Software Vulnerability Assessment and Classification using Recurrent Neural Network

Authors: Ali Hussein, Azri Hj Azmi and Hafiza Abas
E-mail: hussein.a@graduate.utm.my
Razak Faculty Of Technology And Informatics, university teknologi (UTM)
Kuala Lumpur , malaysia

Keywords: Bug prediction; ensembles; segmentation; classification; neural network; class imbalance learning; re-sampling methods; software defect prediction.

Received: April 5/2023

Abstract: *The detection of software defects is useful technique for improving the quality of technology and testing management by providing rapid detection of deficiency simulation models until the actual testing phase starts. These prediction outcomes help designers of technology to effectively devote their available resources that are more vulnerable to deficiencies. In this paper, we propose a software bug prediction by using deep learning approach. We define Recurrent Neural Network for classification of source code including numerous soft computing techniques. Various pre-processing and data filtration techniques have been carried out for data balancing for normalization. TF-IDF and relation features extraction techniques have been used to generate Vector Space Model (VSM). The classification has done using RNN, according to training module for both training and validation dataset. The proposed deep learning framework consists of various optimization techniques where each has its strengths and limitations. We have evaluated all techniques and selected the best one. In order to perform the observations and benchmark the proposed method, various real-time and synthetic accessible databases are evaluated. The results of the assessment indicate that the proposed framework version is superior to other excellently basic models as well as deep learning classification models.*

1. Introduction

Recognition of vulnerabilities in source code or software has become an emerging field of research. Even though earlier studies have demonstrated the usefulness of multiple detection methods, models, and software vulnerability analysis tools in identifying source code vulnerabilities, the enhancement of the efficiency of such detection models and tools remains a major challenge for researchers. Annually, thousands of security issues are identified in virtual instruments, which are released publicly to the general vulnerability database exposed in obfuscated code. Threats also occur in indirect ways which are not evident to the concerned code inspectors or the programmers. It seems necessary to understand the dynamics of vulnerabilities that can lead to system issues directly from the raw data, with abundance of publicly available source code. In this work, we present an information approach to security technology by way of using deep learning. Stimulated by the success of similar researches in the area of recognizing the vulnerabilities in source code/software, we use a theoretical framework to examine its feasibility to aid in finding out the said vulnerabilities. The preliminary results indicate that within the domain of detecting attacks, the definition is feasible[1].

To bridge the domains gap, we can propose that each feature in a program be treated in computer vision as a neural network because fault detectors may only have to say whether a feature is insecure and fully explain vulnerability positions. That is, we want an intelligent interpretation of fault management programs. On either hand, one may recommend approaching each piece of code (i.e., comment, in this study, the two words are used synonymous) as a vulnerability detection unit. There are important exceptions to this diagnosis:
(i) most comments in a program may not contain any uncertainty, indicating that a few samples are susceptible;
(ii) multiple comments are not regarded as a whole that is semantically linked to each other [2].

Using traditional programming by utilizing k-means cluster analysis and the generative adversarial model, a scheme was introduced to check bugs in large amount random codes. To select the optimal code with an interactive analysis framework and software code refactor generation, k-means cluster transformation has been used. Use of a system, described on object-oriented code analysis documented in literature; in the instant case. The model is verified by the tasks of conceptual relationship analysis based on coding pairs and identification of sentiments.Moreover, research study, centered on what form of compilation with communication of massive source code has helped to recognize errors for inexperienced developers and suggest the steps needed to be taken on source code mistakes. The investigation uses the form of a message previously based on the coding system and detects specific code snippets' vulnerability [3].

It is therefore noted that, as classification algorithms for web application, bug identification and vulnerability classification, most investigators have used conventional semi-supervised classifiers, RNNsor CNNs. RNNs are far better than the standard language models, including such n-gram, but they have drawbacks in understanding long sequence data. Based on faults, functional programming identification, archive code identification, and basic error detection, almost all of the studies are found to have used various system software and classification models. On the other hand, the developed scheme of ours explicitly defines logic, grammar, and other system software errors. Additionally, in place of the error spot, the proposed model is used to predict the correct terms. Overall, in pursuing unique objectives, our suggested Language model varies from many other models [4].

This paper introduces a novel active tracking on deep learning to automatically learn features for predicting runtime environment weaknesses. In source code, where contingent code components are spread far apart, For example, combinations of code tokens that are needed to appear simultaneously due to the configuration of the computer program (e.g. in Java) or according to the configuration of API use (e.g. Lock () and activate ()), but do not accompany each other automatically are effectively handled. The interpretation of code symbols (semantic functionality) and the hierarchical structure of source code are appropriately reflected by the learned features (syntactic features). Our automated feature learning strategy removes the need for automated feature selection in conventional methods, which takes up much effort. Finally, testing the framework from a huge repository on many Java programs for the Desktop version reveals that our methodology is highly accurate in explaining code vulnerabilities[5].

2. Proposed System Design:

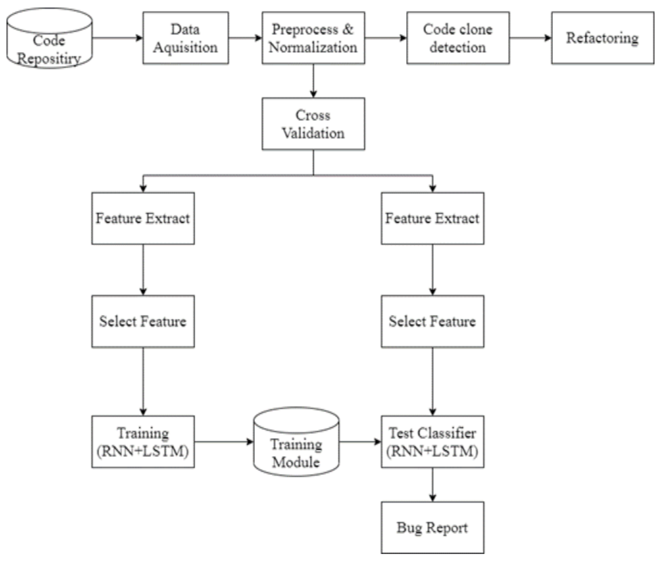

Fig.1: proposed system architecture design

shows a system overview of execution process flow, and delineates how it works with different algorithms.

2.1 Feature Execution

The function is compiled entire source code or modules with real statistics; in this method behavior is analyzed of code for vulnerability detection. During the analysis, four dynamic analysis methods have been used, fault infusion, mutation suitable starting, dynamic taint assessment and dynamic system check to generate the vector Space Model (VSM) from extracted features [6].

2.2. Pre-processing

A start of a few clone discovery come up to, the source code is separated as well as the area of the comparison is first decided. There are three basic types of goals in below steps.

Eliminate section of code: In this step source code uninteresting compared phase is removed.

Determine source units: By removing all the uninteresting code, remaining part of source code is divided in the arrangement of dissimilar sections known as source units.

Determine correlation units/granularity:

Source code parts should be auxiliary divided into smaller parts relying upon the evaluation method utilized a tool [6].

2.3: Extraction

Extraction changes program to the form that is correct while support to the real comparison algorithm. Conditional upon the device, it contains are as following:

Tokenization: If there should be an event of token-based methodologies, every source code line of the program is more dividing into tokens as showed by the lexical regulation of the program design platforms of importance. Apply different tokens of source code lines or forms after that frame of token systems to compare. The entire whitespace and comments between marks are removal from the token groups.

Parsing: For syntactic methods, the whole source code is described to prepare a parse tree or (potentially clarified) abstract syntax tree (AST). The source parts to be studied are then shown as sub trees of the describe tree or the AST, and correlation algorithms search for qualified sub trees to check as clone. Measurements based methodologies can utilize a parse tree depiction to discover clones taking into account sizes for sub trees.

Control and Data Flow Analysis: Semantics-related methodologies products program dependence graphs (PDGs) as of the source code. The nodes of a Program Dependence Graphs show the reports and circumstances of a system, while edges show to control and information conditions. Source units to be matched are shown as sub graphs of these PDGs. Different plans then search in favor of isomorphic sub graphs to discover clones. A few measurements based methodologies use sub graphs to compute info with control stream measurements [6].

2.4 Feature Selection

The various feature selection methods have been used during module training. The function is compiled entire source code or modules with real statistics; in this method behavior is analyzed of code for vulnerability detection. In a broader dataset, all of the variables are less necessary to consider; but, the greater the amount of variables, the greater the difficulty. As a result, it is often preferable to reduce the number of variables in a dataset and to use critical variables. We may reduce the parameter and find the variable's value in a dataset using a Function Selection technique. During the analysis, four dynamic analysis methods have been used, fault infusion, mutation suitable starting, dynamic taint assessment and dynamic system check. For generate the vector Space Model (VSM) from extracted features [6].

2.5 Vulnerability Detection

The vulnerability detection has been performed based on extracted features from the training data set. The vector space model has been generated according to extracted features like TF-IDF, relational features, and some bigram features. The classification has been done with recurrent neural networks, including long short term memory algorithm. This detection is also effective for of prevention of software-as-a-service attacks for web applications. The vulnerable code finds generate internal as well as external attacks and grant un-authorized access to external users. The major objective of detection vulnerability is automatic detection of exception handling and buffer overflow attack during the code execution. In the section proposed algorithm provides better detection accuracy in the code snippet [16].

3. Algorithm Design:

The algorithms furnished below have been used during the calculations of TF-IDF and weight score calculations using RNN.

```
TF-IDF:
Input: Input test instance that contains numerous
        tokens T[i…n]
Output: TF-IDF weight for all T[i]
Step 1:Data_vector = {Data1, Data 2, Data 3…. Data n}
Step 2: Words exist in entire dataset
Step 3: D = {cmt1, cmt2, cmt3, cmtn} and comments available in each document. Calculate the Tf score as
Step 4: tf (t,d) = (t,d)
        t= term
    d= document
Step 5: idf = t    sum(d)
Step 6: Return tf *idf
```

```
Recurrent neural network:
Input: Training dataset TestDBList [], Train dataset TrainDBList[] and Threshold th.
Output: Predicted class according to classification
Step 1: Read train data rules using below formula
```

$$\text{Train}[] = \sum_{n=1}^{k} (\text{Att}_n \ldots\ldots\ldots\ldots \text{Att}_k)$$

```
Step 2: Read test data rules using below formula
```

$$\text{Test}[] = \sum_{m=1}^{k} (\text{Att}_m \ldots\ldots\ldots\ldots \text{Att}_k)$$

```
Step 3: Calculate weight between input and hidden layer
```

$$\text{Instance}[w] = \sum_{n=1}^{k} (\text{Test}_n \ldots\ldots\ldots\ldots \text{Test}_n) \sum_{m=1}^{k} (\text{Train}_m \ldots\ldots\ldots\ldots \text{Train}_k)$$

```
Step 4: Generate feedback layer based on threshold policy
```

$$\text{Feed\_Layer}[] = \sum_{m=1}^{k} (\text{Feed\_Layer.optimized}\ ())$$

```
Step 5: Return  $\text{Feed\_Layer}[0].\text{class}$
```

4. Results and Discussion:

To validate the evaluation of the proposed bug forecast procedure, we have employed RNN classification algorithms that are gainfully utilized for fault prediction including unlabeled datasets. The performance evaluations of software defect prediction are based on the confusion matrix, as shown in Table 1, which includes the measures of precision, recall, as well as F-score.

Actual	Predictive	
True	TP (true positive)	FN (false negative)
False	FP (false positive)	TN (true negative)

Table 1: Confusion Matrix Evaluation

True positive (TP):The number of fake entities anticipated as fake.
False negative (FN): The number of fake entities anticipated as normal.
False positive (FP): The number of normal entities anticipated as fake.
True negative (TN): The number of normal entities anticipated as normal.
In this research, analytical performance procedures are calculated as follows:
Precision: It shows the proportion of faulty identities receive adequate as faulty of all desired objects.
Recall: It is the percentage of faulty identities to all entities that are currently faulty is the proportions of recall.
F-measure: It is the cumulative recall and precision average, with higher estimated coefficients matching higher predictive efficiency.

$$F - Measure = \frac{2 * Precision * Recall}{Recall + Precision}$$

$$Precision = \frac{TP}{TP + FN} \qquad Recall = \frac{TP}{TP + FP}$$

To evaluate the proposed system, we have used machine learning classifiers like ANN, SVM, Adaboost. Also, we have used deep learning framework of RNN with LSTM by using activation functions like Sigmoid, Tanh and ReLU. The results of classification accuracy with confusion matrix with 20 folds cross-validation for all algorithms are shown in Table 2. Measures used to compare the algorithms are Accuracy, Precision, Recall and Micro-score. From the observations, it can be concluded that RNN (ReLU) gives highest performance among the all.

4.1 Experiment using Artificial Neural Network:

The figure 2 shows the classification accuracy of the ANN classification algorithm. Initially, it has been trained using inbuilt functions from the weka tool. Numerous cross-validation techniques have been used for classification, and various parameters has tuned for ANN during the classification. This approach can classify each validation according to probability function, that the reason this algorithm bit high error rate than other supervised classification algorithms.

Table 2: accuracy and confusion matrix for ANN

ANN	Fold 10	Fold 15	Fold 20
Accuracy	85.20	84.20	85.60
Precision	83.60	82.30	84.99
Recall	87.50	85.40	77.72
Micro-Score	85.05	83.35	81.10

The ANN model is easy to build and particularly useful for very large data classification using supervised machine learning technique or Artificial Intelligence (AI). Along with simplicity, ANN is known to outperform even highly sophisticated classification methods. The proposed ANN predicts the possibility for individual instance according to current values.

Figure 2 shows the performance evaluation calculation of ANN classification with 20-fold classification. It achieves around 85.60% accuracy for the given input dataset. We used a multinomial event model, samples represent the frequencies with which certain events have been generated by a multinomial probability of that particular event and based on that probability system predicts the final class.

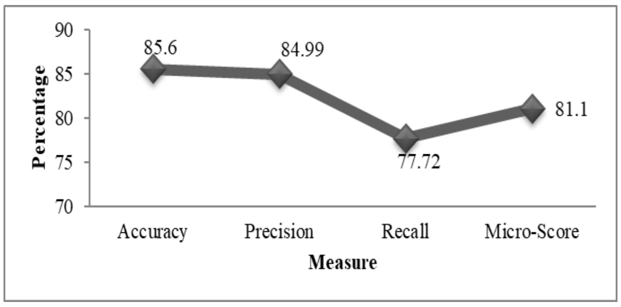

Fig. 2: Analysis of bug and vulnerability detection using ANN with 20-fold data cross validation

4.2 Experiment using Support Vector Machine (SVM):

The below table 5.3 depicts the classification analysis with various cross validation, we conclude 20 fold cross-validation provides the highest 95.2% classification accuracy for SVM.

table 3: accuracy and confusion matrix for SVM

SVM	Fold 10	Fold 15	Fold 20
Accuracy	91.20	91.70	95.20
Precision	91.35	92.10	94.80
Recall	92.30	93.10	96.20
Micro-Score	91.35	92.20	94.75

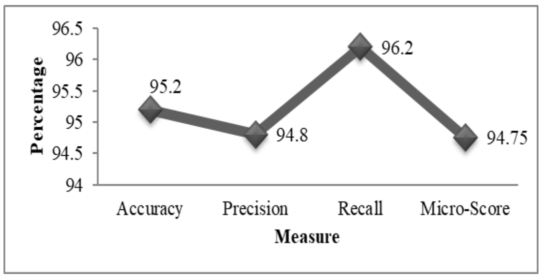

Fig. 3: Analysis of bug and vulnerability detection using SVM with 20-fold data cross validation

Figure 3 describes SVM for 20-fold cross-validation. The labeling circumstances to construct a training set become moment and expensive in many machine learning; it is also helpful to find strategies to reduce supervised classification numbers. By improving performance, the Kernel-Based algorithm has been used to minimize occurrences. We classify all in this algorithm as a point in n-dimensional spaces only with respect of a property direction being the meaning of each characteristic by classification technique; we detect clones by finding the hyper-plane that separates the two groups very well.

4.3 Experiment using Adaboost:

The below table 4 depicts the classification analysis with various cross validation, we conclude 20 fold cross-validation provides the highest 81.30% classification accuracy for Adaboost.

table 4: accuracy and confusion matrix for Adaboost

Adaboost	Fold 10	Fold 15	Fold 20
Accuracy	70.60	78.50	81.30
Precision	72.30	73.50	74.50
Recall	69.90	68.50	70.30
Micro -Score	70.60	71.90	72.30

Adaboost is adaptive in that it tweaks future weak learners in favor of cases misclassified by prior classifiers. It may be less prone to the over fitting issue than other learning algorithms in certain situations. Individual learners may be poor, but as long as their performance is somewhat better than actual guessing, the overall model will converge to a powerful learner.

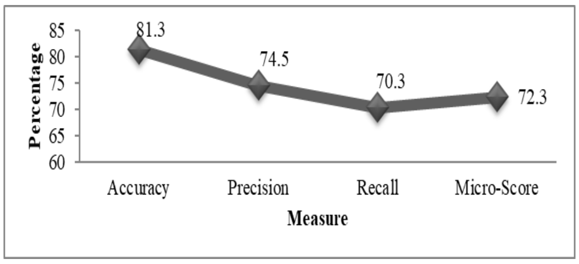

Fig. 4: Analysis of bug and vulnerability detection using Adaboost with 20-fold data cross validation

The figure 4 describes Adaboost classification for fake account detection for 20-fold cross-validation. AdaBoost is a specific training technique for boosted classifiers.

$$F_T(x) = \sum_{t=1}^{T} f_t(x)$$

A boost classifier is a kind of classifier.

Each Ft is a weak learner that accepts an object x as input and returns a value that indicates the object's class. The sign of the weak learner output, for example, specifies the predicted object class in the two-class issue, whereas the absolute value indicates the confidence in that classification. Similarly, if the sample belongs to a positive class, the Tth classifier is positive; otherwise, it is negative.

4.4 Experiment using Recurrent Neural Network (Sigmoid):

we demonstrate classification accuracy of RNN (Sigmoid) using synthetic dataset, the similar experiments has done with various cross validation and results has illustrated in table 5. According to this analysis we conclude 20 fold cross validation provides highest 96.10% classification accuracy using RNN with Sigmoid function.

table 5: accuracy and confusion matrix for RNN (Sigmoid)

RNN (Sigmoid)	Fold 10	Fold 15	Fold 20
Accuracy	95.60	95.90	96.10
Precision	95.80	96.10	97.00
Recall	95.80	96.00	96.30
Micro-Score	94.70	95.90	96.05

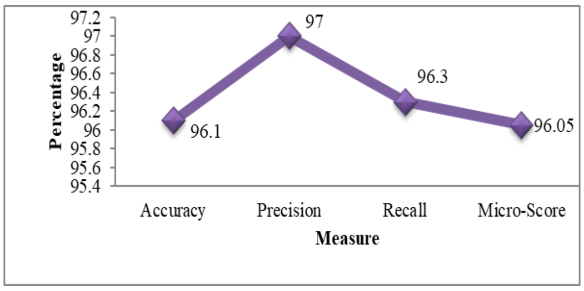

Fig. 5: Detection of accuracy using RNN (Sigmoid) with 20-fold data cross validation

The 20-fold cross validation also achieves 96.10% with RNN with sigmoid function, have been explained in Figure 5, this RNN functions achieve around higher accuracy over the traditional machine learning algorithms during module testing.

4.5 Experiment using Recurrent Neural Network (Tanh):

The figure 6 shows classification accuracy of RNN, the similar experiments has done with various cross validation and results are illustrated in table 6. According to this analysis we conclude that 20 fold cross validation provides highest 97.25% classification accuracy for RNN using Tanh.

Table 6: Classification accuracy with confusion matrix for RNN (Tanh)

RNN (Tanh)	Fold 10	Fold 15	Fold 20
Accuracy	96.90	97.50	97.25
Precision	97.00	97.40	97.60
Recall	97.30	97.50	97.30
Micro-Score	96.80	96.70	96.90

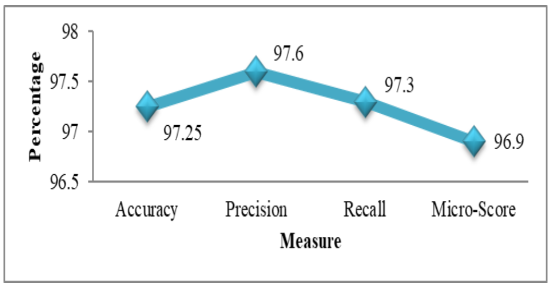

Fig. 6: Detection of accuracy using RNN (Tanh) with 20-fold data cross validation

4.6 EXPERIMENT USING RECURRENT NEURAL NETWORK (ReLU):

In this experiment we analyse the classification accuracy of ReLU using synthetic dataset, the similar experiments has done with various cross validation and results has illustrated in table 7. According to this analysis we conclude system provides highest 97.5% accuracy for 20-fold cross validation classification accuracy for RNN.

Table 7: Classification accuracy with confusion matrix for RNN (ReLU)

RNN (ReLU)	Fold 10	Fold 15	Fold 20
Accuracy	97.20	97.90	97.50
Precision	97.40	96.90	97.60
Recall	95.60	97.20	97.90
Micro-Score	96.20	95.80	97.20

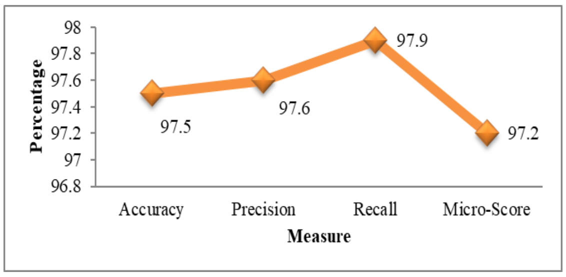

Fig. 7: Detection of accuracy using RNN (ReLU) with 20-fold data cross validation

Above experiments describes a proposed deep learning classification algorithm with a machine learning algorithm. This figure describes the result with and without cross-validation. We have used a minimum of three hidden layers for the detection of code clone. Using this experiment, we conclude RNN with sigmoid provides better detection accuracy than the other two activation functions as well as random forest machine learning algorithm.

In table 8, we have compared all the results of above experiments.

Method / Measure	ANN	SVM	Adaboost	RNN (Sigmoid)	RNN (Tanh)	RNN (ReLU)
Accuracy	85.60	95.2	81.30	96.10	97.25	97.50
Precision	84.99	94.80	74.50	97.00	97.60	97.60
Recall	77.72	96.2	70.30	96.30	97.30	97.90
Micro-Score	81.10	94.75	72.30	96.05	96.90	97.20

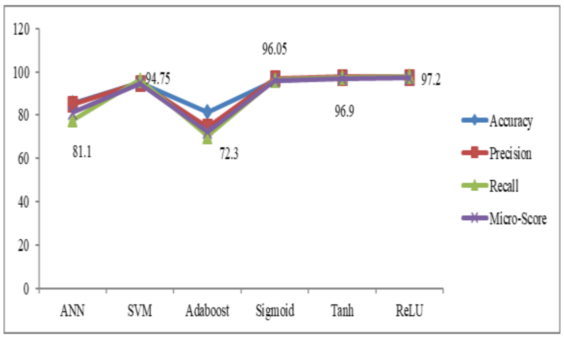

Fig. 8: Classification accuracy with 20-fold cross-validation for all methods

the proposed method obtains the best predictive performance. The suggested solution can be further tested when used in actual software applications. The three data splitting mechanism has use as 10, 15 and 20 fold cross-validation.

table 9: Dataset description of source code extracted from android APK files

Total Size	2500
Training Samples	2000
Testing Samples	500

System describes four evaluations between this research results and the some existing systems results has calculated on the similar as well as multiple dataset.

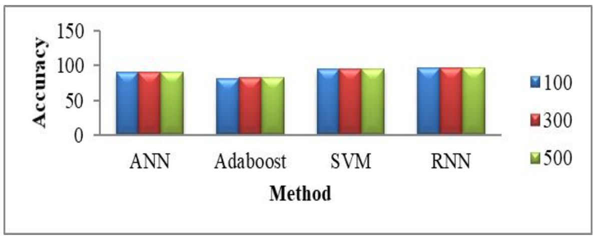

Fig. 9: Comparative analysis of proposed vs. existing classification for vulnerability detection shows two machine learning algorithms used. This figure depicts the proposed RNN provides better detection accuracy over machine learning algorithms.

A classification model is generated using this arrangement or learning set to organize the input courses into corresponding template files or labels. Then a test set is used by gleaning the class labels of orthonormal courses to validate the model. A variety of neural networks are used to identify reviews, such as ANN and Support Vector Machines (SVM) and Adaboost.

5. Conclusion and Future Work:

The vulnerability detection is very tedious work for imbalance source codes; vulnerable code allows generating software attack to remote user. Sometimes, during execution the vulnerable code also generates internal attacks like

buffer overflow, session hijack, bypass authentication etc. In literature, many problems are detected in software every year. Vulnerabilities mostly does not appear in hidden the forms which the software testers can identify. This

system describes the method of finding drawbacks by utilizing deep learning.

In this paper, we have developed a RNN including LSTM for constructing code vulnerability detection and bug triage on various platforms. Numerous tools are not able to support a web-based application to find code vulnerability. The proposed system works on different datasets for feature extraction and is able to detect the vulnerability. RNN provides a better result over traditional machine learning classifiers.

In future, developers needed to detect the code triage for runtime mobile-based application programs, because the existing tools do not support mobile application programs. Another need in software engineering is code clone management. Good quality of design can be achieved with the help of bugs free code clone in developing software.

References:

1- Terada, K.; Watanobe, Y., "Automatic Generation of Fill-in-the-Blank Programming Problems", In Proceedings of the 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, 1–4 October 2019; pp. 187–193.

2- Tai, K.S.; Socher, R.; Manning, C.D., "Improved semantic representations from tree-structured long short-term memory networks", In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing, Beijing, China, 26–31 July 2015; pp. 1556–1566.

3- Pedroni, M.; Meyer, B., "Compiler error messages: What can help novices?", In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, Portland, OR, USA, 12–15 March 2008; pp. 168–172.

4- Saito, T.; Watanobe, Y., "Learning Path Recommendation System for Programming Education based on Neural Networks", Int. J. Distance Educ. Technol. (Ijdet) 2019, 18, 36–64.

5-Teshima, Y.; Watanobe, Y., "Bug detection based on LSTM networks and solution codes", In Proceedings of the 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Miyazaki, Japan, 7–10 October 2018; pp. 3541–3546.

6- Fan, G.; Diao, X.; Yu, H.; Yang, K.; Chen, L., "Software Defect Prediction via Attention-Based Recurrent Neural Network", Sci. Program. 2019, 2019, 6230953.

7- Ohashi, H.; Watanobe, Y., "Convolutional Neural Network for Classification of Source Codes", In Proceedings of the 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Singapore, Singapore, 1–4 October 2019; pp. 194–200.

8- Zhou, Y.; Tong, Y.; Gu, R.; Gall, H., "Combining text mining and data mining for bug report classification", J. Softw. Evol. Process 2016, 28, 150–176.

9- Jin, K.; Dashbalbar, A.; Yang, G.; Lee, J.-W.; Lee, B., "Bug severity prediction by classifying normal bugs with text and meta-field information", Adv. Sci. Technol. Lett. 2016, 129, 19–24.

10- Goseva-Popstojanova, K.; Tyo, J., "Identification of security related bug reports via text mining using supervised and unsupervised classification", In Proceedings of the IEEE International Conference on Software Quality, Reliability and Security, Lisbon, Portugal, 16–20 July 2018; pp. 344–355.

11- Kukkar, A.; Mohana, R., "A Supervised bug report classification with incorporate and textual field knowledge", Procedia Comp. Sci. 2018, 132, 352–361.

12- Tong, H.; Liu, B.; Wang, S., "Software defect prediction using stacked denoising auto encoders and two-stage ensemble learning", Inf. Softw. Technol. 2018, 96, 94–111.

13- More, A., "Survey of resampling techniques for improving classification performance in unbalanced datasets" arXiv 2016, arXiv:1608.06048.

14- Rawat, M.S.; Dubey, S.K., "Software defect prediction models for quality improvement: A literature study", IJCSI Int. J. Comput. Sci. Issues 2012, 9, 288–296

15- Neuhaus, S.; Zimmermann, T.; Holler, C.; Zeller, A. "Predicting Vulnerable Software Components", In Proceedings of the 14th ACM conference on Computer and Communications Security, Alexandria, VA, USA, 28–31 October 2007; pp. 529–540.

16- Markad Ashok Vitthalrao, Mukesh Kumar Gupta, "Software Vulnerability Classification based on Machine Learning Algorithm", International Journal of Advanced Trends in Computer Science and Engineering (IJATSCE), ISSN 2278-3091, Volume 9, No.4, July – August 2020, Page No.6653-6659.

17- Markad Ashok Vitthalrao, Mukesh Kumar Gupta, "Software Vulnerability Classification based on Deep Neural Network", International Journal of Engineering and Advanced Technology (IJEAT), ISSN: 2249-8958 (Online), Volume-9 Issue-1, October 2019, Page No.3146-3150.