Software Quality Journal (2025) 33:7
https://doi.org/10.1007/s11219-024-09704-1

RESEARCH

Domain-specific implications of error-type metrics in risk-based software fault prediction

Khoa Phung[1] · **Emmanuel Ogunshile**[1] · **Mehmet E. Aydin**[1]

Accepted: 27 November 2024

Abstract

In software development, Software Fault Prediction (SFP) is essential for optimising resource allocation and improving testing efficiency. Traditional SFP methods typically use binary-class models, which can provide a limited perspective on the varying risk levels associated with individual software modules. This study explores the impacts of Error-type Metrics on the fault-proneness of software modules in domain-specific software projects. Also, it aims to enhance SFP methods by introducing a risk-based approach using Error-type Metrics. This method categorises software modules into High, Medium, and Low-Risk categories, offering a more granular and informative fault prediction framework. This approach aims to refine the fault prediction process and contribute to more effective resource allocation and project management in software development. We explore the domain-specific impact of Error-type Metrics through Principal Component Analysis (PCA), aiming to fill a gap in the existing literature by offering insights into how these metrics affect machine learning models across different software domains. We employ three machine learning models - Support Vector Machine (SVM), Random Forest (RF), and Extreme Gradient Boosting (XGB) - to test our approach. The Synthetic Minority Over-sampling Technique (SMOTE) is used to address class imbalance. Our methodology is validated on fault data from four open-source software projects, aiming to confirm the robustness and generalisability of our approach. The PCA findings provide evidence of the varied impacts of Error-type Metrics in different software environments. Comparative analysis indicates a strong performance by the XGB model, achieving an accuracy of 97.4%, a Matthews Correlation Coefficient of 96.1%, and an F1-score of 97.4% across the datasets. These results suggest the potential of the proposed method to contribute to software testing and quality assurance practices. Our risk-based SFP approach introduces a new perspective to risk assessment in software development. The study's findings contribute insights into the domain-specific applicability of Error-type Metrics, expanding their potential utility in SFP. Future research directions include refining our fault-counting methodology and exploring broader applications of Error-type Metrics and our proposed risk-based approach.

✉ Khoa Phung
khoa.phung@uwe.ac.uk

Emmanuel Ogunshile
emmanuel.ogunshile@uwe.ac.uk

Mehmet E. Aydin
mehmet.aydin@uwe.ac.uk

[1] University of the West of England, Coldharbour Ln, Bristol BS16 1QY, Stoke Gifford, UK

Published online: 07 January 2025

Keywords Software fault prediction · Error-type metrics · Risk categorisation · Software quality assurance · Principal component analysis · Extreme gradient boosting model · Domain-specific analysis

1 Introduction

The foundation for attaining high-quality software is effective software quality assurance (SQA). This includes activities such as encompassing formal code inspections, code walk-throughs, software testing, validation, and verification. These activities together ensure the aspired software quality is cost-effectively attained by supervising and steering the Software Development Life Cycle (SDLC) (Rathore & Kumar, 2019).

Despite this, the objective of thoroughly testing a software system is practically impossible due to the substantial time and resources demanded (Rathore & Kumar, 2016; Hierons et al., 2009). This problem is underscored by the irregular distribution of faults among software modules, leading to inefficiencies when testing resources and efforts are evenly expended on all modules of the system under test (SUT).

To counter this issue, software fault prediction (SFP) has been suggested as a feasible solution. SFP aims to optimise the allocation of limited SQA resources by predicting the fault-proneness of software modules or classes. For instance, if only a quarter of resources are accessible, understanding the most susceptible areas enables testers and developers to prioritise these resources, concentrating on modules or classes more likely to exhibit faults. This allows for the creation of robust software within the bounds of constrained time and budget. Over the past few decades, SFP techniques enabling early detection of faulty software modules during the SDLC have attracted increasing interest from researchers and software developers alike.

SFP research employs a wide range of statistical and machine learning techniques such as Logistic Regression, Naïve Bayes, Multilayer Perceptron, Decision Tree, Support Vector Machine, etc., to predict the fault-proneness of software modules (Rathore & Kumar, 2016; Malhotra, 2015). The data employed to train these SFP models, comprising software metrics and fault data, are typically derived from similar projects or earlier versions of the same project. These models are then applied to the modules or classes of a specified SUT to classify them as either fault-prone or not. The abundance of available software metrics (Bundschuh & Dekkers, 2008; Al Dallal, 2013; Tahir & MacDonell, 2012; Nagappan et al., 2010; Jiang et al., 2008; Premraj & Herzig, 2011) and open-source data repositories, such as NASA (Petrić et al., 2016) and PROMISE (Shirabad et al., 2005), has catalysed numerous explorations into SFP. However, despite recent syntheses of SFP, providing details on its achievements and current trends, a comprehensive evaluation of various SFP studies and a coherent understanding of the merits and demerits of existing SFP methods remain elusive (Kamei & Shihab, 2016).

In addition, the concept of software fault proneness presents ambiguity. Faults can appear at any stage of the SDLC, and some may remain undetected during testing, only to emerge after release (Rathore & Kumar, 2019). Many SFP methodologies are reliant on binary-class classification, predicting whether a software module is fault-prone or not (Rathore & Kumar, 2019, 2016). However, this binary approach grossly simplifies the complexity of fault prediction. It neglects the fact that some modules are indeed more prone to faults and demand more attention than others. A more granular representation of faultiness in software modules, such as the number of faults in a module, fault-wise ranking of modules, and severity of a fault, would offer greater value to software testers or analysts (Rathore & Kumar, 2019; Menzies et al., 2008).

It has been observed that the performance of techniques and methods used for SFP has plateaued. Merely employing different or seemingly superior techniques does not assure enhanced performance (Rathore & Kumar, 2019; Menzies et al., 2008). To achieve better prediction outcomes, additional information needs to be incorporated into the construction of SFP models (Menzies et al., 2011), and innovative approaches to SFP should be contemplated. To this end, we have previously introduced Error-type Metrics (Phung et al., 2021, 2023), which effectively capture the patterns of runtime errors observed in Java programs. However, it is crucial to fully understand the nature of these new metrics and their interactions. If these metrics exhibit high inter-correlations (multicollinearity), their predictive power could be compromised, leading to unstable and potentially inaccurate SFP models.

In light of the above, this paper makes several key contributions:

1. *Enhanced Risk Categorisation Framework:* Building on our previously introduced Error-type Metrics (Phung et al., 2021, 2023), we develop a comprehensive risk categorisation framework that surpasses traditional binary-class models in SFP. This framework classifies risks into Low, Medium, and High categories, significantly facilitating the decision-making process for high-level management. By abstracting the intricacies of software faults into these categories, our approach aims to streamline risk monitoring and aligns with project management principles to improve project management effectiveness.
2. *Domain-specific Evaluation of Error-type Metrics:* Our investigation extends into the domain-specific impacts of Error-type Metrics across diverse software projects, addressing a gap highlighted by Rathore and Kumar (2019) regarding the contextual evaluation of software metrics. We provide empirical analysis for the impacts of these metrics on specific open-source software projects.
3. *Addressing Multicollinearity in Error-type Metrics:* We examine the multicollinearity among Error-type Metrics before applying them to train SFP models. This investigation, which was overlooked in our previous study, ensures the reliability and validity of the Error-type Metrics employed in this study, strengthening the foundation for their use in SFP.

Given these contributions, our first research question (RQ1) is: *How do Error-type Metrics impact the fault-proneness of software modules across different domains, and do these metrics exhibit multicollinearity issues?* - By posing this question, we aim to provide a comprehensive understanding of the nature and applicability of Error-type Metrics. Specifically, RQ1 explores whether the impact of Error-type Metrics is consistent across different types of software or if their relevance is domain-specific. Furthermore, RQ1 aims to identify the potential multicollinearity issues among these metrics, thereby validating their use in SFP models.

Our second research question (RQ2) is: *Can the integration of Error-type Metrics enhance the prediction and categorisation of software modules into risk levels (Low Risk, Medium Risk, High Risk) with respect to the number of faults?* - This question aims to investigate the potential benefit of combining Error-type Metrics with other software metrics in risk assessment. The investigation in RQ1 lays the groundwork for RQ2 by assuring that the integration of Error-type Metrics will not lead to problematic multicollinearity issues.

The remainder of this paper is structured as follows. Section 2 provides an overview of pertinent literature. Section 3 delineates the methodology for deriving Error-type Metrics. Our proposed approach is articulated in Section 4. The findings of our experiments and the comparison of our method against the current state-of-the-art are discussed in Section

5. Finally, the paper concludes in Section 6 with reflections on the findings and potential directions for future research.

2 Related work

In this section, we conduct a comprehensive review of associated work in three distinct categories: software metrics, data quality issues, and model construction approaches.

2.1 Software metrics

Software metrics, understood as a quantitative assessment of software product characteristics, are crucial in evaluating the quality of software products (Fenton & Bieman, 2015). Each metric is directly linked with specific functional attributes like coupling, cohesion, and inheritance, and aids in assessing crucial external quality factors such as reliability, testability, and fault-proneness (Bansiya & Davis, 2002).

There are various software metrics in the literature, notably Object-Oriented (OO) metrics, including the Chidamber and Kemerer (CK) metrics suite (Chidamber & Kemerer, 1994), MOODS metrics suite (Harrison et al., 1998), and the Bansiya metrics suite (Bansiya & Davis, 2002). On the other hand, Traditional metrics offer a different perspective, including Size metrics like Function Points (FP), Source Lines of Code (SLOC), and Quality metrics like Defects per FP after delivery. Additionally, System Complexity metrics (McCabe, 1976) and Halstead metrics (Halstead, 1977) provide insights into the structural complexity and computational capabilities.

Some observations based on the review of the literature include: the performance of software metrics can vary significantly depending on the context; several OO metrics like Coupling Between Objects (CBO), Response for a Class (RFC), and Weighted Method Count (WMC) are effective in identifying faults; and a positive correlation exists between size metrics and fault-proneness, implying larger software systems may be more fault-prone (Rathore & Kumar, 2019).

Our prior work introduced a new approach using a formal method, namely Stream X-Machine (Dranidis et al., 2012), to extract *Error Specification Machine* (ESM) values from the source code (Phung et al., 2021). The ESM values were used to create *Error-type software metrics*, a novel set of metrics that improved the performance of machine learning models in predicting fault-proneness (Phung et al., 2023). Error-type Metrics, derived from error-type models, are applicable across any domain, thus potentially addressing the challenge of choosing the right combination of metrics for different application domains. A detailed discussion of Error-type Metrics is presented in Section 3.

2.2 Data quality issues

The success of SFP models is significantly influenced by the quality of datasets. While public repositories such as NASA and PROMISE are conveniently accessible, they may harbour erroneous or superfluous information that can compromise the effectiveness of classifiers (Shepperd et al., 2013; Petrić et al., 2016). Despite this, many studies often assume the adequacy of these datasets, neglecting potential data quality issues (Rathore & Kumar, 2019; Bhandari et al., 2022).

One major issue is *high-dimensionality*, where datasets have an excess of features. This can degrade classification accuracy and increase computational costs (Bhandari et al., 2022). Dimensionality reduction methods such as feature selection (e.g., Chi-square, Information Gain, and Principal Component Analysis (PCA) Wold et al., 1987) and feature extraction are solutions to this issue (Malhotra, 2015), reducing the feature set or creating a new set of relevant features by combining existing ones, respectively.

Class imbalance is another challenge, where instances of a "minor class" are outnumbered by the "major class". This can bias learning algorithms towards the major class, compromising minor class prediction performance (Moreno-Torres et al., 2012; Song et al., 2018). Techniques like resampling methods (e.g., Random Under-Sampling - RUS, Synthetic Minority Over-sampling Technique - SMOTE), which alter distribution by oversampling the minority class or undersampling the majority class, can ameliorate this (Wang & Yao, 2013).

Outliers, data points diverging significantly from the general pattern, also pose a quality issue. In the context of SFP, outliers might represent faulty modules, so they should not be arbitrarily eliminated (Li et al., 2018).

Lesser-known issues, such as missing data, repeated values, and redundancy, can also negatively affect classifier performance (Bhandari et al., 2022). Other problems such as high-class overlap can reduce model efficacy (Gupta & Gupta, 2017). The simultaneous occurrence of these issues can greatly impair prediction performance. This necessitates careful data preprocessing, which should be tailored according to the unique characteristics of the dataset under study. One objective of this research is to carefully address these data quality issues before training the SFP models.

2.3 Model construction approaches

SFP models range from binary-class classification to the prediction of fault density or severity. Binary-class classification models have received the most attention in previous studies (Rathore & Kumar, 2019; Malhotra, 2015; Alsolai & Roper, 2020; Kumar & Bansal, 2019). However, to the best of our knowledge, there is limited work focused on predicting fault density, severity, or error-type proneness, which is essential for a more comprehensive understanding of fault proneness in software modules.

In 2005, Ostrand et al. (2005) showcased the effectiveness of Negative Binomial Regression (NBR) in predicting the number and density of faults in software files, using their fault and modification history. The approach was able to accurately identify 20% of the files with the highest predicted number of faults. A similar study by Yu (2012) found that while NBR was not superior to Binary Logistic Regression in predicting fault-prone modules, it was efficient in predicting multiple faults within a single module.

Genetic programming (GP) has also demonstrated significant accuracy in fault count prediction, as shown in open-source projects by Afzal et al. (2008) and Rathore and Kumar (2015). In addition, a comparison of various count models, including the Poisson Regression model (PR), Zero-Inflated Poisson model (ZIP), Negative Binomial Regression model (NBR), Zero-Inflated Negative Binomial model (ZINB), and Hurdle Regression model (HR), highlighted the superior predictive accuracy of ZINB and HR models in predicting fault counts (Gao & Khoshgoftaar, 2007).

Rathore and Kumar (2016) explored the potential of Decision Tree Regression (DTR) for predicting fault count in both intra- and inter-release contexts in open-source projects, with the DTR-based model showing considerable accuracy. On the other hand, Yang et al. (2014) suggested that predicting the exact number of faults is challenging due to noisy data in fault

datasets. They introduced a learning-to-rank (LTR) approach for constructing SFP models, optimising the ranking performance directly. This method proved to be more robust against noisy data and could rank the severity level of software modules directly.

Our previous work (Phung et al., 2021) proposed an innovative SFP approach, merging Stream X-Machine and machine learning techniques to predict whether software modules are susceptible to specific types of runtime errors in Java programs. However, the study faced limitations, such as the smaller dataset size and the potential for unintentional mistakes in the manual extraction of ESM values.

This research distinguishes itself from existing literature and our previous works in several ways.

Firstly, this study formalises the derivation process of each Error-type Metric directly from the source code, providing a detailed mathematical formalism that was previously lacking in our prior work (Phung et al., 2023).

Secondly, our approach uses traditional machine learning techniques, incorporating Error-type Metrics to predict software modules' severity levels based on fault numbers. Despite deep learning's promise (Pandey & Tripathi, 2020; Deng et al., 2020; Qiao et al., 2020; Pandey & Tripathi, 2021; Wang et al., 2021), its drawbacks - such as the need for large datasets, risk of overfitting on small datasets, and high computational costs (Pandey et al., 2023) - make it less suitable for our specific study context. Our research, as outlined in RQ1, assesses the impact of Error-type Metrics on fault-proneness in domain-specific open-source projects. Given deep learning's challenges with performance and generalisability across diverse projects (Pandey et al., 2023), we opt for traditional models including Support Vector Machine, Random Forest, and Extreme Gradient Boosting. These models align with our proposed risk categorisation SFP framework, designed to help high-level managers allocate testing resources efficiently. It highlights that the balance between informed decision-making and the high computational demand of deep learning models does not justify their use in this scenario.

Thirdly, we diverge from the trend of relying on public datasets such as NASA and PROMISE repositories. Instead, we utilise software metrics extracted from actively maintained open-source projects provided by the BugHunter Dataset (Ferenc et al., 2020). The primary reason for not using these public datasets is the absence of source code. Access to source code is essential for extracting Error-type Metrics directly from the codebase, as these metrics require detailed analysis of the source code to capture error-specific characteristics; without source code access, these metrics cannot be derived. Additionally, previous research has substantiated concerns about the quality and reliability of the NASA and PROMISE datasets. Studies by Shepperd et al. (2013) and Petrić et al. (2016) have highlighted issues such as inconsistent data, missing values, and potential errors within these datasets, which can adversely affect the performance and validity of predictive models.

A distinguishing feature of the BugHunter Dataset is that it records both the faulty and fixed states of the same source code, irrespective of the release versions, rather than merely collating characteristics of source code elements at selected release versions. This approach is beneficial for tracking modifications in software metrics during bug-fixing activities. The detailed process of deriving software metrics for the BugHunter Dataset is described in (Ferenc et al., 2020).

Lastly, we address the critical issues of data quality in SFP. As detailed in Subsection 2.2, various data quality issues can significantly compromise the effectiveness of SFP models. In this research, we conduct a comprehensive evaluation of these issues, proposing targeted strategies to effectively mitigate their impacts.

3 Error-type metrics

Building upon our preceding work (Phung et al., 2021, 2023), where we developed a representation model for runtime errors, this paper extends these concepts with new theoretical contributions and practical applications. Previously, we envisioned each error as an operation performed by one object onto another, mathematically formalised as follows:

$$A\ operates\ on\ B \tag{1}$$

In this formula, A and B denote the operands and could either be literals or references. The term "operates on" signifies an action that operand A enacts on operand B. For instance, Number A divided by Number B leads to a potential Arithmetic Exception.

Leveraging the information about A, B, and the "operates on" aspect extracted from (1), we can dissect the characteristics of a particular error. These characteristics allow us to elucidate what the error is, the mechanism of its occurrence, and the specific context within which it emerges. Furthering this notion, we utilise (1) to devise a Stream X-Machine (SXM) representation that characterises a particular type of runtime error. This SXM representation, alternatively known as an Error Specification Machine (ESM), symbolises each type of runtime error. An ESM is an octet tuple, comprised of the following elements:

$$ESM_i = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0) \tag{2}$$

Where:

- Σ is a finite set of input symbols,
- Γ is a finite set of output symbols,
- Q is a finite set of states,
- M is a (possibly) infinite set called memory,
- Φ is a finite set of partial functions (processing functions), which map memory-input pair to output-memory pairs, $\phi : M \times \Sigma \rightarrow \Gamma \times M$,
- F is the next-state partial function, $F : Q \times \Phi \rightarrow Q$,
- $q_0 \in Q$ and $m_0 \in M$ are the initial state and memory, respectively,
- $i \in E$, where E is a finite set of different types of Java Runtime Error (JRE), represented as

$$\begin{aligned} E = \{&\text{Arithmetic},\\ &\text{Null Pointer},\\ &\text{Class Cast},\\ &\text{Index Out Of Bounds}, \ldots\} \end{aligned}$$

An SXM is essentially a finite automaton with arcs labelled by functions that correspond to the types. In our previous work (Phung et al., 2021), we presented a state-transition diagram of the ESM. The associated Finite Automaton (FA) of a Stream X-Machine is denoted as $A_Z = (\Phi, Q, F, I, T)$ and each ESM_i has a corresponding FA, $A_{ESM_i} = (\Phi, Q, F, I, T)$.

Given that each ESM is a Stream X-Machine specification, the associated test cases for each ESM can be generated utilising the state-counting method, a Stream X-Machine testing approach (Ipate & Dranidis, 2016; Ipate, 2006). As a result, each type of runtime error will have a distinct set of test suites, computed as per (3) (Ipate, 2006):

$$U_i = \bigcup_{q \in Q_r} \{p_q\} prefix(V(q)) W_s \tag{3}$$

Where:

- S_r is a non-empty set of realisable sequences such that no state in ESM_i is reached by more than one sequence in S_r,
- p_a is a path in A_{ESM_i} where $p_a = \phi_1 \cdots \phi_k \in \Phi^*$,
- The definition of the set $V(q)$ can be found in (Ipate, 2006),
- W_s is a finite set that separates between separable states of ESM_i. W_s is required to be non-empty,
- $d_s : Q \leftrightarrow Q$ is a relation on the states of ESM_i that satisfies the following conditions: for every two states $q_1, q_2 \in Q$, if $(q_1, q_2 \in d_s)$ then q_1 and q_2 are separated by W_s. The relation d_s identifies pairs of states that are known to be separated by W_s,
- The maximal set $Q_1 \cdots Q_j$ of states of ESM_i that are known to be pairwise separated by W_s,
- $i \in E$, where E represents different types of JRE, as defined in (2),
- U_i is a set of test cases associated with each type of JRE. For instance, $ArithmeticException$ can be represented as $ESM\ Arithmetic$, which can subsequently generate test cases for this runtime error.

Furthering our discussion on the characteristics of the errors and their analysis, the current study introduces a Lemma and its Proof which outline the relationship between the error model and code patterns. These new additions aim to provide a comprehensive understanding that was not fully explored in our previous work.

3.1 Formal derivation of error-type metrics from source code

Lemma 1 *In a given software module, the ESM value for a specific JRE matches the cumulative sum of all code patterns that:*

1. *Align with the pattern illustrated in* (1)*, and*
2. *Align with one or more test cases corresponding to the specific error, which are generated as per the pattern outlined in* (3)*.*

Proof Assume $code\_pattern$ as a function that receives as input a line of code in a software module and returns true if it aligns with the pattern illustrated in (1) and aligns with one or more test cases corresponding to the specific error, generated as per the pattern outlined in (3). Otherwise, it returns false.

For a given software module m with L lines of code, let i signify the i^{th} line of code. Let ESM_value be a function that gives us the ESM value of a line of code.

The ESM value of a software module can be computed by summing up the ESM values of each line of code. This is expressed as follows:

$$ESM\_value(m) = \sum_{i=1}^{L} ESM\_value(i)$$

Assuming that $ESM\_value(i)$ equals 1 if $code\_pattern(i)$ is true and 0 otherwise (as the ESM value of a line of code increases by 1 only if the code pattern aligns with the two conditions), we have:

$$ESM\_value(i) = \begin{cases} 1, & \text{if } code\_pattern(i) \text{ is true} \\ 0, & \text{otherwise} \end{cases}$$

Thus, the ESM value of the software module m is the sum of all the ESM values of the lines of code for which $code\_pattern(i)$ is true, which is the cumulative sum of all code

patterns that align with the two conditions:

$$ESM\_value(m) = \sum_{i=1}^{L} ESM\_value(i)$$
$$= \sum_{i=1}^{L} \begin{cases} 1, & \text{if } code\_pattern(i) \text{ is true} \\ 0, & \text{otherwise} \end{cases}$$

Consequently, the ESM values corresponding to various types of JRE within software modules of a software system can be represented mathematically as follows:

$$ESM\_values = \bigcup_{m \in S} \sum_{i=1}^{L} \sum_{j=1}^{K} matched\_pattern \tag{4}$$

Where:

1. m refers to an individual software module within the software system, denoted by S,
2. i signifies the i^{th} line among the total L lines of code in the software module m,
3. j represents the j^{th} code pattern among the total K code patterns present in the i^{th} line,
4. $matched\_pattern$ denotes the pattern within the i^{th} line of code that fulfils the aforementioned two conditions.

This Lemma and its Proof are significant in that they provide a methodical approach to quantifying the impact of specific code patterns on JREs. They offer a mathematical basis for understanding how different code constructs contribute to software errors, which is crucial for the development of more effective SFP models.

3.2 Practical application and universal applicability

To illustrate the practical application of these concepts, consider a scenario where a software module contains multiple lines of code, each potentially contributing to different types of JREs. Using the framework established by the Error-type Metrics, we can systematically analyse each line of code to determine its contribution to the overall fault proneness of the module.

The intrinsic generic nature of the Error-type Metrics, derived from error-type models, makes them universally applicable across various software domains. This universality is a significant advancement over traditional metrics such as LOC, SLOC, KSLOC, and CK metrics, which often show variable efficacy in different application domains. For instance, in a project predominantly dealing with database operations, Error-type Metrics can effectively capture specific error patterns related to database connectivity and query execution, which might be overlooked by conventional metrics.

4 Methodology

This section outlines our proposed methodology, encompassing a thorough discussion of the software fault datasets under investigation, the strategy for data labelling and our novel risk categorisation approach, the selection of evaluation measures, the necessary data preprocessing techniques used to prepare the datasets and the selection of machine learning models.

Table 1 The selected projects and their descriptions

Project Name	No. Software Modules	kLOC	Description
ANTLR v4	469	68	A popular software in language processing. It is a powerful parser generator for reading, processing, executing, or translating structured text or binary files.
JUnit	351	43	A Java framework for writing and designing unit tests.
OrientDB	5,514	621	A popular document-based NoSQL graph database. Mainly famous for its speed and scalability.
Elastic Search	18,324	995	A popular RESTful search engine.

4.1 Software fault datasets

In this study, fault datasets are selected from four open-source Java software projects whose software metrics are obtained from the BugHunter Dataset (Ferenc et al., 2020). These projects are not only under active maintenance but also represent four distinct domains: ANTLR v4 operates in the field of language processing; JUnit in software testing; OrientDB in databases; and Elastic Search in search engines. The rationale for this diverse selection relates to RQ1, which investigates the impact of Error-type Metrics across different types of software projects and domains. By choosing projects from different domains, we aim to explore whether the Error-type Metrics hold consistently across various types of software, or if their impact is domain-specific. The specifics of these selected projects are further elaborated in Table 1.

For example, our analysis, provided and discussed in Subsection 5.1, indicates that while some Error-type Metrics such as Index Out Of Bounds and Null Pointer are generally impactful across all domains, the Class Cast metric shows a greater impact in projects related to testing frameworks, such as JUnit. This observation suggests that the project's domain can influence which Error-type Metrics are most relevant, affecting their usefulness in fault prediction models.

To ensure a clear understanding of the datasets used in this study, we provide a detailed description of their structure, common across the four selected projects. Each dataset includes a set of `Independent Variables` that consists of four Error-type Metrics and conventional software metrics.

- The Error-type Metrics correspond with four JREs including Index Out Of Bounds (ESM IndexOutOfBounds), Null Pointer (ESM NullPointer), Class Cast (ESM ClassCast), and Arithmetic (ESM Arithmetic).
- The conventional software metrics are measured at the *class level* and can be found in Table 2. These metrics are extracted from the BugHunter Dataset and re-validated using Metrics Reloaded (Ardito et al., 2020). They provide a comprehensive overview of software module characteristics such as size, complexity, and object-oriented metrics.

The `Dependent Variable` across all datasets is the `Number of Faults`, which quantifies the actual faults found in each software module.

The datasets for the four projects ANTLR v4, JUnit, OrientDB, and Elastic Search are available as CSV files on GitHub [1]. To illustrate the composition of each dataset, Tables 3, 4,

[1] https://github.com/dangkhoa0303/Error-type-Metrics-Datasets.git

Table 2 Class-level software metric used in this study

Abbreviation	Full name
CLOC	Comment Lines of Code
LOC	Lines of Code
LLOC	Logical Lines of Code
NL	Nesting Level
NLE	Nesting Level Else-If
NII	Number of Incoming Invocations
NOI	Number of Outgoing Invocations
CD	Comment Density
DLOC	Documentation Lines of Code
TCD	Total Comment Density
TCLOC	Total Comment Lines of Code
NOS	Number of Statements
TLOC	Total Lines of Code
TLLOC	Total Logical Lines of Code
TNOS	Total Number of Statements
PDA	Public Documented API
PUA	Public Undocumented API
LCOM5	Lack of Cohesion in Methods 5
WMC	Weighted Methods per Class
CBO	Coupling Between Object classes
CBOI	Coupling Between Object classes Inverse
RFC	Response set For Class
AD	API Documentation
DIT	Depth of Inheritance Tree
NOA	Number of Ancestors
NOC	Number of Children
NOD	Number of Descendants
NOP	Number of Parents
NA	Number of Attributes
NG	Number of Getters
NLA	Number of Local Attributes
NLG	Number of Local Getters
NLM	Number of Local Methods
NLPA	Number of Local Public Attributes
NLPM	Number of Local Public Methods
NLS	Number of Local Setters
NM	Number of Methods
NPA	Number of Public Attributes
NPM	Number of Public Methods
NS	Number of Setters
TNA	Total Number of Attributes

Table 2 continued

Abbreviation	Full name
TNG	Total Number of Getters
TNLA	Total Number of Local Attributes
TNLG	Total Number of Local Getters
TNLM	Total Number of Local Methods
TNLPA	Total Number of Local Public Attributes
TNLPM	Total Number of Local Public Methods
TNLS	Total Number of Local Setters
TNM	Total Number of Methods
TNPA	Total Number of Public Attributes
TNPM	Total Number of Public Methods
TNS	Total Number of Setters

5, and 6 depict examples from each project, highlighting the Error-type Metrics alongside the other software metrics and the corresponding `Number of Faults`. For instance, in the dataset corresponding to the JUnit project, each module's record includes the ESM values - quantifying specific exception occurrences within the software module - and additional software metrics such as Lines of Code (LOC), Lack of Cohesion in Methods 5 (LCOM5), Weighted Methods per Class (WMC), etc.

4.2 Data labelling and risk categorisation approach

This subsection describes the methodologies employed for labelling data and categorising the risk levels of software modules, which are fundamental to the development of our SFP models.

As discussed in Subsection 4.1, in this study, the target variable is the `Number of Faults`, which quantifies the number of faults within each software module. Each instance in our dataset corresponds to a unique software module. Hence, when referring to instances, we are discussing individual software modules under examination.

Our analysis of fault distribution, as shown in as shown in Fig. 1, reveals that the number of faults varies and is discontinuous across different datasets. For instance, in the ANTLR v4 dataset, 80.77% of modules have no fault while 18.38% have 1 fault and 0.85% have 2 faults. Similarly, in the Elastic Search dataset, 55.45% of modules are faultless, but the rest range from 1 to 8 faults, with varying distribution percentages. Also, it can be seen that most modules contain no or very few faults, as illustrated by the large percentage of modules with zero faults in all datasets. However, the proportion of modules that contain multiple faults differs significantly between datasets, suggesting variability in fault-proneness across different software.

This heterogeneous distribution of faults within each dataset serves as the foundation for our proposed approach: *Risk Categorisation*. The strategy for risk categorisation derives from the statistical concept of quantiles. Specifically, it utilises the distribution of the *Risk Ratio* (RR_i), defined by (5) as one minus the ratio of the number of instances (software modules) with a particular number of faults i to the total number of instances (software modules) in each dataset.

Table 3 Example of the ANTLR v4 dataset

Modules	ESM IndexOutOfBounds	ESM NullPointer	ESM ClassCast	ESM Arithmetic	LOC	LCOM5	WMC	...	Number of Faults
1	1	208	33	3	565	2	78	...	1
2	1	57	13	0	205	5	52	...	1
3	8	260	18	0	833	10	161	...	0
4	189	593	42	18	1067	3	152	...	1

Table 4 Example of the JUnit dataset

Modules	ESM IndexOutOfBounds	ESM NullPointer	ESM ClassCast	ESM Arithmetic	LOC	LCOM5	WMC	...	Number of Faults
1	0	9	0	0	40	5	5	...	0
2	0	3	0	0	72	1	8	...	1
3	4	29	0	0	166	14	23	...	1
4	1	37	5	0	101	3	8	...	3

Table 5 Example of the OrientDB dataset

Modules	ESM IndexOutOfBounds	ESM NullPointer	ESM ClassCast	ESM Arithmetic	LOC	LCOM5	WMC	...	Number of Faults
1	0	8	1	0	38	1	8	...	0
2	8	209	14	0	583	3	135	...	1
3	5	226	82	1	638	5	176	...	2
4	2	1428	38	7	1158	1	226	...	3

Table 6 Example of the elastic search dataset

Modules	ESM IndexOutOfBounds	ESM NullPointer	ESM ClassCast	ESM Arithmetic	LOC	LCOM5	WMC	...	Number of Faults
1	32	263	6	1	161	1	17	...	1
2	70	139	51	60	351	3	18	...	1
3	0	61	0	0	71	1	4	...	2
4	32	77	1	6	72	1	16	...	0

$$RR_i = 1 - \frac{N_i}{N_{total}} \tag{5}$$

Where:

- RR_i represents the Risk Ratio for each number of faults i,
- N_i denotes the number of instances with i faults,
- N_{total} denotes the total number of instances in each dataset.

We introduce two lemmas to formalise this risk categorisation approach.

Lemma 2 ***Risk Ratio Uniformity.*** *Given a dataset with N_{total} software modules, the number of instances with a certain number of faults, i, can be transformed into a risk ratio, RR_i, such that the distribution of the risk ratios becomes more uniform across the dataset.*

Proof Consider a software module with a fault count i, where $i \in [0, I_{max}]$ with I_{max} being the maximum number of faults observed in any software module in the dataset. According to (5), as N_{total} is a constant for any given dataset, and N_i is a number between 0 and N_{total}, the resulting risk ratio RR_i will fall in the range [0, 1]. This allows us to compare the risk across modules and provides a more uniform distribution for risk categorisation.

Lemma 3 ***Risk Ratio Categorisation.*** *The risk ratios can be categorised into three classes - High Risk, Medium Risk, and Low Risk - using quantiles. These classes facilitate effective resource allocation and risk management.*

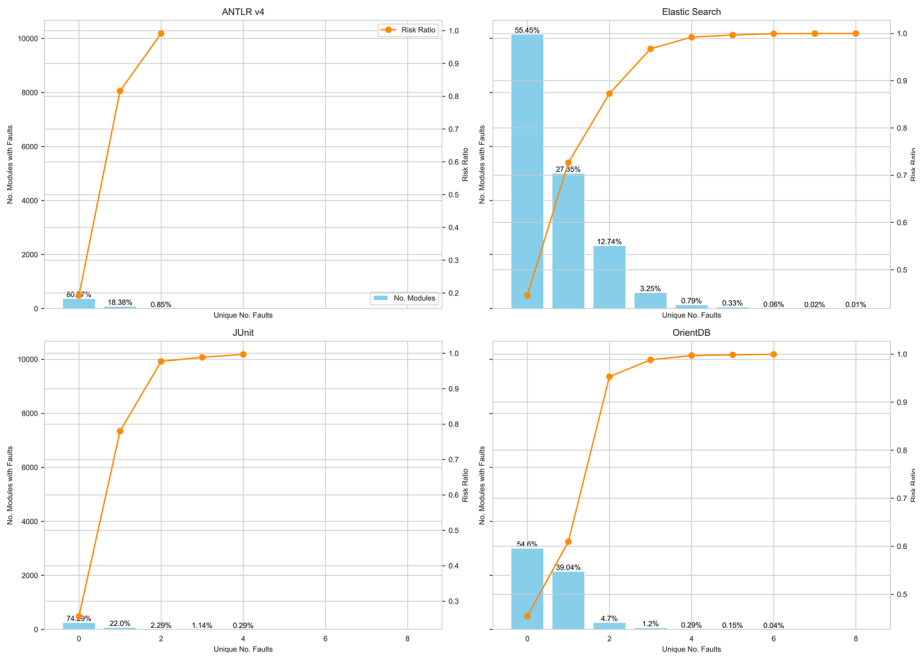


Fig. 1 Fault distributions and risk ratios.

Proof Let RR_i denote the risk ratio of the i^{th} module in the dataset, where $i \in [1, N_{total}]$. Using quantiles, we classify the risk ratios into three categories:

1. High Risk: $RR_i \geq Q_3$ (where Q_3 is the third quartile of the risk ratios).
2. Medium Risk: $Q_1 \leq RR_i < Q_3$ (where Q_1 and Q_3 are the first and third quartiles of the risk ratios, respectively).
3. Low Risk: $RR_i < Q_1$.

With these two lemmas, we formally establish the mathematical and logical grounds of our risk categorisation approach. This approach aims at improving software quality management by providing a means to identify, classify, and manage risks efficiently. The application of quantiles ensures that the risk categories balance the distribution of the Risk Ratio. This data-driven, percentile-based risk categorisation method is a generally accepted statistical approach for dividing data into groups with similar properties (Khoshgoftaar et al., 2004a, b).

We establish three risk categories:

- **High Risk:** Software modules that fall into this category have a Risk Ratio in the top 25% of all modules. This categorisation indicates that a comparatively small proportion of modules contain a significantly higher number of faults. By identifying these High Risk modules, developers can allocate their testing resources where they are most needed.
- **Medium Risk:** Modules in this category have a Risk Ratio between the first quartile (the 25^{th} percentile) and the third quartile (the 75^{th} percentile). These modules contain a moderate number of faults. Identifying Medium Risk modules is also beneficial, as it enables developers to assign adequate resources without over-prioritising these modules at the expense of High Risk ones.
- **Low Risk:** Modules with a Risk Ratio below the first quartile (the 25^{th} percentile) fall into this category. These modules contain fewer or no faults. With the ability to identify Low Risk modules, developers can wisely conserve resources, using them instead where they can have a higher impact on software quality.

This categorisation aligns with the Pareto Principle (80/20 rule), which suggests that a majority of faults are often found in a minority of the modules (Andersson & Runeson, 2007). While the Pareto Principle traditionally refers to an 80/20 distribution, it is a heuristic rather than a strict law, and the exact percentages can vary depending on the specific context and dataset. In our analysis, we observed that the top 25% of modules (High Risk category) accounted for a substantial portion of the total faults in each dataset. For example, in the Elastic Search and OrientDB datasets, although approximately 40% of the modules contained faults, we found that the majority of these faults were concentrated within the top 25% modules with the highest Risk Ratios. Similarly, in the JUnit dataset, 25% of the modules had faults, aligning with our High Risk category. Therefore, to harmonise the Pareto Principle with our empirical findings, we chose a cutoff of 25% for the High Risk modules.

By tailoring actions based on these risk levels - more intensive testing for High Risk modules and less scrutiny for Low Risk ones - this approach assists high-level managers in managing risks more effectively by focusing on broader risk categories rather than the exact number of faults in each module, thereby saving time and effort. This risk-based approach aligns with risk management principles in project management (Olsson, 2008), enabling managers to prioritise resources and efforts efficiently.

Following this categorisation, the Risk Level is used as the new target variable in our subsequent analyses.

4.3 Evaluation measures

To evaluate the performance of the SFP models, we use different types of evaluation measures including Accuracy, F1-score, Precision, Recall, and Matthews Correlation Coefficient (MCC).

The incorporation of Risk Level as a target variable provides a nuanced understanding of a software module's fault-proneness. However, it is noteworthy that the representation of risk levels is not uniform across datasets. For example, Fig. 2 shows that for the ANTLR v4 dataset, an overwhelming majority (80.77%) of modules fall into the Low Risk category, while 18.38% are categorised as Medium Risk, and a minimal 0.85% are categorised as High Risk. Similarly, in the Elastic Search dataset, a significant majority (82.81%) of modules are categorised as Low Risk, 15.99% as Medium Risk, and only 1.20% as High Risk. On the other hand, the JUnit dataset shows a slightly higher percentage of modules in the Medium Risk category at 24.29%, with 74.29% being Low Risk and 1.43% as High Risk. Finally, in the OrientDB dataset, the vast majority of the modules (93.63%) are deemed Low Risk, with 4.70% as Medium Risk, and a relatively small percentage (1.67%) as High Risk.

Fig. 2 Risk level distributions

The significant class imbalance in these datasets underscores the unequal representation of risk classes. The majority of instances correspond to the Low Risk level, with Medium Risk and High Risk making up a considerably smaller portion. Such skewness may potentially lead to complications when assessing model performance.

Traditional performance measures such as accuracy, precision, and recall may present an overly optimistic view of the model's predictive power. For instance, in the ANTLR v4 and Elastic Search datasets, a model could achieve an accuracy of over 80% by solely predicting the majority class (Low Risk). Similarly, in the OrientDB dataset, a model could obtain an accuracy of approximately 93% by predicting only the majority class. However, this would completely overlook the model's ability (or inability) to correctly predict the minority classes, which in this case, represent higher risk levels and are likely of most interest.

For a more balanced measure, particularly relevant in our context of imbalanced datasets, the F1-score, being the harmonic mean of precision and recall, is more useful as it ensures that both false positives and false negatives are taken into account during model performance evaluation.

Although the F1-score offers a more balanced measure, it is not without limitations. Particularly in situations marked by substantial class imbalance, the MCC proves to be a more robust and reliable metric (Yao & Shepperd, 2020). The MCC considers both true and false positives and negatives, providing a more comprehensive evaluation of model performance across all classes. It would not inflate the performance based on correct predictions of the majority class alone, but instead, would also account for the model's ability to correctly predict the instances of higher risk levels (Medium Risk and High Risk). The MCC can be calculated using the following formula:

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{6}$$

Where:

- TP (True Positives) - faulty software modules correctly classified as faulty.
- TN (True Negatives) - fault-free software modules correctly classified as fault-free.
- FP (False Positives) - faulty software modules incorrectly classified as fault-free.
- FN (False Negatives) - fault-free software modules incorrectly classified as faulty.

In conclusion, given the considerable class imbalance in our datasets and the necessity for accurate predictions across all classes, MCC is considered a key measure to evaluate our models. Therefore, we utilise MCC as the primary measure in the model selection procedures (e.g., fine-tuning hyperparameters) and report performance on other additional measures - Accuracy, MCC, F1-score, Precision, and Recall - to ensure a holistic assessment of the model performances.

4.4 Data Preprocessing

Data preprocessing serves as the foundation for addressing our research questions, RQ1 and RQ2. Each preprocessing step is designed to tackle challenges inherent in software fault prediction discussed in Subsection 2.2, thereby enhancing the reliability and accuracy of our findings. We utilised the `scikit-learn` (Bisong & Bisong, 2019) and `imbalanced-learn` (Lemaître et al., 2017) libraries for implementing the preprocessing techniques. Here we elaborate on these steps and relate them to the objectives of our research questions.

4.4.1 Class encoding

To facilitate machine learning algorithms, we encode risk levels numerically (`Low` as 0, `Medium` as 1, and `High` as 2) using basic Python mapping functions. This step is crucial for RQ2, where we aim to predict these risk levels accurately.

4.4.2 Addressing the curse of dimensionality

Our study addresses the "curse of dimensionality", a common challenge in machine learning where high-dimensional data can hinder model performance (Goyal & Bhatia, 2021). To counteract this, we introduce controlled variation into the dataset, specifically targeting the Error-type Metrics. This is achieved through a custom function which modifies a fraction of the instances in Error-type Metric columns with the prefix `ESM_`.

In our experiments, we set a variation fraction of 10%. This means that for each constant column under the `ESM_` prefix, 10% of its instances are altered. The implementation involves identifying constant columns, and then randomly selecting instances which are modified with new values. To ensure reproducibility, we use a fixed random seed from the `numpy` package (`np.random.seed(42)`). This process introduces necessary diversity in the data, mitigating the risk of learning bias from constant values and enhancing model generalisation.

The introduction of this minor but controlled variability in the Error-type Metrics aligns with theoretical recommendations (Hastie et al., 2001). By doing so, we aim to prevent overfitting and augment model performance on unseen data, addressing the adverse effects of high dimensionality while improving the generalisation capability of our models.

4.4.3 Investigating feature insights and handling multicollinearity

Multicollinearity among features can adversely affect machine learning models by inflating variances and leading to less reliable parameter estimates (Hastie et al., 2001). Therefore, to investigate potential correlations among Error-type Metrics and address multicollinearity, we employ Principal Component Analysis (PCA) (Jolliffe, 2002; Jolliffe & Cadima, 2016), which helps in identifying the principal components that capture the most significant variance, thereby reducing redundancy among features (Niu et al., 2020).

The rationale for choosing PCA is that PCA is an unsupervised method that does not require class labels, making it suitable for our exploratory analysis (Karamizadeh et al., 2013). Alternative methods like Linear Discriminant Analysis (LDA) require class labels and are more appropriate when maximising class separability (Izenman, 2013), which was not the primary goal at this stage. Furthermore, PCA has been effectively employed to enhance model performance by reducing dimensionality. By transforming high-dimensional software metrics into a lower-dimensional space, PCA aids in mitigating issues related to multicollinearity and overfitting, thereby improving the predictive accuracy of fault prediction models. For instance, a study by Mahanta et al. (2024) demonstrated that integrating PCA with ensemble learning techniques led to significant improvements in defect prediction accuracy. Similarly, research by Dhamayanthi and Lavanya (2019) highlighted the efficacy of PCA in conjunction with Naïve Bayes for software fault prediction, underscoring its role in enhancing model robustness and generalisability.

In our experiments, we apply PCA selectively to different subsets of our dataset, specifically targeting the Error-type Metrics and their interaction features.

Our approach involves setting up the parameters such as $n\_components = 0.95$, which indicates the amount of variance (95%) we aim to capture in the principal components, and

$random\_state = 42$ for reproducibility. We have options to apply PCA to Error-type Metrics (`apply_pca_esm`), their interaction features (`apply_pca_esm_interactions`), and other features (`pca_others`). The PCA transformation is tailored based on these settings.

For each subset of features, we compute principal components if PCA is to be applied. For instance, if `apply_pca_esm` is True, PCA is performed on Error-type Metrics, reducing them to principal components that capture 95% of their variance. The same process applies to interaction features and other features in the dataset. If PCA is not applied to a particular subset, the original features are retained.

This transformation allows us to identify the unique contributions of each Error-type Metric and their interactions, enabling us to detect underlying correlations and address multicollinearity. By understanding the variance each original metric contributes, we can more effectively scrutinise the impact of Error-type Metrics across different software domains, directly contributing to RQ1.

4.4.4 Feature enrichment

The datasets are then enriched by creating interaction features based on Error-type Metric columns. Particularly, this is done using a custom Python function which generates new features by pairwise multiplication of these metrics. For example, if the dataset contains `ESM_Arithmetic` and `ESM_NullPointer`, a new feature `ESM_Arithmetic_x_ESM_NullPointer` is created. This method is relevant for RQ2 as it aims to uncover complex relationships between different runtime error types, potentially enhancing the predictive accuracy of our models. This approach aligns with the principles in (Hastie et al., 2001), advocating for the inclusion of interaction terms in predictive models to capture more nuanced relationships between variables.

4.4.5 Feature scaling

Non-constant columns in the dataset undergo scaling using `scikit-learn`'s `PowerTransformer` employing the Yeo-Johnson transformation to ensure an equal contribution of each feature to the learning algorithms (Riani et al., 2023; Hastie et al., 2001). The transformed, Gaussian-like data aids in improving the performance of many machine learning models. This transformation is particularly vital for RQ2, where the goal is to predict risk levels across diverse software projects.

4.4.6 Class balancing

Addressing the significant class imbalance in our datasets is crucial for building effective SFP models, as detailed in Subsection 4.3. Common techniques for handling class imbalance include random oversampling and undersampling. However, random oversampling can lead to overfitting due to the duplication of minority class instances, while random undersampling may result in the loss of valuable information by removing instances from the majority class (He & Garcia, 2009).

We choose the Synthetic Minority Over-sampling Technique (SMOTE) for its balance of simplicity and effectiveness. SMOTE generates synthetic samples by interpolating between existing minority class examples, effectively mitigating overfitting without discarding important data (Chawla et al., 2002). Although more sophisticated algorithms like

Borderline-SMOTE (Han et al., 2005), ADASYN (He et al., 2008), and SMOTEENN (Batista, 2004) offer refined resampling processes, they introduce additional complexity and computational overhead (Fernández et al., 2018). Additionally, its proven efficacy in SFP scenarios further supports our choice. Studies have demonstrated that SMOTE improves classification performance in SFP by effectively handling class imbalance (Gupta et al., 2024; Pak et al., 2018; Tamanna et al., 2022). Its ability to enhance model accuracy without significant drawbacks associated with other methods makes it particularly suitable for our research.

In this study, we employ SMOTE from the `imbalanced-learn` library (Lemaître et al., 2017), using its default settings and setting `random_state` to 42 for reproducibility. Notably, SMOTE is applied solely to the training set to maintain the integrity of the evaluation process, ensuring that performance on the test set remains unbiased and accurately reflects the model's ability to generalise to unseen data.

4.4.7 Nested cross-validation

To ensure a robust measure of expected performance, particularly crucial for the predictive models in RQ2, we employ nested cross-validation (CV) through `scikit-learn`'s `GridSearchCV` and `StratifiedKFold` modules into our evaluation pipeline. Nested CV serves as a robust mechanism for both hyperparameter tuning and performance evaluation, providing a more reliable performance estimate for the model on unseen data (Cawley & Talbot, 2010). The technique comprises an outer loop to estimate the generalisation error and an inner loop for model selection, such as hyperparameter tuning. This approach prevents information leakage from the test set to the model training process, ensuring unbiased performance estimates.

In our experiments, for the outer loop of the nested cross-validation, we set n_splits to 5. This determines the number of folds for the outer loop, ensuring a comprehensive and robust generalisation error estimation. For the inner loop, which focuses on model selection and hyperparameter tuning, n_splits is set to 3. For both loops, we set $shuffle$ to `True`. This ensures that the data is shuffled before being split into folds, contributing to the randomness and thereby enhancing the robustness of the cross-validation process. The use of 'shuffle' is particularly important in scenarios where data might have an implicit order that could bias the validation process.

Stratified K-Fold cross-validation is particularly chosen over standard K-Fold cross-validation to ensure that each fold is a good representative of the whole dataset. It maintains the same proportion of each target class as the complete dataset in each fold. This is especially crucial for RQ2, where the distribution of risk levels (Low, Medium, High) must be consistent across all folds to get an accurate estimate of model generalisation performance. Furthermore, `StratifiedKFold` provides a more reliable performance evaluation for imbalanced class distribution, which is often the case in SFP scenarios. This method ensures that each fold has the same distribution of classes, thereby preventing any single class from being over-represented or under-represented. Although it can be computationally demanding, the use of `StratifiedKFold` is justified in our context where an accurate estimate of generalisation performance is paramount.

4.5 Selection of machine learning models

The choice of machine learning models in our research methodology is essential, particularly for addressing RQ2. Informed by comprehensive studies in the field, such as those by Rathore

and Kumar (2019) and Malhotra (2015), our selection of models is grounded in both historical performance and current best practices in SFP.

Malhotra (2015) identified several machine learning techniques as predominant in SFP, including C4.5 in the Decision Tree category, Naïve Bayes in Bayesian learners, Multi-layer Perceptron in Neural Networks, and Random Forest in Ensemble learners. Notably, Random Forest (RF) was highlighted for its significant usage (59%) in the Ensemble learners category, alongside its proven effectiveness in SFP scenarios. The five techniques that performed the best in SFP have been C4.5, NB, MLP, Support Vector Machine (SVM), and RF.

Drawing from this insight, we have selected SVM, RF, and Extreme Gradient Boosting (XGBoost or XGB) for our study. The choice of SVM and RF is directly influenced by their historical performance and noted effectiveness in the domain of SFP. SVM is renowned for its robustness in high-dimensional spaces, making it suitable for datasets with a large number of features, such as those in our study. RF, being one of the most effective models in SFP as per the cited studies, offers excellent performance and interpretability, which are critical in understanding fault prediction dynamics. Additionally, we incorporate XGB, an advanced implementation of gradient boosting algorithms, known for its ability to handle imbalanced datasets effectively. This choice is particularly relevant given the imbalanced nature of fault data in software projects.

To tailor these models to our datasets effectively, we configured them based on the sizes of the datasets:

- *Small Models* are for datasets with fewer than 500 instances.
- *Medium Models* are suitable for datasets with 500 to 10,000 instances.
- *Large Models* are designed for datasets with more than 10,000 instances.

This configuration ensures that each model is optimally configured to the specific characteristics of the dataset size, enhancing performance and interpretability.

4.5.1 Support vector machines (SVM)

SVMs are highly effective in classification tasks, particularly in high-dimensional spaces, due to their ability to find an optimal hyperplane for class separation (Cortes & Vapnik, 1995). For our study, the SVM models were configured with a radial basis function (RBF) kernel. The regularisation parameter C was varied across different models:

- *Small Models*: `C` [0.1, 1, 10].
- *Medium Models*: `C` [1, 5, 10].
- *Large Models*: `C` [1, 5, 10, 50].

These configurations, particularly the regularisation parameter C, are chosen to balance the trade-off between classification boundary and misclassification rate.

4.5.2 Random forest (RF)

RF's robustness to overfitting and its capability to capture complex relations in data make it suitable for our study (Breiman, 2001). The RF model in our study was configured with a range of hyperparameters:

- *Small Models*: `n_estimators` [10, 50, 100], `max_depth` [3, 5], `min_samples_split` [5, 10], `min_samples_leaf` [5, 10].

- *Medium Models*: `n_estimators` [100, 200], `max_depth` [5, 10], `min_samples_split` [5, 10], `min_samples_leaf` [5, 10].
- *Large Models*: `n_estimators` [500], `max_depth` [10, 20], `min_samples_split` [5, 10], `min_samples_leaf` [5, 10].

These settings help in tuning the RF model to effectively handle different dataset complexities and sizes.

4.5.3 XGBoost (XGB)

XGB is renowned for its performance with imbalanced and high-dimensional datasets (Chen & Guestrin, 2016) making it highly relevant for our study. In our experiments, XGB was configured with various hyperparameters:

- *Small Models*: `n_estimators` [10, 50, 100], `learning_rate` [0.01, 0.1], `max_depth` [5, 10].
- *Medium Models*: `n_estimators` [100, 200], `learning_rate` [0.01, 0.1], `max_depth` [5, 10].
- *Large Models*: `n_estimators` [500], `learning_rate` [0.001, 0.01], `max_depth` [10, 20].

These configurations enable XGB to handle the distinct challenges presented by different sizes of datasets, particularly in terms of balancing bias and variance.

5 Experimental results

This section presents our study's findings, addressing the two research questions outlined in Section 1. We then compare our approach with existing methodologies and discuss the potential limitations.

5.1 Impact of error-type metrics and multicollinearity investigation (RQ1)

This subsection addresses the first research question (RQ1), exploring the impact of Error-type Metrics across different software domains and investigating potential multicollinearity among these metrics. As described in Section 4.4, we used Principal Component Analysis (PCA) to identify the unique contributions of each Error-type Metric and to deal with multicollinearity.

We applied this analytical approach to four software project datasets: ANTLR v4, Elastic Search, JUnit, and OrientDB. The PCA results, illustrated in Fig. 3, highlight four principal components across all four datasets that account for 95% of the variance.

5.1.1 Multicollinearity among error-type metrics

To evaluate the presence and extent of multicollinearity among Error-type Metrics, we employed Pearson's correlation test (Sedgwick, 2012) within each dataset. The results from the Pearson's correlation test, detailed in Table 7, together with the PCA findings depicted in Fig. 3, fulfil two purposes: they enhance our comprehension of the Error-type Metrics by identifying the most significant metrics and elucidating which components account for

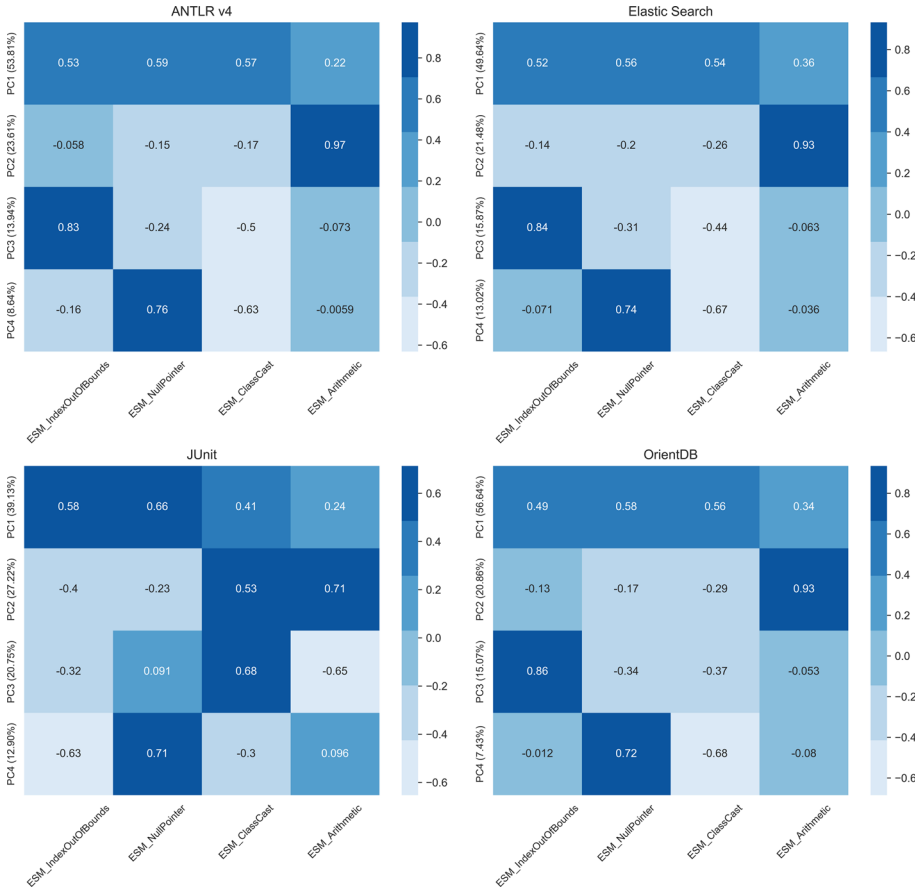


Fig. 3 PCA results on error-type metrics

the greatest variance. This holistic strategy not only assists in identifying instances of multicollinearity that correlation coefficients alone might not reveal but also conducts an in-depth analysis of the interactions among Error-type Metrics across various datasets. Such analysis yields critical insights into both the singular and collective impacts of these metrics on the precision and efficiency of the models.

We formulate our hypothesis testing framework as follows:

Null Hypothesis (H0): There is no significant linear correlation between the Error-type Metrics. This suggests that any observed correlation is due to random chance.

Alternative Hypothesis (H1): There is a significant linear correlation between the Error-type Metrics. This indicates a true relationship between the metrics, beyond random chance.

This analysis is crucial in understanding the relationship between different Error-type Metrics and how they collectively influence the principal components identified in our PCA analysis. Significant correlations may indicate potential multicollinearity issues, while low or insignificant correlations suggest that each Error-type Metric contributes unique information to our models. The Pearson's correlation test results are illustrated in Table 7.

In the ANTLR v4 dataset, PC1, accounting for 53.81% of the variance, is significantly influenced by ESM ClassCast and ESM NullPointer, with a Pearson correlation of 0.65. This

Table 7 Pearson correlation coefficients between error-type metrics

Dataset	Metric Pair	Pearson Coefficient	p-value
ANTLR v4	ESM_IndexOutOfBounds & ESM_NullPointer	0.53	< 0.001
	ESM_IndexOutOfBounds & ESM_ClassCast	0.46	< 0.001
	ESM_IndexOutOfBounds & ESM_Arithmetic	0.16	0.0004
	ESM_NullPointer & ESM_ClassCast	0.65	< 0.001
	ESM_NullPointer & ESM_Arithmetic	0.15	0.001
	ESM_ClassCast & ESM_Arithmetic	0.13	0.0037
Elastic Search	ESM_IndexOutOfBounds & ESM_NullPointer	0.40	< 0.001
	ESM_IndexOutOfBounds & ESM_ClassCast	0.38	< 0.001
	ESM_IndexOutOfBounds & ESM_Arithmetic	0.22	< 0.001
	ESM_NullPointer & ESM_ClassCast	0.48	< 0.001
	ESM_NullPointer & ESM_Arithmetic	0.23	< 0.001
	ESM_ClassCast & ESM_Arithmetic	0.20	< 0.001
JUnit	ESM_IndexOutOfBounds & ESM_NullPointer	0.45	< 0.001
	ESM_IndexOutOfBounds & ESM_ClassCast	0.06	0.2282
	ESM_IndexOutOfBounds & ESM_Arithmetic	0.05	0.3868
	ESM_NullPointer & ESM_ClassCast	0.24	< 0.001
	ESM_NullPointer & ESM_Arithmetic	0.05	0.3505
	ESM_ClassCast & ESM_Arithmetic	0.18	0.0008
OrientDB	ESM_IndexOutOfBounds & ESM_NullPointer	0.47	< 0.001
	ESM_IndexOutOfBounds & ESM_ClassCast	0.38	< 0.001
	ESM_IndexOutOfBounds & ESM_Arithmetic	0.25	< 0.001
	ESM_NullPointer & ESM_ClassCast	0.70	< 0.001
	ESM_NullPointer & ESM_Arithmetic	0.31	< 0.001
	ESM_ClassCast & ESM_Arithmetic	0.24	< 0.001

strong correlation suggests a combined influence on PC1. PC2, which accounts for 23.61% of the variance, is predominantly influenced by ESM Arithmetic, showing a distinct correlation pattern with other metrics.

The Elastic Search dataset reveals that PC1, contributing 49.64% to the total variance, is shaped by ESM ClassCast and ESM NullPointer with a correlation of 0.48. The distinct influence of ESM Arithmetic on PC2, accounting for 21.48% of the variance, is evident from its unique correlation with other metrics.

For JUnit, PC1, covering 39.13% of the variance, is influenced by ESM NullPointer and ESM IndexOutOfBounds with a correlation of 0.45. PC2, accounting for 27.22% of the variance, shows distinct contributions from ESM ClassCast and ESM Arithmetic, indicated by a lower correlation of 0.27.

In OrientDB, PC1, which accounts for 56.64% of the variance, is largely influenced by ESM ClassCast and ESM NullPointer, with a correlation of 0.70. The significant impact of ESM Arithmetic on PC2, which accounts for 20.86% of the variance, is reinforced by its distinct correlation patterns.

Overall, the outcomes of the Pearson's correlation test, as presented in Table 7, show various degrees of correlation strengths between the Error-type Metrics, accompanied by different levels of statistical significance. In specific datasets such as ANTLR v4 and Ori-

entDB, we observe strong correlations between certain pairs of metrics, evidenced by very low p-values (for instance, ESM NullPointer & ESM ClassCast in ANTLR v4 with a correlation of 0.65 and a p-value of < 0.001), signifying a substantial linear relationship and leading to the rejection of the null hypothesis for these combinations. Such significant correlations suggest these metrics have a collective impact on the principal components. On the other hand, certain metric pairs across the datasets display moderate to negligible correlations, indicated by higher p-values. A case in point is the JUnit dataset, where ESM IndexOutOfBounds & ESM ClassCast show a weak correlation (correlation: 0.06, p-value: 0.2282), hinting at their distinct contributions to the principal components. In these instances, the null hypothesis, positing no significant linear correlation, remains tenable.

The detection of both robust and minimal correlations among various Error-type Metric pairs, validated by their respective p-values, reveals a complex landscape of relationships. Some metrics exhibit significant interconnections, whereas others appear relatively independent. This intricate pattern of metric interrelations deepens our insight into the individual and combined roles of Error-type Metrics in influencing the principal components. It affirms our analytical strategy, demonstrating that despite certain significant correlations among pairs of Error-type Metrics, the overall diversity in their information contribution helps address concerns about pervasive multicollinearity within these metrics.

5.1.2 Domain-specific impact of error-type metrics

In our investigation of the domain-specific effects of Error-type Metrics on SFP, we employed the Kruskal-Wallis test (Ostertagova et al., 2014), a non-parametric method ideal for comparing multiple independent groups. This test is particularly apt for our analysis as it does not require the data to follow a normal distribution, making it well-suited for evaluating the principal component scores obtained from the PCA of Error-type Metrics. Our hypothesis testing framework was established as follows:

Null Hypothesis (H0): The distributions of principal component scores are the same across different datasets, indicating no significant domain-specific variations in the impact of Error-type Metrics.

Alternative Hypothesis (H1): There is at least one dataset whose principal component scores' distribution significantly differs from others, suggesting domain-specific variations in the impact of Error-type Metrics.

Applying the Kruskal-Wallis test to the scores of each principal component across the four datasets (ANTLR v4, Elastic Search, JUnit, and OrientDB) was essential for identifying any significant disparities in their distribution, a key step in assessing the variable impact of Error-type Metrics across software domains.

The findings, as documented in Table 8, provided the following insights:

- The p-values for PC1 and PC2 were found to be 0.5251 and 0.1431, respectively, indicating that the distributions of these components do not significantly differ across the

Table 8 Kruskal-wallis test results for principal components

Principal Component	Statistic	p-value
PC1	2.2349	0.5251
PC2	5.4260	0.1431
PC3	4.3493	0.2261
PC4	8.4559	0.0375

datasets. This result supports the uniformity in the effect of Error-type Metrics across various software domains, affirming the null hypothesis for these principal components.
- Similarly, PC3 showed no substantial domain-specific variation in the impact of Error-type Metrics, as evidenced by a p-value of 0.2261, further aligning with the null hypothesis.
- Contrastingly, PC4 exhibited a notable exception with a p-value of 0.0375, falling below the conventional alpha threshold of 0.05 for determining statistical significance. This outcome suggests a significant difference in the distribution of PC4 scores across the datasets, warranting the null hypothesis's rejection for this component. This indicates meaningful domain-specific variation in the influence of Error-type Metrics for PC4, underscoring the importance of a detailed comprehension of these metrics across various software environments.

Overall, the outcomes from the Kruskal-Wallis test primarily reinforce the view that, for the majority of principal components, the impact of Error-type Metrics on SFP models does not significantly diverge across different software domains. Nonetheless, the distinct finding for PC4 underscores the presence of domain-specific disparities in the influence of these metrics, shedding light on the nuanced and variable nature of applying Error-type Metrics in diverse software contexts and underscoring the critical need for a contextually aware application of these metrics in SFP.

5.1.3 Implications and summary

The implications of our findings in the context of Error-type Metrics' influence on SFP and their correlation behaviours are significant. The key implications are as follows:

- *Independence of Error-type Metrics:* Pearson's correlation analysis revealed that while some Error-type Metrics are strongly correlated (e.g., ESM NullPointer and ESM Class-Cast in ANTLR v4 with a correlation of 0.65), others are weakly correlated. This suggests that, despite some relationships, each metric largely retains predictive independence, enhancing their utility in SFP models without significant multicollinearity concerns.
- *Domain-Specific Relevance of Error-type Metrics:* The Kruskal-Wallis test indicated domain-specific variations in the impact of Error-type Metrics (e.g., a significant result for PC4 with a p-value of 0.0375). Certain metrics have a more pronounced influence in specific software environments; for instance, the Class Cast metric showed a substantial impact in the JUnit dataset. This allows for a tailored application of these metrics in different software contexts.
- *Enhanced Fault Prediction Models:* Recognising the unique contributions of Error-type Metrics enables the development of more accurate SFP models. Understanding each metric's specific impact across various environments helps practitioners allocate resources effectively and focus on the most relevant error types, aligning with the need for context-specific evaluation as emphasised by Rathore and Kumar (2019).

5.2 Predicting risk level using error-type metrics (RQ2)

This subsection focuses on the second research question (RQ2), which investigates the efficacy of Error-type Metrics in predicting the risk level of software modules. We employ three machine learning models - Support Vector Machines (SVM), Random Forest (RF), and XGBoost (XGB) as discussed in Subsection 4.5 - for this purpose.

5.2.1 Model performance and insights

The comparative analysis of SVM, RF, and XGB on four diverse datasets - ANTLR v4, Elastic Search, JUnit, and OrientDB, as shown in Table 9 and Fig. 4, is illustrative of the different performance facets of these machine learning models.

For the ANTLR v4 dataset, XGB outperforms the other models with an accuracy of 92.2%, demonstrating its capability to correctly classify a higher percentage of instances. The MCC score for XGB is 0.887, which surpasses the MCC scores for SVM (0.741) and RF (0.755), pointing to a higher correlation between the observed and predicted classifications. In terms of the F1-score, XGB again leads the race with a score of 0.921, as opposed to SVM and RF, which lag at 0.821 and 0.833, respectively.

For the Elastic Search dataset, a similar pattern is seen, with XGB dominating with the highest accuracy of 94.8%, an MCC of 0.923, and an F1-score of 0.947. However, RF also exhibits strong performance, with an accuracy of 92.6%, MCC of 0.890, and F1-score of 0.926. SVM, although performing better than in the ANTLR v4 dataset, still falls behind with an accuracy of 82.5%, MCC of 0.739, and F1-score of 0.821.

The trend of XGB outperforming the other models continues in the JUnit dataset. XGB reports an accuracy of 91.3%, an MCC of 0.873, and an F1-score of 0.912. The RF model performs competitively but still falls short with an accuracy of 84.9%, MCC of 0.774, and an F1-score of 0.848. The performance of SVM is lower, with an accuracy of 84%, MCC of 0.762, and an F1-score of 0.838.

In the OrientDB dataset, XGB performs exceptionally well with an impressive accuracy of 97.4%, an MCC of 0.961, and an F1-score of 0.974. The RF model also shows good performance with an accuracy of 95.1%, an MCC of 0.928, and an F1-score of 0.951. The SVM model trails behind with lower scores: an accuracy of 85.6%, an MCC of 0.786, and an F1-score of 0.857.

The comparative analysis conducted on the ANTLR v4, Elastic Search, JUnit, and OrientDB datasets not only showcases the intrinsic strengths of SVM, RF, and XGB in managing varying data complexities but also highlights how the size of the dataset influences the performance of these models. Throughout the analysis, a clear pattern is observed where

Table 9 Performance comparison of SVM, random forest, and XGBoost across different datasets

Dataset	Model	Accuracy	MCC	F1-score	Precision	Recall
ANTLR v4	SVM	0.824	0.741	0.821	0.826	0.825
	RF	0.835	0.755	0.833	0.835	0.835
	XGB	**0.922**	**0.887**	**0.921**	**0.930**	**0.922**
Elastic Search	SVM	0.825	0.739	0.821	0.822	0.825
	RF	0.926	0.890	0.926	0.927	0.926
	XGB	**0.948**	**0.923**	**0.947**	**0.950**	**0.948**
JUnit	SVM	0.840	0.762	0.838	0.842	0.840
	RF	0.849	0.774	0.848	0.849	0.849
	XGB	**0.913**	**0.873**	**0.912**	**0.920**	**0.913**
OrientDB	SVM	0.856	0.786	0.857	0.864	0.856
	RF	0.951	0.928	0.951	0.953	0.951
	XGB	**0.974**	**0.961**	**0.974**	**0.974**	**0.974**

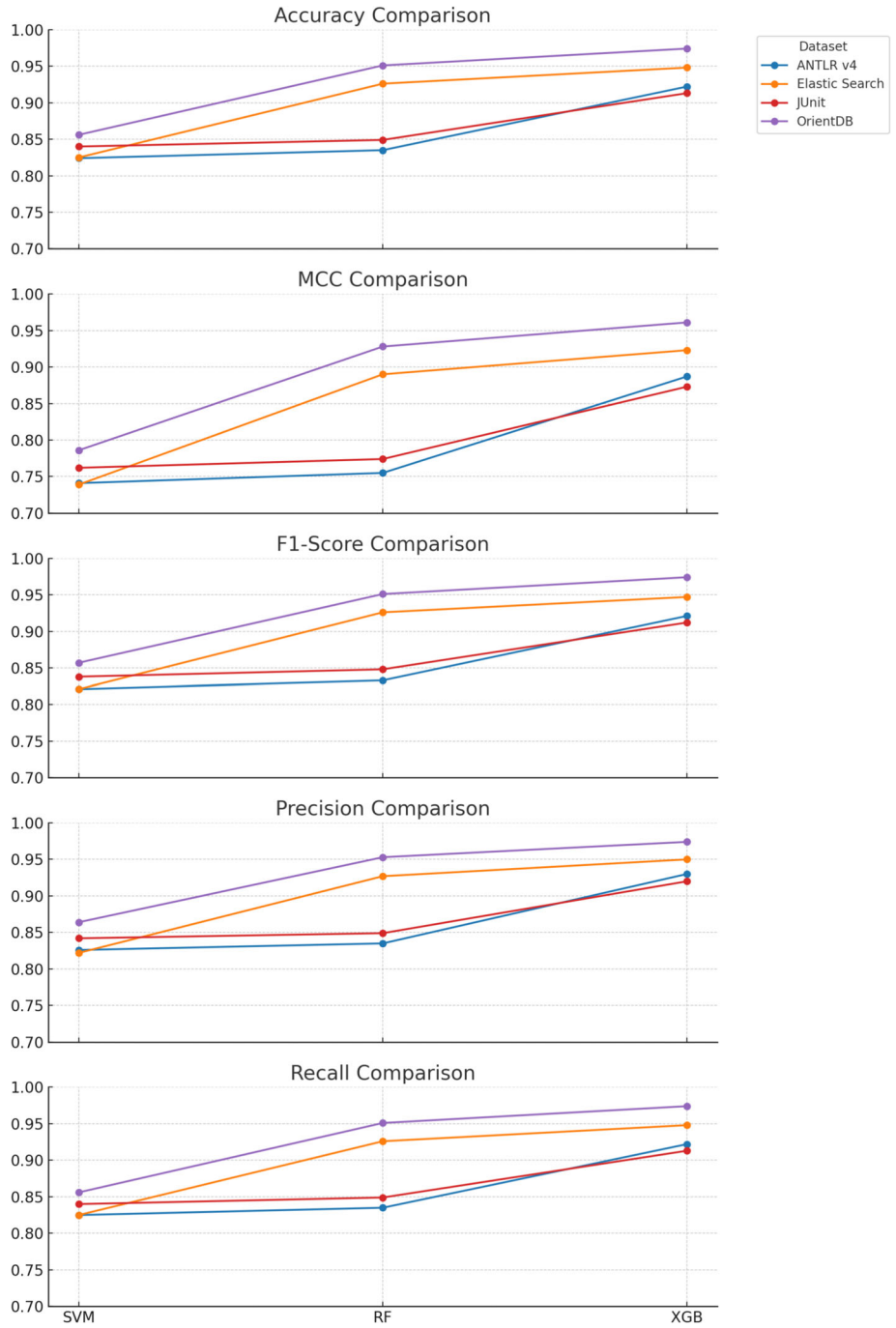


Fig. 4 Performance comparison of SVM, random forest, and XGBoost across different datasets

XGB consistently surpasses both SVM and RF across all examined datasets. This prevailing trend is largely attributed to the size of the datasets, which varies from the smaller JUnit dataset, containing around 300 instances, to the significantly larger Elastic Search dataset, with approximately 18,000 instances.

An important insight, as depicted in Fig. 4, is the evident direct relationship between the size of the dataset and the performance of the models. Notably, models trained on larger datasets, like OrientDB and Elastic Search, outperform those trained on smaller datasets such as JUnit and ANTLR v4. This suggests that an increase in dataset size markedly boosts model accuracy and dependability. For example, the leap in XGB's accuracy from the JUnit to the Elastic Search dataset by about 3.5% (from 91.3% to 94.8%), and an even more impressive increase from JUnit to OrientDB by about 6.1% (from 91.3% to 97.4%), exemplifies this point. The trend is similarly reflected in the MCC and F1-Score, which increase by approximately 0.088 and 0.062, respectively, from JUnit to OrientDB, illustrating XGB's adeptness at discerning and leveraging complex data relationships to minimise prediction errors.

XGB's gradient boosting framework, which builds models sequentially to amend predecessors' errors, greatly benefits from the rich diversity and size of data in larger datasets, as demonstrated by its exceptional outcomes on the OrientDB and Elastic Search datasets, highlighted by accuracy rates of 97.4% and 94.8%, MCC scores of 0.961 and 0.923, and F1-Scores of 0.974 and 0.947, respectively. Conversely, SVM, despite certain improvements, consistently shows lower performance across the datasets. SVM's performance on JUnit, the smallest dataset, results in an MCC of 0.762, and this performance dips further on the larger Elastic Search dataset to 0.739, indicating SVM's scaling challenges. This can be partly attributed to SVM's computational demands and the criticality of the kernel and regularisation parameter tuning. Furthermore, the implementation of SMOTE, intended to balance the dataset, introduces synthetic points that may not entirely replicate the minority class's true distribution. SVM's performance is particularly sensitive to these synthetic instances, potentially degrading its efficacy. This, combined with SVM's scalability issues on larger datasets, contrasts sharply with the adaptability and efficiency of ensemble techniques like RF and XGB.

Furthermore, the analysis points out that RF, while typically outperforming SVM, shows a positive but more modest increase in correlation with dataset size. For instance, RF's performance improvement from the smaller JUnit dataset to the larger Elastic Search dataset, from an MCC of 0.774 to 0.890, signifies a 15% uplift, demonstrating RF's ability to handle large datasets without overfitting. However, this improvement is relatively modest when compared to XGB's performance boost in the same datasets, indicating a 5.7% increase in MCC. Such comparisons highlight that although RF benefits from larger datasets, enhancing its accuracy and consistency, it does not reach the high-performance benchmark set by XGB. For example, on the OrientDB dataset, RF achieves an MCC of 0.928, commendable yet slightly below XGB's peak score of 0.961, with a relative difference of about 3.6% favouring XGB. This subtle difference further illuminates the slight variances in their ability to exploit large datasets for improved predictive accuracy.

5.2.2 Implications and summary

The implications of these results are manifold:

1. The performance of XGB indicates that Error-type Metrics can be useful predictors for software fault-proneness, providing support for RQ2.

2. The lower performance of SVM compared to RF and XGB suggests that linear models may not capture the complexity of Error-type Metrics as effectively as ensemble or boosting methods. This insight may be valuable for practitioners when choosing a model for software risk prediction.
3. The application of SMOTE for mitigating class imbalance appears to enhance our model's ability to identify high-risk modules, which is important for prioritising quality assurance efforts.

In summary, by leveraging Error-type Metrics, we have developed a model that shows promise in predicting faults across different risk categories. This capability could make our approach a useful tool for researchers and practitioners in software engineering, contributing to the field of SFP.

5.3 Comparison with state-of-the-art approaches

Our research integrates Error-type Metrics and a refined risk-based classification system into SFP models, exhibiting competitive performance with state-of-the-art SFP approaches that commonly use traditional metrics such as complexity, churn, or code metrics (Malhotra, 2015).

We break from convention by employing Error-type Metrics, verified in our prior study (Phung et al., 2023), offering a nuanced prediction framework. We assign software modules to three risk categories - Low Risk, Medium Risk, and High Risk - improving the granularity of fault distribution understanding compared to conventional binary-class models. The use of fault datasets sourced from open-source software projects, as opposed to widely used benchmark datasets from repositories like NASA and PROMISE, enhances the practical relevance of our work. The key distinctions between our study and existing SFP methodologies are shown in Table 10. In what follows, we discuss a detailed comparison of our study with each of these state-of-the-art approaches.

The study by Goyal (2022) employed a Neighbourhood-based Under-Sampling (N-US) algorithm, primarily focusing on the binary-class classification of fault-prone modules. Their model yielded an impressive AUC score of 95.6% and an accuracy of 96.9% using five different SFP classifiers (Artificial Neural Network - ANN, SVM, Decision Trees, K-Nearest Neighbour, and Naïve Bayes) on the NASA repository datasets. However, our XGB model, using a more nuanced risk-based classification and SMOTE to counter class imbalance, demonstrated consistent accuracy, MCC, and F1-score exceeding 96% across all datasets.

Rhmann et al. (2020) utilised software change metrics for fault prediction and evaluated various machine learning models (RF, Multi-layer Perceptron - MLP, and Decision Tree) and Hybrid Search-Based Algorithms (Fuzzy-AdaBost Del Jesus et al., 2004 and Logitboost Otero and Sánchez, 2006) on Android project data. Their results positioned Logitboost as the leading algorithm with 0.822 precision and 0.992 recall, while the RF model yielded precision and recall scores of 0.616 and 0.618 respectively. However, our approach improved these performance measures: our RF model achieved over 83% in both precision and recall, escalating to over 91% in the OrientDB and Elastic Search datasets. Our XGB ensemble model outperformed all others with precision and recall rates exceeding 92%, reaching peak values of up to 97%.

Yang et al. (2014) proposed an innovative application of the learning-to-rank (LTR) approach for SFP on real-world datasets (Yang et al., 2014). Their study's primary evaluation metric was the fault-percentile-average (FPA), which calculates the average proportion of actual faults in the top m ($m = 1, 2, \cdots, k$) modules relative to all faults. FPA has been

Table 10 Comparison with state-of-the-art approaches

Study	Technique	Datasets	Output	Primary Metric	Performance
Goyal (2022)	N-US, ANN, SVM, DT, KNN, NB	NASA	Fault Proneness	AUC	95.6%
Our Study	SMOTE, XGB	ANTLR v4, Elastic Search, JUnit, OrientDB	Risk Categorisation	MCC	> 96%
Rhmann et al. (2020)	RF, MLP, DT, Fuzzy-AdaBost, Logitboost	Android	Number of Faults	Precision, Recall	0.822, 0.992
Our Study	RF, XGB	ANTLR v4, Elastic Search, JUnit, OrientDB	Risk Categorisation	Precision, Recall	> 92%
Yang et al. (2014)	LTR	Real-world	Ranking of Faults	FPA	0.787
Our Study	XGB	ANTLR v4, Elastic Search, JUnit, OrientDB	Risk Categorisation	MCC, Accuracy, F1-score, Precision, Recall	> 92%

shown to correspond closely with the area under the Alberg diagram, a method proven to be effective for assessing software fault prediction models' ranking ability (Ohlsson & Alberg, 1996; Weyuker et al., 2010). The LTR model demonstrated strong performance, achieving an FPA of up to 0.787. In contrast, our study implemented MCC as the primary metric for model selection and reported the results in terms of accuracy, MCC, F1-score, precision, and recall. Despite the methodological differences, our XGBoost model demonstrated exceptional performance, achieving scores of up to 92% across all metrics and datasets. This suggests that our risk-based approach may offer an effective alternative to conventional ranking-based methods in SFP.

In conclusion, despite methodological differences with conventional approaches, our SFP models, especially XGB, demonstrated consistently superior performance across all performance measures, underscoring the promise of our approach.

5.4 Threats to validity

This empirical study, like all others, is potentially influenced by various validity threats affecting the generalisability and applicability of our results.

External validity Our experiments were based on four popular open-source software systems. While these systems have broad utility and span a range of application domains, they do not encompass the entire spectrum of possible software systems. Therefore, our study's results may not be directly transferable to other software systems, especially those from differing domains or differing scales. Future studies should incorporate a wider software system range to further validate our results.

Construct validity This research predominantly used Error-type Metrics. Although they provided a robust framework for the SFP approach in this study, they are not without limitations. Their effectiveness can vary depending on the complexity and specific characteristics of the software being analysed. Also, while we have performed thorough analyses to ensure these metrics did not exhibit multicollinearity issues, the way faults were counted, based on the version control commits, might introduce observational errors. The commit logs may reflect zero faults in a software module, even though latent faults may persist. This discrepancy could impact the accuracy of the prediction models. We acknowledge this limitation and have made efforts to mitigate this risk by using the most reliable and current data available. However, the interpretation of commit logs and the attribution of faults to software modules may have been influenced by subjective decisions, introducing potential bias.

An additional consideration is the 25% cutoff used for defining the High Risk category in our risk categorisation approach. This threshold was determined based on the fault distributions observed in our datasets and aligns with our empirical findings, as discussed in Subsection 4.2. While this cutoff is consistent with the notion that a majority of faults are often found in a minority of modules - an idea related to the Pareto Principle (Andersson & Runeson, 2007) - it may not be universally applicable. Different datasets might exhibit varying fault distribution patterns, necessitating adjustments to the cutoff rate to accurately reflect fault concentration. Therefore, the applicability of the 25% cutoff is limited to the datasets used in this study, and future research should consider revising the cutoff when applying the approach to other datasets.

Internal validity The main internal threat concerns the preprocessing techniques employed to address the class imbalance. Although we utilised the SMOTE technique, other methods

such as ADASYN, Borderline-SMOTE, and random oversampling could potentially provide better results. The optimisation of parameters for the SMOTE technique and machine learning models was specific to our datasets, which might not produce similar performance with different datasets.

Conclusion validity Our model selection was largely informed by the MCC performance measure. Despite its effectiveness, alternative evaluation metrics, such as Receiver Operating Characteristic curve (AUC-ROC) or Precision-Recall curves, might offer more nuanced insights into model performance and should be considered in future research (Chicco & Jurman, 2020).

6 Conclusion

This research introduced an SFP approach, demonstrating a novel application of Error-type Metrics for risk categorisation of software modules into Low, Medium, and High Risk with respect to the number of faults.

A major contribution of this research is the exploration of the domain-specific impact of Error-type Metrics in SFP. This addition addresses a gap in existing literature, identified by Rathore and Kumar (2019), regarding the need for context-specific evaluation of software metrics. Our work offers empirical evidence to validate the utility of domain-specific metrics, enriching the toolkit for practitioners in different software environments.

Another significant contribution of this work lies in the utility of Error-type Metrics for making informed resource allocation decisions. Our methodology enables the identification of the top 25% of software modules that are classified as High Risk. This is not only beneficial for developers in optimising the allocation of testing resources but also greatly aids high-level management. By enabling granular classification into Low, Medium, and High-Risk categories, our approach abstracts the complex details of software faults into more digestible risk categories. This simplification potentially facilitates easier risk monitoring and aligns well with the principles of project management, thereby streamlining decision-making processes and enhancing the effectiveness of project management.

To validate our approach, we addressed the issue of potential multicollinearity among Error-type Metrics and evaluated the performance of three machine learning models - Support Vector Machine (SVM), Random Forest (RF), and Extreme Gradient Boosting (XGB). The results showed that XGB significantly outperformed SVM and RF, yielding impressive scores (up to 97.4% of accuracy, 96.1% of MCC, and 97.4% of F1-score) across different datasets. The RF model also showed strong performance, further substantiating the effectiveness of our approach. The performance, however, may vary with different datasets.

Despite our contributions, this research is limited by several threats to validity discussed in Subsection 5.4. In future work, we plan to address these threats by further investigating them to ensure the validity and generalisability of our approach. Additionally, we will expand our research to include more industrial projects across diverse domains. This expansion will enhance the generalisability and practical applicability of our findings, potentially validating our current results in a broader context and uncovering new insights and challenges specific to industrial software projects.

Author Contributions Khoa Phung (First author) conceptualised and led the research, performed the experiments, and drafted the manuscript, while Dr. Emmanuel Ogunshile and Dr. Mehmet E. Aydin provided critical feedback and guidance throughout the study.

Funding Not Applicable.

Data Availability The complete data associated with this research is part of ongoing PhD work and will be made available upon request after the completion of the PhD project.

Code Availability The code associated with this research is part of ongoing PhD work and will be made available upon request after the completion of the PhD project.

Declarations

Consent for publication All authors have given their consent for the publication of this work.

References

Afzal, W., Torkar, R., Feldt, R. (2008). Prediction of fault count data using genetic programming. In: *2008 IEEE International Multitopic Conference*, IEEE, (pp. 349–356).

Al Dallal, J. (2013). Incorporating transitive relations in low-level design-based class cohesion measurement. *Software: Practice and Experience, 43*(6), 685–704.

Alsolai, H., & Roper, M. (2020). A systematic literature review of machine learning techniques for software maintainability prediction. *Information and Software Technology, 119*, 106214.

Andersson, C., & Runeson, P. (2007). A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering, 33*(5), 273–286.

Ardito, L., Coppola, R., Barbato, L., & Verga, D. (2020). A tool-based perspective on software code maintainability metrics: a systematic literature review. *Scientific Programming, 2020*, 1–26.

Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering, 28*(1), 4–17.

Batista, G. E. A. P. A. (2004). A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explorations, 6*(1), 20–29.

Bhandari, K., Kumar, K., & Sangal, A. L. (2022). Data quality issues in software fault prediction: a systematic literature review. *Artificial Intelligence Review,* 1–70.

Bisong, E., & Bisong, E. (2019). Introduction to scikit-learn. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners,* 215–229.

Breiman, L. (2001). Random forests. *Machine Learning, 45*, 5–32.

Bundschuh, M., & Dekkers, C. (2008). *The IT Measurement Compendium: Estimating and Benchmarking Success with Functional Size Measurement* (1 aufl). Berlin, Heidelberg: Springer.

Cawley, G. C., & Talbot, N. L. (2010). On over-fitting in model selection and subsequent selection bias in performance evaluation. *The Journal of Machine Learning Research, 11*, 2079–2107.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). Smote: synthetic minority oversampling technique. *Journal of Artificial Intelligence Research, 16*, 321–357.

Chen, T., Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In: *Proceedings of the 22nd Acm Sigkdd International Conference on Knowledge Discovery and Data Mining*, (pp. 785–794).

Chicco, D., & Jurman, G. (2020). The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics, 21*, 1–13.

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

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine learning, 20*, 273–297.

Del Jesus, M. J., Hoffmann, F., Navascués, L. J., & Sánchez, L. (2004). Induction of fuzzy-rule-based classifiers with evolutionary boosting algorithms. *IEEE Transactions on Fuzzy Systems, 12*(3), 296–308.

Deng, J., Lu, L., & Qiu, S. (2020). Software defect prediction via LSTM. *IET Software, 14*(4), 443–450.

Dhamayanthi, N., Lavanya, B. (2019). Software defect prediction using principal component analysis and naïve bayes algorithm. In: *Proceedings of International Conference on Computational Intelligence and Data Engineering: Proceedings of ICCIDE 2018*, Springer, (pp. 241–248).

Dranidis, D., Bratanis, K., Ipate, F. (2012). Jsxm: A tool for automated test generation. In: *International Conference on Software Engineering and Formal Methods*, Springer, (pp. 352–366).

Fenton, N. E., & Bieman, J. (2015). *Software Metrics: A Rigorous and Practical Approach* (3rd ed.). Boca Raton, Florida: CRC Press.

Ferenc, R., Gyimesi, P., Gyimesi, G., Tóth, Z., & Gyimóthy, T. (2020). An automatically created novel bug dataset and its validation in bug prediction. *Journal of Systems and Software, 169*, 110691.

Fernández, A., Garcia, S., Herrera, F., & Chawla, N. V. (2018). Smote for learning from imbalanced data: progress and challenges, marking the 15-year anniversary. *Journal of Artificial Intelligence Research, 61*, 863–905.

Gao, K., & Khoshgoftaar, T. M. (2007). A comprehensive empirical study of count models for software fault prediction. *IEEE Transactions on Reliability, 56*(2), 223–236.

Goyal, S. (2022). Handling class-imbalance with KNN (neighbourhood) under-sampling for software defect prediction. *Artificial Intelligence Review, 55*(3), 2023–2064.

Goyal, S., & Bhatia, P. K. (2021). Software fault prediction using lion optimization algorithm. *International Journal of Information Technology, 13*, 2185–2190.

Gupta, M., et al. (2024). Software fault prediction with imbalanced datasets using smote-tomek sampling technique and genetic algorithm models. *Multimedia Tools and Applications, 83*(16), 47627–47648.

Gupta, S., & Gupta, A. (2017). A set of measures designed to identify overlapped instances in software defect prediction. *Computing, 99*, 889–914.

Halstead, M. H. (1977). *Elements of Software Science*. New York: North-Holland.

Han, H., Wang, W., Mao, B. (2005). Borderline-smote: A new over-sampling method in imbalanced data sets learning. In: *Advances in Intelligent Computing*, Springer, Berlin, Heidelberg, (pp. 878–887).

Harrison, R., Counsell, S. J., & Nithi, R. V. (1998). An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering, 24*(6), 491–496.

Hastie, T., Tibshirani, R., & Friedman, J. H. (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (1st edn). New York, NY: Springer.

He, H., Bai, Y., Garcia, E.A., Li, S. (2008). Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In: *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, IEEE, Hong Kong, China, (pp. 1322–1328).

He, H., & Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering, 21*(9), 1263–1284.

Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., et al. (2009). Using formal specifications to support testing. *ACM Computing Surveys (CSUR), 41*(2), 1–76.

Ipate, F. (2006). Testing against a non-controllable stream x-machine using state counting. *Theoretical Computer Science, 353*(1–3), 291–316.

Ipate, F., & Dranidis, D. (2016). A unified integration and component testing approach from deterministic stream x-machine specifications. *Formal Aspects of Computing, 28*(1), 1–20.

Izenman, A.J. (2013). Linear discriminant analysis. In: *Modern Multivariate Statistical Techniques*, Springer, New York, NY, (pp. 237–280).

Jiang, Y., Cukic, B., & Ma, Y. (2008). Techniques for evaluating fault prediction models. *Empirical Software Engineering, 13*, 561–595.

Jolliffe, I.T. (2002). Principal component analysis for special types of data. In: *Principal Component Analysis*, Springer, New York, NY, ((pp. 199–22)2.

Jolliffe, I. T., & Cadima, J. (2016). Principal component analysis: a review and recent developments. *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences, 374*(2065), 20150202.

Kamei, Y., Shihab, E. (2016). Defect prediction: Accomplishments and future challenges. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, (vol. 5, pp. 33–45).

Karamizadeh, S., Abdullah, S. M., Manaf, A. A., Zamani, M., & Hooman, A. (2013). An overview of principal component analysis. *Journal of Signal and Information Processing, 4*(3), 173–175.

Khoshgoftaar, T.M., Liu, Y., Seliya, N. (2004). Module-order modeling using an evolutionary multi-objective optimization approach. In: *10th International Symposium on Software Metrics*, 2004. Proceedings., IEEE, (pp. 159–169).

Khoshgoftaar, T. M., Liu, Y., & Seliya, N. (2004). A multiobjective module-order model for software quality enhancement. *IEEE Transactions on Evolutionary Computation, 8*(6), 593–608.

Kumar, A., Bansal, A. (2019). Software fault proneness prediction using genetic based machine learning techniques. In: *2019 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU)*, IEEE, (pp. 1–5).

Lemaître, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research, 18*(1), 559–563.

Li, Z., Jing, X.-Y., & Zhu, X. (2018). Progress on approaches to software defect prediction. *IET Software, 12*(3), 161–175.

Mahanta, A. K., Pradhan, S. R., Sahoo, B., & Pradhan, D. (2024). An automated pca-lda based software fault prediction model using machine learning classifier. *International Journal of Engineering Research & Technology (IJERT),13*(1).

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing, 27*, 504–518.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 4*, 308–320.

Menzies, T., Butcher, A., Marcus, A., Zimmermann, T., Cok, D. (2011). Local vs. global models for effort estimation and defect prediction. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, IEEE, (pp. 343–351).

Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., Jiang, Y. (2008). Implications of ceiling effects in defect predictors. In: *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, (pp. 47–54).

Moreno-Torres, J. G., Raeder, T., Alaiz-Rodríguez, R., Chawla, N. V., & Herrera, F. (2012). A unifying view on dataset shift in classification. *Pattern Recognition, 45*(1), 521–530.

Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., Murphy, B. (2010). Change bursts as defect predictors. In: *2010 IEEE 21st International Symposium on Software Reliability Engineering*, IEEE, (pp. 309–318).

Niu, T., Wang, J., Lu, H., Yang, W., & Du, P. (2020). Developing a deep learning framework with two-stage feature selection for multivariate financial time series forecasting. *Expert Systems with Applications, 148*, 113237.

Ohlsson, N., & Alberg, H. (1996). Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering, 22*(12), 886–894.

Olsson, R. (2008). Risk management in a multi-project environment: An approach to manage portfolio risks. *International Journal of Quality & Reliability Management*.

Ostertagova, E., Ostertag, O., & Kováč, J. (2014). Methodology and application of the kruskal-wallis test. *Applied Mechanics and Materials, 611*, 115–120.

Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering, 31*(4), 340–355.

Otero, J., & Sánchez, L. (2006). Induction of descriptive fuzzy classifiers with the logitboost algorithm. *Soft Computing, 10*, 825–835.

Pak, C., et al. (2018). An empirical study on software defect prediction using over-sampling by smote. *International Journal of Software Engineering and Knowledge Engineering, 28*(6), 811–830.

Pandey, S. K., Haldar, A., & Tripathi, A. K. (2023). Is deep learning good enough for software defect prediction? *Innovations in Systems and Software Engineering,* 1–16.

Pandey, S. K., & Tripathi, A. K. (2020). Bcv-predictor: A bug count vector predictor of a successive version of the software system. *Knowledge-Based Systems, 197*, 105924.

Pandey, S. K., & Tripathi, A. K. (2021). Dnnattention: A deep neural network and attention based architecture for cross project defect number prediction. *Knowledge-Based Systems, 233*, 107541.

Petrić, J., Bowes, D., Hall, T., Christianson, B., Baddoo, N. (2016). The jinx on the nasa software defect data sets. In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, (pp. 1–5)

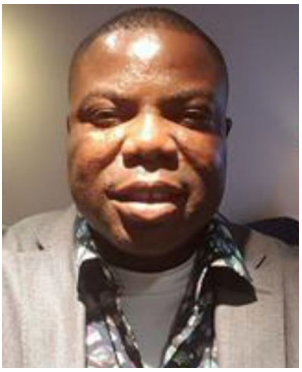
Petrić, J., Bowes, D., Hall, T., Christianson, B., Baddoo, N. (2016). The jinx on the nasa software defect data sets. In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, (pp. 1–5).

Phung, K., Ogunshile, E., Aydin, M. (2021). A novel software fault prediction approach to predict error-type proneness in the java programs using stream x-machine and machine learning. In: *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*, IEEE, (pp. 168–179).

Phung, K., Ogunshile, E., Aydin, M. (2023). *Error-type–a novel set of software metrics for software fault prediction*. IEEE Access.

Premraj, R., Herzig, K. (2011). Network versus code metrics to predict defects: A replication study. In: *2011 International Symposium on Empirical Software Engineering and Measurement*, IEEE, (pp. 215–224).
Qiao, L., Li, X., Umer, Q., & Guo, P. (2020). Deep learning based software defect prediction. *Neurocomputing, 385*, 100–110.
Rathore, S. S., & Kumar, S. (2015). Predicting number of faults in software system using genetic programming. *Procedia Computer Science, 62*, 303–311.
Rathore, S. S., & Kumar, S. (2016). A decision tree regression based approach for the number of software faults prediction. *ACM SIGSOFT Software Engineering Notes, 41*(1), 1–6.
Rathore, S. S., & Kumar, S. (2019). A study on software fault prediction techniques. *Artificial Intelligence Review, 51*, 255–327.
Rhmann, W., Pandey, B., Ansari, G., & Pandey, D. K. (2020). Software fault prediction based on change metrics using hybrid algorithms: An empirical study. *Journal of King Saud University-Computer and Information Sciences, 32*(4), 419–424.
Riani, M., Atkinson, A. C., & Corbellini, A. (2023). Automatic robust box-cox and extended yeo-johnson transformations in regression. *Statistical Methods & Applications, 32*(1), 75–102.
Sedgwick, P. (2012). Pearson's correlation coefficient. *BMJ,345*.
Shepperd, M., Song, Q., Sun, Z., & Mair, C. (2013). Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering, 39*(9), 1208–1215.
Shirabad, J.S., Menzies, T.J., et al. (2005). *The promise repository of software engineering databases*. School of information technology and engineering, University of Ottawa, Canada. (vol. 24, pp. 3).
Song, Q., Guo, Y., & Shepperd, M. (2018). A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering, 45*(12), 1253–1269.
Tahir, A., MacDonell, S.G. (2012). A systematic mapping study on dynamic metrics and software quality. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, (pp. 326–335).
Tamanna, T., et al. (2022). Feature reduction techniques for software bug prediction. In: *AIP Conference Proceedings*. American Institute of Physics, Melville
Wang, S., & Yao, X. (2013). Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability, 62*(2), 434–443.
Wang, H., Zhuang, W., & Zhang, X. (2021). Software defect prediction based on gated hierarchical LSTMs. *IEEE Transactions on Reliability, 70*(2), 711–727.
Weyuker, E. J., Ostrand, T. J., & Bell, R. M. (2010). Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering, 15*, 277–295.
Wold, S., Esbensen, K., & Geladi, P. (1987). Principal component analysis. *Chemometrics and Intelligent Laboratory Systems, 2*(1–3), 37–52.
Yang, X., Tang, K., & Yao, X. (2014). A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability, 64*(1), 234–246.
Yao, J., & Shepperd, M. (2020). Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. *Proceedings of the evaluation and assessment in software engineering,* 120–129.
Yu, L. (2012). *Using negative binomial regression analysis to predict software faults: A study of apache ant*.

Khoa Phung received his Bachelor's degree in Computing and Master's degree in Information Technology from the University of the West of England (UWE), Bristol, in 2018 and 2019, respectively. He is currently a Lecturer in Computer Science at UWE. He joined the School of Computing and Creative Technologies (SCCT) as a Lecturer in April 2022. He is also pursuing a PhD in Computer Science, focusing on "Errortype Metrics in Risk Categorisation Software Fault Prediction through Integration of Formal Methods and Machine Learning."



Dr Emmanuel Ogunshile is a Senior Lecturer in Computer Science and Chair of the Athena SWAN process at the University of the West of England (UWE), Bristol, UK. He is actively engaged in innovative teaching, research, scholarship, and administration within Computer Science and Software Engineering. From 2nd September 2013 to 31st August 2014, Dr Ogunshile worked at Loughborough University, UK, as a Senior Research Fellow on a prestigious EPSRC and Jaguar Land Rover Collaborative Research Project, valued at over £12 million, involving six major UK universities. Prior to this, from 1st September 2011 to 31st August 2013, he was a Research Fellow at the Surrey Space Centre, University of Surrey, UK. Dr Ogunshile is a dedicated British Computer Scientist, Software Engineer, System Designer, Scientific Inventor, and professional with a strong drive for achieving goals. He has a wide range of experience, having worked successfully as a Software Engineer, Intranet Manager, Systems Manager, Teaching Assistant, and Senior Research Scientist in Software and Systems Engineering. With over 15 years of professional experience, Dr Ogunshile's expertise spans analysis and design, development, service, testing, technical writing, user training, team leadership, and negotiation. He advocates for collaboration and the adoption of new technologies to enhance information sharing and improve software usability, focusing on user-centric approaches to solve real-world problems. Dr Ogunshile holds an MSc(Eng) in Advanced Software Engineering (2005), a BEng(Hons) in Software Engineering (2003), and a PhD in Computer Science (2011), all from the University of Sheffield, England, UK, a member of the prestigious Russell Group of researchintensive universities. His PhD thesis, titled *"A Machine with Class: A Framework for Object Generation, Integration and Language Authentication (FROGILA)"*, explored innovative approaches in software engineering.

Dr Mehmet E. Aydin is a Senior Lecturer in Computer Science at the University of the West of England (UWE). He joined the Computer Science and Creative Technologies (CSCT) department in January 2015. Before this, he held academic and research positions at various universities, including the University of Bedfordshire, London South Bank University, and the University of Aberdeen. Dr Aydin has led numerous research projects as both Co-Investigator (CoI) and Principal Investigator (PI) and has published over 100 articles in international peer-reviewed journals and conferences. He serves on the editorial boards of several international peerreviewed journals and is an active committee member for various international conferences. His research interests include machine learning, multi-agent systems, planning, and scheduling. Dr Aydin is a member of the EPSRC Review College, a senior member of IEEE and ACM, and a Fellow of the Higher Education Academy.