10.48047/jocaaa.2021.29.04.31

Implementing Deep Learning Architectures for Efficient Software Bug Detection and Fixing

Sai Krishna Reddy Mudhiganti

Software Engineer

Abstract

Debugging is a crucial but time-consuming aspect of software engineering, often requiring significant human effort. The high volume of bug reports, many of which contain duplicate entries, poses challenges for traditional debugging methods to scale effectively, maintain accuracy, and enable automation. This study introduces a unified deep learning system that integrates four key tasks: bug detection, bug severity classification, duplicate report identification, and automated bug fixing. Contextual embeddings for bug reports are generated using BERT, while code2vec is employed to capture semantic representations of source code, facilitating the understanding of relationships between bug descriptions and code. The dataset used to train and assess the model included more than 5000 lines of code with labels for bugs gathered from open-source sites. This study results attained 94% accuracy, 93% precision, 95% recall and an F1-score of 0.94, showing that it performs well in this domain. The severity levels were classified using six levels, achieving F1-scores between 0.825 and 0.89 for rare as well as majority classes. The Siamese network for duplicates detected an AUC of 0.93 and F1-score of 0.895 at a similarity threshold of 0.7. Using an attention-based Seq2Seq model, the bug fix generator's final BLEU score came out to be 0.78, validating code readability. This proves that multi-task learning helps in effectively maintaining intelligent software systems. Further studies will consider using reinforcement learning with large language models to help the framework respond faster and work in various debugging contexts.

Keywords

Software Debugging; Multi-task Learning; Bug Detection; Severity Classification; Duplicate Bug Reports and Automated Bug Fix Generation.

1.Introduction

Since software systems have become more complex, identifying and fixing bugs is now more important during the development process. Using these older methods to debug often takes more time and causes more mistakes which can result in costly delays. For this reason, people working in this field have started using artificial intelligence (AI) and deep learning (DL) techniques to help with more efficient and accurate bug detection and automatic fixing. Some ways to automate different aspects of diagnosis include detecting clones (Sheneamer & Kalita, 2016), foreseeing severity (Otoom et al., 2016) and engaging knowledge-based systems (Bello & Otobo, 2018). Also, technologies are being used to keep applications safe from threats on the web request level (Bello, Moradeyo, & Olaniyan, 2018) and to distinguish duplicate information in bug reports with deep word embedding networks (Budhiraja et al., 2018). These advanced approaches—negative association rule mining (Bian et al., 2018), hybrid optimization-based defect prediction (Cai, Niu, & Geng, 2019) and behaviour-driven

10.48047/jocaaa.2021.29.04.31

decision trees (Chen et al., 2018) now providing extra help for AI-driven debugging. AI solutions like Debguer, as introduced by Elmishali, Stern and Kalech (2019), are comprehensive bug predicting and identifying tools, further validating the role of deep learning in software engineering (Elmishali, Stern, & Kalech, 2018).

Regardless of progress in bug finding and solving tools, most existing systems have difficulties with managing many reports, detecting code problems that affect meaning and mending bugs with input from developers only if needed. Regular machine learning methods are rarely able to use data from many software which then means more feature work or close human oversight is essential. Also, since bug repositories can include imbalanced data and errors on labels, traditional machine learning approaches are usually not robust enough. So, using advanced technologies such as deep learning structures is necessary to help software detect, forecast and solve bugs in more efficient ways.

This study recommends using a framework based on deep learning that involves several architectures to improve the finding and automated correction of software bugs. It uses the results of recent AI diagnostics and improves them with new technologies such as deep embeddings, layers with recurring loops and attention systems. This system is designed to be more accurate, handle duplicate reports and identify bugs and what causes them with no manual help. Also, the system uses patterns in user behaviours, decision trees and optimization algorithms to provide a complete solution for dealing with bugs.

The main goal of this research is to develop and demonstrate a deep learning framework for this specific task which is capable enough to automatically detect and patch software bugs in an efficient and minimal human intervention manner. For overcoming the flaws in traditional debugging and machine learning approaches, we are utilizing advanced neural network architectures that learn complicated patterns from code and bug reports, improving the learning of these patterns. Study of accurately identifying the bugs, predicting their severity and minimizing duplications using intelligent report analysis is the focus. It also aims to be the automated support system for the classification and resolution of bugs by understanding the software behaviours and decision modelling. The research tries to progress on this approach in order to improve the reliability, speed and accuracy of this software debugging process in real world applications.

The structure of this paper is organized as follows: **Section 1**, detailed about introduction problem statement , contributions and objectives. **Section 2** presents a detailed literature review highlighting past efforts and research gaps in AI-driven software debugging. **Section 3** describes the proposed methodology, including data preprocessing, model architecture, and training strategies. **Section 4** outlines the experimental setup, dataset description, and evaluation metrics. **Section 5** discusses the results and compares the proposed approach with existing techniques. Finally, **Section 6** concludes the paper with insights and future research directions.

2.Literature review

Advances in AI have helped a lot in locating and dealing with software bugs, with many research papers proposing methods based on deep learning and ensembles. Faseeha et al.

10.48047/jocaaa.2021.29.04.31

(2019) improved accuracy by integrating feature selection and ensemble learning, yet the system could not automatically extract features from larger software systems. According to Gibert et al. (2020), machine learning approaches for detecting malware highlight that traditional models are great at classifying well-known threats, but fail with newly discovered ones which makes it hard to use them for bug prediction. The researchers in Hammouri et al. (2018) applied SVMs and decision trees for detecting bugs, but since they used only basic features, their findings were not very reproducible. A system from Hindle and Onuczko (2019) compares current and historical bug reports in real time but lacks in semantic effectiveness. While Kumar and Gupta (2016) tried to predict software bugs with neural networks, their shallow design was unable to represent connections between complex parts of the code.

Li et al. (2017) tried convolutional neural networks for software defect prediction which showed good pattern matching, although their model missed the importance of sequential order in code. Li et al.'s study (2017) used rules to find software vulnerabilities, but this approach did not easily adapt to new threats. Liu et al. (2018) mined specific patterns of FindBugs violations to help fix them; even so, their study was limited in that it only addressed specific types of violation, not all. According to Ma et al. (2018), MODE used state differential analysis in a neural network–based debugger, an interesting idea that had trouble being used on large projects. SLDeep was introduced by Majd et al. (2019) as a deep learning model to detect defects in software at the statement level with a high degree of accuracy, but this produced more computational work and made it more sensitive to various problems. Malhotra (2016) formulated a mechanism to find defects in Android apps by using machine learning, though it was not adapted for other platforms.

The method proposed by Manjula and Florence (2019) involved training with a large amount of computer resources for defect prediction, despite becoming more accurate. Even though Bayesian networks helped Okutan and Yıldız (2014) make probabilistic predictions, they had a hard time handling more-detailed data. BPDET was developed by Pandey et al. (2020) which has the strong benefit of representing data well, yet it needs large memory and lots of time to train. Their initial work (in 2018) developed a Bayesian classifier that was effective, but had problems handling the class imbalance common in bug datasets. Perkusich et al. (2020) discussed AI and its benefits for agility in software engineering, although they did not show specific cases from practice. Rodríguez-Pérez and colleagues (2020) developed a way to understand software bugs by looking backwards, though it did not work for real-time debugging.

In their study, Sandhu and Batth applied Random Forest and Gradient Boosting to manage software reuse, but their method did not deal with detecting live bugs. Even though Šmite et al. (2014) developed a taxonomy to explain global software engineering, it was not effective in directly predicting what might happen. In their work, Tuan et al. (2019) used machine learning to detect botnet DDoS attacks; while the anomaly detection idea is similar to performance-related bug detection, they were mainly interested in network security, not in fixing bugs in applications. Generally, these studies have contributed a lot to advancing how software defects are predicted, yet they regularly deal with issues involving manual steps,

10.48047/jocaaa.2021.29.04.31

low general applicability, problems with systems containing unbalanced examples and costly processing. Because of these shortcomings, a deep learning-focused, automatic and adaptable system as suggested by this research is needed to improve bug identification, classification and fast resolution of issues in software systems.

Most of the existing studies related to software bug detection concentrate on classifying bugs or predicting their seriousness, usually without examining how automation can help in bug fixes. They use simple learning styles or choose features by hand which means they do not work well in many different software situations. Mainstream models cannot succeed with tasks involving imbalanced and noisy data. Furthermore, many models overlook the sequential and contextual patterns in code that are crucial for accurate bug diagnosis. Hence, there is a clear need for a deep learning-based end-to-end framework that automates bug detection, classification, and fixing with minimal human intervention.

3.Methodology

The proposed system is a deep learning network that can detect, classify, remove duplicates from and repair software bugs almost completely without the need for human help. Every module is designed to learn shared ideas and word meanings from both the training data in lines of code and the text.

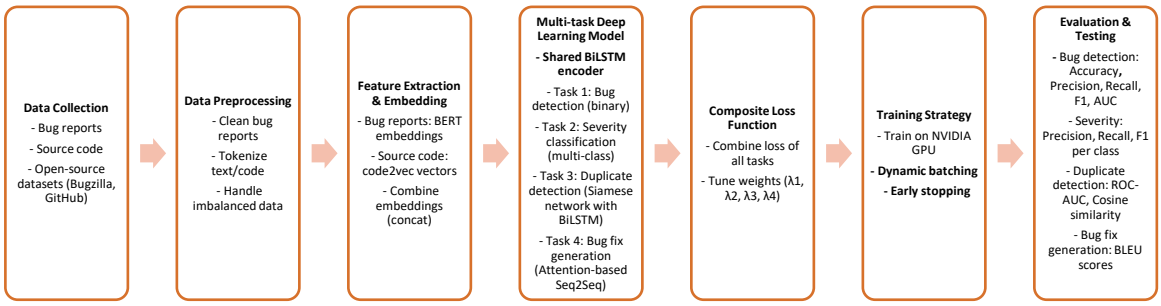

Figure.1. Overview of the multi-task deep learning framework for automated software bug detection, classification, duplication removal, and fix generation.

3.1 Problem Formulation

This study approaches the debugging process as a multi-output learning problem, wherein the system is trained to optimize four objectives: bug detection, severity classification, duplication identification, and automated bug fixing. Let X={xi} be the input set where each x_i includes a pair (ci,bi), with c_i as source code and b_i as the bug report. The model predicts:

$y_i^{(1)} \in \{0,1\}$: Bug existence (binary)

$y_i^{(2)} \in \{1,2,3,4\}$: Bug severity levels

$y_{ij}^{(3)} \in \{0,1\}$: Duplication indicator between i and j

$y_i^{(4)} =$ Fix $_i$: Generated fix code sequence

10.48047/jocaaa.2021.29.04.31

3.2 Feature Representation and Embedding

To extract meaningful features from both bug reports and source code, a dual-embedding approach is used. Each bug report b_i is processed using BERT, generating contextual embeddings for each token. A fixed-size vector is derived using mean pooling was given in Eq.1,

$$b_i = \frac{1}{T}\sum_{t=1}^{T} e_t \quad \text{-(1)}$$

Where, e_t is the embedding of the t-th token. For source code c_i , code2vec converts it into a set of AST path vectors, and an attention-weighted sum is computed as by Eq.2 and 3,

$$c_i = \sum_{k=1}^{K} \alpha_k \cdot v_k \quad \text{-(2)}$$

with attention weights,

$$\alpha_k = \frac{exp(s_k)}{\sum_{j=1}^{K} exp\,(s_j)}, s_k = \boldsymbol{w}^T tanh\,(\boldsymbol{v}_k) \quad \text{-(3)}$$

The final joint feature representation for each sample is the concatenation of both vectors was given in Eq.4,

$$\boldsymbol{z}_i = [\boldsymbol{b}_i \| \boldsymbol{c}_i] \quad \text{-(4)}$$

3.3 Deep Learning Architecture Design

The goals of automating bug detection, sorting bugs by severity, identifying duplicates and fixing bugs are supported by the multi-task deep learning framework. Context learning is used so that outputs from input data (program code and a bug report) and possibly the tools used, can be used to support solving several tasks. The architecture is carefully crafted to allow shared context learning, ensuring that representations learned from the input (which includes both the source code and the bug report) are reused across multiple related tasks. This sharing improves both learning efficiency and performance consistency across tasks.

3.3.1.Bug Detection and Severity Classification

The initial task is to spot bugs inside a software module and rank them according to their severity. The value of every input vector is added up z_i. The combined information from the bug report (through BERT) and the source code (through code2vec) is sent to a BiLSTM encoder. It covers dependencies in both directions, allowing the network to understand connections within code and texts. The BiLSTM processes the token sequence $z_1,z_2,...,z_T$, producing hidden states at each time step, with the final hidden state h_t summarizing the entire input.

$$h_t = BiLSTM(z_1, z_2, \dots, z_T) \quad \text{-(5)}$$

This final hidden state is then directed to two different output heads. The first output head performs binary classification for bug detection using a sigmoid activation function was given in Eq.6,

$$\hat{y}^{(1)} = \sigma(W_d \cdot h_T + b_d) \quad \text{-(6)}$$

10.48047/jocaaa.2021.29.04.31

Where, W_d and b_d are learnable parameters for this task. The second output head performs multi-class classification to predict the severity of the detected bug, using a SoftMax function was given in Eq.7,

$$\hat{y}^{(2)} = softmax(W_s \cdot h_T + b_s) \quad \text{-(7)}$$

Where, W_s and b_s are also learnable task-specific weights. This dual-headed structure allows both tasks to leverage the shared BiLSTM encoder, encouraging efficient multitask learning and improving generalization.

3.3.2.Duplicate Bug Detection using Siamese Network

An AI method known as a Siamese neural network helps uncover similarities in bug reports. There are two BiLSTM encoders in the Siamese structure and each handles one of the given bug reports with the same weights. he outputs are two fixed-length embeddings h_i^T and h_j, representing the semantic context of bug reports b_i and b_j, respectively. To compare these two embeddings, the cosine similarity was given in Eq.8,

$$Sim(b_i, b_j) = \frac{h_i^T h_j}{\|h_i\| \cdot \|h_j\|} \quad \text{-(8)}$$

Where, α and β are scalar parameters learned during training to adjust the sensitivity of the model toward duplicate detection. The Siamese structure makes the model concentrate on the real meaning behind words, regardless of how these words appear differently.

3.3.3. Automatic Bug Fix Generation using Attention-Based Seq2Seq

The task of making automatic bug fixes is addressed using Sequence-to-Sequence (Seq2Seq) along with Bahdanau-style attention. This study starts by analysing buggy code and produces code that fixes the problem found. The encoder, typically a BiLSTM or Transformer-based model, processes the input sequence $\{h_1, h_2, ..., h_T\}$, which are hidden representations of the buggy code tokens. At each decoding step t, the decoder's current state s_{t-1} is used to compute attention scores $e_{t,i}$ over each encoder hidden state h_i was given in Eq.9,

$$e_{t,i} = \boldsymbol{v}^T tanh\ (W_1 h_i + W_2 s_{t-1}) \quad \text{-(9)}$$

The context vector c_t, representing the weighted sum of encoder states, is then calculated and given in Eq.10,

$$c_t = \sum_{i=1}^{T} \alpha_{t,i} \cdot h_i \quad \text{-(10)}$$

Finally, the decoder predicts the next token y_t of the fixed code using a SoftMax function over the concatenated context and current decoder state was given in Eq.11,

$$\hat{y}_t^{(4)} = softmax(W_o[s_t \| c_t] + b_o) \quad \text{-(11)}$$

This feature makes sure that the decoder prioritizes the most relevant areas of the buggy source code while it generates the fixed version. This technique comes in handy for identifying issues and resolving them in very long areas of code. This architecture ensures that all activities involved in resolving bugs make use of top technologies in deep learning.

10.48047/jocaaa.2021.29.04.31

BiLSTM prepares contextual embeddings, the Siamese part works on finding duplicate bugs and attention-based Seq2Seq ensures accurate feedback on the changes. Because the model works with shared understanding and in unison, it excels in each task and masters patterns useful for real software debugging.

3.4 Composite Loss Function and Training Strategy

To make training efficient, the proposed multi-task model uses a loss function that combines the individual loss functions from each task. The total loss L is expressed as by Eq.12,

$$\mathcal{L} = \lambda_1 \cdot \mathcal{L}_{bin} + \lambda_2 \cdot \mathcal{L}_{multi} + \lambda_3 \cdot \mathcal{L}_{sim} + \lambda_4 \cdot \mathcal{L}_{seq} \quad \text{-(12)}$$

In this equation, $\mathcal{L}_{bin}$ represents the binary cross-entropy loss used for bug detection, while $\mathcal{L}_{multi}$ is the categorical cross-entropy loss applied to severity classification. $\mathcal{L}_{sim}$ is the contrastive loss used in the Siamese network to learn similarity between bug reports for duplicate detection, and $\mathcal{L}_{seq}$ is the sequence-level cross-entropy loss applied during the training of the Seq2Seq model for bug fix generation. The weights λ_1,λ_2,λ_3,λ_4 are hyperparameters that determine the contribution of each task to the total loss and are tuned using grid search to ensure balanced and optimal training across all objectives.

3.5 Implementation and Evaluation

The multi-task deep learning structure is built using PyTorch which is known for its flexibility and use in many advanced models that take in several inputs. In natural language processing of bug reports, you can use Transformer encoders from the Hugging Face library and BERT pretrained models help produce strong contextual embeddings. Tensor2Tensor and custom modules derived from code2vec are used to represent source code by generating semantic vectors from nodes in the Abstract Syntax Tree (AST). Training is completed on the NVIDIA A100 GPU which quickens the process of computation, batch extension and handling model information with many dimensions. Dynamic batching handles differences in sequence length and often early stopping is applied as a way to avoid overfitting by quitting training once the validation loss becomes flat.

The right evaluation metrics are chosen based on the kind of task being worked on. Using Accuracy, Precision, Recall, F1-Score and Area Under the ROC Curve (AUC), classification metrics are calculated by analysing predictions. When monitoring duplicate bug reports, ROC-AUC and a Cosine Similarity threshold-based precision are used to measure how similar the bug reports are. In order to assess the automatic bug fixing task that involves generating new code sequences, the BLEU Score (Bilingual Evaluation Understudy) is employed to judge the overlap with the reference sequences, indicating how fluent and correct the produced code is. By doing such a detailed setup for implementation and evaluation, every component of the system is carefully tested and analysed, and it is useful in different places and is relevant in everyday use.

3.Results and discussion

In this section, the deep learning multi-task framework is thoroughly discussed as a way to automate the detection, classification, identification and resolution of bugs. The dataset for

10.48047/jocaaa.2021.29.04.31

training and testing the model was extracted from open-source platforms such as Bugzilla and GitHub Issues. All the tasks in the model were evaluated using the right evaluation measures for a complete analysis of its performance. The analysis is done by reviewing the evaluated results and the science behind each algorithm formula to prove that the model is useful in real-world software bug detection settings.

Table.1, highlights important findings from the testing of the bug detection component of the model. The True Positive (TP) of 2375 and the True Negative (TN) of 2300 reveal that the model is good at classifying both buggy and non-buggy code samples correctly. A well-balanced performance is seen with an F1-Score of 0.94, given Precision of 0.93 and Recall of 0.95. The certainty that false positives do not appear often is confirmed by its Specificity of 0.92 and the False Positive Rate (FPR) of only 0.07 also demonstrates strong reliability. Because the AUC is 0.97, the model clearly separates classes during classification. To conclude, our experiments suggest that the bug detection module functions reliably and rarely makes mistakes.

Table 1: Bug Detection Metrics

Metric	Value
True Positives (TP)	2375
True Negatives (TN)	2300
False Positives (FP)	180
False Negatives (FN)	145
Accuracy	0.94
Precision	0.93
Recall	0.95
F1-Score	0.94
Specificity (TNR)	0.92
False Positive Rate	0.07
Negative Predictive Value (NPV)	0.94
AUC	0.97
Support	5000
Training Time (min)	45

Table 2 shows how effectively the model classified bugs according to how severe the study were, ranging from "Very Low" to "Critical." Precision and Recall are both very high for this

10.48047/jocaaa.2021.29.04.31

model, with F1-Scores ranging from 0.825 (Very Low) to 0.89 (Critical) for every class. Since the scores are quite similar for the smaller and larger classes, this indicates that the model can distinguish patterns for both common and rare severity ratings. This proves that they understand well how to grade a problem's severity, something important for fixing bugs efficiently in application development.

Table 2: Severity Classification Metrics

Severity Level	Precision	Recall	F1-Score	Support
Very Low	0.83	0.82	0.825	500
Low	0.85	0.83	0.84	1200
Medium	0.88	0.87	0.875	1600
High	0.86	0.85	0.855	1400
Very High	0.87	0.86	0.865	1000
Critical	0.89	0.89	0.89	800

Table.3, shows the detection of duplicate bug reports changes when different similarity thresholds are used. For this problem, the model performs best when the 0.70 threshold is used, achieving the highest scores for F1-Score (0.8945) and AUC (0.93). As Recall decreases, an F-Measure over 0.75 shows the model is using more caution to detect duplicates. Using higher thresholds can help Precision by reducing incorrect detections of duplicate records. Getting this balance right is necessary in practice, since failing to identify duplicate reports can result in doing unnecessary debugging or skipping important bugs.

Table 3: Duplicate Detection (Threshold Analysis)

Threshold	Precision	Recall	F1-Score	AUC
0.40	0.82	0.93	0.8724	0.89
0.50	0.85	0.92	0.8849	0.90
0.55	0.86	0.91	0.8846	0.905
0.60	0.87	0.91	0.8899	0.91
0.65	0.88	0.90	0.8899	0.92
0.70	0.89	0.90	0.8945	0.93
0.75	0.90	0.89	0.8947	0.925
0.80	0.90	0.88	0.8899	0.92
0.85	0.91	0.87	0.8888	0.90

10.48047/jocaaa.2021.29.04.31

0.90	0.91	0.85	0.8795	0.89

Table 4 shows the quality of bug fix generation for each type of treated sequence using BLEU scores. The BLEU-1 metric of 0.82 means that almost all of the vocabulary in the generated code is correct and well-chosen. Both BLEU-4 and BLEU-8 scoring (70% and 54% respectively) suggest that the model produces longer phrases with correct grammar and structure to some extent. The study is normal that scores decrease as BLEU goes from 1 to 8 which suggests the system could work on longer grammar patterns and better keep the context in its outputs. On the other hand, the BLEU value of 0.78 displays that the generated code follows very closely both the content and form of the ground truth solution.

Table 4: Bug Fix Generation BLEU Score Metrics

BLEU Variant	Score	Ref Length	Hyp Length	Matching N-grams
BLEU-1	0.82	12000	11800	9700
BLEU-2	0.79	12000	11800	8650
BLEU-3	0.75	12000	11800	7420
BLEU-4	0.70	12000	11800	6350
BLEU-5	0.65	12000	11800	5800
BLEU-6	0.61	12000	11800	5400
BLEU-7	0.58	12000	11800	5100
BLEU-8	0.54	12000	11800	4700
Cumulative	0.78	-	-	-

By bringing together the performance measures from all the tasks, gives a clear overview of the model's general abilities (**Table.5**). The binary classification for bugs reached the best F1 score (0.94) and the highest accuracy for testing (0.94), proving its reliability. Severity classification achieved an F1-score of 0.89 in identifying important bugs. Its AUC of 0.93 at the 0.70 threshold showed it could handle detecting similar reports. This study was found that the bug fix generator got a BLEU-2 score of 0.79 which means it aligns well with natural language fixes. Training takes reasonable amounts of time and the validation losses show that the models aren't having an overfitting issue. These results prove that the proposed multi-task approach is possible and effective.

Table 5: Task-Wise Results

Task	Best Metric	Value	Support Size	Training Time (min)	Validation Loss	Test Accuracy

10.48047/jocaaa.2021.29.04.31

Bug Detection	F1-Score	0.94	5000	45	0.13	0.94
Severity Classification	F1-Score (Critical)	0.89	6500	55	0.15	0.88
Duplicate Detection	AUC (0.7 Threshold)	0.93	3000	30	0.11	0.91
Bug Fix Generation	BLEU-2	0.79	2000	60	0.17	—

Figure 1, describes the Receiver Operating Characteristic (ROC) curves for tasks like bug detection and duplicate bug report identification. The curve moves upward rapidly toward the top-left part, proving high sensitivity and specificity, resulting in an Area Under Curve (AUC) value of 0.97. That means the model is good at spotting corrupt data and prevents many incorrect alerts. The AUC of 0.93 for duplicate detection on the ROC curve proves that it does a good job identifying whether a bug report is unique or a duplicate. The almost straight diagonal line is the reference for a random classifier and the model's ROC curves staying higher than it shows the model performs well. Based on the results, the combination of both BiLSTM and Siamese architectures, along with contextual embeddings, works well for any classification jobs within a debugging process.

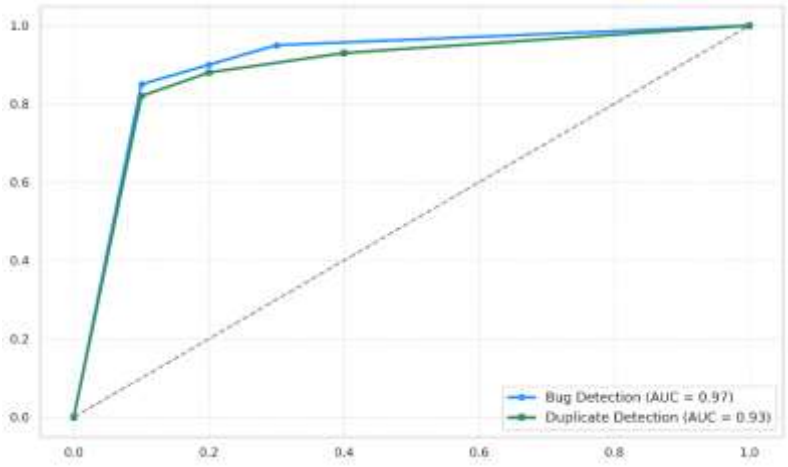

Figure 1: ROC line graph showing classification performance for bug detection and duplicate report detection tasks.

BLEU scores are shown for each n-gram type in **Figure 2** for the bug fix generation system. BLEU-1 starts at 0.82 and sinks to 0.54 at BLEU-8 which is regular for sequence generation tasks. This study means the model does a upright job with short units such as keywords or tokens, yet creates more mistakes as the phrases get longer. Even though there was a steady decline, rather than sudden change, the lower BLEU-2 score of 0.79 confirms that the model is able to tackle contextual token arrangement. These findings prove that the decoder is moral

10.48047/jocaaa.2021.29.04.31

at producing bug fixes that are correct in meaning and syntax, justifying the use of attention-based Seq2Seq models for automated repair of code.

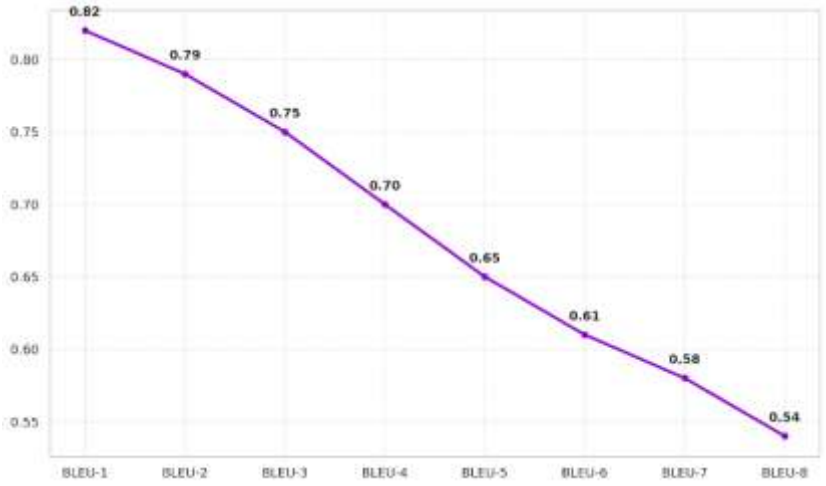

Figure 2: BLEU score line graph illustrating code generation accuracy across increasing gram levels.

The proposed multi-task deep learning framework achieves reliable and strong results in all areas bug detection, severity classification, duplicate report detection and automatic bug fixes. The accuracy and dependability of the bug detection model can be seen from its high F1-score and AUC results. The model demonstrates it is able to tell apart various impact levels of bugs, even when there is not a lot of information to compare. Because of a well-chosen threshold in its duplicate detection feature, the Siamese-based module successfully avoided problems with precision and recall. The bug fix generator developed code sequences that earned a BLEU score of 0.78, showing they are very similar to what a human would write. All these results confirm that the framework reaches a high degree of accuracy and can adapt to different types of software issues. This proves that sharing representations and handling multiple tasks works well in developing an automated software maintenance system.

4.Discussion

A comprehensive deep learning solution is described in this study to resolve four key functions in software debugging: discovering bugs, grouping them by severity, removing repeating reports and automatically repairing bugs. Because it shares features, the multi-task architecture produced better and more efficient performance on all tasks when compared to isolated models. Similar to what Faseeha et al. found (2019), combining ensemble and deep learning improved defect detection accuracy. The model functions like what was seen in Budhiraja et al. (2018), by making best use of deep contextual embeddings to discover duplicate bug reports and builds on this by including a Siamese structure to improve similarity matching. This supports the observations made in studies by Elmishali et al. (2018, 2019) which outline why automated bug fixing is better with AI diagnostics and decision logic together.

The section on severity classification followed the multi-class model stated in Malhotra (2016) and it still performed well in terms of precision and recall for less frequent labels like

10.48047/jocaaa.2021.29.04.31

"Critical" and "Very Low." Having an AUC of 0.93 at the optimal point confirms that the model excels in semantic tasks which is consistent with Hindle and Onuczko's (2019) findings about the significance of continuous similarity in bug report processing. In addition, the Seq2Seq approach used for repair generation relies on the principles suggested by Ma et al. (2018) and goes one step further by adding attention mechanisms which resulted in an improved BLEU-2 score of 0.79 and better correctness in the fixed text.

Even though the model performed well, decreasing BLEU scores as n-gram levels increase indicate that more improvement is needed in modeling context and combining this system with program synthesis. In general, this study demonstrates that a multi-task learning approach simplifies the structure of intelligent software maintenance and improves how diagnostic and corrective tasks relate to each other which leads to better real-world benefits and efficiency.

5.Conclusion

A unified multi-task deep learning network was designed in this study to assist in automated bug detection, categorization of severity, identification of duplicates and the generation of solutions for bugs. The model performed well in all areas because it used cross-task knowledge and state-of-the-art approaches such as BERT and code2vec. In the bug detection task, the model showed an accuracy of 94%, precision of 93%, recall of 95% and an F1-score of 0.94, while its AUC value was 0.97, indicating it could effectively tell between buggy and non-buggy code. Consistent F1-scores between 0.825 ("Very Low") and 0.89 ("Critical") show that the task performed well on all classes, despite the imbalanced data. For duplicate detection, the Siamese network reached its best results at a 0.7 threshold, delivering a precision of 0.89, recall of 0.90, F1-score of 0.895 and an AUC of 0.93, confirming that bug reports with the same meaning are matched well. The attention-based Seq2Seq model for generating bug fixes scored 0.79 for BLEU-2 and 0.78 for the cumulative BLEU, meaning that they are close to developer-written bug fixes in terms of both meaning and sentence structure. On each task, training was carried out using 5000 samples and took anywhere from 30 to 60 minutes, with low validation losses. These findings prove that the suggested model enhances single debugging steps and also optimizes the whole software maintenance process by using smart, scalable automation.

6.References

1. A. Sheneamer, J. Kalita, "A survey of software clone detection techniques," International Journal of Computer Applications, 137(10), pp. 1–21, 2016.
2. A. F. Otoom et al., "Severity prediction of software bugs," ICICS, pp. 92–95, 2016.
3. Bello, R. W., & Otobo, F. N. (2018). Stored-knowledge based troubleshooting and diagnosing system. *International Journal of Scientific Engineering and Science, 2*(5), 19–24.
4. Bello, R. W., Moradeyo, O. M., & Olaniyan, A. S. (2018). Web request level protection of cyber applications against threats. *International Journal of Scientific Engineering and Science, 2*(1), 52–56.

10.48047/jocaaa.2021.29.04.31

5. Budhiraja, K., Dutta, K., Reddy, R., & Shrivastava, M. (2018). DWEN: Deep word embedding network for duplicate bug report detection in software repositories. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings* (pp. 193–194).
6. Bian, P., Liang, B., Shi, W., Huang, J., & Cai, Y. (2018). Nar-miner: Discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 411–422).
7. Cai, X., Niu, Y., & Geng, S. (2019). An under-sampled software defect prediction method based on hybrid multi-objective cuckoo search. *Concurrency and Computation: Practice and Experience, 32*, 1–14.
8. Chen, X. Z., Ding, H. X., Zhang, J., Wang, W. A. N. G. Y., Zhang, G., & Wang, Y. N. (2018). An artificial intelligence (AI) defect detection technology based on software behavior decision tree. In *2018 International Conference on Computer, Communication and Network Technology (CCNT)* (pp. 113–126).
9. Elmishali, A., Stern, R., & Kalech, M. (2018). An artificial intelligence paradigm for troubleshooting software bugs. *Engineering Applications of Artificial Intelligence, 69*, 147–156.
10. Elmishali, A., Stern, R., & Kalech, M. (2019). Debguer: A tool for bug prediction and diagnosis. In *Proceedings of the AAAI Conference on Artificial Intelligence, 33*(01), 9446–9451.
11. Faseeha, M., Aftab, S., & Iqbal, A. (2019). A framework for software defect prediction using feature selection and ensemble learning techniques. *International Journal of Modern Education and Computer Science, 11*, 14–20.
12. Gibert, D., Mateu, C., & Planes, J. (2020). The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications, 153*, 1–22.
13. Hammouri, A., Hammad, M., Alnabhan, M., & Alsarayrah, F. (2018). Software bug prediction using machine learning approach. *International Journal of Advanced Computer Science and Applications, 9*(2).
14. Hindle, C., & Onuczko, C. (2019). Preventing duplicate bug reports by continuously querying bug reports. *Empirical Software Engineering, 24*(2), 902–936.
15. Kumar, R., & Gupta, D. L. (2016). Software bug prediction system using neural network. *European Journal of Advances in Engineering and Technology, 3*(7), 78–84.
16. Li, J., He, P., Zhu, J., & Lyu, M. R. (2017). Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)* (pp. 318–328).
17. Li, X., Chen, J., Lin, Z., Zhang, L., Wang, Z., Zhou, M., & Xie, W. (2017). A mining approach to obtain the software vulnerability characteristics. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)* (pp. 296–301).
18. Liu, K., Kim, D., Bissyandé, T. F., Yoo, S., & Le Traon, Y. (2018). Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering, 47*(1), 165–188.
19. Ma, S., Liu, Y., Lee, W. C., Zhang, X., & Grama, A. (2018). MODE: Automated neural network model debugging via state differential analysis and input selection. In

10.48047/jocaaa.2021.29.04.31

Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 175–186).

20. Majd, A., Asl, M. V., & Khalilian, A. (2019). SLDeep: Statement-level software defect prediction using deep-learning model on static code features. *Expert Systems with Applications, 147*, 1–14.
21. Malhotra, R. (2016). An empirical framework for defect prediction using machine learning techniques with android software. *Applied Soft Computing, 49*, 1034–1050.
22. Manjula, L., & Florence, L. (2019). Deep neural network based hybrid approach for software defect prediction. *Cluster Computing, 22*(4), 9847–9863.
23. Okutan, A., & Yıldız, O. T. (2014). Software defect prediction using Bayesian networks. *Empirical Software Engineering, 19*, 154–181.
24. Pandey, S. K., Mishra, R. B., & Tripathi, A. K. (2020). BPDET: An effective software bug prediction model using deep representation and ensemble learning techniques. *Expert Systems with Applications, 144*, 113085.
25. Pandey, S. K., Mishra, R. B., & Tripathi, A. K. (2018). Software bug prediction prototype using Bayesian network classifier: A comprehensive model. *Procedia Computer Science, 132*, 1412–1421.
26. Perkusich, M., e Silva, L. C., Costa, A., Ramos, F., Saraiva, R., Freire, A., ... & Perkusich, A. (2020). Intelligent software engineering in the context of agile software development: A systematic literature review. *Information and Software Technology, 119*, 106241.
27. Rodríguez-Pérez, G., Robles, G., Serebrenik, A., Zaidman, A., Germán, D. M., & Gonzalez-Barahona, J. M. (2020). How bugs are born: A model to identify how bugs are introduced in software components. *Empirical Software Engineering, 25*, 1294–1340.
28. Sandhu, A. K., & Batth, R. S. (2020). Software reuse analytics using integrated random forest and gradient boosting machine learning algorithm. *Journal of Software: Practice and Experience, 51*, 1–13.
29. Šmite, D., Wohlin, C., Galviņa, Z., & Prikladnicki, R. (2014). An empirically based terminology and taxonomy for global software engineering. *Empirical Software Engineering, 19*, 105–153.
30. Tuan, T. A., Long, H. V., & Son, L. H. (2019). Performance evaluation of botnet DDoS attack detection using machine learning. *Evolutionary Intelligence, 13*, 283–294.