A novel software fault prediction approach to predict error-type proneness in the Java programs using Stream X-Machine and machine learning

Author
Organisation
City
Email

Author
Organisation
City
Email

Author
Organisation
City
Email

Abstract—**Software fault prediction makes software quality assurance process more efficient and economic. Most of the works related to software fault prediction have mainly focused on classifying software modules as faulty or not, which does not produce sufficient information for developers and testers. In this paper, we explore a novel approach using a streamlined process linking Stream X-Machine and machine learning techniques to predict if software modules are prone to having a particular type of runtime error in Java programs. In particular, Stream X-Machine is used to model and generate test cases for different types of Java runtime errors, which will be employed to extract error-type data from the source codes. This data is subsequently added to the collected software metrics to form new training data sets. We then explore the capabilities of three machine learning techniques (Support Vector Machine, Decision Tree, and Multi-layer Perceptron) for error-type proneness prediction. The experimental results showed that the new data sets could significantly improve the performances of machine learning models in terms of predicting error-type proneness.**

Keywords-component; software fault prediction; Stream X-Machine; error-type proneness prediction;

I. INTRODUCTION

Software quality assurance (SQA), which includes formal code inspections, code walkthroughs, software testing, validation, verification, and software fault prediction, ensures the desired software quality at a lower cost by monitoring and controlling the Software Development Life Cycle (SDLC) [1]. However, complete testing of a software system is practically not possible as it consumes an enormous amount of time and resources [2] [3]. Also, faults are not uniformly distributed among software modules, which makes it less efficient when spending the same amount of testing resources and efforts to every module of the system under test (SUT). Therefore, software fault prediction (SFP) comes to solve this problem. SFP aims to economically optimise the allocation of limited SQA resources with prior prediction of the fault-proneness of software modules/classes. For instance, if there are only 25% resources available, the prior knowledge of the more vulnerable areas will help testers/developers prioritise the available resources on fixing the modules/classes that are more prone to faults. Hence, within a limited time and budget, a robust software can still be produced. Over the last three decades, the use of SFP techniques to identify faulty software modules as early as possible within the SDLC has gained considerable attention from researchers and software developers.

According to Rathore and Kumar [1], the definition of software fault proneness is very ambiguous and can be measured in different ways since a fault can happen in any phase of the SDLC and some faults remain undetected during the testing phase and forwarded to regular use in the field. Also, most of the SFP approaches are based on the binary-class classification which predicts whether a software module is fault prone or not fault prone [1] [2]. However, this approach provides an ambiguous picture of fault prediction because some modules are indeed more fault-prone and thus, require more attention than the others [1]. In reality, it would be more beneficial for software testers or analysts to focus on more severe areas of the system if the SFP models can provide more information about the faultiness of the software modules such as the number of faults in a module, ranking of modules fault-wise, severity of a fault, etc. [1] [2] [4].

Also, in their studies, Menzies et al. [4] and Rathore and Kumar [1] pointed out that the techniques/approaches used for SFP have hit the "performance ceiling". Thus, simply applying different or better techniques will not guarantee an improved performance. In order to achieve better prediction performance, Menzies et al. [5] suggested the use of additional information when building SFP models while Rathore and Kumar [1] recommended researchers to consider new approaches for SFP.

Currently, only a few researchers have paid attention to predicting the number of faults and the severity of faults in the software modules. In [2] [6] [7], the authors presented their SFP approaches with the severity ranking of the software modules and the number of defects taken into consideration. Some SFP studies demonstrated that a few numbers of modules contain most of the faults in the system. For instance, Ostrand et al. [8] proposed a study to detect the number of faults in top 20% of the files. However, to the best of our knowledge, *there have been no SFP approaches that provide information about the **types of faults** existing in each software module*. Therefore, in this paper, we are motivated to employ Stream X-Machine (a formal specification method) and machine learning techniques to propose a novel SFP approach that can predict if a software module is prone to having an error type. As Menzies et al. [5] suggested that researchers should concentrate on finding solutions that work best for the groups of related projects rather than trying to seek general solutions that can be applied to many projects, our approach will primarily focus on runtime error types in the Java programming language (JPL).

The rest of the paper is structured as follows. Section II contains the background research about software fault prediction and Stream X-Machine, respectively. Section III outlines the proposed methodology which illustrates the link between Stream X-Machine and machine learning in the SFP context. Section IV explains how the experiments were set up. Section V consists of the experimental results. Section VI concludes the paper and provides the directions for future work.

II. Background

1. Software fault prediction

Software fault prediction aims to predict the fault-proneness of modules in a given SUT. The process of SFP typically includes training a prediction model using the underlying properties of the software project, and subsequently using the prediction model to predict faults for unknown software projects. Figure 1 illustrates an overview of the SFP process. Firstly, software project repositories are collected to extract software fault data that is related to the SDLC such as source code, change logs, and fault information. Secondly, independent variables (a.k.a. features or inputs to be analysed) are collected by extracting values of multiple software metrics (e.g., Lines of code – LOC) while dependent variables are the required fault information with respect to the fault prediction (e.g., number of faults, fault prone or not fault prone). Thirdly, statistical and machine learning techniques are used to construct the SFP models. Finally, different measures (e.g., accuracy, precision, recall, F1-score, and Area Under the Curve – AUC) [1] [9] are applied to evaluate the performance of the built SFP model. In the following subsections, we further elaborate software fault data set and prediction techniques used in SFP with related works.

1.1. Software fault data set

Software fault data set plays a role as the training data set and testing data set during SFP process. It consists of three main components: *set of software metrics*, *fault information*, and *meta information* about the software project. In the upcoming subsections, these components are reviewed in detail.

1.1.1. Software metrics

Software metrics are the timely and continuously measured information of different characteristics of a software product [10]. Software metrics can be used to quantitatively analyse and evaluate the quality of a software product [1]. According to Bansiya and Davis [11], each software metric is related to some functional properties such as coupling, cohesion, inheritance, etc., and is used to derive an external quality attribute such as reliability, testability, or fault-proneness. There are various software metrics in the literature such as *Object-Oriented (OO) metrics* with *CK metrics suite* [12], *MOODS metrics suite* [13], *Bansiya metrics suite* [11], etc.; or *Traditional metrics* with *Size metrics* (e.g., Function Points – FP, Source lines of code – SLOC, Kilo-SLOC – KSLOC), *Quality metrics* (e.g., Defects per FP after delivery, Defects per SLOC or KSLOC after delivery), *System complex metrics* [14], *Halstead metrics* [15], etc. According to Rathore and Kumar [1], various works have been conducted to evaluate the capabilities of software metrics for SFP; however, with the availability of the NASA and PROMISE data repositories, many researchers have started to perform their studies using open-source software projects (OSS). The benefit of using OSS is that it enables anyone to replicate the study and verify its findings.

Based on the study of Rathore and Kumar [1], some observations drawn from the software metrics literature are as follows:

- The metrics that perform well in one environment may not perform similarly in another (e.g., open-source environment vs. commercial environment).
- Most of the studies confirmed that OO metrics (e.g., coupling between objects – Coupling Between Objects – CBO, Response for a Class – RFC, and Weighted Method Count – WFC) are the best predictors of faults.
- Many studies have reported the positive correlation between size metric (e.g., Lines of Code – LOC) and fault proneness.

1.1.2. Project's fault information

The fault information indicates how faults are recorded in a software module and their severity levels. Fault data is collected and recorded in an associated database during requirements, design, development, and in various testing phases of the software project [16]. According to Radjenovic et al. [17], there are three fault data repositories that can be used for SFP including *Private/commercial*, *Partially public/freeware*, and *Public* (e.g., NASA and PROMISE repositories). Some of the fault data sets included information on both the number of faults and severity of faults (e.g., KC1, KC2, KC3, PC4, and Eclipse 2.0, 2.1, 3.0, etc. from the PROMISE data repository) [1], which makes it easier for software engineers to focus their testing efforts on the most sever modules first or to allocate the testing resources optimally [18].

1.1.3. Meta information about project

Meta information about the software project contains contextual information about various characteristics (properties) of that project such as the domain of software development, the number of revisions, etc. [1]. According to Hall et al. [19], the current knowledge about the influence of context variables on the SFP models is still limited; therefore, most of the studies did not pay much attention on the context variables before building the SFP models. Some of the basic contextual variables/factors that are applied in SFP are *Source of Data*, *Maturity of the System*, *Size*, *Application Domain*, and *The Granularity of Prediction*. In their systematic literature review, Hall et al. [19] analysed 19 papers and figured out that context variables affect the performance of SFP model. Also, they found that large-sized software projects tend to have higher probability of fault detection. Additionally, the maturity of the system, the programming language used, or the granularity level of prediction has little or no impacts on the model's performance.

1.1.4. Data quality

According to Rathore and Kumar [1], the quality of SFP models highly depends on the quality of the fault data set. Public data sets (e.g., NASA and PROMISE data repositories), which are typically used in SFP studies, may deteriorate the performance of the classifiers as they may contain irrelevant or unnecessary information. From their study, Rathore and Kumar [1] found evidence that data quality issues have not been handled adequately in many SFP approaches. Therefore, the performances of the learners are not up to the mark. According to Gray et al. [20], there are a number of quality issues associated with software fault data sets that researchers need to properly handle before using them to construct SFP models.

- *Outlier*: Outliers are the data points that do not meet the general behaviour of the data [1]. Outliers are essential in SFP as they may indicate faulty modules. Therefore, arbitrarily removing outliers can potentially lead to insignificant results.
- *Missing value*: Values that are left blank in the data set. According to Gray et al. [21], some prediction techniques can automatically deal with missing values and no special care is required.
- *Repeated value*: Two or more attributes have the same values for each instance. Gray et al. [21] suggested removing one of the attributes so that the values are only represented once in the data set.
- *Redundant and irrelevant value*: Same features (attributes) describe multiple modules with the same class label [1]. These data points are problematic in the context of SFP. Therefore, Gray et al. [21] suggested that the classifiers should be tested upon such data points independently of those used during training [1]. This issue needs to be addressed before building any prediction model.
- *Class imbalance*: Certain types of instances (minor class) are mostly dominated in the data set by the other types of instances (major class) [1]. In such cases, the classifiers may have biases towards the instances of the major class. Therefore, poor results can be produced for the minor class instances [22].
- *Data shift problem*: Data shifting is a problem where the joint distribution of the training data is different from the distribution of the testing data.
- *High dimensionality of data*: The data set is stuffed with unnecessary features. According to Gao et al. [23], higher dimensional data can potentially lead to lower classification accuracy, higher computational cost, and higher memory usage.

To sum up, this subsection has provided an overview of the software fault data set used in most of the SFP approaches by outlining its main components with related works and figuring out the potential quality issues that researchers need to pay attention to when using public data sets. Also, it can be seen that the existing fault data sets do not contain information about error types, which makes it impossible for the SFP models to predict the proneness of error types in software modules.

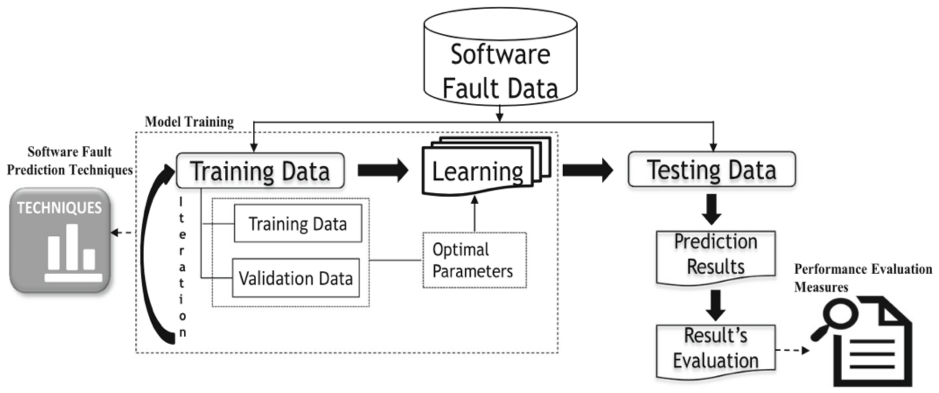


Figure 1. Software fault prediction process [1]

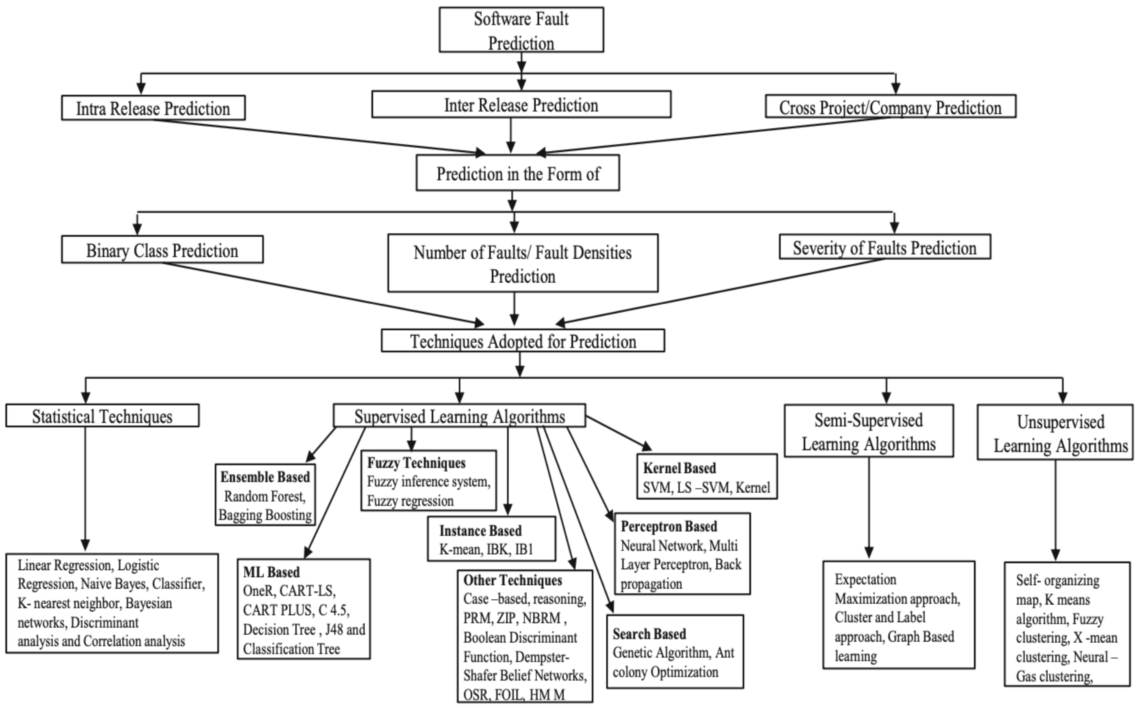

Software Fault Prediction
Intra Release Prediction
Inter Release Prediction
Cross Project/Company Prediction
Prediction in the Form of
Binary Class Prediction
Number of Faults/ Fault Densities Prediction
Severity of Faults Prediction
Techniques Adopted for Prediction
Statistical Techniques
Supervised Learning Algorithms
Semi-Supervised Learning Algorithms
Unsupervised Learning Algorithms
Ensemble Based Random Forest, Bagging Boosting
Fuzzy Techniques Fuzzy inference system, Fuzzy regression
Kernel Based SVM, LS –SVM, Kernel
Instance Based K-mean, IBK, IB1
Perceptron Based Neural Network, Multi Layer Perceptron, Back propagation
Linear Regression, Logistic Regression, Naive Bayes, Classifier, K- nearest neighbor, Bayesian networks, Discriminant analysis and Correlation analysis
ML Based OneR, CART-LS, CART PLUS, C 4.5, Decision Tree , J48 and Classification Tree
Other Techniques Case –based, reasoning, PRM, ZIP, NBRM , Boolean Discriminant Function, Dempster-Shafer Belief Networks, OSR, FOIL, HM M
Search Based Genetic Algorithm, Ant colony Optimization
Expectation Maximization approach, Cluster and Label approach, Graph Based learning
Self- organizing map, K means algorithm, Fuzzy clustering, X -mean clustering, Neural –Gas clustering,

Figure 2. Taxonomy of software fault prediction techniques [1]

2. *Methods to build SFP models*

In the literature, there is a wide range of machine learning techniques for SFP. Based on their systematic review, Rathore and Kumar [1] summarised various schemes used for SFP as in Figure 2. Additionally, according to the study conducted by Malhotra [9], the most frequently used machine learning techniques for SFP have been C4.5 in Decision Tree (DT) category (46%), Naïve Bayes (NB) in Bayesian learners (BL) category (74%), Multi-layer Perceptron (MLP) in Neural Networks (NN) category (85%), and Random Forest (RF) in Ensemble learners (EL) category (59%). The five techniques that performed the best in SFP were C4.5, NB, MLP, Support Vector Machine (SVM), and RF. According to [9] and [24], the strengths and weaknesses of these techniques are provided in Table I.

TABLE I. STRENGTHS AND WEAKNESSES OF THE TOP FIVE MACHINE LEARNING TECHNIQUES USED IN SOFTWARE FAULT PREDICTION

Technique	Strengths	Weaknesses
C4.5	- Requires less efforts for data preparation during pre-processing. - Easy to build and apply. - Comprehensive capability.	- A small change in the data can cause a large change in the structure of the model.
Random Forest (RF)	- It can efficiently handle large data and is a consistent performer. - Robust to noisy and missing data. - Fast to train, robust towards parameter settings. - Comprehensive capability.	- Requires much more computational power and resources. - Requires much more time to train compared to C4.5.
Naïve Bayes (NB)	- Robust in nature. - Easy to interpret and construct. - Computationally efficient.	- Does not consider feature correlation. - Implicitly assumes that all the attributes are mutually independent, which is almost impossible in practice. - Unable to discard irrelevant attributes.
Support Vector Machine (SVM)	- Has good tolerance for high-dimensional space and redundant features. - Robust in nature. - Can handle complex functions and non-linear problems. - A small change in the data does not greatly affect the model.	- Could be tricky and complex when handling non-linear data. - Requires a lot of memory when training. - Takes a significant amount of time to train on large data sets. - Difficult to understand and interpret by human beings.

Multi-layer Perceptron (MLP)	- Can infer complex non-linear input/output transformation.	- Requires huge amount of data. - Computationally expensive to train. - The classifier produced is incomprehensive to interpret.

In general, a prediction model is used to predict the fault-proneness of software modules in one of the three categories: binary-class classification of faults, number of faults/fault density prediction, and severity of fault prediction [1]. The use of SFP models for binary-class classification has been investigated by various researchers. The systematic reviews and analysis of some of these studies can be found in [1] [9] [25] [26]. However, there have been very few efforts that focused on predicting the fault density or fault severity of software modules. With the motivation of addressing the lack of information issue in SFP (discussed in Section I), in what follows, we only focus on analysing and critically reviewing the SFP approaches that could provide more useful information (e.g., prediction of number of faults and severity of faults in software modules) for testers and developer.

In 2005, Ostrand et al. [8] proposed an approach for predicting the number of faults and fault density using negative binomial regression (NBR) technique. The study was performed over the code of the file in the current release, and fault and modification history of the file from previous releases. The prediction aimed to identify top 20% of files with the highest percentage of the predicted number of faults. The analysis indicated that NBR-based models could produce accurate results for the number of faults and fault density predictions. However, no comparison and evaluation were provided with respect to the actual value of faults. A similar type of work was also reported in [27]. A few years later, Yu [28] conducted a deeper study to investigate the effectiveness of NBR in the context of Apache Ant software system. The results showed that NBR could not outperform Binary Logistic Regression in predicting fault prone modules. However, the study demonstrated that (1) the performance of forward assessment is better than or at least the same as the performance of self-assessment; and (2) NBR is effective in predicting multiple faults in one module.

In another study, Afzal et al. [29] applied genetic programming (GP) for predicting the number of faults in a given project. The independent variables used to train the model were the weekly fault count data collected from three industrial projects. The empirical results indicated a significant accuracy rate of GP-based model for fault count prediction. Also, in [30], Rathore and Kumar presented an approach for predicting the number of faults using GP over several open-source software projects. The results demonstrated the significant accuracy and completeness of GP-based model in predicting the number of faults in software modules. In [31], Gao et al. presented a comprehensive analysis of five count models including Poisson Regression model (PR) [32], Zero-Inflated Poisson model (ZIP) [33], NBR model, Zero-Inflated Negative Binomial model (ZINB) [34], and Hurdle Regression model (HR) [35]. The results showed that ZINB and HR models produced better prediction accuracy for fault counts. Recently, Rathore and Kumar [2] explored the capability of Decision Tree Regression (DTR) for the number of faults prediction in two different scenarios, intra-release and inter-release predictions for a given software project. Five open-source software projects with their nineteen releases collected from the PROMISE data repository were chosen to perform the experimental study. The results indicated that DTR-based model could produce significant accuracy in both the considered scenarios.

Yang et al. [6] believed that predicting the exact number of faults in a software module is difficult due to noisy data that exists in the fault data set; therefore, the authors introduced a learning-to-rank (LTR) approach to construct the SFP models by directly optimising the ranking performance. The LTR approach has two major benefits which are (1) more robust against noisy data and (2) unlike the other approaches which have to predict the number of faults in the software modules before ranking them, it provides a way to rank the level of severity of the software modules directly. The empirical results showed the effectiveness of the LTR approach for the ranking task.

In general, it can be seen that all of the SFP studies reviewed in this section fall into one of the following three categories: (i) binary-class classification of faults, (ii) number of faults/fault density prediction, and (iii) severity of fault prediction. There are *no approaches that can indicate/identify the types of faults existing in the software modules*. With the lack of useful information for analysts that most existing SFP approaches are facing, we believe that it would be much more useful for testers and/or developers if SFP models can predict the types of faults besides the number of faults in a module or the ranking of the modules fault-wise. The prediction of a fault's type and its location will enable developers to have identify/spot out what the fault is and thus, handle it early. Therefore, our proposed approach differs from these studies in several ways. Firstly, beside using the set of OO metrics recommended in [2] and [9] for constructing the SFP model, we will introduce a data set containing information about the types of runtime errors that can be found in the JPL by using a formal specification method, especially Stream X-Machine, which will be further elaborated in Section II.3. Secondly, in our proposed work, we will explore the capabilities of the best performing machine learning techniques (from the most commonly used ones presented in Table I) for the purpose of predicting if software modules are prone to having a particular type of runtime error.

3. *Stream X-Machine*

A Stream X-Machine (SXM) is a comprehensive and powerful modelling formalism that extends finite state machines with a memory (data) structure, X, and a set of labelled processing functions, Φ, which operate on X [36]. An SXM can potentially model very general systems as the data set X can contain information about the system internal memory as well as different output behaviours. One of the great benefits of using an SXM to specify a system is its associated testing method which was initially developed for deterministic SXM [37] [38] and was further extended to non-deterministic SXM [39] and communicating SXM [40]. Under certain *design-for-test conditions* [37], this method can produce a test suite that can be used to verify the correctness of the implementation under test (IUT), provided that the processing functions of the SXM specification have been correctly implemented [41]. SXM has been applied extensively in various modelling areas (e.g., modelling agents in biology [42]) and automating software testing to evaluate the behaviours of object-oriented systems [40] [43] [44]. The automation helps to reduce human involvement to avoid human errors and increase reliability of the software testing process [36].

Definition: A Stream X-Machine is a tuple:

$Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$
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 φ (processing functions) that map memory-input pairs to output-memory pairs, φ: $M \times \Sigma \rightarrow \Gamma \times M$,
- 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 initial memory respectively.

Intuitively, an SXM can be thought as a finite automaton with the arcs labelled by functions from the type Φ. The automaton $A_Z = (\Phi, Q, F, I, T)$ is called *the associated finite automaton (FA)* of Z and is usually described by a state-transition diagram. An example of a three-state SXM is shown in Figure 3.

In general, from the definition of SXM and the areas where SXM is employed, it can be seen that SXM is not related to SFP. However, in this paper, we introduce a novel SFP where SXM will be applied to model Java runtime errors. The details of the proposed methodology are outlined in Section III.

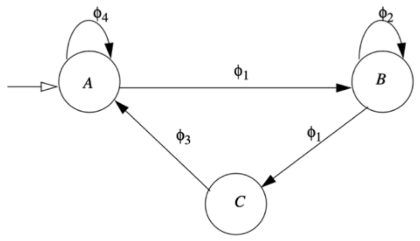


Figure 3. A three-state Stream X-Machine diagram

III. Methodology

In Section II, we reviewed the SFP process in general, machine learning techniques that are used in SFP, and SXM in the context of automating software testing. In this section, we present our proposed methodology which highlights the link between SXM and machine learning in the context of SFP. There are 5 essential steps which are presented as follows:

- **Step 1: Create and generate test cases for JPL runtime errors using SXM** (Figure 4).
  - In particular, a runtime error might occur when one object operates an action on another object. Therefore, the general equation is as follows:

    $$\boldsymbol{A}\ operates\ on\ \boldsymbol{B} \quad (1)$$

    Where:
    - **A** and **B** are either literals (e.g., literal string, literal integer, etc.) or references (e.g., string reference, integer reference, object, etc.)
    - **"operates on"** denotes any action that A acts on B (e.g., Number A divides Number B → *ArithmeticException error might occur*; Array A accesses Bth element of itself → *IndexOutOfBoundsException error might occur*, etc.)
  - With the information about A, B, and "operates on" from equation (1), we can determine some features of an error such as what the error is, how it happens, and in which context it happens. Therefore, we represent equation (1) as a SXM for a runtime error. This runtime error SXM is also known as an Error Specification Machine (ESM). Figure 5 illustrates the ESM's state-transition diagram with five states (*state 1*, *state 2, state 3*, *state 4*, and *state 5*) and four processing functions (*getA*, *getOperation*, *getB*, and *getErrorLabel*).
  - SXM is used to model the formal specification for each Java runtime error (a.k.a. ESM) from the ESM's state-transition diagram. Subsequently, the associated SXM testing method is applied to generate cases for each ESM. An example of the test cases for *ArithmeticException* is given in Figure 6.
- **Step 2: Extract ESM values for different types of runtime errors from software modules.**
  - In a software module, an ESM value for a runtime error (e.g., *IndexOutOfBoundsException*) is the sum of all the lines of code that have the pattern "**A operates on B**" and that pattern matches one or more test cases (generated from Step 1) of that error. An example of how ESM values can be extracted is provided in Figure 9.

- **Step 3: Extract OO software metrics from software modules.**
  - The metrics we collect include Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), Lack of Cohesion in Methods (LCOM), Number of Children (NOC), Response for a Class (RFC), Weighted Methods per Class (WMC), Lines of Code (LOC), and Comment Lines of Code (CLOC).

- **Step 4: Create new training data sets.**
  - Combine software metrics with the ESM value of a runtime error to create a new training data set of the error. This means that for each particular runtime error, there is a corresponding training data set that includes the ESM value of that error, alongside with the software metrics. Figure 7 illustrates an overview of the data set for *IndexOutOfBoundsException*.

  - In each data set, the independent variables are the software metrics and the ESM value of the error. The dependent variable (a.k.a. result or outputs) has the value of Yes or No which indicates if the software module has the error or not. A sample data set for *IndexOutOfBoundsException* is provided in Figure 8.

- **Step 5: Explore the performances of different machine learning algorithms on the new training data sets.**
  - Accuracy and F1-score are used to evaluate the performances of the trained models.

    - Accuracy score indicates the fraction or the count of correct predictions [45].
    - F1-score is a weighted average between precision and recall [45]. F1-score's best value is 1 and worst value is 0. The formula for the F1-score is as follow:

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

To sum up, in this proposed methodology, SXM is used to represent formal specifications of different Java runtime errors (a.k.a. ESM) and generate test cases for these ESMs, which will be used to extract ESM values from Java source codes. ESM values will then be combined with the existing OO software metrics to construct new SFP data sets which contain information about error types. Finally, machine learning algorithms are employed to train on these new data sets to evaluate their effectiveness.

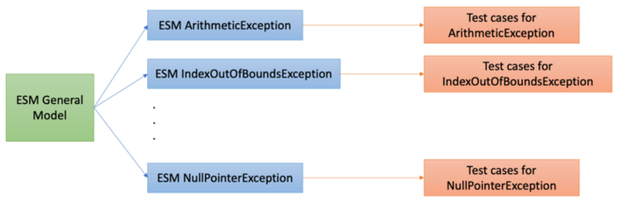

ESM General Model
ESM ArithmeticException
ESM IndexOutOfBoundsException
ESM NullPointerException
Test cases for ArithmeticException
Test cases for IndexOutOfBoundsException
Test cases for NullPointerException

Figure 4. Process of creating and generating test cases for JPL runtime errors using SXM

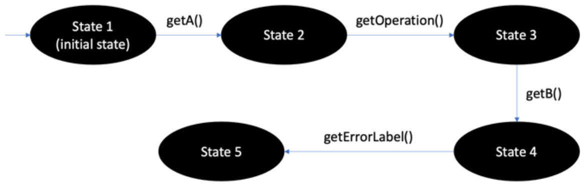
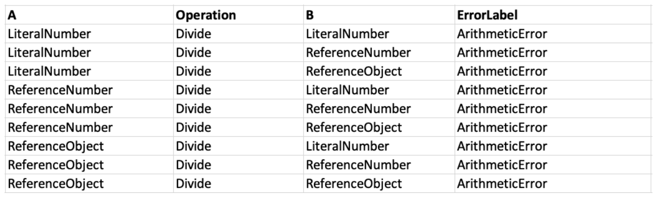


Figure 5. ESM's state-transition diagram



A	Operation	B	ErrorLabel
LiteralNumber	Divide	LiteralNumber	ArithmeticError
LiteralNumber	Divide	ReferenceNumber	ArithmeticError
LiteralNumber	Divide	ReferenceObject	ArithmeticError
ReferenceNumber	Divide	LiteralNumber	ArithmeticError
ReferenceNumber	Divide	ReferenceNumber	ArithmeticError
ReferenceNumber	Divide	ReferenceObject	ArithmeticError
ReferenceObject	Divide	LiteralNumber	ArithmeticError
ReferenceObject	Divide	ReferenceNumber	ArithmeticError
ReferenceObject	Divide	ReferenceObject	ArithmeticError

Figure 6. Example test cases for *ArithmeticException*

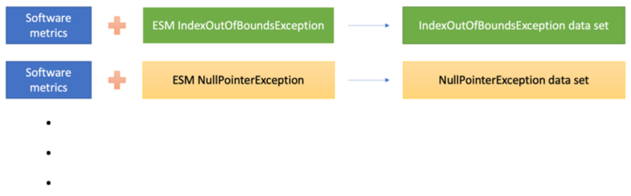
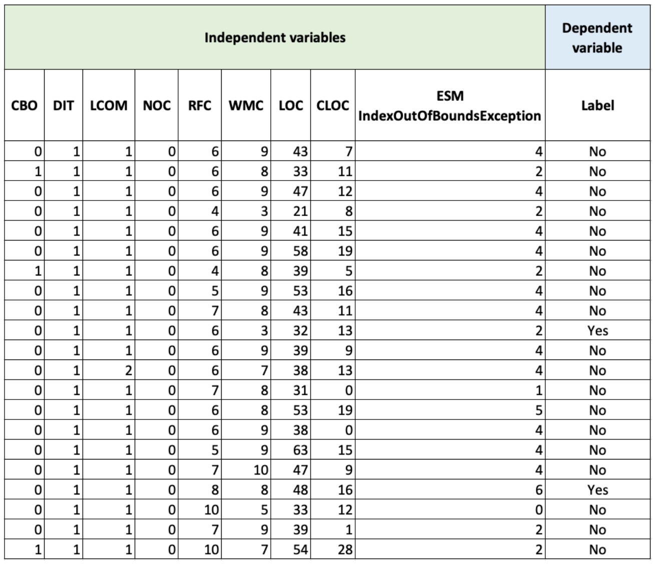
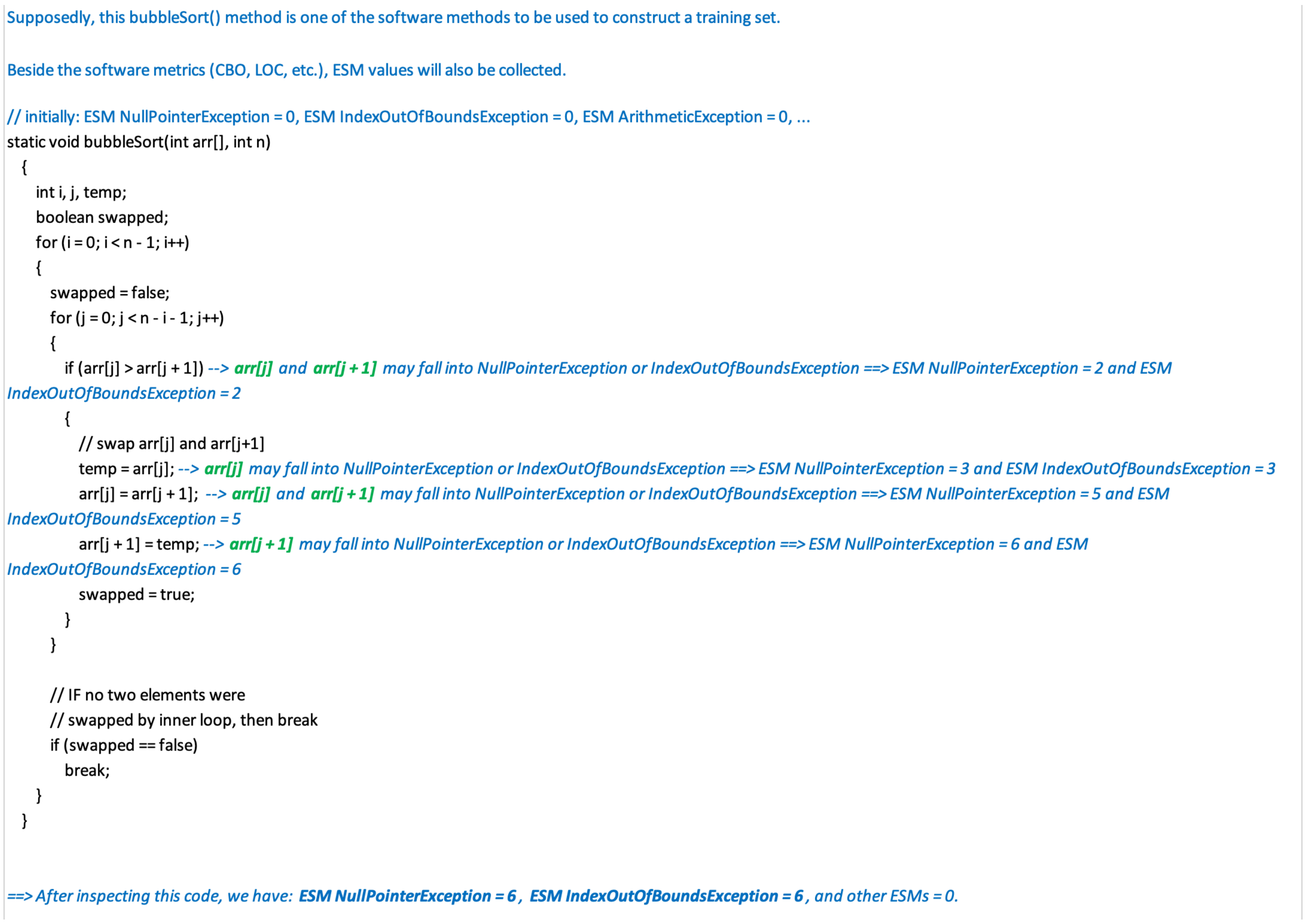

Software metrics
ESM IndexOutOfBoundsException
IndexOutOfBoundsException data set
Software metrics
ESM NullPointerException
NullPointerException data set

Figure 7. Overview the training data sets corresponding to different types of runtime errors



Independent variables									Dependent variable
CBO	DIT	LCOM	NOC	RFC	WMC	LOC	CLOC	ESM IndexOutOfBoundsException	Label
0	1	1	0	6	9	43	7	4	No
1	1	1	0	6	8	33	11	2	No
0	1	1	0	6	9	47	12	4	No
0	1	1	0	4	3	21	8	2	No
0	1	1	0	6	9	41	15	4	No
0	1	1	0	6	9	58	19	4	No
1	1	1	0	4	8	39	5	2	No
0	1	1	0	5	9	53	16	4	No
0	1	1	0	7	8	43	11	4	No
0	1	1	0	6	3	32	13	2	Yes
0	1	1	0	6	9	39	9	4	No
0	1	2	0	6	7	38	13	4	No
0	1	1	0	7	8	31	0	1	No
0	1	1	0	6	8	53	19	5	No
0	1	1	0	6	9	38	0	4	No
0	1	1	0	5	9	63	15	4	No
0	1	1	0	7	10	47	9	4	No
0	1	1	0	8	8	48	16	6	Yes
0	1	1	0	10	5	33	12	0	No
0	1	1	0	7	9	39	1	2	No
1	1	1	0	10	7	54	28	2	No

Figure 8. Sample data set for *IndexOutOfBoundsException*



Supposedly, this bubbleSort() method is one of the software methods to be used to construct a training set.

Beside the software metrics (CBO, LOC, etc.), ESM values will also be collected.

```
// initially: ESM NullPointerException = 0, ESM IndexOutOfBoundsException = 0, ESM ArithmeticException = 0, ...
static void bubbleSort(int arr[], int n)
  {
    int i, j, temp;
    boolean swapped;
    for (i = 0; i < n - 1; i++)
    {
      swapped = false;
      for (j = 0; j < n - i - 1; j++)
      {
        if (arr[j] > arr[j + 1]) --> arr[j] and arr[j + 1] may fall into NullPointerException or IndexOutOfBoundsException ==> ESM NullPointerException = 2 and ESM
IndexOutOfBoundsException = 2
        {
          // swap arr[j] and arr[j+1]
          temp = arr[j]; --> arr[j] may fall into NullPointerException or IndexOutOfBoundsException ==> ESM NullPointerException = 3 and ESM IndexOutOfBoundsException = 3
          arr[j] = arr[j + 1]; --> arr[j] and arr[j + 1] may fall into NullPointerException or IndexOutOfBoundsException ==> ESM NullPointerException = 5 and ESM
IndexOutOfBoundsException = 5
          arr[j + 1] = temp; --> arr[j + 1] may fall into NullPointerException or IndexOutOfBoundsException ==> ESM NullPointerException = 6 and ESM
IndexOutOfBoundsException = 6
          swapped = true;
        }
      }

      // IF no two elements were
      // swapped by inner loop, then break
      if (swapped == false)
        break;
    }
  }
```

==> After inspecting this code, we have: ***ESM NullPointerException = 6***, ***ESM IndexOutOfBoundsException = 6***, *and other ESMs = 0.*

Figure 9. Example of how to extract ESM values from source codes

IV. Experimental setup

This section provides the details of the whole experimental setup including the software fault data set used in this study. Also, we present the objectives of the experiments, what machine learning techniques were used and what tools were applied to train SFP models.

1. *Experiement objectives*

The experiment was conducted with three objectives:

i) Examine the correlation between the ESM value of each type of runtime error and the software metrics.
ii) Investigate if the software metrics alone can be used to predict the error-type proneness.
iii) Investigate if the software metrics and the ESM value together can be used to predict the error-type proneness.

2. *Software fault data set*

The data set of the faults used in this study was taken from the Java source codes created by a group of volunteers (not expert programmers) who have been asked to create a solution for a given problem. The reason we did not use public data sets (e.g., NASA and PROMISE data repositories) is that the source codes of the projects featured in these data sets are not available for us to extract the ESM values.

We used a plugin from IntelliJ IDEA [46] to obtain the software metrics from the source codes. Also, due to not having a dedicated tool to extract the ESM values, we did it manually by examining each line of code carefully.

3. *SXM tool*

Among the SXM tools (e.g., SXMtool [47], *JSXM* [36], and T-SXM [48]), in this study, we employed T-SXM as the SXM tool for modelling and generating test cases for the ESMs. The reason for using this tool is (1) it can automatically generate test cases for a specification and (2) it allows using Java as the language for modelling SXM specification while the other tools require the specification to be written using the X-Machine Design Language (XMLD) [49] which is not comprehensive when logical code is required within the model's description.

4. *SFP models and evaluation measures*

The experiment in this study was conducted using *scikit-learn* [45], which is an open-source library for predictive data analysis. The three machine learning algorithms we used to build our SFP models were Support Vector Machine (SVM), Decision Tree (DT), and Multi-layer Perceptron (MLP).

V. Experimental results

In this section, we present the experimental results corresponding to the objectives outlined in Section IV.2.

1. *The correlation between the ESM value and the software metrics*

From Table II and Table III, it can be seen that the ESM values of *IndexOutOfBoundsException* and *NullPointerException* have no correlations with the software metrics; hence, software metrics cannot be used to predict ESM values.

TABLE II. Correlations between ESM IndexOutOfBoundsException and software metrics

	ESM IndexOutOfBoundsException
CBO	-0.36
RFC	-0.25
WMC	0.52
LOC	0.58
CLOC	0.27

TABLE III. Correlations between ESM NullPointerException and software metrics

	ESM NullPointerException
CBO	0.02
RFC	0.18
WMC	0.49
LOC	0.63
CLOC	0.47

2. *Investigate if the software metrics alone can be used to predict the error-type proneness*

In this experiment, for *IndexOutOfBoundsException*, we trained three models with three machine learning algorithms (SVM, DT, and MLP) on the training data set that only consists of the software metrics (we removed the ESM IndexOutOfBoundsException from the independent variables). The F1-scores and accuracies of these models are shown in Table IV. It can be seen that SVM can potentially be used to predict *IndexOutOfBoundsException* proneness by just using software metrics, while DT and MLP did not perform well with low F1-scores and accuracy rates.

TABLE IV. F1-scores and accuracies of SFP models trained on the data set with only software metrics

	SVM	**DT**	**MLP**
F1-score	0.89	0.57	0.75
Accuracy	0.80	0.40	0.60

3. *Investigate if the software metrics and the ESM value together can be used to predict the error-type proneness*

In this experiment, for *IndexOutOfBoundsException*, we trained three models with three machine learning algorithms

(SVM, DT, and MLP) on the full training data set. The F1-scores and accuracies of these models are shown in Table V. By comparing the results in Table IV and Table V, it can be seen that the presence of the ESM IndexOutOfBoundsException in the training set has significantly improved the performances of the models, especially DT and MLP, in terms of predicting *IndexOutOfBoundsException* proneness. Therefore, in the case of *IndexOutOfBoundsException*, the ESM value plays a significant role in the training data set to help improve the performances of the machine learning algorithms.

TABLE V. F1-SCORES AND ACCURACIES OF SFP MODELS TRAINED ON THE DATA SET WITH BOTH SOFTWARE METRICS AND ESM VALUE

	SVM	**DT**	**MLP**
F1-score	0.97 (increased by 8.9%)	0.89 (increased by 56%)	0.91 (increased by 21%)
Accuracy	0.91 (increased by 14%)	0.80 (increased by 100%)	0.86 (increased by 43%)

4. *Limitations and threats to validity*

Experiments are always associated with a set of threats which could potentially hinder the findings. Here, we present some limitations and threats to the validity of the study. The first concern is the size of the software fault data set. We did not use public data sets like the other SFP approaches; instead, we established the data set by collecting source codes from voluntarily non-expert programmers. Therefore, the size of our data set is much smaller than the public data sets such as PROMISE and NASA. We performed the experiments with 8 software metrics which are measured at the class level (not at the method level). Software metrics at method level (e.g., Halstead metrics [15]) should also be used. We have built and evaluated SFP models over small projects created by non-expert programmers while the other industrial projects may possess different fault patterns. Therefore, it is recommended that one should take care of the underlying pattern of software system before applying the proposed approach. Also, in the experiments, we extracted the ESM values from the source codes manually which could potentially lead to some unwanted mistakes.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel approach for predicting error-type proneness in software modules using a streamlined process linking SXM and machine learning techniques. We assessed the performances of SVM, DT, and MLP in two scenarios: data set with only software metrics and data set with both software metrics and the ESM value. The experimental results showed that the trained models (especially DT and MLP) performed much better on the data set with the ESM value. However, there are still a number of limitations and threats (discussed in Section IV.4) that can potentially affect the findings of the study.

In the future, we will develop a tool which has the capability of extracting ESM values automatically from the source codes, which will help us speed up the process of collecting data and reducing mistakes. We will use more software metrics from industrial projects with larger sizes to improve the generalisation of the results and consolidate the findings of our investigation.

REFERENCES

[1] S. S. Rathore and S. Kumar, “A study on software fault prediction techniques,” *Artificial Intelligence Review,* vol. 51, no. 2, pp. 255-327, 2019.

[2] S. S. Rathore and S. Kumar, “A Decision Tree Regression based Approach for the Number of Software Faults Prediction,” *ACM SIGSOFT Software Engineering Notes,* vol. 41, no. 1, pp. 1-6, 2016.

[3] R. Hierons, K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause and G. Lüttgen, “Using formal specifications to support testing,” *ACM Computing Surveys (CSUR),* vol. 41, no. 2, pp. 1-76, 2009.

[4] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic and Y. Jiang, “Implications of ceiling effects in defect predictors,” *Proceedings of the 4th international workshop on Predictor models in software engineering,* pp. 47-54, 2008.

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

[6] X. Yang, K. Tang and X. Yao, “A Learning-to-Rank Approach to Software Defect Prediction,” *IEEE Transactions on Reliability,* vol. 64, no. 1, pp. 234-246, 2014.

[7] Z. Yan, X. Chen and P. Guo, “Software defect prediction using fuzzy support vector regression,” *International symposium on neural networks,* pp. 17-24, 2010.

[8] T. Ostrand, E. Weyuker and R. Bell, “Predicting the Location and Number of Faults in Large Software Systems,” *IEEE Transactions on Software Engineering,* vol. 31, no. 4, pp. 340-355, 2005.

[9] R. Malhotra, “A systematic review of machine learning techniques for software fault prediction,” *Applied Soft Computing,* vol. 27, pp. 504-518, 2015.

[10] P. Goodman, Practical Implementation of Software Metrics, London: McGraw-Hill, 1993.

[11] J. Bansiya and C. Davis, “A hierarchical model for object-oriented design quality assessment,” *IEEE Transactions on software engineering,* vol. 28, no. 1, pp. 4-17, 2002.

[12] S. Chidamber and C. Kemerer, “A metrics suite for object-oriented design,” *IEEE Transactions on software engineering,* vol. 20, no. 6, pp. 476-493, 1994.

[13] R. Harrison, S. Counsell and R. Nithi, “An evaluation of the mood set of object-oriented software metrics,” *IEEE Transactions on Software Engineering,* vol. 24, no. 6, pp. 491-496, 1998.

[14] “A complexity measure,” *IEEE Transactions on software Engineering,* vol. 4, pp. 308-320, 1976.

[15] M. Halstead, Elements of software science, New York: Elsevier, 1977.

[16] M. Jureczko, “Significance of different software metrics in defect prediction,” *Software Engineering: An International Journal,* vol. 1, no. 1, pp. 86-95, 2011.

[17] D. Radjenović, M. Heričko, R. Torkar and A. Živkovič, "Software fault prediction metrics: a systematic literature review," *Information and software technology,* vol. 55, no. 8, pp. 1397-1418, 2013.

[18] P. Shanthi and K. Duraiswamy, "An empirical validation of software quality metric suites on open source software for fault-proneness prediction in object oriented systems," *European journal of Scientific Research,* vol. 5, no. 2, pp. 168-181, 2011.

[19] T. Hall, S. Beecham, D. Bowes, D. Gray and S. Counsell, "A systematic review of fault prediction performance in software engineering," *Software Engineering Softw. Eng. IEEE Trans,* p. 1, 2011.

[20] T. Hall, S. Beecham, D. Bowes, D. Gray and S. Counsell, "The misuse of the nasa metrics data program data sets for automated software defect prediction," *A Systematic Review of Fault Prediction Performance in Software Engineering Softw. Eng. IEEE Trans,* p. 1, 2011.

[21] D. Gray, D. Bowes, N. Davey, Y. Sun and B. Christianson, "The misuse of the nasa metrics data program data sets for automated software defect prediction," *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011),* pp. 96-103, 2011.

[22] J. Moreno-Torres, T. Raeder, R. Alaiz-Rodríguez, N. Chawla and F. Herrera, "A unifying view on dataset shift in classification," *Pattern recognition,* vol. 45, no. 1, pp. 521-530, 2012.

[23] K. Gao, T. Khoshgoftaar, H. Wang and N. Seliya, "Choosing software metrics for defect prediction: an investigation on feature selection techniques," *Software: Practice and Experience,* vol. 41, no. 5, pp. 579-606, 2011.

[24] N. Kumar, "The Professionals Point," 2019. [Online]. Available: http://theprofessionalspoint.blogspot.com. [Accessed 06 02 2021].

[25] H. Alsolai and M. Roper, "A systematic literature review of machine learning techniques for software maintainability prediction," *Information and Software Technology,* vol. 119, 2020.

[26] A. Kumar and A. Bansal, "Software Fault Proneness Prediction Using Genetic Based Machine Learning Techniques," *2019 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU),* pp. 1-5, 2019.

[27] T. Ostrand, E. Weyuker and R. Bell, "Where the bugs are," *ACM SIGSOFT Software Engineering Notes,* vol. 29, no. 4, pp. 86-96, 2004.

[28] L. Yu, "Using negative binomial regression analysis to predict software faults: A study of apache ant," 2012.

[29] W. Afzal, R. Torkar and R. Feldt, "Prediction of fault count data using genetic programming," *2008 IEEE International Multitopic Conference,* pp. 349-356, 2008.

[30] S. Rathore and S. Kumar, "Predicting number of faults in software system using genetic programming," *SCSE,* pp. 303-311, 2015.

[31] K. Gao and T. Khoshgoftaar, "A comprehensive empirical study of count models for software fault prediction," *IEEE Transactions on Reliability,* vol. 56, no. 2, pp. 223-236, 2007.

[32] P. Consul and F. Famoye, "Generalized Poisson regression model," *Communications in Statistics-Theory and Methods,* vol. 21, no. 1, pp. 89-109, 1992.

[33] M. Xie, B. He and T. Goh, "Zero-inflated Poisson model in statistical process control," *Computational statistics & data analysis,* vol. 38, no. 2, pp. 191-201, 2001.

[34] M. Ridout, J. Hinde and C. Demétrio, "A score test for testing a zero-inflated Poisson regression model against zero-inflated negative binomial alternatives," *Biometrics,* vol. 57, no. 1, pp. 219-223, 2001.

[35] S. Gurmu, "Semi-parametric estimation of hurdle regression models with an application to Medicaid utilization," *Journal of applied econometrics,* vol. 12, no. 3, pp. 225-242, 1997.

[36] D. Dranidis, K. Bratanis and F. Ipate, "JSXM: A tool for automated test generation," *International Conference on Software Engineering and Formal Methods,* pp. 352-366, 2012.

[37] M. Holcombe and F. Ipate, Correct Systems: Building a Business Process Solution, Berlin: Springer, 1998.

[38] F. Ipate and M. Holcombe, "An integration testing method that is proved to find all faults," *Internat. J. Comput. Math,* vol. 63, pp. 159-178, 1997.

[39] F. Ipate and M. Holcombe, "Generating test sequences from non-deterministic generalized stream X-machines," *Formal Aspects of Comput.,* vol. 12, no. 6, pp. 443-458, 2000.

[40] F. Ipate and M. Holcombe, "Testing conditions for communicating stream X-machine systems," *Formal Aspects of Comput.,* vol. 13, no. 6, pp. 431-446, 2002.

[41] F. Ipate and D. Dranidis, "A unified integration and component testing approach from deterministic stream X-machine specifications," *Formal Aspects of Computing,* vol. 28, no. 1, pp. 1-20, 2016.

[42] S. Coakley, R. Smallwood and M. Holcombe, "Using x-machines as a formal basis for describing agents in agent-based modelling," *Simulation Series,* vol. 38, no. 2, p. 33, 2006.

[43] F. Ipate and M. Holcombe, "Testing data processing-oriented systems from stream X-machine models," *Theoretical Computer Science,* vol. 403, no. 2-3, pp. 176-191, 2008.

[44] F. Ipate, M. Gheorghe and M. Holcombe, "Testing(Stream)X-machines," *Applicable Algebra in Engineering, Communication and Computing,* vol. 14, no. 3, pp. 217-237, 2003.

[45] "Scikit-learn," [Online]. Available: https://scikit-learn.org/stable/. [Accessed 07 05 2021].

[46] I. IntelliJ, "The most intelligent Java IDE," JetBrains, 2011. [Online]. Available: https://www. jetbrains. com/idea/.

[47] C. Ma, J. Wu and T. Zhang, "Sxmtool: A tool for stream x-machine testing," *World Congress on Software Engineering,* 2010.

[48] K. Phung and E. Ogunshile, "An algorithm for implementing a minimal stream X-Machine model to test the correctness of a system," *2020 8th International Conference in Software Engineering Research and Innovation (CONISOFT),* pp. 93-101, 2020.

[49] P. Kapeti and P. Kefalas, "A design language and tool for X-machines specification," *Advances in Informatics,* pp. 134-145, 2000.

[50] S. Eilenberg, "Automata, languages and machines," *Academic Press,* vol. A, 1974.

[51] F. Ipate and M. Holcombe, "A method for refining and testing generalized machine specifications," *J. Computer Math,* vol. 68, pp. 197-219, 1998.