IJITCE

ISSN 2347- 3657

International Journal of Information Technology & Computer Engineering

Email : ijitce.editor@gmail.com or editor@ijitce.com

International Journal of Information Technology & Computer Engineering

ISSN 2347–3657

Volume 6, Issue 3, 2018

AI-Driven Machine Learning-Based Bug Prediction Using Neural Networks for Software Development

[1]Rahul Jadon
F5 Networks
USA
rahul.jadon0@gmail.com

[2]Aiswarya RS
Bethlahem Institute of Engineering,
Nagercoil, India
aiswaryars112@gmail.com

Abstract

Software development for project success. Conventional bug detection techniques like human code review and rule-based inspection are powerless to deal with complexity within large dynamic codebases. This article introduces a machine learning-based bug predictor system that entails the application of neural networks via a Gated Recurrent Unit (GRU) model to identify potential bugs in software. The GRU model is trained upon the source code, commit history, and bug reports of popular code bases such as GitHub and GitLab. The system employs token-based processing of data through the tokenization, text homogenization, and feature extraction processes. Preprocessed data is used to input into the GRU model in order to predict bugs. The performance of the model is measured with impressive metrics such as accuracy of 98.75%, precision of 97.90%, recall of 96.62%, F1-score of 97.45%. The results show that the GRU-based model performs well in code bug prediction, improving software quality, and lowering maintenance costs in the future. Some of the limitations of bug prediction using neural networks, including high computational cost, un-interpretable models, and large amounts of labeled data, are also discussed in the paper. Methods like model pruning, transfer learning, and explainable AI are debated controversially regarding overcoming limitations and being capable of being utilized more effectively in real-world applications.

Keywords: *Machine Learning, Bug Prediction, Gated Recurrent Unit (GRU), Neural Networks, Software Development, Accuracy, AUC-ROC, Tokenization, Feature Extraction.*

1. Introduction

Software development, having high-quality code up top and low bug-counts down is more imperative than ever before [1]. Human-done code review methods and rule-driven testing are now no longer that appropriate considering that they have very long cycle times and cannot keep up with maintaining complex, big codebases [2]. The addition of ML and AI has been the most important aspect through history-making innovations in automated bug predictor systems [3]. Among various ML approaches, neural networks have proved to be very efficient as they can learn nonlinear and complex patterns of data [4]. The models can be trained on vast sets of source code and bug reports such that they recognize repeating patterns with software faults. Because they are predictive, the models allow development teams to discover potential bugs at earlier phases in the software life cycle, increasing overall reliability and reducing long-term maintenance costs [5]. Neural network architectures such as feedforward networks, CNNs can be implemented to handle a range of input from structured code to sequential programming flow, and thus they are highly adaptable based on current software engineering demands [6].

Several reasons are to blame for the widespread occurrence of bugs in modern software systems, and most of them are due to greater complexity and quicker development rates [7]. Developers are under pressure to deliver features early on tight schedules, and during this process, poor coding can take place, with not much time left for adequate testing [8]. Team-based development environments especially with open-source coders or large development teams add complexity in the guise of variable coding conventions, integration issues, and opposing logic [9]. With growing size and sophistication of software systems, even traditional bug-detection methods are stumped to deal with sheer quantity and variability of code to allow malicious bugs to remain undiscovered. The build and release pipelines that fuel DevOps cycles today also fuel the risk of adding new bugs with every release [10]. Dynamic dependencies, third-party libraries, and platform variability also add unknown code execution features. These create a need for intelligent, automated bug detection systems. Machine learning through the application of neural networks provides a scalable solution by learning past trends in source code and bugs to detect defects at an early stage and accurately [11].

While their potential power is exciting, bug prediction utilities based on neural networks suffer a number of very significant disadvantages to general use [12]. Among the most significant is extremely high computational demand required to train and execute deep learning algorithms, which often calls for specialized hardware such as expensive high-end GPUs or TPUs. This can be a cost barrier, particularly to small- to medium-sized development teams [13]. There is another serious concern that is the uninterpretability of these models. Neural networks are generally regarded as "black-box" systems, and it is difficult for developers to understand or have confidence in the reasoning that goes into a specific prediction [14]. That untransparency is problematic in mission-critical systems where decisions must be traceable. Furthermore, these models depend heavily on large, well-labeled, high-quality datasets for effective training something that is not always easily available, especially for newer or proprietary projects. There is also the potential for overfitting, where the model performs well on training data but struggles to generalize to unseen code. Finally, incorporating these advanced systems into existing development workflows can be technically difficult and require additional expertise and infrastructure [15].

Neural network-based bug prediction, several efficient solutions have been suggested and proven by the software development community. Transfer learning, model pruning, and quantization greatly minimize the computational needs, making models free to execute on humble hardware as well. All these optimizations enable one to leverage sophisticated bug prediction in real-time and at scale. To address the "black-box" problem, explainable AI (XAI) techniques are being proposed increasingly that provide transparency in model operation and enable developers to understand why a particular line of code is deemed buggy. Publicly accessible repositories such as GitHub, with big data at disposal, and open-source bug-labeled datasets help train more solid and accurate models. In addition, cloud machine learning platforms like Google Cloud AI and AWS Sage Maker enable effortless model deployment and scaling with limited on-premises infrastructure. Paired with the contemporary MLOps practices and automated CI/CD pipelines, these technologies enable organizations with tools to easily roll out neural bug prediction tools into their process and improve software quality, development speed, and overall product quality.

1.1 Objectives

- Describe ML and AI deployment for automation of software bug prediction in software development.
- Describe the capacity of neural networks in predicting software bugs by learning source code and bug reports patterns.
- Deploy neural network-based models early for bug prediction in the development stage to improve software quality and reduce maintenance costs.
- Explain performance of various neural network architectures, such as Feedforward Networks, CNNs, and RNNs, on applications to bug prediction.
- Propose remedies such as interpretable AI and model optimization techniques to address the computational requirement and model interpretability bug forecast issues.

2. Literature Survey

The proposed paper discusses a new IoT-based health framework that supports patient monitoring through smooth integration with cloud infrastructures [16]. It resolves scalability and quality issues in the data by using k-NN for imputation of missing values, Z-score normalization, and ChaCha20 encryption for secure data storage. Data from IoT sensors is preprocessed, encrypted, and stored in the cloud for effective management. Performance metrics reveal higher encryption levels and latency times with growing data, verifying the efficiency of the architecture for real-time healthcare monitoring [17]**.** A safe document clustering scheme for IoT applications is proposed through the integration of Multivariate Quadratic Cryptography (MQC) with Affinity Propagation (AP). Strong encryption ensures confidentiality of the data while adaptive encrypted document clustering is performed by the system. With an objective to alleviate scalability issues, efficiency of clustering, and computation overhead, the proposed framework enhances secure sharing of data in IoT applications like smart cities and healthcare. Test results confirm accuracy improvement, security boosting, and better performance, positioning it for the right handling of sensitive IoT information [18]**.**

This research examines the impacts of internet-based finance and cloud computing on urban-rural income disparities in an e-commerce era [19]. With its ability to provide financial inclusion and online connectivity, they can equal economic development. Panel data estimation over a period of years on proxy economic and demographic data sets is used in the research to determine how digital finance usage induces income levels and decreases interregional income disparities in a fast-changing digital economy [20]. Cloud computing infrastructure must be optimized to enhance big data processing performance, scalability, efficiency, and cost. The principal challenges are security, energy efficiency, resource utilization, and system dependability [21]. Load balancing, auto-scaling, and dynamic allocation of resources are key measures. Employing vertical and horizontal scaling, high-grade security controls, and power-saving controls support firm operations [22]. Automation,

monitoring in real time, and support for compliance ease the cloud environment, reduce expenses, and ensure a robust infrastructure to support diverse workloads. [23] targets machine learning application for improving chronic disease management in the elderly. As per evidence developed by the SURGE-Ahead Project, it targets developing customized AI-based tools for geriatric care. Its aim is to improve decision-making at the clinic utilizing solid information for the real-time prediction of a specific patient. Methods involve utilizing Support Vector Machines, Decision Trees, and Neural Networks, facilitated by feature selection and preprocessing aid, in attempts to effectively predict chronic disease and allow for prompt, individualized healthcare intervention.

[24] investigates how cloud computing and AI-powered sentiment analysis are revolutionizing customer relationship management (CRM). Through the analysis of customer interactions via various channels, AI models are able to segment sentiments and predict behavior, enabling specific communications strategies. The aim is to enhance customer satisfaction, engagement, and retention through sentiment-based CRM strategies. Using cloud platforms promises scalability and real-time processing to enable companies to deliver customized, data-driven responses in sync with the sentiment and expectations of the customers, thus improving the overall relationship management [25]. This paper presents an AI-based architecture for secure mHealth systems using Hierarchical Identity-Based Encryption (HIBE), Role-Based Access Control (RBAC), and Secure Multi-Party Computation (SMC). Data sharing is kept private with hierarchical encryption of data and role-based data access. AI facilitates effective role delegation and computation of data for security and scalability. The architecture effectively performs privacy and collaboration requirements and thus is an effective solution to securely and efficiently manage mHealth data.

2.1 Problem statement

AI in critical sectors like healthcare, finance, and city management has raised concerns about data security, scalability, and processing speed [26]. Modern systems generally lack the capacity to deal with growing complexity and size of data, especially when dealing with real-time and sensitive data [27]. Long-standing issues are data heterogeneity, differing data quality, privacy breaches, high computational intensity, and unbalanced access to digital infrastructure [28]. These issues undermine the performance and fairness of digital services. Thus, intelligent and adaptive machine learning paradigms with integrated robust encryption techniques, self-tuning clustering mechanisms, and scalable processing models are a pressing need [29]. Such systems need to be able to guarantee safe handling of data, privacy, and efficient provision of service in heterogeneous domains. Solving such challenges will aid in the construction of robust, fair, and efficient systems with the ability to support next-generation digital infrastructure and real-time analytics requirements in complex environments [30].

3. Proposed Methodology

The method employs a GRU model to make bug predictions in source code. It starts from data collection, where data about software such as source code files, commit history, and bug reports are collected from platforms such as GitHub or GitLab. The data collected is tokenized and cleaned using text cleaning to format and sanitize the data for model input. Feature extraction is performed to get semantically correct code metrics and features. These are fed into the GRU model, which is trained to separate bug-prone code snippets and bug-free code snippets. The GRU structure recognizes sequential patterns within the code in a way that allows for early detection of bugs and consequently better software quality and reduced development time. Figure 1 shows the GRU unit architecture for bug prediction.

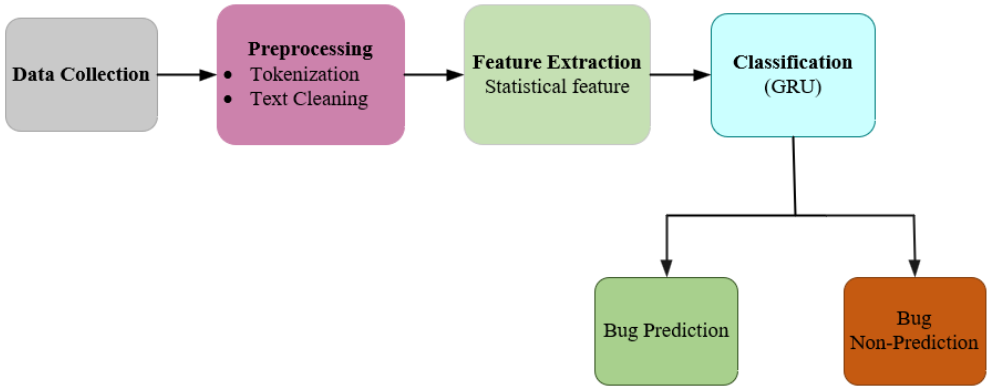

Figure 1: GRU Unit Architecture for Bug Prediction

3.1 Data Collection

Data Collection is the initial step of the bug prediction process in which software-related data from different authentic sources is combined. It involves the extraction of source code files, commit history, version control information, issue tracking information, and bug fix comments from the repositories like GitHub, GitLab, or Bitbucket. Metadata such as timestamps, contributor details, and commit messages are also extracted to provide context to changes in the software. The data gathered is the main input for the following preprocessing and model training processes to allow the system to learn historical patterns and provide precise predictions of bug-prone modules. Diverse, high-quality, and representative data are essential to improving the performance and generalizability of predictive models.

3.2 Preprocessing

Preprocessing is an important process that cleanses raw software data to ready it for effective modeling and analysis. Preprocessing is a series of important steps starting with tokenization, by which the source code is disassembled into very small units like identifiers, operators, and keywords in an effort to make the syntax comprehensible. There is also text cleansing for the elimination of unnecessary features like comment, whitespaces, and unnecessary special characters from bug prediction. It is performed to prevent the dataset having solely meaningful words, eliminating noise, and enhancing learning ability of the model. Preprocessing and normalizing input data lead to enhanced quality of feature extraction and straight contribution to efficiency and accuracy of a classification model on the basis of a neural network.

3.2.1 Tokenization

Tokenization is the process of transforming source code into a stream of small meaningful units referred to as tokens that reflect the syntactic form of the code. Tokens can be keywords such as if for return identifiers such as variable and function names literals such as constants operators such as addition or comparison and punctuation characters such as semicolon or braces. Tokenization converts unstructured code into structured form that is machine learning ready. It holds patterns of syntax and assists in neural networks through the capability of representing code as a numerical and analyzable form.

$$T(C) = \{t_1, t_2, \dots, t_n\} \quad (1)$$

Where represent C the original source code, $T(C)$ represent the tokenization function applied to C, $t_1, t_2, \dots, t_n$ are the resulting tokens such as keywords, identifiers, literals, and operators.

3.2.2 Text Cleaning

Text Cleaning is a crucial preprocessing technique of stripping distracting or noisy data from the source code to enhance input data quality for machine learning algorithms. It involves stripping comments, unneeded whitespaces, special characters, and non-executable characters that don't form part of the semantic construct of the code. By minimizing noise, text cleaning increases the cohesiveness and intelligibility of the code representation, which better suits precise feature extraction and bug prediction. Cleaned code lets neural networks focus on meaningful patterns instead of distracting with redundant data.

3.3 Feature Extraction

Cleaned, tokenized source code into a well-structured array of numerical features that represent underlying attributes relevant to bug prediction. Such features may be static code metrics such as lines of code, cyclomatic complexity, number of methods, and historical metrics such as frequency of commits or prior bug occurrence. Besides, semantic attributes are obtained using embedding techniques like mean, variance Standard Deviation or AST-based representations that preserve the syntactic and contextual meaning of code. Through this transformation, neural network models are able to learn patterns and relationships in the data pointing to buggy or error-prone modules, thus facilitating intelligent and accurate bug prediction.

3.3.1 Mean

Mean or average is a basic statistical metric that determines the central tendency of data. Mean is determined by adding up all the individual values in a dataset and then dividing the sum by the number of values. Mean provides an estimate of the approximate "center" or mean value of the data. Mean can be represented mathematically as:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \quad (2)$$

Where μ This is the mean or average of the data set. N represents is the number of values, x_i These are individual values in the data set and $\sum_{i=1}^{N} x_i$ This is the sum of all the values in the data set.

3.3.2 Variance

Variance measures the spread or dispersion of data points from the mean. It tells us how far, on average, each data point in the dataset is from the mean. The larger the variance, the more dispersed the data points are, while the smaller the variance, the closer they are to the mean. Mathematically, variance is the mean of the square of the difference between every value and the mean of the dataset. Variance is represented by the formula:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} (x_i - \mu)^2 \quad (3)$$

Where σ^2 is the variance, x_i is each individual value in the dataset, μ is the dataset mean and N is the number of values in the dataset.

3.3.3 Standard Deviation

Standard Deviation quantifies how much individual points in a dataset vary from the mean. It provides a better sense of the spread of the data since it is in the same units as the original data, unlike variance, which is in squared units. Standard deviation is simply variance's square root, and it tells us on average to what degree data values vary from the mean. Formula for standard deviation is:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N} (x_i - \mu)^2} \quad (4)$$

Where σ is standard deviation, x_i is every individual value of the dataset, μ is mean of the dataset and N is number of values in the dataset.

3.4 Classification Using GRU

Classification (GRU) is the application of a neural network to examine the features extracted from source code and predict whether or not a code snippet would have bugs. GRUs are a form of RNN that are particularly well suited to processing sequential data, like sequences of tokens or temporally ordered software modifications. They learn long dependencies and context of code features in an effective way by using gate mechanisms that govern information exchange as well as memory. The input feature sequence in the GRU model is employed to generate binary or probability class bug-prone or bug-free status. Classification of this sort facilitates early indication of possible defects in software that increases software quality as well as debugging figure 2 shows the gated recurrent unit of architecture.

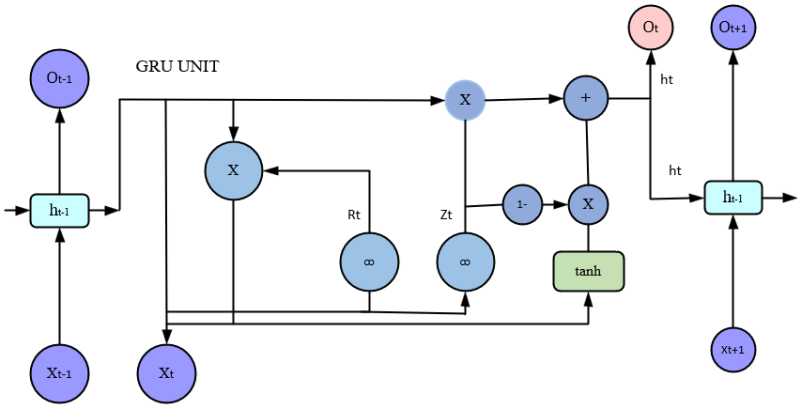

Figure 2: Architecture of Gated Recurrent Unit

The process begins utilizing the current input X_t and passed hidden state h_{t-1} as inputs to the GRU cell and tries to compute its two most significant gating mechanisms: Reset gate and update gate. These gates govern how data flows through the unit and how information gets updated in memory. Reset gate R_t determines the percentage of the previous hidden state h_{t-1} to be forgotten. It is given as sigmoid activation of weighted sum of previous hidden state and input at time step t:

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 6, Issue 3, 2018

$$R_t = \sigma(W_r \cdot X_t + U_r \cdot h_{t-1} + b_r) \quad (5)$$

This gate allows the model to selectively ignore unessential past history information when generating the new candidate hidden state. secondly, the update gateZ_t is computed to decide how much past information from h_{t-1} to retain and how much new candidate state to add. It is also computed with a sigmoid function:

$$Z_t = \sigma(W_z \cdot X_t + U_z \cdot h_{t-1} + b_z) \quad (6)$$

A large Z_t value leads to a small update from the previous memory; conversely, a small Z_t value causes the network to update more from new incoming data. The next candidate hidden state $\tilde{h}_t$ is calculated from the reset-transformed hidden state and new input. This is when the GRU produces a new candidate state at the current time step:

$$\tilde{h}_t = \tanh\left(W \cdot X_t + U \cdot (R_t \odot h_{t-1}) + b\right) \quad (7)$$

This equation ensures that the GRU dynamically decides whether to retain the old information or update it with the new candidate state, enabling it to capture long-term dependencies efficiently. the hidden state h_t is then passed as output O_t and also forwarded as the next hidden input h_{t+1} for the subsequent time step, allowing the GRU to maintain memory across sequences. this formula guarantees that the GRU makes dynamic decisions on whether or not to keep the previous information or replace it with the new candidate state to effectively capture long-range dependencies. the hidden state h_t is then passed as output O_t and also propagated as the next hidden input h_{t+1} for the next time step to keep the GRU remembering across sequences.

4. Result and Discussion

The proposed machine learning model of bug prediction from AI and utilizing a GRU architecture has proven to be highly efficient in determining code snippets containing bugs. The model was pre-trained on a vast dataset that was extracted from repositories such as GitHub and GitLab and was comprised of source code files, commit history, and bug reports. With preprocessing phases of tokenization, cleaning of text, and feature extraction, the model was trained to acquire robust patterns in the data so that it could accurately predict bugs in the code. The performance measures a robust and consistent model that can detect probable software bugs at initial stages during the development process, thus enhancing software quality and avoiding maintenance cost. Additionally, the confusion matrix analysis validated the model accuracy with dramatic decrease in false positives and false negatives. The work indicates the possibility in the convergence of software development pipelines and neural network-based solutions towards reliability enhancement and process improvement in bug detection.

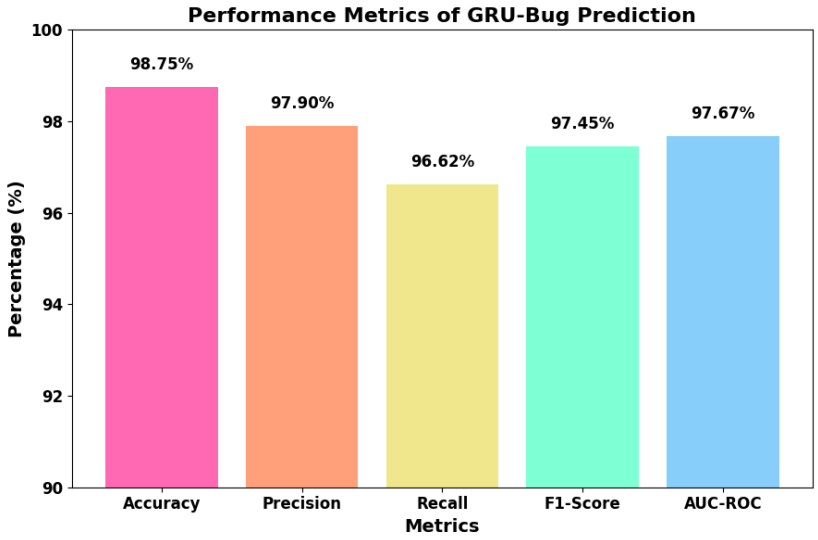

Figure 3: Performance Metrics of GRU-Based Bug Prediction

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 6, Issue 3, 2018

Figure 3 represents the performance metrics of a GRU based bug prediction model for bug prediction. It depicts five important evaluation metrics: Accuracy, Precision, Recall, F1-Score, and AUC-ROC, as bar plots for each of the metrics. The model is performing excellently with an accuracy of 98.75%, which represents its capability to identify fraudulent transactions well in most of the instances. Precision (97.90%) is the fraction of fraud predictions correctly made by the model, and Recall (96.62%) is the ability of the model in the selection of fraud cases from the data set. F1-Score (97.45%) is a balance between precision and recall, and AUC-ROC (97.67%) is a great ability in fraud classification and non-fraud transactions discrimination. The excellent performance on all metrics proves the superiority of the GRU-based model for bug prediction tasks.

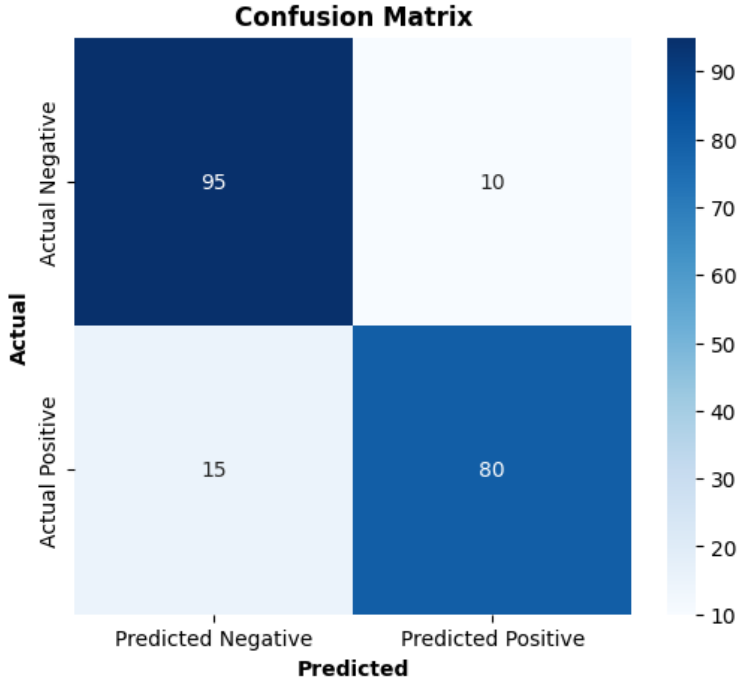

Figure 4: Confusion Matrix for Model Classification Performance

Figure 4 indicates the confusion matrix for model classification performance above plot indicates the model's performance in classification, and the number of true negatives, false positives, false negatives, and true positives is represented. The model predicted 95 examples to be negative correctly, and it incorrectly predicted 10 examples to be positive. In addition, it also correctly predicted 15 examples as negative but were positive, and it correctly predicted 80 examples as positive. This matrix is important in determining how well the model can assess and distinguish between the negative and positive classes to the point that false positives and false negatives are reduced while true positives and true negatives are increased.

5. Conclusion

Machine learning-based bug prediction model with a Gated Recurrent Unit (GRU) model. early bug detection is vital in improving the reliability, usability, and maintainability of software programs and avoiding bug fixing at huge expense in the long run. The approach employs vital preprocessing steps such as tokenization and feature extraction to enable the model to be able to label vast amounts of data and with great accuracy indicate buggy code. Others are still unsolved problems, including record usage of the computer hardware by model training machinery and the black-box nature of the neural network, less than salubrious to consider when analyzing the process employed by the model. Massive amounts of labeled data must be provided while training the model as a function of their description. The model was greatly excellent in performance metrics such as accuracy (98.75%) and AUC-ROC (97.67%), demonstrating that the model is highly efficient in identifying software bugs at an early stage. Future research will have to examine optimization techniques such as transfer learning, model pruning, and use of explainable AI (XAI) techniques in an effort to make the model more transparent and scalable. Overcoming these limitations will enhance the model's applied validity in actual software development contexts. In general, this study is useful in propelling evidence-based automated bug prediction towards AI-driven solutions for ensuring software quality.

References

[1] Drury-Grogan, M. L., Conboy, K., & Acton, T. (2017). Examining decision characteristics & challenges for agile software development. Journal of Systems and Software, 131, 248-265.

[2] Antinyan, V., Staron, M., & Sandberg, A. (2017). Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. Empirical Software Engineering, 22, 3057-3087.

[3] Dilsizian, S. E., & Siegel, E. L. (2014). Artificial intelligence in medicine and cardiac imaging: harnessing big data and advanced computing to provide personalized medical diagnosis and treatment. Current cardiology reports, 16, 1-8.

[4] Fadlullah, Z. M., Tang, F., Mao, B., Kato, N., Akashi, O., Inoue, T., & Mizutani, K. (2017). State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems. IEEE Communications Surveys & Tutorials, 19(4), 2432-2455.

[5] Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., Törner, F., ... & Höglund, C. (2014). Selecting software reliability growth models and improving their predictive accuracy using historical projects data. Journal of Systems and Software, 98, 59-78.

[6] Mou, L., Li, G., Zhang, L., Wang, T., & Jin, Z. (2016, February). Convolutional neural networks over tree structures for programming language processing. In Proceedings of the AAAI conference on artificial intelligence (Vol. 30, No. 1).

[7] Wong, W. E., Li, X., & Laplante, P. A. (2017). Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. Journal of Systems and Software, 133, 68-94.

[8] Lavallée, M., & Robillard, P. N. (2015, May). Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (Vol. 1, pp. 677-687). IEEE.

[9] Wu, B., & Wang, A. I. (2012). A Guideline for Game Development-Based Learning: A Literature Review. International Journal of Computer Games Technology, 2012(1), 103710.

[10] Alajrami, S., Romanovsky, A., & Gallina, B. (2016). Software development in the post-PC era: towards software development as a service. In Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17 (pp. 662-671). Springer International Publishing.

[11] Lam, A. N., Nguyen, A. T., Nguyen, H. A., & Nguyen, T. N. (2015, November). Combining deep learning with information retrieval to localize buggy files for bug reports (n). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 476-481). IEEE.

[12] Ruffieux, S., Spycher, N., Mugellini, E., & Abou Khaled, O. (2017, September). Real-time usage forecasting for bike-sharing systems: A study on random forest and convolutional neural network applicability. In 2017 intelligent systems conference (intellisys) (pp. 622-631). IEEE.

[13] Beer, S., Gómez, T., Dallinger, D., Momber, I., Marnay, C., Stadler, M., & Lai, J. (2012). An economic analysis of used electric vehicle batteries integrated into commercial building microgrids. IEEE Transactions on Smart Grid, 3(1), 517-525.

[14] Moeyersoms, J., de Fortuny, E. J., Dejaeger, K., Baesens, B., & Martens, D. (2015). Comprehensible software fault and effort prediction: A data mining approach. Journal of Systems and Software, 100, 80-90.

[15] Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE access, 5, 3909-3943.

[16] Hassanalieragh, M., Page, A., Soyata, T., Sharma, G., Aktas, M., Mateos, G., ... & Andreescu, S. (2015, June). Health monitoring and management using Internet-of-Things (IoT) sensing with cloud-based processing: Opportunities and challenges. In 2015 IEEE international conference on services computing (pp. 285-292). IEEE.

[17] Yuan, X., Wang, X., Wang, C., Weng, J., & Ren, K. (2016). Enabling secure and fast indexing for privacy-assured healthcare monitoring via compressive sensing. IEEE Transactions on Multimedia, 18(10), 2002-2014.

[18] Chen, L., Thombre, S., Järvinen, K., Lohan, E. S., Alén-Savikko, A., Leppäkoski, H., ... & Kuusniemi, H. (2017). Robustness, security and privacy in location-based services for future IoT: A survey. Ieee Access, 5, 8956-8977.

[19] Shenglin Ben, S. B., Romain Bosc, R. B., Jinpu Jiao, J. J., Wenwei Li, W. L., Felice Simonelli, F. S., & Ruidong Zhang, R. Z. (2017). Digital Infrastructure: Overcoming the digital divide in China and the European Union. CEPS Research Report, November 2017.

[20] Shi, K., Chen, Y., Yu, B., Xu, T., Chen, Z., Liu, R., ... & Wu, J. (2016). Modeling spatiotemporal CO2 (carbon dioxide) emission dynamics in China from DMSP-OLS nighttime stable light data using panel data analysis. Applied Energy, 168, 523-533.

[21] Rawat, D. B., & Reddy, S. R. (2016). Software defined networking architecture, security and energy efficiency: A survey. IEEE Communications Surveys & Tutorials, 19(1), 325-346.

[22] Musa, A., Gunasekaran, A., Yusuf, Y., & Abdelazim, A. (2014). Embedded devices for supply chain applications: Towards hardware integration of disparate technologies. Expert Systems with Applications, 41(1), 137-155.

[23] Ravì, D., Wong, C., Deligianni, F., Berthelot, M., Andreu-Perez, J., Lo, B., & Yang, G. Z. (2016). Deep learning for health informatics. IEEE journal of biomedical and health informatics, 21(1), 4-21.

[24] Rathore, B. (2016). Building next-generation marketing teams navigating the role of AI and emerging digital skills. Eduzone: International Peer Reviewed/Refereed Multidisciplinary Journal, 5(2), 1-7.

[25] Hu, H., Wen, Y., Chua, T. S., & Li, X. (2014). Toward scalable systems for big data analytics: A technology tutorial. IEEE access, 2, 652-687.

[26] Lo'ai, A. T., Mehmood, R., Benkhlifa, E., & Song, H. (2016). Mobile cloud computing model and big data analysis for healthcare applications. IEEE Access, 4, 6171-6180.

[27] Verma, S., Kawamoto, Y., Fadlullah, Z. M., Nishiyama, H., & Kato, N. (2017). A survey on network methodologies for real-time analytics of massive IoT data and open research issues. IEEE Communications Surveys & Tutorials, 19(3), 1457-1477.

[28] Burg, A., Chattopadhyay, A., & Lam, K. Y. (2017). Wireless communication and security issues for cyber–physical systems and the Internet-of-Things. Proceedings of the IEEE, 106(1), 38-60.

[29] Baccarelli, E., Naranjo, P. G. V., Scarpiniti, M., Shojafar, M., & Abawajy, J. H. (2017). Fog of everything: Energy-efficient networked computing architectures, research challenges, and a case study. IEEE access, 5, 9882-9910.

[30] Menouar, H., Guvenc, I., Akkaya, K., Uluagac, A. S., Kadri, A., & Tuncer, A. (2017). UAV-enabled intelligent transportation systems for the smart city: Applications and challenges. IEEE Communications Magazine, 55(3), 22-28.