoUbJx8KZu9A6k6.

Acknowledgement

The authors thank the reviewers and the handling editor for their constructive feedback which helped us to improve the clarity and readability of the paper.

References

Agrawal A, Menzies T (2018) Is "Better Data" Better Than "Better Data Miners"? In: Int. Conf. Softw. Eng., pp 1050–1061, DOI 10.1145/3180155.3180197, 1705.03697

Ahmad MJ (2021) Analysis and classification of software fault-proneness and vulnerabilities. PhD thesis, West Virginia University, URL https://researchrepository.wvu.edu/etd/8323

Ahmad MJ, Goseva-Popstojanova K, Lutz RR (2022) Online supplemental document for the untold impact of learning approaches on software fault-proneness predictions. URL https://tinyurl.com/UntolImpact

Alshehri YA, Goseva-Popstojanova K, Dzielski DG, Devine T (2018) Applying machine learning to predict software fault proneness using change metrics, static code metrics, and a combination of them. In: IEEE SOUTHEASTCON, pp 1–7, DOI 10.1109/SECON.2018.8478911

Amasaki S (2020) Cross-version defect prediction: Use historical data, cross-project data, or both? Empir Softw Eng 25(2):1573–1595, DOI 10.1007/s10664-019-09777-8

Apache Software Foundation (2022a) Apache Projects. URL http://www.apache.org/index.html/projects-list, Last accessed: 2022-06-16

Apache Software Foundation (2022b) Apache Software Foundation (ASF) Bugzilla. URL https://bz.apache.org/bugzilla/, Last accessed: 2022-06-16

Arisholm E, Briand LC, Fuglerud M (2007) Data mining techniques for building fault-proneness models in telecom Java software. In: ISSRE, pp 215–224, DOI 10.1109/ISSRE.2007.16

Arisholm E, Briand LC, Johannessen EB (2010) A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J Syst Softw 83(1):2–17, DOI 10.1016/j.jss.2009.06.055

Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2009) Putting it all together: Using socio-technical networks to predict failures. In: Proc. ISSRE, pp 109–119, DOI 10.1109/ISSRE.2009.17

Bluemke I, Stepień A (2016) Selection of metrics for the defect prediction. In: Adv. Intell. Syst. Comput., Springer, Cham, vol 470, pp 39–50, DOI 10.1007/978-3-319-39639-2-4

Bowes D, Hall T, Petrić J (2018) Software defect prediction: Do different classifiers find the same defects? Softw Qual J 26(2):525–552, DOI 10.1007/s11219-016-9353-3

Catal C (2011) Software fault prediction: A literature review and current trends. Expert Syst Appl 38(4):4626–4636, DOI 10.1016/j.eswa.2010.10.024

Chapman P, Clinton J, Kerber R, Khabaza T, Reinartz TP, Shearer C, Wirth R (2000) CRISP-DM 1.0: Step-by-step data mining guide. Tech. rep.

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: Synthetic minority over-sampling technique. J Artif Intell Res 16:321–357, DOI 10.1613/jair.953, 1106.1813

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: Proc. - Int. Conf. Softw. Eng., pp 31–41, DOI 10.1109/MSR.2010.5463279

Devine T, Goseva-Popstojanova K, Krishnan S, Lutz RR (2016) Assessment and cross-product prediction of software product line quality: Accounting for reuse across products, over multiple releases. Autom Softw Eng 23(2):253–302, DOI 10.1007/s10515-014-0160-4

Devine TR, Goseva-Popstajanova K, Krishnan S, Lutz RR, Li JJ (2012) An empirical study of pre-release software faults in an industrial product line. In: ICST 2012, pp 181–190, DOI 10.1109/ICST.2012.98

Elish KO, Elish MO (2008) Predicting defect-prone software modules using support vector machines. J Syst Softw 81(5):649–660, DOI 10.1016/j.jss.2007.07.040

Falessi D, Huang J, Narayana L, Thai JF, Turhan B (2020) On the need of preserving order of data when validating within-project defect classifiers. Empir Softw Eng 25:4805–4830, DOI https://doi.org/10.1007/s10664-020-09868-x

Fiore A, Russo A, Gravino C, Risi M (2021) Combining CNN with DS3 for detecting bug-prone modules in cross-version projects. In: Proc. - 2021 47th Euromicro Conf. SEAA 2021, pp 91–98, DOI 10.1109/SEAA53835.2021.00021

Galar M, Fernandez A, Barrenechea E, Bustince H, Herrera F (2011) A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42(4):463–484

Gao K, Khoshgoftaar TM, Wang H, Seliya N (2011) Choosing software metrics for defect prediction: An investigation on feature selection techniques. Softw - Pr Exp 41(5):579–606, DOI 10.1002/spe.1043

Ghotra B, McIntosh S, Hassan AE (2015) Revisiting the impact of classification techniques on the performance of defect prediction models. In: Proc. - Int. Conf. Softw. Eng., vol 1, pp 789–800, DOI 10.1109/ICSE.2015.91

Giger E, Pinzger M, Gall HC (2011) Comparing fine-grained source code changes and code churn for bug prediction. In: Proc. - Int. Conf. Softw. Eng., pp 83–92, DOI 10.1145/1985441.1985456

Giger E, D'Ambros M, Pinzger M, Gall HC (2012) Method-level bug prediction. In: Int. Symp. Empir. Softw. Eng. Meas., pp 171–180, DOI 10.1145/2372251.2372285

Gondra I (2008) Applying machine learning to software fault-proneness prediction. J Syst Softw 81(2):186–195, DOI 10.1016/j.jss.2007.05.035

Gong L, Rajbahadur GKK, Hassan AE, Jiang S (2021) Revisiting the impact of dependency network metrics on software defect prediction. IEEE Trans Softw Eng DOI 10.1109/TSE.2021.3131950, 2202.06145

Goseva-Popstojanova K, Ahmad MJ, Alshehri YA (2019) Software fault proneness prediction with Group Lasso regression: On factors that affect classification performance. In: Proc. - Int. Comput. Softw. Appl. Conf., vol 2, pp 336–343, DOI 10.1109/COMPSAC.2019.10229

Goyal S (2022) Handling class-imbalance with KNN (neighbourhood) under-sampling for software defect prediction. Artif Intell Rev 55(3):2023–2064, DOI 10.1007/s10462-021-10044-w

Guo L, Ma Y, Cukic B, Singh H (2004) Robust prediction of fault-proneness by random forests. In: Proc. - ISSRE, pp 417–428, DOI 10.1109/ISSRE.2004.35

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A Systematic Literature Review on Fault Prediction Performance in Software Engineering. IEEE Trans Softw Eng 38(6):1276–1304, DOI 10.1109/TSE.2011.103

Hamill M, Goseva-Popstojanova K (2009) Common trends in software fault and failure data. IEEE Trans Softw Eng 35(4):484–496, DOI 10.1109/TSE.2009.3

He H, Garcia EA (2009) Learning from imbalanced data. IEEE Transactions on knowledge and data engineering 21(9):1263–1284

He P, Li B, Liu X, Chen J, Ma Y (2015) An empirical study on software defect prediction with a simplified metric set. Inf Softw Technol 59:170–190, DOI 10.1016/j.infsof.2014.11.006, 1402.3873

He Z, Peters F, Menzies T, Yang Y (2013) Learning from open-source projects: An empirical study on defect prediction. In: Int. Symp. Empir. Softw. Eng. Meas., pp 45–54, DOI 10.1109/ESEM.2013.20

Hosseini S, Turhan B, Gunarathna D (2019) A systematic literature review and meta-analysis on cross project defect prediction. IEEE Trans Softw Eng 45(2):111–147, DOI 10.1109/TSE.2017.2770124

Jiang Y, Cukic B, Ma Y (2008a) Techniques for evaluating fault prediction models. Empir Softw Eng 13(5):561–595, DOI 10.1007/s10664-008-9079-3

Jiang Y, Cukic B, Menzies T (2008b) Can data transformation help in the detection of fault-prone modules? In: DEFECTS, pp 16–20, DOI 10.1145/1390817.1390822

Jiang Y, Cukic B, Menzies T, Bartlow N (2008c) Comparing design and code metrics for software quality prediction. In: Proc. - Int. Conf. Softw. Eng., ACM, pp 11–18, DOI 10.1145/1370788.1370793

Jira (2022) Issue Project Tracking Software — Atlassian. URL https://www.atlassian.com/software/jira/features/bug-tracking, Last accessed: 2022-06-16

Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proc. 6th Int. Conf. Predict. Model. Softw. Eng., pp 1–10, DOI 10.1145/1868328.1868342

Jureczko M, Spinellis D (2010) Using object-oriented design metrics to predict software defects. Model Methods Syst Dependability Oficyna Wydawnicza Politech Wrocławskiej pp 69–81, DOI 10.1.1.226.2285

Jureczko M, Spinellis D (2011) CKJM extended - An extended version of Tool for Calculating Chidamber and Kemerer Java Metrics (and many other metrics). URL http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

Kabir MA, Keung J, Turhan B, Bennin KE (2021) Inter-release defect prediction with feature selection using temporal chunk-based learning: An empirical study. Appl Soft Comput 113:107870, DOI 10.1016/j.asoc.2021.107870

Kamei Y, Matsumoto S, Monden A, Matsumoto KI, Adams B, Hassan AE (2010) Revisiting common bug prediction findings using effort-aware models. In: IEEE Int. Conf. Softw. Maint., pp 1–10, DOI 10.1109/ICSM.2010.5609530

Khoshgoftaar TM, Seliya N (2004) Comparative assessment of software quality classification techniques: An empirical case study. Empir Softw Eng 9(3):229–257, DOI 10.1023/B:EMSE.0000027781.18360.9b

Khoshgoftaar TM, Gao K, Seliya N (2010) Attribute selection and imbalanced data: Problems in software defect prediction. In: Proc. ICTAI, vol 1, pp 137–144, DOI 10.1109/ICTAI.2010.27

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: Proc. - Int. Conf. Softw. Eng., pp 481–490, DOI 10.1145/1985793.1985859

Koru AG, Liu H (2005) Building effective defect-prediction models in practice. IEEE Softw 22(6):23–29, DOI 10.1109/MS.2005.149

Krishnan S, Strasburg C, Lutz RR, Goseva-Popstojanova K (2011) Are change metrics good predictors for an evolving software product line? In: Proc. 7th Int. Conf. Predict. Model. Softw. Eng., pp 1–10, DOI 10.1145/2020390.2020397

Krishnan S, Strasburg C, Lutz RR, Goseva-Popstojanova K, Dorman KS (2013) Predicting failure-proneness in an evolving software product line. Inf Softw Technol 55(8):1479–1495, DOI 10.1016/j.infsof.2012.11.008

Layman L, Kudrjavets G, Nagappan N (2008) Iterative identification of fault-prone binaries using inprocess metrics. In: Proc. Empir. Softw. Eng. Meas., pp 206–212, DOI 10.1145/1414004.1414038

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans Softw Eng 34(4):485–496, DOI 10.1109/TSE.2008.35

Li J, He P, Zhu J, Lyu MR (2017) Software defect prediction via convolutional neural network. In: Proc. - 2017 IEEE Int. Conf. Softw. Qual. Reliab. Secur. QRS, pp 318–328, DOI 10.1109/QRS.2017.42

Ma Y, Luo G, Zeng X, Chen A (2012) Transfer learning for cross-company software defect prediction. Inf Softw Technol 54(3):248–256, DOI 10.1016/j.infsof.2011.09.007

Madeyski L, Jureczko M (2015) Which process metrics can significantly improve defect prediction models? An empirical study. Softw Qual J 23(3):393–422, DOI 10.1007/s11219-014-9241-7

Mahmood Z, Bowes D, Hall T, Lane PCR, Petrić J (2018) Reproducibility and replicability of software defect prediction studies. Inf Softw Technol 99:148–163, DOI 10.1016/j.infsof.2018.02.003

Malhotra R, Jain J (2020) Handling imbalanced data using ensemble learning in software defect prediction. In: Proc. Conflu. 2020 - 10th Int. Conf. Cloud Comput. Data Sci. Eng., pp 300–304, DOI 10.1109/Confluence47617.2020.9058124

Malhotra R, Raje R (2015) An empirical comparison of machine learning techniques for software defect prediction. In: Proc. Int. Conf. Bio-inspired Inf. Commun. Technol., pp 320–327, DOI 10.4108/icst.bict.2014.257871

Mende T, Koschke R (2009) Revisiting the evaluation of defect prediction models. In: Proc. 5th Int. Conf. Predict. Model. Softw. Eng., pp 1–10, DOI 10.1145/1540438.1540448

Menzies T, Greenwald J, Frank A (2007) Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng 33(1):2–13, DOI 10.1109/TSE.2007.256941

Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A (2010) Defect prediction from static code features: Current results, limitations, new approaches. Autom Softw Eng 17(4):375–407, DOI 10.1007/s10515-010-0069-5

Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proc. - Int. Conf. Softw. Eng., pp 181–190, DOI 10.1145/1368088.1368114

Nagappan N, Ball T, Murphy B (2006) Using historical in-process and product metrics for early estimation of software failures. In: Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE, pp 62–71, DOI 10.1109/ISSRE.2006.50

Nagappan N, Murphy B, Basili VR (2008) The influence of organizational structure on software quality: An empirical case study. In: Proc. - Int. Conf. Softw. Eng., pp 521–530, DOI 10.1145/1368088.1368160

Nagappan N, Zeller A, Zimmermann T, Herzig K, Murphy B (2010) Change bursts as defect predictors. In: Proc. ISSRE, pp 309–318, DOI 10.1109/ISSRE.2010.25

Nam J, Pan SJ, Kim S (2013) Transfer defect learning. In: Proc. - Int. Conf. Softw. Eng., IEEE Press, pp 382–391, DOI 10.1109/ICSE.2013.6606584

Nam J, Fu W, Kim S, Menzies T, Tan L (2018) Heterogeneous defect prediction. IEEE Trans Softw Eng 44(9):874–896, DOI 10.1109/TSE.2017.2720603

NIST (2023) Artificial Intelligence Risk Management Framework (AI RMF 1.0). Tech. rep., URL https://doi.org/10.6028/NIST.AI.100-1, [Online; accessed 10-July-2023]

OECD (2022) OECD Framework for the Classification of AI systems. Tech. rep., OECD Digital Economy Papers, No. 323, OECD Publishing, Paris, URL https://doi.org/10.1787/cb6d9eca-en, [Online; accessed 10-July-2023]

Okutan A, Yıldız OT (2014) Software defect prediction using Bayesian networks. Empir Softw Eng 19(1):154–181, DOI 10.1007/s10664-012-9218-8

Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31(4):340–355, DOI 10.1109/TSE.2005.49

Pang Y, Xue X, Wang H (2017) Predicting vulnerable software components through deep neural network. In: Proc. - Int. Conf. Softw. Qual. Reliab. Secur, pp 6–10, DOI 10.1145/3094243.3094245

Pinzger M, Nagappan N, Murphy B (2008) Can developer-module networks predict failures? In: Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng., ACM, pp 2–12

Sayyad S, Menzies T (2005) The PROMISE repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa, Canada, URL http://promise.site.uottawa.ca/SERepository

Schapire RE (2013) Explaining AdaBoost, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 37–52

Shepperd M, Bowes D, Hall T (2014) Researcher bias: The use of machine learning in software defect prediction. IEEE Trans Softw Eng 40(6):603–616, DOI 10.1109/TSE.2014.2322358

Shepperd M, Hall T, Bowes D (2018) Authors' reply to 'Comments on 'Researcher bias: The use of machine learning in software defect prediction". IEEE Trans Softw Eng 44(11):1129–1131, DOI 10.1109/TSE.2017.2731308

Song Q, Jia Z, Shepperd M, Ying S, Liu J (2011) A general software defect-proneness prediction framework. IEEE Trans Softw Eng 37(3):356–370, DOI 10.1109/TSE.2010.90

Song Q, Guo Y, Shepperd M (2019) A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Trans Softw Eng 45(12):1253–1269, DOI 10.1109/TSE.2018.2836442

Stavropoulos A, Caroni C (2008) Rank test statistics for unbalanced nested designs. Stat Methodol 5(2):93–105, DOI 10.1016/j.stamet.2007.06.001

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) Comments on "Researcher bias: The use of machine learning in software defect prediction". IEEE Trans Softw Eng 42(11):1092–1094, DOI 10.1109/TSE.2016.2553030

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2017) An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. IEEE Trans Softw Eng 43(1):1–18, DOI 10.1109/TSE.2016.2584050

Tosun A, Bener A, Turhan B, Menzies T (2010) Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. In: Inf. Softw. Technol., vol 52, pp 1242–1257, DOI 10.1016/j.infsof.2010.06.006

Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empir Softw Eng 14(5):540–578, DOI 10.1007/s10664-008-9103-7

Wang S, Yao X (2013) Using class imbalance learning for software defect prediction. IEEE Trans Reliab 62(2):434–443, DOI 10.1109/TR.2013.2259203

Wang S, Liu T, Tan L (2016) Automatically learning semantic features for defect prediction. In: Proc. - Int. Conf. Softw. Eng., pp 297–308, DOI 10.1145/2884781.2884804

Watanabe S, Kaiya H, Kaijiri K (2008) Adapting a fault prediction model to allow inter language reuse. In: Proc. Int. Conf. Softw. Eng., pp 19–24, DOI 10.1145/1370788.1370794

Wirth R, Hipp J (2000) CRISP-DM: Towards a standard process model for data mining. In: Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining, Manchester, vol 1, pp 29–39

Wu R, Zhang H, Kim S, Cheung SC (2011) ReLink: Recovering links between bugs and changes. In: SIGSOFT/FSE, pp 15–25, DOI 10.1145/2025113.2025120

Xu Z, Li S, Tang Y, Luo X, Zhang T, Liu J, Xu J (2018) Cross version defect prediction with representative data via sparse subset selection. In: Proc. - Int. Conf. Softw. Eng., pp 132–143, DOI 10.1145/3196321.3196331

Xu Z, Li S, Luo X, Liu J, Zhang T, Tang Y, Xu J, Yuan P, Keung J (2019) TSTSS: A two-stage training subset selection framework for cross version defect prediction. J Syst Softw 154:59–78, DOI 10.1016/j.jss.2019.03.027

Zahalka A, Goseva-Popstojanova K, Zemerick J (2010) Empirical evaluation of factors affecting distinction between failing and passing executions. In: ISSRE, pp 259–268, DOI 10.1109/ISSRE.2010.44

Zhao K, Xu Z, Yan M, Tang Y, Fan M, Catolino G (2021) Just-in-time defect prediction for Android apps via imbalanced deep learning model. In: Proc. ACM Symp. Appl. Comput., vol 1, pp 1447–1454, DOI 10.1145/3412841.3442019

Zhou T, Sun X, Xia X, Li B, Chen X (2019) Improving defect prediction with deep forest. Inf Softw Technol 114:204–216, DOI 10.1016/j.infsof.2019.07.003

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: PROMISE'07, pp 9–9, DOI 10.1109/PROMISE.2007.10

Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In: ESEC-FSE'09, pp 91–100, DOI 10.1145/1595696.1595713