Educational Administration: Theory and Practice
2024, 30(4), 9973-9986
ISSN: 2148-2403
https://kuey.net/

Research Article

Machine Learning Model for Automated Software Testing and Defect Analysis in Early Maintenance Applications

Kodanda Rami Reddy Manukonda[1*]

[1*]Test Architect, IBM, Email: reddy.mkr@gmail.com

Citation: Kodanda Rami Reddy Manukonda et al. (2024) Machine Learning Model for Automated Software Testing and Defect Analysis in Early Maintenance Applications, *Educational Administration: Theory and Practice*, *30*(4), 9973-9986
Doi: 10.53555/kuey.v30i4.6130

ARTICLE INFO | **ABSTRACT**

For cognitive operations that statistically resemble human actions, ML solutions are finding their way into an expanding variety of software applications. A lot of human work goes into testing this kind of software by coming up with appropriate picture, text, and voice inputs and then judging if the program is processing them as humans do. When bad behavior is detected, it's not always easy to tell whether it originated in the cognitive ML API itself or in the code that uses it. When organizing software testing, ML classifiers for module fault prediction are helpful. The majority of these studies primarily assess the ML classifier's performance based on its accuracy. The ML model develops bias, however, when the classifier is trained and tested on imbalanced datasets. Hidden accuracy is caused by biased ML models. Within this framework, this research suggests a method to improve ML classifier performance in forecasting software module failures, even when dealing with imbalanced datasets. First, there was a marked improvement in the efficiency of software testing; second, there was a notable improvement in the number of modules that were accurately identified as defective; third, there was an increase in the number of modules that were tested; but, there were also some negative outcomes: a decrease in overall accuracy; an increase in the number of false positives; a decrease in software testing efficiency; an increase in the number of modules that were tested; and an increase in the number of modules that were tested. That is why it is important to think about the trade-off that the recommended strategy imposes when organizing software testing tasks. Lastly, this paper suggests a method to assist managers in handling these trade-offs while taking resource limits into account.

Keywords—software testing, machine learning API; efficiency; efficacy; defect prediction; software testing effort

Introduction

For cognitive duties that statistically resemble human actions, ML solutions are finding their way into an expanding variety of software applications. A lot of human work goes into testing this kind of software by coming up with appropriate picture, text, and voice inputs and then judging if the program is processing them as humans do. When bad behavior is detected, it's not always easy to tell whether it originated in the cognitive ML API itself or in the code that uses it [1]. Learning the scientific foundations of machine learning as well as the many ML APIs is a huge challenge for students and working software engineers interested in ML. Making APIs that are intuitive and simple to use, particularly for newcomers, is a problem that authors aim to solve. Nevertheless, the way this may be accomplished while maintaining expressiveness remains unclear [2]. When you put software through its paces, you're engaging in software testing. There are two ways to test it: manually and automatically. Automated testing is a method for evaluating software that involves creating scripts to run tests automatically. While automated testing may be appropriate for some software development projects, manual testing is still necessary for others [3]. To guarantee the quality of a program while it is being developed, software testing is essential. When a possible defect is discovered, a defect Report is created with details on how to recreate and characterize the problem. Many testing teams in large organizations often open hundreds of BRs per week, all of which must be thoroughly examined and corrected [4]. Maintaining quality standards with frequent product introductions is an ongoing concern for consumer electronic production firms. The purpose of a manufacturing test is to detect and confirm any flaws in the product's functioning. Product recalls are possible outcomes of incomplete testing [5]. Due to the ever-increasing complexity of systems, testing approaches in software engineering are always evolving. Because of this, new approaches to testing have emerged, with the goal of combining AI with software testing instruments. The term "automation testing" describes a set of practices and technologies that generate and run test cases automatically, eliminating the need for humans to do mundane, repetitive operations that are prone to human error [6]. Cloud APIs for ML make it easy for programmers to integrate learning solutions into existing applications. Due to their specific semantics, data needs, and accuracy-performance tradeoffs, ML APIs may be difficult to utilize effectively and efficiently. The use of ML APIs by open-source apps has received less attention than the development of ML APIs and ML cloud services [7]. Since conventional approaches can't keep up with the exponential growth of software systems, testing video games has become an ever more daunting challenge. Due to the high expense associated with the time and effort required for manual testing, this method is rapidly becoming obsolete. Automated testing scripts are cheap, but they don't work in non-deterministic settings, and it's much more difficult to determine when to perform each test. Unfortunately, the existing industry standard and approach for game testing—which involves building scripts and doing manual testing—is about to be replaced [8]. Due to the growing complexity of software systems, testing is now an integral part of software development to guarantee a high-quality end result. The requirement for human participation in manual testing, however, may make the process time-consuming and expensive. This makes it harder to run a great deal of test cases in a limited amount of time, which slows down the process of finding software bugs. However, conventional methods have their limits, and automated testing may cut down on testing time and costs [9]. Automatic software testing has its greatest challenge in the development of test oracles. These days, it's easy to create a ton of test cases for any system and get a ton of coverage, but test cases can only be as good as the test diviners that can identify failed executions [10]. The use of ML libraries is on the rise, particularly in the Python community. For optimal performance and safety, use the most recent versions of these libraries. It takes a lot of effort to change usages of deprecated APIs when upgrading to a newer release of a machine learning library [11]. For some security-related tasks, such software security, cryptography is an essential tool. However, when it comes to encryption and its APIs, the majority of developers just don't know enough. As a result, software programs become vulnerable to exploits and are used incorrectly [12]. ML libraries, frequently developed in Python, have become much more accessible as a result of the proliferation of AI applications. Updates to ML libraries are common, and these updates might deprecate older APIs, therefore it's important for developers to change their use of these libraries accordingly [13]. When compared to traditional computing, quantum computing claims to solve complicated problems much more quickly. Present and future quantum computers do, however, exhibit noise. To the point where it is hard to tell whether a test case failed because of noise or actual errors, noise is an inevitable part of quantum software testing (to obtain confidence in the correctness of quantum software) [14]. For cognitive operations that statistically resemble human actions, ML solutions are finding their way into an expanding variety of software applications. A lot of human work goes into testing this kind of software by coming up with appropriate picture, text, and voice inputs and then judging if the program is processing them as humans do. The question of whether the offender is located within the cognitive ML API or in the code that makes use of the API remains unanswered even after misconduct has been identified [15]. One of the most important steps in creating software is the testing phase. As a rule, developers' mistakes are addressed in subsequent stages of software development. The effect of the flaw is amplified in this way. This is avoidable if software flaws can be foreseen in the early stages of development, when testing resources may be more effectively used. Software modules are categorized as either defect prone or non-defect prone as part of the defect prediction process [16].One of the most important parts of making software is testing it. The feasibility of using conventional manual testing techniques is diminishing as software system complexity rises. One interesting trend in software testing that has been developing recently is the use of AI [17].

Here is the outline for the rest of the paper: Section II presents the literature review on software prediction techniques. Section III discusses the background knowledge and describes the Machine Learning algorithms used. Sections IV and V present the dataset with assessment methodology as well as experimental results, respectively. Section VI discusses the conclusion and future work.

I. RELATED WORK

Machine learning approach for automated testing adoption prediction was created in their study [18]. To discover the connection between components, they utilized the chi-square test, and to build the model, they used the PART classifier. Their suggested model has a 93.1624% accuracy rate. An essential (and costly) process to automate in order to increase the efficiency of testing teams is Escaped Defect Analysis, which is the subject of the study [19]. ML methods are used to automate EDA. According to their research, Escaped Defects occur when one team inadvertently discovers a defect or problem that should have been reported by another. Because they are often associated with testing activity failures, EDs pose a danger when they occur. Software developers often do EDA by hand, reading the content of each BR to determine whether it is an ED. Doing this will take a lot of effort and time. As part of their method, textual operations are used to preprocess the BR's content. Then, an ML classifier is used to adopt a feature representation and return the likelihood of EDA labels. The Motorola Mobility subsidiary Comércio de Produtos Eletrônicos Ltda supplied the dataset of 3767 BRs used in the experiments. In a cross-validation experiment, classifiers were built using several ML methods, and the AUC values were high, often above 0.8. According to the findings of the study cited in [20], CEM organizations may save time and money by using a universal computerized testing system. The purpose of developing a universal hardware interface was to facilitate the connection of UUTs with COTS test equipment. The LabVIEW environment is used to create a machine learning-based software application. Roughly one hundred test locations have had their data gathered. The program uses a common hardware interface to choose drivers for commercial off-the-shelf testing equipment, as well as to connect to UUT and take test measurements at designated test locations. In addition, it uses machine learning to gather data from tests in real-time, analyze it, provide reports and KPIs, and finally, offer suggestions. Additionally, it keeps track of past data in a database for the purpose of bettering production methods. The goal of incorporating a deep reinforcement learning robot into the program's state representation is to enhance generative testing, as described in [21]. This will allow for more efficient and automatically extractable state information from the software being tested. Applying the suggested DeepRNG framework to the Cosmos SDK as a testbed, they demonstrate a statistically significant enhancement in testing of the intricate software library including more than 350,000 lines of code. Anyone may access the DeepRNG framework's source code on the internet. For the goals of structure-based learning as well as hyperparameter optimization, they constructed an AutoML pipeline in [22]. Three primary automated steps make up the pipeline. Using the Kaggle API, the first one retrieves and prepares the dataset from the Kaggle database. To optimize hyperparameters, the second one makes use of the Keras-Bayesian optimizer tuning module. Step three involves using the hyperparameter settings calculated in the previous step to train the ML model. The model is then evaluated on testing data using the Neptune AI. Popular tools like as Git for version control, Google Cloud Platform for virtual machines, Jenkins for server administration, Docker for containerization, and Ngrok for reverse proxy are essential to building a reliable and repeatable machine learning pipeline. The authors of [23] employed two cutting-edge API recommendation methods—BIKER and DeepAPI—to do an empirical investigation on six popular Python-based ML packages. Two key issues contribute to the current methods' poor performance:(1) Python-based ML activities often need lengthy API sequences, and(2) present methods are unable to handle typical API usage habits in Python-based ML programming duties. In light of their research, they recommend FIMAX, a straightforward method based on frequent itemset mining, to improve API recommendation algorithms. This would mean taking advantage of the common API usage data found in SO questions to make existing API recommendation algorithms better at ML programming tasks involving Python. As far as API recommendations go, their testing reveals that FIMAX outperforms current best practices by as much as 54.3% in MRR and 57.4% in MAP. Additional evidence of FIMAX's usefulness for API suggestion is provided by their user survey with fourteen developers. Create an automated method for updating obsolete API use in [24]. In order to learn how to update Python's deprecated ML API usages, researchers first provide an empirical research. Scikit-Learn, TensorFlow, and PyTorch's deprecated APIs (112) are part of the dataset used in the research. Their empirical analysis informs their proposal of MLCatchUp, an automated tool for updating Python deprecated API usages. By comparing the signatures of the deprecated with updated APIs, it can deduce the API migration transformation. In a Domain Specific Language, these changes are articulated. Using a dataset consisting of 267 files and 551 API usages retrieved from public GitHub projects, they assess MLCatchUp. With a flawless score of 80.6%, MLCatchUp is able to identify deprecated API usages in their dataset and update them appropriately. To reduce the impact of noise on quantum program testing outcomes, the authors of [25] suggest a noise-aware method they call QOIN. The QOIN algorithm learns the quantum computer's noise impact and filters it out of the outputs of quantum programs by using machine learning methods (e.g., transfer learning). The filtered results are then fed into test case evaluations, which determine whether a test case operation passes or fails when compared to a test oracle. Authors tested QOIN on a number of different platforms, including IBM's 23 noise models, Google's two publicly accessible noise models, with Rigetti's Quantum Virtual Machine, which has nine actual quantum programs and one thousand simulated ones. The results demonstrate that, for most noise models, QOIN may reduce the noise impact by over 80%. Two big problems with defect prediction are data imbalance and high complexity of the defect datasets; this study tries to address both of these difficulties to a lesser extent [26]. This study employs feature selection methods like ElasticNet, Boruta, Lasso, Ridge, Elastic Feature Elimination, and Correlation-based feature selection to assess a number of software metrics. Decision Trees, the K-Nearest Neighbor, Logistic Regression, Support Combining feature extraction and selection approaches with various machine learning algorithms has allowed for the classification of software modules as either fault prone or non-defect prone. The suggested model takes use of a mix of techniques to reduce dimensions, including Partial Least Square Regression with RFE. To account for the datasets' imbalance, it also incorporates the Synthetic Minority Oversampling Technique. This review article seeks to provide a comprehensive analysis of where software testing with AI stands at the present time [27]. This review will take a look at the methods, tools, and approaches utilized in this field and see how well they work. Using an extensive search string method, the papers chosen for this study were pulled from several research databases. It all started with forty papers culled from various academic libraries. Twenty publications were ultimately chosen for the research after a process of incremental sifting. After reviewing all of the articles, they have concluded that AI (Machine Learning as well as Deep Learning) can effectively automate a number of testing processes, including test case generations, defect forecasting, and test case prioritization. White Box Testing, Metamorphic Testing, Android Testing, and Test Case Validation. Put forth a method for assessing machine learning models based on their properties in [28]. Their method, MLCheck, has two parts: (1) a language for describing properties and (2) a way to generate test cases in a systematic way. There are similarities between the specification language and languages used for property-based testing. Using state-of-the-art verification technology, test case generation constructs test suites systematically, depending on properties, and does not need any extra generator functions from the user. They assess MLCheck by means of three distinct domains' specifications and datasets: software prejudice, learning on knowledge graphs, and security. Their results demonstrate that MLCheck can beat specialized testing methods with the same amount of time spent running, despite its generalizability. The goal of the study is to classify traffic in SDN networks using machine learning models and to compare their efficacy [29]. This is how the "D-ITG" function, part of a basic SDN architecture, came to be. In order to train and test the machine learning models, processing scripts are used to store traffic data. Authors used the most effective supervised model to label the traffic according to the categories produced by the unsupervised model. They selected the group centroids in advance and compared them to results from a novel approach that used randomly generated centroids; this yielded an accuracy of 83.5%—better than in the literature—for such a small number of groups, allowing us to achieve the best possible classification outcome for the unsupervised classification model. Based on five machine learning techniques—artificial neural network, support vector machine, decision trees, k-Nearest Neighbor, and Naïve Bayes Classifiers (NBC)—and six datasets—CM1, KC1, KC2, PC1, JM1, as well as a combined one—the article compares thirty software quality prediction models. These models use static code metrics, specifically McCabe complexity measures, to predict quality. They compare the thirty predictors based on their accuracy, area under the curve, overall receiver operator curve. The findings demonstrate that the ANN method shows promise for reliable software quality prediction across all datasets. Researchers in [31] opted to do a literature study on software bug forecasting with machine learning to help us better understand how to build the prediction model. They do more than just look for examples of machine learning methods; they also evaluate the datasets, measurements, and performance metrics that were utilized to build the models. They identified six distinct kinds of machine learning approaches and reduced the number of key research to 31. The prediction model's most used metrics are object-oriented ones, and two open data sets are the most utilized ones overall. When assessing the efficacy of a model, it is common practice to use both numerical and visual metrics. Using commonplace components like open-source software and sensors found in homes or businesses, the study in [32] lays out a cheap prototype. To intelligently monitor river segments, this prototype may be modified to accommodate various sensors and monitoring needs. To get the most out of the sensors, we'll employ sensor fusion. Using principal component analysis, they find that six dimensions account for 97% of the variation, whereas four dimensions account for 87%. More testing will shed light on this, but for now, test datasets are modest. An integral part of these prototypes will be a built-in Random Forest model that can be trained on extensive datasets related to water quality. It will have an F-score of 0.85 that can use the real-time data obtained from the onboard sensors to dictate navigation settings.

II. ML CLOUD APIS

Researchers have recently become interested in the convergence of two large fields of ongoing research: ML and software testing. A study of research activities centered on applying ML algorithms to enhance software testing was the primary focus of our systematic mapping. In our opinion, this mapping study gives a good summary of the current status of ML applied to testing software, which might be helpful for academics and professionals who want to learn more about this area of study or who want to make a contribution to it.

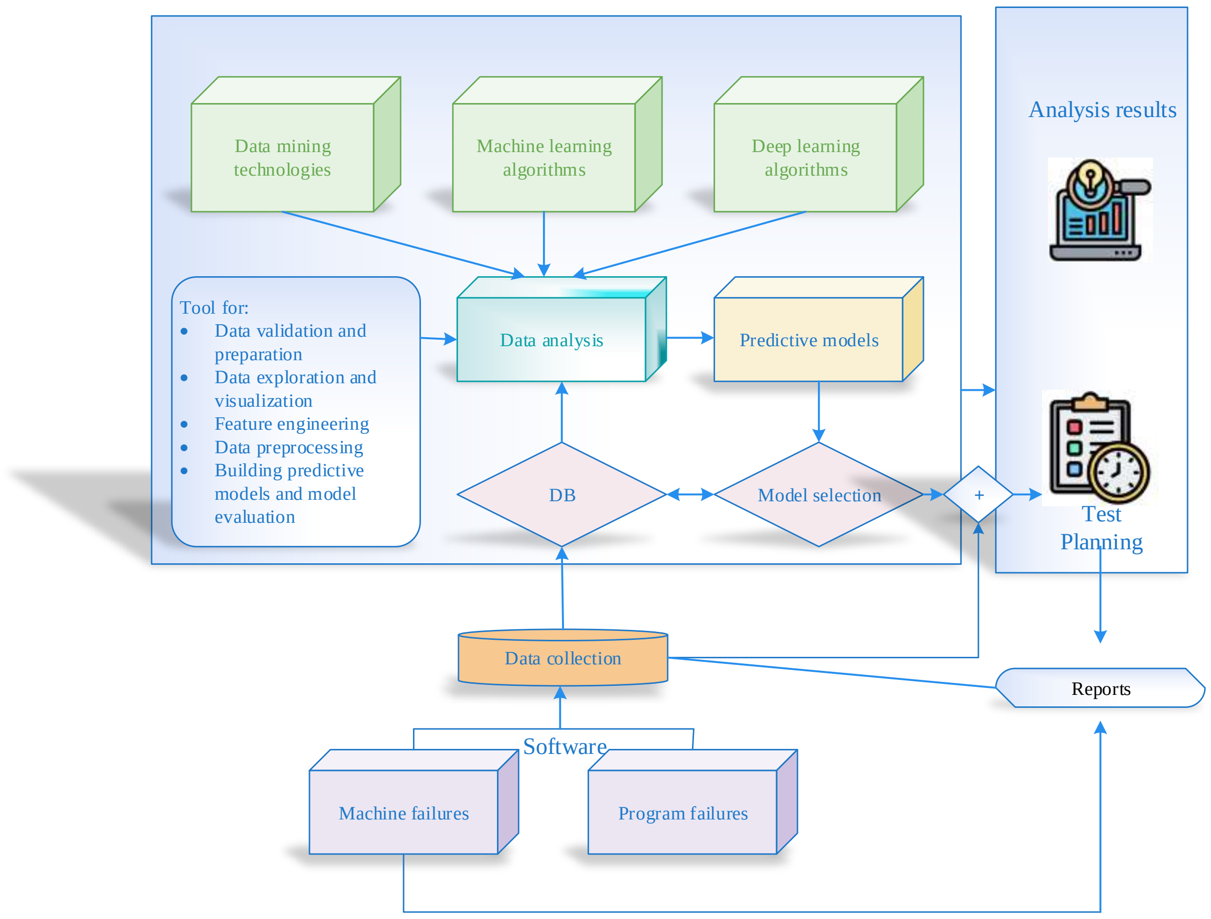

Fig.1 Procedure for Software Testing using ML Techniques

The goal of software testing is to find bugs by analyzing how software systems work. Because of the high expense and complexity of most testing tasks, automating software testing has become a popular approach. An increasing number of software engineering tasks, particularly those pertaining to testing, are being considered for automation via the use of ML.
Our research shows that ML algorithms are most often used for creating, improving, and evaluating test cases. Test oracle construction evaluations and testing-related activity cost predictions have also made use of ML. In order to assist researchers comprehend the present level of research concerning ML used to software testing, this article outlines the most prevalent ML algorithms applied to automate software-testing processes. Additionally, they discovered that more thorough empirical investigations into the usage of ML algorithms for software testing automation are required.
Developing algorithms—a series of instructions that, when executed, transform an input (or collection of inputs) into an output (or sets of outputs)—is the backbone of computer-assisted problem solving. As an example, several sorting algorithms have been suggested over the years. Such algorithms accept a collection of items (like numbers) as input and return a sorted list (like a list of numbers arranged in either ascending or descending order, for example). Traditional algorithms, however, are not well-suited to solving many issues. Predicting the efficacy of a test case is an example of a problem that is difficult to tackle with conventional algorithms. For example, we know that the input for a program that uses a sorting algorithm is a list of items (e.g., integers) as this is the SUT that applies the algorithm. Additionally, we are aware of the desired output, which is a sequential list of components. However, we are unsure of the dataset that is probably going to reveal bugs, or the inputs that will test various sections of the program's code. An algorithm does not exist for every issue. As a result, conventional algorithmic approaches to these challenges have yielded only modest results. The good news is that a mountain of information on these issues has only recently become accessible. Researchers and practitioners are now exploring solutions that use ML algorithms, which learn from data, due to the increased availability of data. The exponential growth of computing power has allowed computers to tackle ever-more-complex issues using ML, and the growing number of powerful ML tools has also contributed to the current widespread adoption of ML algorithms. The other major factor is the explosion of information being captured and stored.
These developments have allowed academics and industry professionals to apply machine learning (ML) techniques to a growing number of fields. Weather prediction, search engines for the web, natural language processing, voice recognition, computer vision, among robotics are just a few of the disciplines that have used ML techniques to tackle challenges. We should return to the issue of determining the efficacy of test cases. In

order to determine what a good test case resembles when dealing with issues like these, data becomes relevant. We assume that the obtained data (such as a collection of inputs for an application which run-time behaviour was also recorded) will include some successful test cases, even though we may not know how to generate them. You may utilize the findings of an ML algorithm to generate predictions if it can learn from the test-case data and the software being tested did not change greatly from the version that was used to gather the data. The machine learning (ML) method might not be able to deduce the whole procedure for evaluating test cases, but it can nevertheless uncover some patterns and structures within the data. In this case, a model or approximation is what comes out of the algorithm. At its most basic level, ML algorithms generate models by processing the data that is already accessible. Inferring and better characterizing issues, such as estimating the efficacy of test cases, is made possible by the patterns embodied by the resultant models. Outlining the most researched themes, the strength of proof for, and the advantages and limits of ML algorithms, this document gives current details on the studies at the confluence of ML with software testing. Since researchers will be able to draw on the most up-to-date information, we anticipate that this systematic mappings will lead to better ML-based testing methods. Also, although ML won't solve every problem with software testing, we think this research is a good starting point for moving forward with ML in software testing. The main take away from this study is that it might help academics and practitioners choose the most appropriate ML algorithms for their specific challenges.

Learn the basics of machine learning application programming interfaces (APIs), including what they are, how they work, and what data they take in and produce.

- **ML Cloud APIs:** There are three primary types of machine learning jobs that all ML APIs from various vendors address: vision, language, and voice. According to Table 1, all of the most popular APIs from these three families are managed by Keeper. Apis from the vision, language, and voice families are not presently handled by Keeper because to their low usage in open-source apps. These families include video intelligence, translation, and speech synthesis. A machine learning API's output could include several records, such as various categorization outcomes, discovered objects, and so on. There is a score of trust field that shows the likelihood that this result is accurate, and an important outcome field that is usually of a string or a listing type. This sort of field is used to store information like an image's categorization label or a face's mood. Table 1 summarizes the major result fields that are referred to as ML API output throughout the remainder of the article, unless otherwise stated.
- Keep in mind that a few of these APIs really provide additional supplemental data. Take the face recognition API as an example; it not only detects faces in the input picture but also produces their bounding boxes. Although control flow choices might be made using these auxiliary result fields, a prior ML API research did not find any evidence of such use in any of the 360 programs that were gathered. machine learning programs. On occasion, ML APIs are only tangentially related to the rest of the program, and their results are written out without being used in any way. It is not the goal of Keeper to test this kind of program as testing the ML APIs as well as the rest of the software independently is sufficient. The outcomes of ML APIs are used to influence the management process of program execution in other instances when there is a closer relationship between the two. The main emphasis of this work is on these situations since they illustrate the increased obstacles to software testing that were mentioned in Section 1.

With the use of successful pseudo inversion, search engines provide a number of vision as well as language APIs with a collection of realistic pictures and sentences that correspond to a given term. There are a number of features of search engines that are useful for Keeper's experiments. As a first benefit, they provide excellent semantic inversion; after all, there are a number of engines for searching that have served thousands of millions of happy customers for years. As a rule, their top results from searches are in line with what the average person would think. Next, we shall use engines other than Google's to reduce the likelihood of correlations, and they are not a statistical reversal of ML APIs. Thirdly, they can handle a broad variety of search terms and provide several ranked results, which translate to a substantial amount of high-quality data for testing.

III. PROPOSED WORK

A. ML Algorithms

The purpose of this research was to evaluate the accuracy of many ML algorithms in predicting software failures. Here is a brief overview of these ML algorithms:

- **Random Forest** (RF)

The RF algorithm can learn both regression and supervised classification tasks. Instead of making decisions based on individual trees, it creates a forest of them and then uses committee voting to choose the best one.

- **Decision Table (DT)**

DT is a structured approach of integrating business rules with requirements. Complex logic may be described using this classifier. True (T) and false (F) are the labels given to conditions in a DT. A business logic constraint specifies the unique mix of instances in each column of the table.

- **Linear Regression (LR)**

LR makes use of supervised learning techniques. A linear regression connection is established among an independent variable (x) and the dependent variable's value (y) in order to forecast the outcomes.

For this Machine Learning (ML) task, we opted for a Multilayer Perceptron (MLP). Mainly Layered Perceptrons (MLP) are among the most popular ANNs because to their modest training set needs, speed, and ease of implementation. The input, hidden, and output layers make up a multi-layer perceptron (MLP) (Figure 1). Neurones in the hidden layer are linked to both the output neuron and all of the input neurons. Add all of the input signals (x_i) from all of the neurons in the network, and then multiply each result by its connection weight (w_{ji}). To calculate the output signal (y_j) of every neuron j, we feed its activation function its weighted sum of inputs. The activation function of an artificial neuron may take several forms, including the basic threshold, sigmoid, and hyperbolic tangent functions. The activation function used in this investigation was the sigmoid. Therefore, Eq. 4 gives the output of every neuron (j).

$$y_j = f\left(\sum w_{ji} x_i\right) \qquad (1)$$

In order to train the MLP, the input values are fed into the input neurons, and the result, y_j, is calculated. The next step is to utilize Eq. 5 to get the total of the squared deviations between the output neurons' actual values (y_j) and their intended outputs (y_{dj}). Lastly, in order to reduce the value E, the weights (w_{ji}) are modified. The current research makes use of backpropagation, one of the best-known algorithms.

$$E = \frac{1}{2}\sum_j \left(y_{dj} - y_j\right)^2 \qquad (2)$$

The study's MLP setup is shown in Figure 1. Using a learning rate of 0.001 and a momentum of 0.05, the backpropagation algorithm trained the MLP across 500 epochs. For training purposes, the seed value was 1. There were 100 validations since the MLP was induced using a 10×10-fold cross-validation technique. For this research, we opted for K-fold with k=10 as opposed to leave-one-out cross-validation for a number of reasons, including that it produces superior results relative to the dataset size and that it reduces variance, which facilitates the comparison of different ML models' performances. To further quantify the model's accuracy in reality (generalize to an alternative dataset), cross-validation was used instead of a train/test split to mitigate issues like underfitting and overfitting.

Hence, it is not a realistic aim to do comprehensive software testing. Making ensuring the program is acceptable in terms of the likelihood of failure owing to dormant flaws is a reasonable objective. But what works for a mobile app isn't good enough for an airplane's autopilot. Thus, the need for software testing varies between applications based on the desired dependability (and safety).

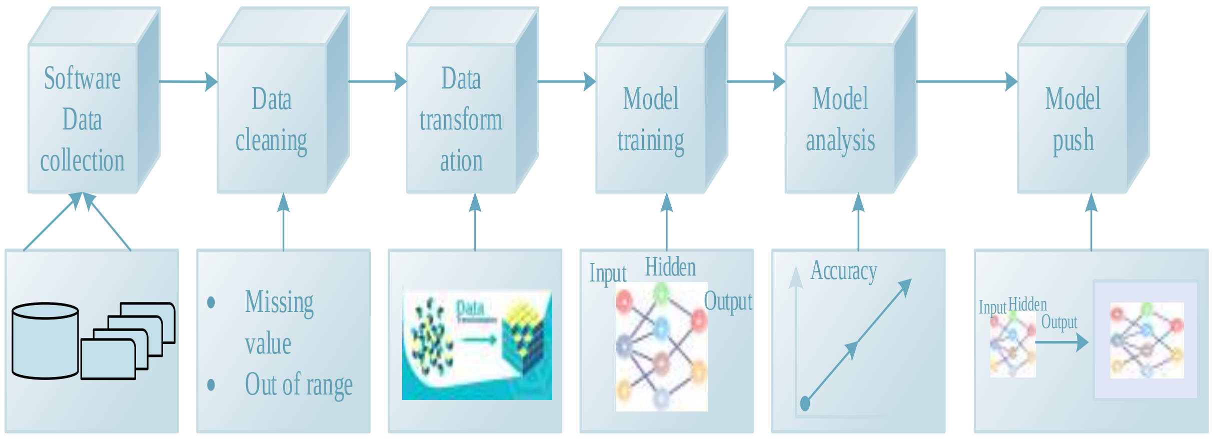

Fig.2 ML Classifier for Software Testing

B. Proposed Kernel ELM

For this study, we tested software using an Extreme Learning Model and the SMO algorithm. Software flaws may be categorized according to the data provided using the suggested ELM based SMO method. Two blocks, the block diagram and the flow diagram, make up the classification process in Fig. 3, which uses Kernel based ELM.

TABLE I. NUMERIC ATTRIBUTES DESCRIPTION OF KC1 DATASET

Item No.	Attribute	Description
1	PERCENT_PUB_DATA	The percentage of data that is public and protected data in a class.
2	ACCESS_TO_PUB_DATA	The number of times that a class's public and protected data is accessed.

3	COUPLING_BETWEEN_OBJECTS	The number of distinct non-inheritance-related classes on which a class depends.
4	DEPTH	The level for a class.
5	LACK_OF_COHESION_OF_METHODS	The percentage of the methods in the class.
6	NUM_OF_CHILDREN	The number of classes derived from a specified class.
7	DEP_ON_CHILD	Whether a class is dependent on a descendant.
8	FAN_IN	Count of calls by higher modules.
9	RESPONSE_FOR_CLASS	Count of methods implemented within a class.
10	WEIGHTED_METHODS_PER_CLASS	Count of methods implemented within a class.
11	NUMDEFECTS	No. of Defects

The following is a definition of the weight values used in the output of the hidden node to classify software defects,

$$FL(x) = \sum_{n=1}^{L} \alpha_n \mu_n(X) \qquad (3)$$

Where, α_n portray the concealed node's weight values and J(x) = $[j_n(X), .., j_L(X)]$ symbolize the result that ELM's hidden layer produces. Here is the definition of the hidden layer output matrix,

$$J = [j(X_1) \vdots j(X_N)] = [G(p_1, q_1, X_1) \cdots G(p_L, q_L, X_L) \vdots \ddots \vdots G(p_1, q_1, X_N) \cdots G(p_L, q_L, X_N)] \quad (4) \qquad ()$$

Where TM stands for the following-defined training matrix,

$$TM = [r_1 \vdots r_N] \quad (5)$$

Decreases in weight values are the end goal of ELM; the series of γ is $\gamma_1, \gamma_2 > 0, a, b = 0, \frac{1}{2}, 1, 2, \ldots, +\infty$. Defining the objective function mathematically is as follows,

$$||\alpha||\gamma_1\, a + C||\, J\alpha - R||\gamma_2\, b \qquad (6)$$

Here is a definition of the process of kernel-based ELM-based classification,

Algorithm for Kernel based ELM

Step 1: Given a extracted features (F) as an input
Step 2: Initialize training with F
Step 3: Generate training matrix TM
Step 4: Randomly generate the hidden node parameters $J(X)$
Step 5: Calculate the weight values (W) of the hidden node
Step 6: Determined the weight value $(\alpha_n)$of the nth hidden node
Step 7: Calculate the original output of ELM with respect to F

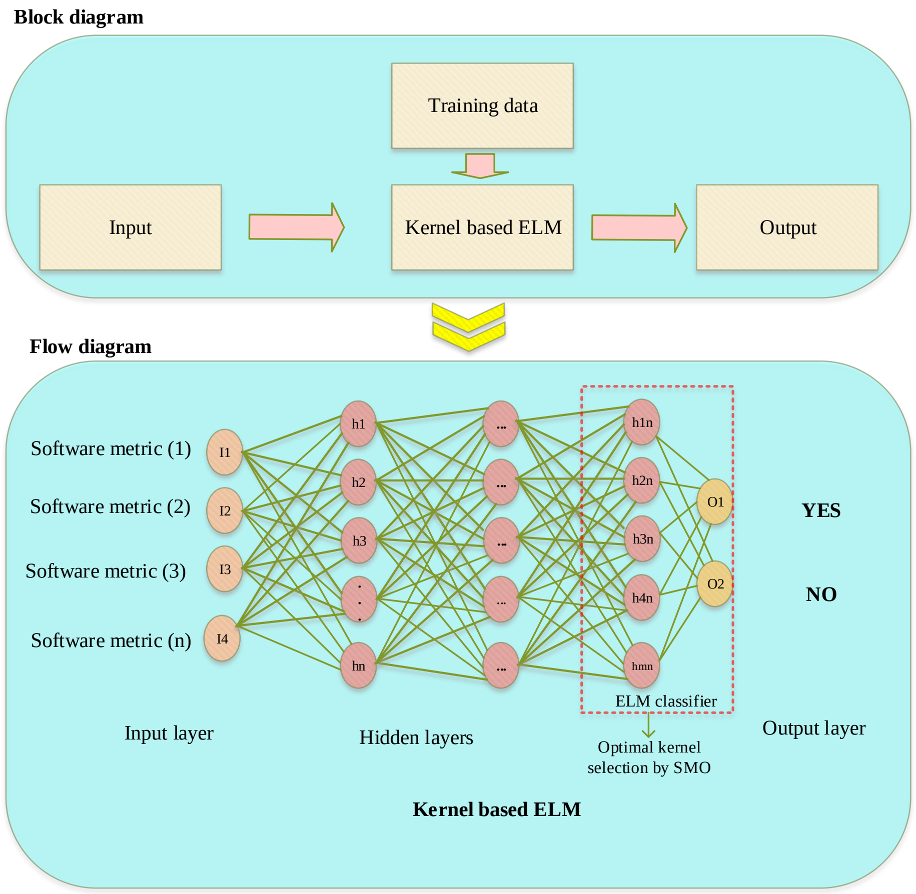

Fig.3 Classification using Kernel based ELM

IV. RESULTS & DISCUSSION

To forecast software issues, this study deployed the previously mentioned eight machine learning (ML) methods on the Weka 3.8.3 platform. The tested dataset has percentage split testing modes of 60%, 70%, 80%, and 90% as well as cross-validation (10-fold). To assess how well the classifiers predicted software problems in the given dataset, we may look at Table II, which displays the statistical measures used for this purpose.

A. Dataset Details

This study made use of class-level data for KC1, a product of NASA, which is a publicly accessible dataset from the PROMISE repository for software engineering. There are properties in the dataset that pertain to both classes and methods. By using the operations of the highest, lowest, average, and sum, Koru et al. transformed 21 characteristics from the method level into 84 attributes from the class level. The dataset has 84 characteristics in total, with an additional 10 class-level attributes adding up to 94 attributes in total. The NUMDEFECTS numerical attribute is the final one and it shows how many class faults were documented. It will be used by ML systems for prediction purposes as a response variable. One of NASA's KC1 products is included in the collection, which includes actual measured data, in 145 occasions.

B. Evaluation metrics

A classifier's output quality may be assessed using a variety of criteria. A perfect ML classifier would not mistake a positive class for a negative one (FP) or vice versa. The classes that were properly identified as negative are called True Negative (TN), whereas the ones that were correctly identified as positive are called True Positive (TP). Consequently, the connection between TP, TN, FP, and FN is shown by several assessment measures. Here, we focus on accuracy, recall, and precision, three of the most popular measures used to evaluate ML classifiers.

The percentage of faulty modules that a classifier accurately identifies is called its precision. (n_{tp}) fractioned by the sum of all faulty module counts ($n_{tp} + n_{fp}$). In this study's sphere of application, the malfunctioning modules will define the area of the test that has to be run ($n_{tp} + n_{fp}$) on the premise that... Nevertheless, only a small percentage of those units will really be determined to be faulty (n_{tp}). Since accuracy reflects the efficacy of the testing procedure, it is called efficiency in this research. Put simply, it shows the percentage of tested modules that were found to be faulty out of the overall amount of modules. Since a near proximity to 1 (or

100%) indicates an extremely effective test effort (i.e., all chosen modules are faulty), this is the ideal value that a manager would aim for.

$$Efficiency = Precision = \frac{n_{tp}}{(n_{ep} + n_{fp})} \quad (7)$$

The classifier's recall is the percentage of really faulty software modules to the total number of modules that were properly predicted as defective (n_{tp}). ($n_{tp} + n_{fn}$). Because it translates the effectiveness of the test attempt, recall is called efficacy in this research. Put simply, it shows the percentage of the target accomplished by the testing effort, where the objective is to identify all faulty modules that are already in existence. To reflect a very low likelihood of omitting a damaged module, a management would want this value to be as near to 1 (or 100%) as feasible.

$$Efficacy = Recall = \frac{n_{lp}}{(n_{tp}+n_{fn})} \quad (8)$$

Lastly, the accuracy of the classifier is the proportion of the overall amount of modules that properly identify software defects (TP) and non-defects (TN) (n_{total}). There would be no FP or FN output from a classifier that was 100% correct. Therefore, it is desirable for a management to aim for an accuracy level as near to 1 (or 100%) as feasible. While perfect accuracy is always desirable, it may be an indication of overfitting and a lack of generalizability beyond the initial dataset. Furthermore, a classifier with poor generalizability will have a harder time becoming useful when faced with fresh data from other parts of the system.

$$Accuracy = \frac{(n_{tp}+n_{fn})}{n_{total}}(9)$$

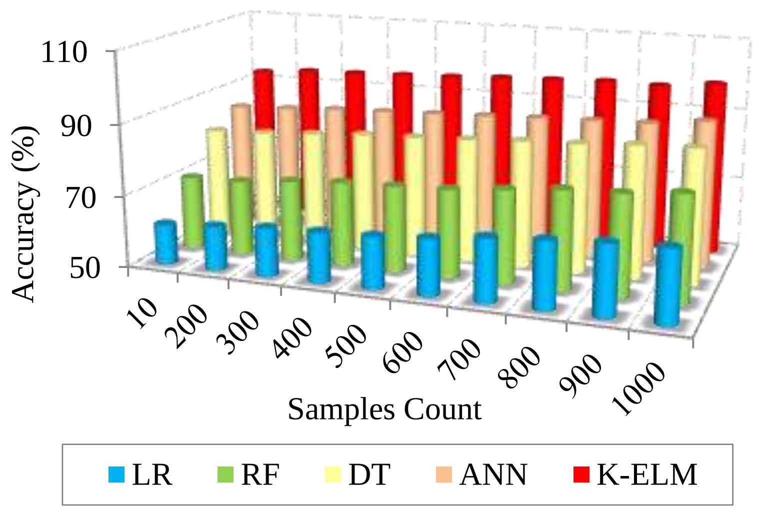

Fig.4 Accuracy vs. Samples Count

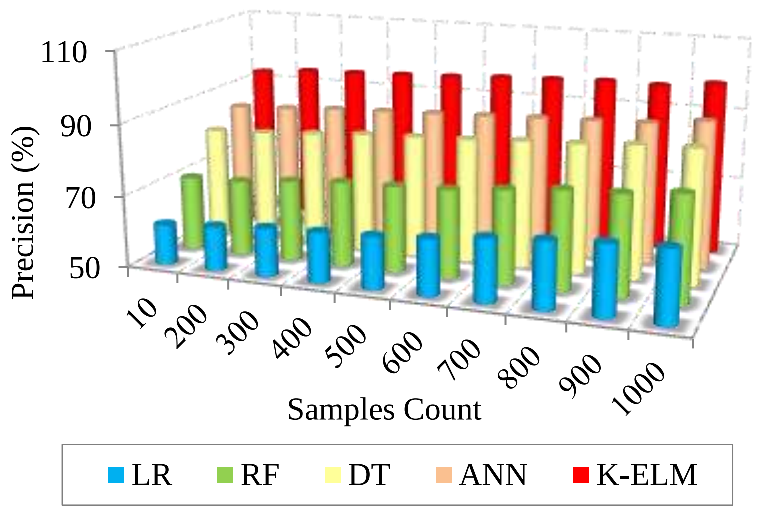

Fig.5 Precision vs. Samples Count

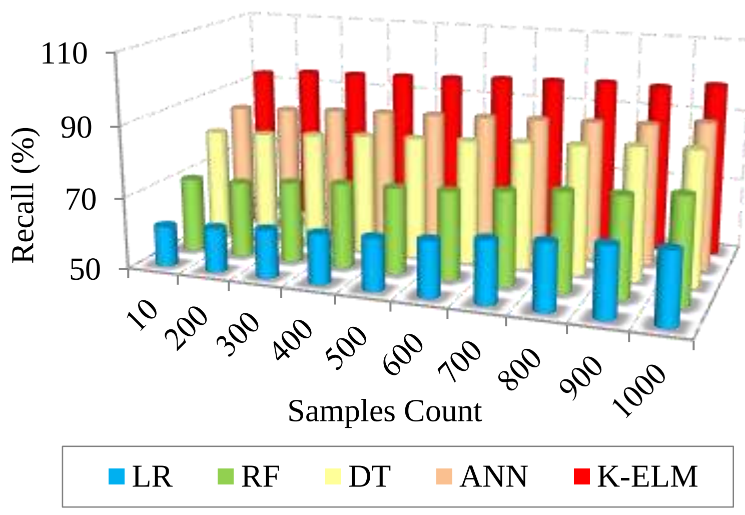

Fig.6 Recall vs. Samples Count

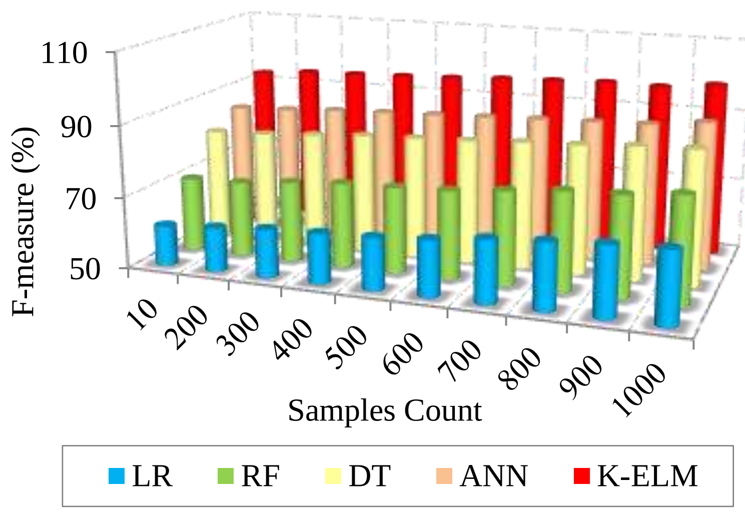

Fig.7 F-score vs. Samples Count

We have already established that the dataset in question is imbalanced. The induced ML models are likely to be skewed toward classifying additional components as non-defective than faulty, given that the majority of the data pertains to modules that are not defective. As a result, a substantial FN ratio is anticipated. This is concerning since FNs encourage testers to exclude or severely limit testing of problematic modules in the software-based vital systems area. The result is a decline in product quality and the possibility of disastrous losses caused by flawed software.
The analysis and prediction procedure were conducted using the KC1 dataset. Using 10-folds cross-validation, the dataset was trained and tested.

Afterwards, several metrics and error measurements are used to assess the classifiers' output. Among the statistical measures are the following: R^2 for correlation, MAE for mean absolute error, RMSE for root mean squared error, RAE for relative absolute error, and RRSE for root relative squared error. These metrics assess the degree to which the dataset's actual defect count differs from its projected defect count. Presuming that E is the count of real faults, $E\tilde{}$ is the count of anticipated flaws, Ei is the average of E, where n is the count of occurrences. In order to make an accurate forecast, we employed the following formulae to measure performance:

1. $$R^2\ R^2 = 1 - \frac{\sum_{i=1}^{n} (Ei - E\tilde{}i)^2}{\sum_{i=1}^{n} (Ei - E\tilde{}i)^2} \qquad (10)$$

2. MAE

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |Ei - E\tilde{}i| \qquad (11)$$

3. RMSE

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \square \, (Ei - \bar{E}i)^2} \qquad (12)$$

4. RAE

$$RAE = \frac{\sum_{i=1}^{n} \square \, |Ai - \bar{A}i|}{\sum_{i=1}^{n} \square \, |Ai - \bar{A}i|} \qquad (13)$$

5. RRSE

$$RRSE = \sqrt{\frac{\sum_{i=1}^{n} \square \, (Ai - \bar{A}i)^2}{\sum_{i=1}^{n} \square \, (Ai - \bar{A}i)^2}} \qquad (14)$$

R^2 finds the degree of correlation between the expected and actual values. The values may be anything from -1 to 1, with 1 denoting an accurate positive linear relationship, 0 indicating no connection, and -1 denoting an accurate negative linear relation. When comparing actual and anticipated values, the mean absolute error (MAE) is the mean of the two. While both RMSE and MAE measure the discrepancy between actual and anticipated values, RMSE takes the square root of that difference to get the average. Major mistakes will therefore get more attention than minor ones. Because RAE provides an average of the real values, the error may be defined as the sum of all absolute errors. Although the error is the sum of the squared error, RRSE is comparable to the RAE in that it is the average of the true values.
Table II shows that in the KC1 dataset, the LR technique had the greatest correlation coefficient (R^2) value, while the SMOreg approach came in second. Contrarily, for the other measures, the SMOreg algorithm produced the best results in terms of error rates. Hence, out of the eight ML methods, the SMOreg classifier had the highest performance results. When compared to other algorithms, the ANN algorithm performed the poorest. Also, the algorithm failed to forecast legitimate values due to the constant values in all the input data, resulting in error rates of over 100% for both RAE and RRSE.

Table Ii. Statistical Performance Results For The Ml Algorithms Using 10 Folds Cross-Validation Testing Mode

CLASSIFIER	R^2	MAE	RMSE	RAE	RRSE
K-ELM	0.1584	6.9869	14.7296	115.6751	134.899
ANN	0.4995	4.4208	9.3905	73.1899	86.0011
Random Forest (RF)	0.2663	4.9232	11.2294	81.5082	102.8431
Decision Table (DT)	0.7594	6.5889	11.1759	109.0851	102.3526
Linear regression (LR)	0.7383	4.3159	7.4378	73.1085	68.1179

V. Conclusion

Better user experiences may be achieved by enhancing current software with intelligent behaviors powered by machine learning. Our software-dependent society stands to gain a lot from the information shared in this study, which might lead to new innovations or investments in software that is powered by machine learning.Applying ML classifiers to forecast software module defects has been the subject of several research. These methods may be helpful in assisting with test planning as testing often makes do with far less resources than are really required. Actually, knowing which software modules are more likely to contain defects might help testing resources be distributed more evenly.The majority of these studies primarily assess the ML classifier's performance based on its accuracy. On the other hand, actual accuracy could be compromised if the datasets used for training and assessing the classifier are imbalanced. Indeed, failing to adequately address the imbalance might lead to possibly biased accuracy. Users may mistakenly allocate test resources due to their erroneous impressions of the classifier's quality caused by this bias. When it comes to applications that need absolute safety, a large number of FNs may be disastrous. Within this framework, this research offered a method to improve ML classifier performance in forecasting software module failures, regardless of the dataset's imbalance. As an incentive for the FNs to train separate MLP models, RC values between 1 and 10 were applied using a cost-sensitive technique coupled with an MLP. First, the number of FNs decreased significantly as the assigned RC for FN increased. Second, the software testing effectiveness increased significantly. Third, the the amount of components correctly indicated as faulty increased. On the other hand, the model suggested a larger test scope, software testing efficiency decreased. Sixth, the number of FPs increased. Lastly, overall accuracy decreased. That is why it is important to think about the trade-off that the recommended strategy imposes when organizing software testing tasks.

References

[1] Klemmer, J.H., Horstmann, S.A., Patnaik, N., Ludden, C., Burton, C., Powers, C., Massacci, F., Rahman, A., Votipka, D., Lipford, H.R., Rashid, A., Naiakshina, A., Security, S.F., Bochum, R.O., Bristol, U.O., University, T., Amsterdam, V.U., Trento, U.O., University, A., & Charlotte, U.O. (2024). Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns.
[2] Wang, L. (2023). AI in Software Engineering: Case Studies and Prospects. ArXiv, abs/2309.15768.
[3] Eisenreich, T., Speth, S., & Wagner, S. (2024). From Requirements to Architecture: An AI-Based Journey to Semi-Automatically Generate Software Architectures. ArXiv, abs/2401.14079.
[4] Pandi, S.B., Binta, S.A., & Kaushal, S. (2023). Artificial Intelligence for Technical Debt Management in Software Development. ArXiv, abs/2306.10194.
[5] Vasilev, Y., Arzamasov, K., Kolsanov, A., Vladzymyrskyy, A.V., Omelyanskaya, O.V., Pestrenin, L.D., & Nechaev, N.B. (2023). EXPERIENCE OF APPLICATION ARTIFICIAL INTELLIGENCE SOFTWARE ON 800 THOUSAND FLUOROGRAPHIC STUDIES. Vrach i informacionnye tehnologii.
[6] Bharadwaj, R., & Parker, I. (2023). Combining Foundation Models and Symbolic AI for Automated Detection and Mitigation of Code Vulnerabilities.
[7] Bharadwaj, R., & Parker, I. (2023). Double-edged sword of LLMs: mitigating security risks of AI-generated code. Defense + Commercial Sensing.
[8] Juneja, S., Bhathal, G.S., & Sidhu, B.K. (2024). Development of optimised software fault prediction model using machine learning. Intelligent Decision Technologies.
[9] Mahapatra, S., & Mishra, S. (2020). Usage of Machine Learning in Software Testing.
[10] Jude, A., & Uddin, J. (2024). Explainable Software Defects Classification Using SMOTE and Machine Learning. Annals of Emerging Technologies in Computing.
[11] Moin, A., Challenger, M., Badii, A., & Gunnemann, S. (2021). A model-driven approach to machine learning and software modeling for the IoT. Software and Systems Modeling, 21, 987 - 1014.
[12] Khan, I., Tunesi, L., Masood, M.U., Ghillino, E., Bardella, P., Carena, A., & Curri, V. (2021). Automatic Management of N × N Photonic Switch Powered by Machine Learning in Software-Defined Optical Transport. IEEE Open Journal of the Communications Society, 2, 1358-1365.
[13] Mustaqeem, M., & Siddiqui, T. (2023). A Hybrid Software Defects Prediction Model for Imbalance Datasets Using Machine Learning Techniques: (S-SVM Model). Journal of Autonomous Intelligence.
[14] Akimova, E.N., Bersenev, A.Y., Deikov, A.A., Kobylkin, K.S., Konygin, A.V., Mezentsev, I.P., & Misilov, V.E. (2021). PyTraceBugs: A Large Python Code Dataset for Supervised Machine Learning in Software Defect Prediction. 2021 28th Asia-Pacific Software Engineering Conference (APSEC), 141-151.
[15] A Shatnawi, R. (2023). Predicting Software Change-Proneness From Software Evolution Using Machine Learning Methods. Interdisciplinary Journal of Information, Knowledge, and Management.
[16] S.Shanmugapriya, & P.Devika, M. (2023). A NOVEL SOFTWARE ENGINEERING APPROACH TOWARD USING MACHINE LEARNING FOR IMPROVING THE EFFICIENCY OF HEALTH SYSTEMS. international journal of engineering technology and management sciences.
[17] Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., & Tonella, P. (2020). Testing machine learning based systems: a systematic mapping. Empirical Software Engineering, 25, 5193 - 5254.
[18] Noor, M.N., Khan, T.A., Haneef, F., & Ramay, M.I. (2022). Machine Learning Model to Predict Automated Testing Adoption. Int. J. Softw. Innov., 10, 1-15.
[19] Nascimento, L.P., Prudêncio, R.B., Mota, A.C., Filho, A.D., Cruz, P.H., Oliveira, D.C., & Moreira, P.R. (2023). Machine Learning Techniques for Escaped Defect Analysis in Software Testing. Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing.
[20] Siddiqui, A., Zia, M.Y., & Otero, P. (2021). A Universal Machine-Learning-Based Automated Testing System for Consumer Electronic Products. Electronics.
[21] Tsai, C., & Taylor, G.W. (2022). DeepRNG: Towards Deep Reinforcement Learning-Assisted Generative Testing of Software. ArXiv, abs/2201.12602.
[22] Filippou, K., Aifantis, G., Papakostas, G.A., & Tsekouras, G.E. (2023). Structure Learning and Hyperparameter Optimization Using an Automated Machine Learning (AutoML) Pipeline. Inf., 14, 232.
[23] Wei, M., Huang, Y., Wang, J., Shin, J., Harzevili, N.S., & Wang, S. (2022). API recommendation for machine learning libraries: how far are we? Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.
[24] Haryono, S.A., Thung, F., Lo, D., Lawall, J.L., & Jiang, L. (2021). Characterization and Automatic Updates of Deprecated Machine-Learning API Usages. 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), 137-147.
[25] Muqeet, A., Yue, T., Ali, S., & Arcaini, P. (2023). Mitigating Noise in Quantum Software Testing Using Machine Learning.
[26] Mehta, S., & Patnaik, K.S. (2021). Improved prediction of software defects using ensemble machine learning techniques. Neural Computing and Applications, 33, 10551 - 10562.
[27] Islam, M., Khan, F., Alam, S., & Hasan, M. (2023). Artificial Intelligence in Software Testing: A Systematic Review. TENCON 2023 - 2023 IEEE Region 10 Conference (TENCON), 524-529.
[28] Sharma, A., Demir, C., Ngomo, A.N., & Wehrheim, H. (2021). MLCheck- Property-Driven Testing of Machine Learning Models. ArXiv, abs/2105.00741.

[29] Vulpe, A., Girla, I., Craciunescu, R., & Berceanu, M.G. (2021). Machine Learning based Software-Defined Networking Traffic Classification System. 2021 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom), 1-5.
[30] Goyal, S. (2020). Comparison of Machine Learning Techniques for Software Quality Prediction. Int. J. Knowl. Syst. Sci., 11, 20-40.
[31] Saharudin, S.N., Wei, K.T., & Na, K.S. (2020). Machine Learning Techniques for Software Bug Prediction: A Systematic Review. Journal of Computer Science.
[32] Maguire, T., & Meehan, K. (2023). Autonomous River Boat Sensor Platform: Monitoring Rivers using AI. 2023 IEEE World AI IoT Congress (AIIoT), 0554-0560.

BIOGARPHY

Kodanda Rami Reddy Manukonda is a Test Architect with 20+ years of experience in IT Industry. Telecom applications technology expert focused on design, development and deployment of end to end operations algorithms that forms the software backbone of the telecom industry, specifically for services in Telecom Billing applications, Service Delivery applications, Service Assurance and Telecom Ordering applications