# Software Defect Prediction using Deep Learning by Correlation Clustering of Testing Metrics

Original Scientific Paper

**Kamal Kant Sharma**
Department of Information Technology, KIET Group of Institutions, Delhi-NCR, Ghaziabad
Dr. A.P.J. Abdul Kalam Technical University, Lucknow, India
knmietkamal@gmail.com

**Amit Sinha**
Department of Information Technology, ABES Engineering College, Ghaziabad
Dr. A.P.J. Abdul Kalam Technical University, Lucknow, India
amit.sinha@abes.ac.in

**Arun Sharma**
Department of AI and Data Sciences
Indira Gandhi Delhi Technical University for Women, Delhi, India
arunsharma2303@gmail.com

***Abstract*** *– The software industry has made significant efforts in recent years to enhance software quality in businesses. The use of proactively defect prediction in the software will assist programmers and white box testing in detecting issues early, saving time and money. Conventional software defect prediction methods focus on traditional source code metrics such as code complexities, lines of code, and so on. These capabilities, unfortunately, are unable to retrieve the semantics of source code. In this paper, we have presented a novel Correlation Clustering fine-tuned CNN (CCFT-CNN) model based on testing Metrics. CCFT-CNN can predict the regions of source code that contain faults, errors, and bugs. Abstract Syntax Tree (AST) tokens are extracted as testing Metrics vectors from the source code. The correlation among AST testing Metrics is performed and clustered as a more relevant feature vector and fed into Convolutional Neural Network (CNN). Then, to enhance the accuracy of defect prediction, fine-tuning of the CNN model is performed by applying hyperparameters. The result analysis is performed on the PROMISE dataset that contains samples of open-source Java applications such as Camel Dataset, Jedit dataset, Poi dataset, Synapse dataset, Xerces dataset, and Xalan dataset. The result findings show that the CCFT-CNN model increases the average F-measure by 2% when compared to the baseline model.*

***Keywords****: Software Engineering, Software Testing, Abstract Syntax Tree, Machine Learning, Convolution Neural Network*

## 1. INTRODUCTION

Software engineering is a vast field that is extensively used in software-based industrial applications. Humans have become increasingly reliant on software-based applications in subsequent years, with software functionality being regarded as the most important aspect of customer experience [1]. Numerous consequences, like a breakdown of software functioning and incorrect output, could arise during real-time software-based applications, resulting in consumer dissatisfaction. Nonetheless, when a large number of activities are processed and performed by software, the software's performance may be reduced. Real-time applications frequently experience software failures and flaws. Program error appears when performance differs from predicted results, causing disagreement between real and necessary software functioning, often known as a software flaw or bug [2]. As the need for user apps grows, so do the complexities of software applications, which has an impact on the operations of numerous software corporations and leads to the production of inadequate software uses. Due to a large number of software applications, experts find it difficult to identify, locate, and discover software defects. Moreover, in the field of software fault prediction and diagnosis, defect density is a challenging problem. Previous research and expertise in this field can help forecast software product faults. Initially, manual testing was used to uncover software defects in the software industry. In the overall development of a program, manual software testing demands 27% human labor [3]. Manual testing takes more time and labor, and it can't catch all of the faults in the program. Defect forecasting models are commonly employed by companies to solve this problem. During the development process, these models allow firms in fault forecasting, effort estimate, software reliability testing, risk assessment, and other functions [4]. This can also contribute to minimizing risk by utilizing software quality forecasting models early in

the software development lifecycle (SDLC), resulting in increased customer contentment and cost reductions, as well as reduced human work. To solve the problem of software fault forecasting, many strategies have been developed. In this context, software metrics are useful for forecasting software product problems by examining the link between software metrics and output product reliability. Process metrics, project metrics, and product metrics are the three types of software metrics. Software development and maintenance utilize process metrics to ensure the software's operation and longevity. Product metrics explain different attributes like layout features, quality level necessity, and performance level need for any specific software program, whereas project metrics offer various parameters including the total number of developers, developer expertise, work scheduling, organization, and software size [5].

Machine learning and data mining approaches are regarded to be the most effective in the field of software engineering and fault predictions. Machine learning is highly utilized in a variety of implementations due to its potential to meet or surpass human-level functionality. The accessibility of huge data samples, high-speed processing techniques, and their capabilities to accomplish excellent functionality has developed the way for DNNs to enter safety-critical applications which include automated vehicle driving, healthcare diagnostic test, safety, and so on. Because such implementations are safety-critical, sufficient testing of these machine learning is required before implementation. Nevertheless, unlike conventional software, machine learning lacks a clear control-flow structure. They learn their judgment call strategy by instructing on a big dataset and progressively modifying variables utilizing various means to accomplish the required precision. As a result, conventional software checking techniques such as operational coverage, branch coverage, and so forth cannot be implemented in machine learning, posing a challenge to their usage in safety-critical implementations [1].

Conventional software screening techniques perform badly when implemented in machine learning since machine learning code contains no details about a machine learning internal decision-making rationality. Machine learning comprehends its regulations from instructing information and does not have the control-flow structure found in conventional software [3]. As a result, conventional coverage criteria such as code coverage, branch coverage, functional coverage, and so on are inapplicable to machine learning. Fig 1 depicts a high-level depiction of the majority of established Whitebox evaluation techniques for machine learning. The machine learning, evaluation inputs, and coverage metric are used as inputs to the evaluation procedure to guarantee that all sections of the program logic have been examined. An oracle determines whether the machine learning action is accurate for the examined inputs. A coordinated test input generation technique can also be utilized to produce test input information with wider reach and expose more corner case behavioral patterns. Established analysis techniques typically produce either an estimate of system appropriateness or an adversarial ratio [3].

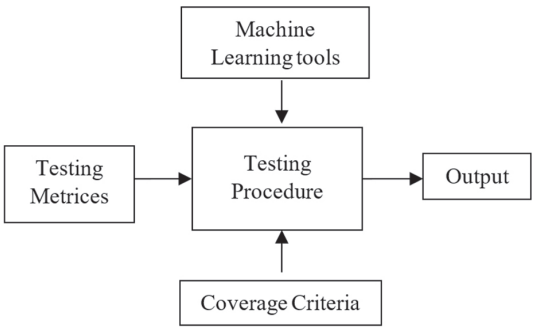



**Fig 1.** Representation of Testing using Machine Learning

Machine learning (ML) systems are highly being used in safety-critical areas as a result of subsequent advancements in the domains [4]-[10]. As a result, several software analysis techniques are necessary to guarantee the dependability and security of ML systems. Because ML framework regulations are interpreted from training information, it is challenging to understand the execution regulations for every ML system behavior. At the same time, Random Testing (RT) is a prominent testing technique, and it does not require any understanding of software execution. As a result, RT is an excellent choice for evaluating ML systems. Furthermore, the established methods for evaluating ML frameworks rely highly on RT by labeled testing datasets. Random Testing (RT), being one of the most basic and widely used software screening techniques, could be used to evaluate ML systems. The advantage of RT is that it is simple in theory and convenient to apply [5]. Most notably, if researchers have no understanding of software execution, it could be the only practicable method [6]. Undoubtedly, due to the change in advancement framework caused by ML, this feature is also an essential advantage for evaluating ML systems [7]. Conventional software is built axiomatically by attempting to write down the regulations as program codes, and the system's behavior is controlled by these known regulations. The execution guidelines for ML systems, on the other hand, are derived inductively from retraining information and are not precisely known to us. Because of the paradigm shift, several of the key characteristics of ML systems have no source code. As a result, RT is an excellent evaluation technique for ML systems. In reality, the conventional way for evaluating ML systems is to collect and physically categorize as much test information as possible, then utilize this data to evaluate ML systems using the RT technique [8].

## 2. RELATED WORK

Gerasimou et al. [8] focused on altering typical software engineering assessment requirements to increase certainty in their right conduct. Nevertheless, they fall

short of representing the essential features that these systems display. Wang et al [9] presented Robustness-Oriented Assessment, a unique assessment methodology (RobOT). A major component of RobOT is a quantitative assessment of 1) the utility of every testing ground in enhancing model durability (typically via additional training) and 2) the model resilience improvement's converging reliability. Guo et al [10] presented Audee, a unique method for evaluating DL systems and locating flaws. Audee uses a search-based method and three distinct mutation algorithms to produce a variety of testing cases by combining model architectures, variables, evaluations, and inputs. Audee can identify three different kinds of problems: logical flaws, failures, and Not-a-Number (NaN) issues. Ma et al [11] presented a mutation assessment method for DL systems to evaluate the integrity of testing information. To achieve so, they provided a collection of source-level mutation operators to introduce flaws into the sources of DL, in a similar manner as mutation assessment in conventional software (i.e., training data and training programs). Ma et al [12] presented DeepGauge, a collection of multi-granularity evaluating requirements for DL systems that tries to generate a multi-faceted depiction of the testing ground, in this study. On two well-known databases, five DL systems, and four state-of-the-art confrontational threat approaches against DL, the in-depth examination of their recommended assessment requirements is proven. DeepGauge's potential utility offers insight into the development of more comprehensive and powerful DL systems. Du et al [13] took the initial move toward evaluating RNN-based stateful DL systems in this research. They characterized RNN as an abstract state transition system, from which they derived a series of stateful DL-specific testing coverage requirements. Wang et al [14] presented a new method for detecting challenging specimens in real-time. When they applied mutations to the DNN, they found that adversarial specimens are substantially more sensitive than regular specimens. They initially constructed a vulnerability metric. CBIL is the name of the hybrid model that was proposed by Farid et al. [15]. CBIL can identify problematic parts of source code. It does it by pulling vector representations of Abstract Syntax Tree (AST) tokens out of the source code. The transformation of integer vectors into dense vectors can be accomplished through mapping and word embedding. The AST tokens' semantics are then extracted by a CNN, which stands for the convolutional neural network. After then, the Bidirectional Long Short-Term Memory (Bi-LSTM) system will retain the most important features while ignoring the less important features to improve the accuracy of software defect prediction. Fan et al. [16] developed a hierarchical method to automatically evaluate the degree of similarity between mapped statements and tokens using a variety of distinct algorithms. The author was able to identify whether or not each of the compared algorithms provides erroneous mappings for a sentence or the tokens that make up the claim by doing the comparison. The study of the results reveals that the precision is between 0.98 and 1.00, and the recall is between 0.65 and 0.75.

## 3. METHODOLOGY

Human error is the primary cause of software bugs. There will be a defect (fault, bug) in the code because of these mistakes. There is no bug-free software. The majority of applications are classed as Critical, High, Medium, or Low because of the number of flaws they include. Problems of all sizes can be caused by the Metrics used.

Defect prediction is used to identify the sections of software that are most likely to cause problems in future versions. Choosing the appropriate testing Metrics is the first process. Then, if there are any errors in the data, it is marked as defective. Otherwise, it is marked as clean. The key features or testing Metrics of each software code file are extracted and collected. This step is critical. Finally, the model is constructed and tested using the labeled data and collected features. Finally, machine learning is used to determine whether or not the new instance is faulty or otherwise clean.

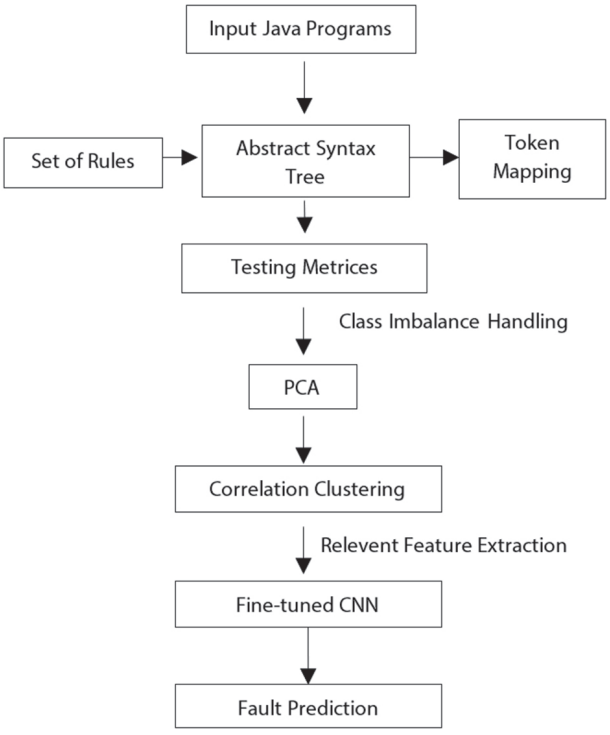



**Fig 2.** Flowchart of Methodology

In the model, there are two datasets: the training set is used to create and train the model, and the test set is used to evaluate and evaluate the trained model. Software defect prediction models can be divided into two categories. Within-Project Defect Prediction (WPDP) and Cross-Project Defect Prediction (CPDP) are two types of defect prediction. As a result of this, WPDP uses data from previous editions of the project. The same project is used for both the training and testing sets.

Two separate projects are used to train and evaluate the application in the CPDP, with the training set coming from one project and the test set coming from a separate platform. In this paper, we have adopted the CPDP model for performance evaluation.

In this paper, cascaded correlation clustering fine-tuned CNN model is proposed and termed (CCFTCNN). Fig. 2 depicts the complete framework for the proposed software testing module using machine learning. In this paper, testing Metrics are extracted using Abstract Syntax Tree (AST) for all java sources in the learning and testing groups. The vector of textual characters is then transformed into a numeric vector by creating mapping dictionaries between symbols and numbers. Then, text encoding is used to convert every numeric vector into a dense vector. Finally, the resulting dense vector is utilized as information for developing and training the proposed framework i.e., CCFTCNN. Then its functionality is examined using the PROMISE database.

### 3.1 DATA PREPROCESSING

In this stage, data is pre-processed before learning to increase its correctness and address the class balancing problem, which might have an adverse effect on the outcomes of the framework.

### 3.2 ABSTRACT SYNTAX TREE (AST)

Initially, an Abstract Syntax Tree (AST) is developed for the chosen Java items in the PROMISE database. Furthermore, the types of AST nodes that will be created as symbols are as follows: (1). Control flows elements, like if/while/do words, which are documented as node kinds. (2). nodes of class example constructions and technique incantation that are reported without parenthetical as the titles of types or techniques (3). proclamation elements, including technique/categories/ declarations, whose titles are marked. Other AST nodes are deleted because they can have an impact on the significance of the AST nodes that have been chosen. Figure 6 depicts the chosen AST nodes. To convert the source code into AST, researchers use the Python package "java lang".

### 3.3 MAPPING TOKENS

Textual elements are generated for each file whenever the source code is translated. However, using them immediately as inputs into a DL framework is problematic. As a result, the text characters must be converted to an integers vector. The spectrum of one to the entire amount of characters is used to build the mapping between text characters and numbers. As a result, every word will be represented by a single number. To transform all textual characters into a series of numbers, the "Keras Text tokenization utility" module is used. The size of all integer vectors must be consistent. The length that has been determined would be chosen. As a result, if the vector size is less than the specified value, it is filled in with 0. Furthermore, if the vector size exceeds the specified value, the excess size will be eliminated.

### 3.4 CLASS IMBALANCE HANDLING

Majority of the cases, the information is unbalanced because the amount of damaged documents is lesser than the number of good documents. As a consequence, the forecasting outcomes would be labeled as clear because cleaned records make up most of the data. For handling class imbalance correlation clustering is performed here. Traditional ways of removing data imbalance include logistic regression, up-sampling, down-sampling, and over-sampling [17]. This approach is less accurate than PCA-based data imbalance elimination. PCA may be used to reduce a high-dimensional point to a lower-dimensional point, and then filters can be used to rank the relevance of the chosen features. The variance-covariance structure of a group of variables is described by principal component analysis (PCA) in terms of fewer new variables that are linear combinations of the original variables. The additional variables may be readily produced via eigenanalysis of the original data's covariance matrix or correlation matrix. It is advisable to do PCA on the sample correlation matrix if the variables are assessed on scales with widely differing ranges or if the units of measurement are not comparable. Finally, the PCA provides a new PC transform that is constructed by utilizing the data correlation matrix to select the best PCs among all of the features. Therefore, after encoding of data, PCA is applied to them to handle their class imbalance issue.

### 3.5 RELEVANT FEATURE EXTRACTION

The Data attributes are fed into this algorithm and relevant features are extracted. Finding the characteristics of features as well as selecting meaningful and non-redundant features from such information is a big challenge. Various evaluation metrics are used in feature selection features to find the optimal feature subset, yet they do not search for feature structures. Clustering is a more effective method that recognizes the structure of features and eliminates noisy or unnecessary information. In this paper, we have applied correlation clustering is used.

To uncover the structure of the database as well as build the clusters, the K-means clustering algorithm is utilized. As initial cluster centers, choose 'k' features at random from the original dataset D. The cluster is given to the most comparable item based on the distance between both the features and the cluster mean. Within every cluster, a new mean number is evaluated. The procedure was repeated till there was no further redistribution of features from any of the clusters. A user must define the number of clusters in advance when using the k-means clustering algorithm. In this work, two clusters are created on the whole dataset. The Euclidean distance function is used to determine the similarity between the two features, as indicated in Eq (1):

$$d_{a,b}{}^2 = (a_1 - b_1)^2 + (a_2 - b_2)^2$$
$$d_{a,b} = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2} \quad (1)$$

Where, A= [$a_1$, $a_2$] and B= [$b_1$, $b_2$] are testing feature vectors.

After the clusters have been constructed, the features that do not fit into any of them are removed. The next step is to clear each cluster of duplicated features. This correlation filtering technique is employed to do this. Eq. (1) is used to compute the correlation between features. Pearson's linear correlation coefficient is defined for features $P$ having values p as well as classes $Q$ with values q, where $P$ and $Q$ are viewed as random variables:

$$\varrho(X, C) = \frac{E(PQ) - E(P)E(Q)}{\sqrt{\sigma^2}(P)\sqrt{\sigma^2}(Q)} \quad (2)$$

**Algorithm 1**: Correlation Clustering of Testing Metrics

Input: Data Attributes, $D\{d_1, d_2, d_3, \ldots d_n\}$
No. of clusters, $n$
Output: Optimal Features, $F$
Procedure
1: Begin
2: Initialize, $k$ cluster centers
3: Calculate Euclidian Distance between $D$ and $k$ (Eq. 1)
4: Join closet cluster based on similarity
5: Compute new mean
6: Repeat steps 3 to 5 until the Convergence criteria met
7: Remove irrelevant features in $D$
8: For $i$:1 to $i$<=$k$
9: Calculate correlation (Eq. 2)
10: Select best features ($F$)
11: End
12: Return $F$
End

### 3.6 FINE-TUNED CNN CLASSIFICATION

In the fine-tuned model, the output from the last convolution block of ResNet-50 is cascaded with denseconv layers with parametric ReLU activation and batch normalization layers. This network is retrained and features were extracted from the proposed fine-tuned model and further flattened to classify the data. Fine-tuning was performed by applying a parametric ReLU activation layer and using a hybrid loss function. This loss function is designed to solve the problem of class imbalance. The loss function is described below:

$$Loss_H = \frac{1}{N}\sum_{i=1}^{I}\sum_{j=1}^{J} L_{ij} \quad (3)$$

Where, $J$ = Number of loss functions, $I$ = Number of layers in the model and $L_{ij}$ = Focal loss

For fine-tuned model, parametric ReLU is adopted because it fine-tunes the learning parameters on its learning rate without any vanishing gradient problem.

## 4. RESULT ANALYSIS

### 4.1 DATASET DESCRIPTION

From the PROMISE database, seven open-source initiatives built in Java were selected. It's a publicly available database that's utilized to forecast software flaws. These initiatives include an XML converter, a textual web search engine, and data communication drivers, among others. Table 1 contains information on every initiative, such as its title, two releases, overall documents, and fault percentage. Every initiative has two deployments, the first of which is utilized for training the framework and the second of which is employed to evaluate it. The initiatives in PROMISE include standard characteristics for every Java file. Table 2 lists all of the testing metrics.

**Table 1.** Description of the PROMISE dataset

| Project | Releases | Total files |
|---|---|---|
| camel | 1.0, 1.2, 1.4, 1.6 | 2784 |
| Jedit | 3.2, 4.0, 4.1, 4.2, 4.3 | 1749 |
| Poi | 1.5, 2.0, 2.5, 3.0 | 1378 |
| Synapse | 1.1, 1.2 | 478 |
| Xalan | 2.4, 2.5, 2.6, 2.7 | 3320 |
| Xerces | 1.2, 1.3 | 893 |

**Table 2.** Testing Metrics of PROMISE Dataset

| Testing Metrics | |
|---|---|
| Weighted methods per class | Lines of code |
| Depth of inheritance | Response of class |
| No. Of children | Data access metric |
| Coupling between object classes | Measure of aggregation |
| The measure of function abstraction | Afferent coupling |
| Lack of cohesion in methods | Efferent coupling |
| Cohesion among methods of a class | Inheritance coupling |
| Lack of cohesion in methods | Average method complexity |
| No, of public methods | Coupling between methods |

### 4.2 EVALUATION METRICS

The confusion matrix is utilized to demonstrate the performances of proposed frameworks. It incorporates the forecasting findings of the framework. It also produces the following groups of outcomes: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) (FN). Where TP represents the number of anticipated faulty codes which are already faulty, TN represents the number of anticipated non-faulty codes which are already non-faulty, FP represents the amount of anticipated faulty documents that are correct, and FN represents the amount of anticipated correct documents that are faulty. The used Metrics in this paper are discussed below:

$$\text{Accuracy} = (\text{TP} + \text{TN})/\text{TP} + \text{TN} + \text{FP} + \text{FN} \quad (4)$$

$$\text{Recall} = \text{TP}/\ (\text{TP}+\text{FN}) \tag{5}$$

$$\text{Precision} = \text{TP}/\ (\text{TP}+\text{FP}) \tag{6}$$

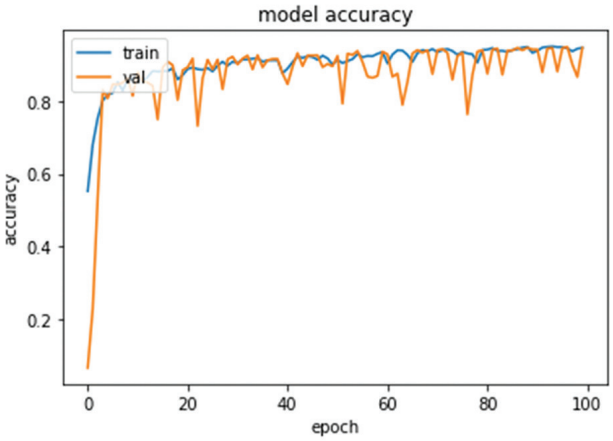
$$\text{F1\_Score} = \frac{2 * (\text{Precision} * \text{Recall})}{(\text{Precision} + \text{Recall})} \tag{7}$$

### 4. 3 IMPLEMENTATION DETAILS

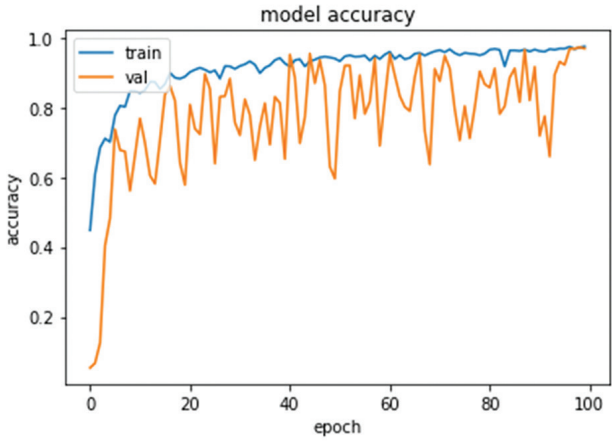
This section presents the training details of the proposed software testing model for fault prediction. For the experiment, we used python for training and testing purposes. For this, the entire dataset is divided into a 70:30 ratio. The minimum batch size for the training autoencoder was taken to be 128. The maximum epoch was taken to be 100. Notably, these model utilizes approx. 10GB RAM on high-performance GPU computing. Therefore, this model was trained on computing service provided by google i.e., google colab.

### 4. 4 RESULT ANALYSIS

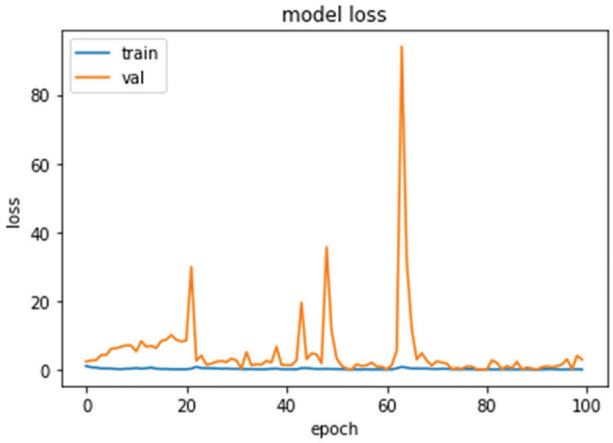
Fig-3(a)-3(f) shows the model accuracy for six datasets Camel Dataset, Jedit dataset, Poi dataset, Synapse dataset, Xerces dataset, and Xalan dataset. The accuracy is given for training and validation sets. The epoch varied from 0-100 at the x-axis. Because for every graph the convergence rate is for less than 100 epochs. Fig-3(a) shows the training and validation accuracy for Camel Dataset. The graph converges at initial epochs nearly in between 20-40. The average accuracy for Camel dataset is 95 %. Fig-3(b) shows the training vs validation accuracy over the number of epochs varied from 0-100 for JEDIT dataset. The accuracy varied from 0-20 epochs and then start to converge after 20 epochs. The average accuracy for JEDIT dataset is 97%. For the Synapse dataset, Fig-3(c) demonstrates the training versus validation accuracy across a range of epochs from 0-100. The accuracy ranged from 0 to 60 epochs, and after 60 epochs, it began to converge. The Synapse dataset has an average accuracy of 88 percent.Fig-3(d) shows the training vs validation accuracy for the Poi dataset over a range of epochs from 0-100. The accuracy ranged from 0 to 20 epochs, and it started to converge after 20 epochs. The Poi dataset has a 79 percent accuracy rate. The training vs validation accuracy over a range of epochs from 0-100 is shown in Fig-3(e) for the Xalan dataset. The precision fluctuated from 0 to 20 epochs, and it started to converge before 20 epochs. The accuracy of the Xalan dataset is 77 percent on average. Fig-3(f) shows the training vs validation accuracy for the Xerces dataset across a range of epochs from 0-100. The accuracy fluctuated from 0 to 20 epochs, with the accuracy beginning to converge after 20 epochs. The Xerces dataset is 95.4 percent accurate on average.



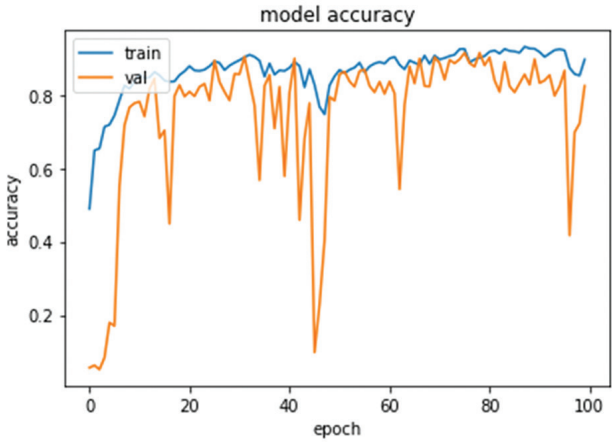

(a)



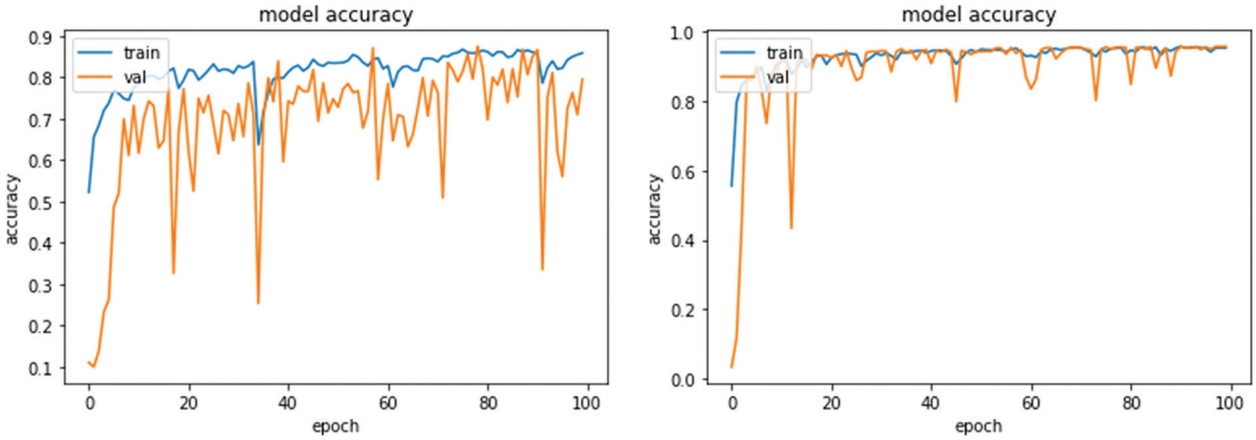

(b)




(c)




(d)




(e) (f)

**Fig. 3.** Training and Validation Accuracy over Given Datasets

Table 3 shows the Accuracy, Precision, Recall, and F–Measure comparison of all the datasets. For camel dataset Accuracy, Precision, Recall, and F–Measure is 95 %. For Jedit dataset accuracy is 97 %, precision is 96.7 %, Recall is 96.9% and F-Measure is 97 %. For Poi dataset accuracy is 82 %, precision is 80 %, Recall is 82 % and F-Measure is 79 %. For Synapse dataset accuracy is 82 %, precision is 77 %, Recall is 81% and F-Measure is 78 %. For Xalan dataset accuracy is 80 %, precision and Recall are 79 % and F-Measure is 77 %. For Xerces dataset accuracy is 96 %, precision is 95.6 %, Recall is 96 % and F-Measure is 95.4 %. Overall for all dataset average accuracy is 88.6 %, average precision is 85.5 %, average Recall is 88.3 % and average F-Measure is 86.9 %. The maximum accuracy I attained with the Jedit dataset i.e. 97 %. Maximum Precision is attained by Jedit i.e.96.7 %. Maximum recall and F-measure are also maximum for Jedit Dataset around 96.9 % and 97 % respectively. Table 4 shows the comparison of different techniques discussed in [15] like RF, DBN, CNN, RNN, and BCIL along with our proposed work. It is clear from the table that our proposed method surpasses all the techniques given in [15] for all six datasets used in our study. For camel dataset, our evaluated f-measure is 95 %, which is 55.4 % higher than the lowest F-measure attained in RF[15]. Similarly, for Jedit dataset F-measure is 97% which is 49% higher than DBN[15] technique. For poi dataset, our proposed work is 13% better than the RF dataset. For Xalan F-measure is 77% and for Xerces it is 95.4 which is 77% better than the RF[15].

**Table 3.** Performance of Proposed Model

| | Accuracy | Precision | Recall | F_measure |
|---|---|---|---|---|
| **Camel** | 0.95 | 0.95 | 0.95 | 0.95 |
| **Jedit** | 0.97 | 0.967 | 0.969 | 0.97 |
| **Poi** | 0.82 | 0.80 | 0.82 | 0.79 |
| **Synapse** | 0.82 | 0.77 | 0.81 | 0.78 |
| **Xalan** | 0.80 | 0.79 | 0.79 | 0.77 |
| **Xerces** | 0.96 | 0.95.6 | 0.96 | 0.954 |
| **Average** | 0.886 | 0.855 | 0.883 | 0.869 |

**Table 4.** F-measure of Proposed and State-of-art models

| Project | RF [15] | DBN [15] | CNN [15] | RNN [15] | CBIL [15] | SMO [18] | Ours |
|---|---|---|---|---|---|---|---|
| **Camel** | 0.396 | 0.335 | 0.505 | 0.515 | 0.935 | 0.427 | 0.95 |
| **Jedit** | 0.550 | 0.480 | 0.631 | 0.595 | 0.850 | 0.734 | 0.97 |
| **Poi** | 0.669 | 0.780 | 0.778 | 0.722 | 0.852 | 0.390 | 0.79 |
| **Synapse** | 0.414 | 0.503 | 0.512 | 0.487 | 0.889 | 0.617 | 0.88 |
| **Xalan** | 0.638 | 0.681 | 0.676 | 0.606 | 0.716 | 0.432 | 0.77 |
| **Xerces** | 0.185 | 0.261 | 0.311 | 0.262 | 0.951 | 0.612 | 0.954 |
| **Average** | 0.475 | 0.506 | 0.5688 | 0.5311 | 0.865 | 0.535 | 0.885 |

## 5. CONCLUSION

Software testing is the most critical element in the development life cycle since software systems are continually developing. Deep learning has subsequently made significant progress in automatically extracting semantic characteristics and improving software defects predictions performance and accuracy. This study presents the Correlation Clustering fine-tuned CNN (CCFT-CNN) model. It aids in improving code review and software testing by predicting the parts of source code that are faulty. The CCFT-CNN model is tested on the Camel Dataset, Jedit Dataset, Poi Dataset, Synapse Dataset, Xerces Dataset, and Xalan Dataset from the PROMISE dataset. The findings show that CCFT-CNN outperforms the existing models of Random Forest, DBN, CNN, RNN, and CBIL in terms of performance. The proposed CCFT-CNN outperforms the RF model, which has the lowest F-Measure in the baseline model, by 41%. For the DBN, CNN, and RNN models, the CCFT-CNN improves by 37.9%, 34.7 percent, and 35.4 percent, respectively. CBIL, on the other hand, has the best performance in the baseline model, and its comparison improves the outcome by 2%. Quality measures such as defect density, defect kinds, and defect severity should be included in the future. CCFT-CNN may also be used on a variety of open-source projects developed in a va-

riety of programming languages. Furthermore, to improve the quality of public datasets, new data preparation methods should be included.

## 6. REFERENCES


[1] F. Salfner, M. Lenk, M. Malek, "A survey of online failure prediction methods", ACM Computing Surveys, Vol. 42, No. 3, 2010.

[2] N. S. Chauhan and A. Saxena, "A green software development life cycle for cloud computing", IT Professional, Vol. 15, No. 1, 2013, pp. 28-34.

[3] J. Wang et al. "Robot: robustness-oriented testing for deep learning systems", Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering, Madrid, Spain, 22-30 May 2021.

[4] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, 1984, pp. 438-444.

[5] S. Anand et al. "An orchestrated survey of methodologies for automated software test case generation", Journal of Systems and Software, Vol. 86, No. 8, 2013, pp. 1978-2001.

[6] H. B. Braiek, F. Khomh, "On testing machine learning programs", arXiv:1812.02257, 2018.

[7] S. Abrecht et al. "Testing deep learning-based visual perception for automated driving", ACM Transactions on Cyber-Physical Systems, Vol. 5, No. 4, 2021, pp. 1-28.

[8] S. Gerasimou et al. "Importance-driven deep learning system testing", Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering, Seoul, Korea, 5-11 October 2020.

[9] J. Wang, et al. "Robot: robustness-oriented testing for deep learning systems Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering, Madrid, Spain, 22-30 May 2021.

[10] Q. Guo et al. "Audee: Automated testing for deep learning frameworks", Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, VIC, Australia, 21-25 September 2020.

[11] L. Ma et al. "Deepmutation: Mutation testing of deep learning systems", Proceedings of the IEEE 29th International Symposium on Software Reliability Engineering, Memphis, TN, USA, 15-18 October 2018.

[12] L. Ma et al. "Deepgauge: Multi-granularity testing criteria for deep learning systems", Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, September 2018, pp. 120-131.

[13] X. Du et al. "Deepcruiser: Automated guided testing for stateful deep learning systems", arXiv:1812.05339, 2018.

[14] J. Wang et al. "Adversarial sample detection for deep neural network through model mutation testing", Proceedings of the IEEE/ACM 41st International Conference on Software Engineering, Montreal, QC, Canada, 25-31 May 2019.

[15] A. B. Farid, E. M. Fathy, A. S. Eldin, L. A. Abd-Elmegid, "Software defect prediction using hybrid model (CBIL) of convolutional neural network (CNN) and bidirectional long short-term memory (Bi-LSTM)", PeerJ Computer Science, Vol. 7, 2021, pp. 1-22.

[16] Y. Fan, X. Xia, D. Lo, A. E. Hassan, Y. Wang, S. Li, "A Differential Testing Approach for Evaluating Abstract Syntax Tree Mapping Algorithms", Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering, Madrid, Spain, 22-30 May 2021., pp. 1174-1185.

[17] S. Goyal, "Handling Class-Imbalance with KNN (Neighbourhood) Under-Sampling for Software Defect Prediction", Artificial Intelligence Review 2021 553, Vol. 55, No. 3, 2021, pp. 2023-2064.

[18] Z. Sun, J. Zhang, H. Sun, X. Zhu, "Collaborative filtering based recommendation of sampling methods for software defect prediction", Applied Soft Computing, Vol. 90, 2020, p. 106163.