IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal

Impact Factor: 7.301

Volume 3, Issue 2, July 2023

Improving Software Quality through Deep Learning: A Comprehensive Literature Study on Error Prediction in Software Development

Amna Shipra and Avaish Ansari
Students, Master of Computer Application
Late Bhausaheb Hiray S.S Trust's Hiray Institute of Computer Application, Mumbai, India
amnashipra@gmail.com and ansariavaish01@gmail.com

Abstract*: The paper explores the significance of error prediction in software development and discusses the use of deep learning approaches to address this task. It emphasizes the need for proactive error prevention and the limitations of reactive bug- fixing strategies. The study examines various deep learning models, including Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and Graph Convolutional Networks (GCNs), and their applicability in error prediction. The conclusions drawn from the study highlight the strengths of each model. RNNs are effective in capturing temporal dependencies and sequential patterns in error data, enabling the analysis of error progression over time. CNNs excel at extracting relevant features and local patterns from software artefacts by treating them as image-like data. GCNs leverage the graph structure of software artefacts to capture structural dependencies and interactions between code elements. To leverage the benefits of both temporal and structural information, the study proposes a hybrid model that combines RNNs with GCNs for error prediction. This hybrid model harnesses the power of deep learning to identify patterns and model relationships, offering promising results in accurate error forecasting and prevention in software development. The adoption of proactive error prediction techniques facilitated by deep learning has the potential to enhance software quality, resource efficiency, and user experience. By proactively identifying and addressing errors, development teams can reduce the impact of issues before they manifest, leading to improved software reliability and customer satisfaction. Overall, the paper highlights the importance of error prediction in software development and demonstrates the potential of deep learning approaches to enhance error prevention strategies.*

Keywords: error prediction, software development, deep learning, RNNs, CNNs, GCNs, proactive error prevention, software quality, temporal dependencies, sequential patterns, structural dependencies, hybrid model

I. INTRODUCTION

In the context of software development, the term "error prediction" refers to the practice of foreseeing and predicting probable faults or flaws that could exist in a software system. Software error prediction models are a significant part of software quality assurance and are commonly used to detect faulty software modules based on software measurement data [1]. Error prediction tries to proactively identify and avoid errors before they have an influence on the software's operation and user experience by utilizing techniques like machine learning and data analysis. Software bugs are typically found and fixed reactively, frequently after users have reported them or during the testing stage. However, using a reactive strategy may be time-consuming, expensive, and unsatisfactory to users. For a number of reasons, error prediction is essential in software development. It offers proactive issue prevention, allowing programmers to find and fix any errors before they have serious consequences. Error prediction uses methods like machine learning and data analysis to help find patterns, trends, and indicators that might result in mistakes in software systems. Thanks to this proactive strategy, development teams may take preventative action to reduce risks, raise software quality, and improve user experience overall.

Error prediction in software development can provide some difficulties, though. Data quality and availability provide important challenges since it might be challenging to get enough data of appropriate quality for error prediction. Prediction model performance can also be impacted by imbalanced datasets when the frequency of mistakes is very low compared to instances without errors. It can take a lot of time and effort to pick useful features through feature engineering, which calls for careful thought to capture pertinent parts of mistakes. In order to comprehend the motivations behind forecasts and win the trust of developers, it is also crucial to make sure that models are interpretable, particularly in the case of deep learning.

The dynamic nature of software systems also makes mistake prediction more difficult. New mistake patterns are continually being introduced by software systems, which may escape the attention of current models. For precise predictions, it is essential to take context into account, such as code alterations or user interactions. Scalability is a problem since error prediction algorithms must deal with big datasets and the computing demands of deep learning models. Despite these difficulties, it is crucial for precise and successful error prediction to find solutions through research and innovation. Software quality, developer productivity, and user experience will all be improved by addressing difficulties with data availability, class imbalance, feature engineering, interpretability, dynamic environments, contextual information, and scalability

Fault prediction modelling is an important area of research and the subject of many previous studies that produce fault prediction models, which allows software engineers to focus development activities on fault-prone code, thereby improving software quality and making better use of resources of the system with a high fault probability [2]. For error prediction in software development, a variety of deep learning algorithms may be used. Recurrent neural networks (RNNs), which operate best with sequential data, are one such approach. RNNs can identify temporal correlations and trends in software logs, enabling them to forecast mistakes based on the timeline of occurrences. The analysis of software code or log files can also be done using convolutional neural networks (CNNs). CNNs can recognize structural patterns and anomalies that might point to potential mistakes by applying convolutional operations to code snippets or log entries. Another method is GCN (Graph Convolutional Network) can be employed in error prediction in software development by utilizing the graph structure of software artefacts. The process involves representing the artefacts as a graph, assigning features to nodes, designing a GCN model architecture, training the model using labelled data, making predictions on new artefacts, and evaluating the model's performance. By capturing relationships and dependencies between entities, GCN enables the early detection of potential errors, aiding developers in focusing on critical areas during software development. Deep autoencoders can also be used for unsupervised learning to spot anomalous behaviour or outliers in software systems, offering important insights into potentially error-prone regions. Deep learning approaches provide a wide range of tools to predict faults in software development, enabling more precise and proactive error prevention procedures.

II. LITERATURE REVIEW

Traditional machine learning methods have been tested for their efficacy in Software Fault Prediction (SFP), but they have drawbacks such as managing imprecise data and needing feature engineering. Deep learning techniques have benefits like automated feature extraction and domain generalisation. Even though deep learning has primarily been employed in pre- processing phases in SFP, some research has used it for classification tasks, such as when utilising convolutional neural networks, deep belief networks, and stacked denoising autoencoders. Deep learning is a potential method for SFP due to its capacity to handle vast amounts of data and build hierarchical representations.

In this section, some related work of Deep Learning techniques is explained.

A. Hasanpour et al. (2004) [3] This study investigates the classification of imbalanced and inadequately sampled NASA datasets using the Stack Sparse Auto-Encoder (SSAE) and Deep Belief Network (DBN) deep learning models. According to experimental findings, accuracy is improved for datasets with enough samples, and the SSAE model beats the DBN model in the majority of assessment measures.

Y. Ma et al. (2012) [4] This paper addresses the issue of cross-company software defect prediction, where data from different companies is used to build prediction models. The authors propose a novel algorithm called Transfer Naive Bayes (TNB) that utilizes transfer learning to leverage information from diverse training data. The experimental results demonstrate that TNB outperforms existing methods in terms of accuracy (measured by AUC) while requiring less runtime. The study suggests that transferring knowledge from different- distribution training data at the feature level can improve classifier performance, potentially reducing software testing costs and enhancing effectiveness.

Das et al. (2018) [5] recommended utilising a recurrent neural network as a technique for calculating the frequency of errors or failures in the software. Recurrent Neural Networks (RNNs) are powerful models commonly used in NLP tasks, where they process sequential data by unfolding the network over time, using hidden states to capture information from previous steps, and generating outputs based on inputs and previous outputs, with parameters remaining the same throughout iterations.

Hammouri A. et al. (2018) [6] discussed software bugs have a significant impact on software reliability, quality, and maintenance costs, making bug prediction crucial in software engineering. This paper explores the use of machine learning techniques, specifically Naïve Bayes, Decision Tree, and Artificial Neural Networks classifiers, to predict faulty modules using historical fault data and software metrics. The study compares the performance of these classifiers on different datasets, assessing accuracy, precision, recall, F-measure, and ROC curves.

Farid et al. (2021) [7] research proposed a hybrid model called CBIL, combining CNN and Bi-LSTM, to enhance code review and software testing by predicting defective areas in source code, achieving significant performance improvements over baseline models in terms of F-measure and AUC.

Batool et al. (2022) [8] presented a thorough assessment of the literature on software defect prediction, with an emphasis on data mining, machine learning, and deep learning approaches. On the basis of a collection of 68 primary papers, they analyse prior reviews and studies, formulate research objectives, assess the effectiveness of various methodologies, and respond to their research questions.

R. Malhotra (2015) [9] conducted a systematic review of 64 primary studies from 1991 to 2013, focusing on machine learning techniques for software fault prediction. The results demonstrate these techniques' prediction capability and superiority over traditional statistical models in estimating software fault proneness. However, the application of machine learning techniques in this area is still limited, and further research is needed to obtain more robust and generalizable results.

III. DEEP LEARNING MODEL FOR ERROR PREDICTION

A branch of machine learning called "Deep Learning" focuses on teaching artificial neural networks with numerous layers to automatically recognise and extract useful representations from large amounts of complicated data. It draws inspiration from the design and operation of the human brain. Massive volumes of data are processed by deep learning algorithms, which also develop hierarchical representations at various levels of abstraction.

3.1 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are well-suited for speech recognition and natural language processing tasks due to their ability to efficiently analyze sequential data and capture relationships across time. In the context of predicting software errors, RNNs are valuable as they can effectively model temporal relationships in sequential software artefacts. Sophisticated variations of RNNs, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), address the vanishing gradient problem, enabling the capture of long-term dependencies in the data. During training, RNNs automatically extract relevant features from software artefacts, eliminating the need for explicit feature engineering. By leveraging labelled data, RNNs can make accurate predictions on new, unseen artefacts, estimating error likelihood. Performance evaluation metrics assess their effectiveness in error prediction. The application of RNNs in software error prediction enhances the overall quality and reliability of software systems. By aiding in early mistake identification and prevention, RNNs enable software practitioners to proactively address potential issues, leading to improved software development practices and more dependable software products.

3.2 Convolution0al Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are widely used in computer vision but have also shown promise in predicting software errors by treating software artefacts as image-like data. CNNs excel at extracting relevant features and local patterns related to errors, eliminating the need for manual feature engineering. The model architecture consists of convolutional, pooling, and fully connected layers, enabling automatic feature learning during training. By training CNNs on labelled data, they can make accurate predictions on new artefacts. Performance evaluation metrics assess the effectiveness of error prediction. Utilizing CNNs allows for early error detection and

IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal

Impact Factor: 7.301

Volume 3, Issue 2, July 2023

mitigation, enhancing the overall quality and reliability of software systems. Their ability to process software artefacts as images provides a unique perspective on error patterns, contributing to the advancement of error prediction techniques in software development.

3.3 GCN, or Graph Convolutional Network

In software error prediction, combining Recurrent Neural Networks (RNNs) and Graph Convolutional Networks (GCNs) can enhance accuracy. First, software artefacts are represented as graphs, with nodes representing elements and edges denoting relationships. GCNs are then applied, extracting features and capturing structural information. Next, the graph representations are converted into sequential data based on order or relevance. RNNs process this sequential data, capturing temporal dependencies and patterns. The combination of GCNs and RNNs allows the model to leverage both structural and temporal information. The final layers of the model predict error occurrences. By training the model with labelled data, it learns to identify error-prone regions. The performance is evaluated using metrics like precision and recall, demonstrating the effectiveness of this approach in early software error detection. Further customization and exploration can improve the model's performance in software error prediction tasks.

IV. DEEP LEARNING HYBRID MODEL FOR ERROR PREDICTION

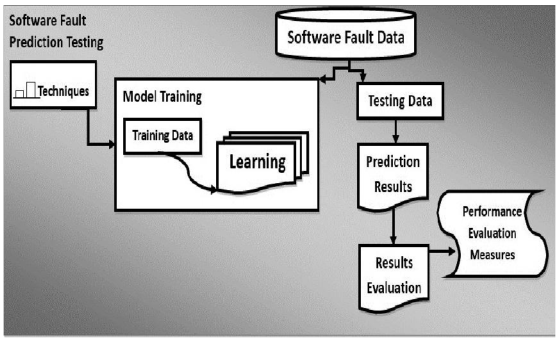
Error prediction in software development is essential for seeing and stopping possible problems before they have an impact on the system. Recurrent neural networks (RNNs) and graph convolutional networks (GCNs) are two effective methods for error prediction. Each of these methods has particular advantages that may be used to raise the precision and potency of mistake prediction in software development.

The temporal relationships and sequential patterns in error data are particularly well-suited for recurrent neural networks (RNNs). They are excellent at analysing how mistakes change over time, which helps them comprehend the dynamics of software faults. RNNs may identify patterns that suggest the possibility of upcoming errors and simulate the progression of coding faults. RNNs are able to forecast probable future errors by learning from prior errors that have already occurred. This skill is extremely useful when creating software since it enables programmers to take preventative action by learning from past mistakes.

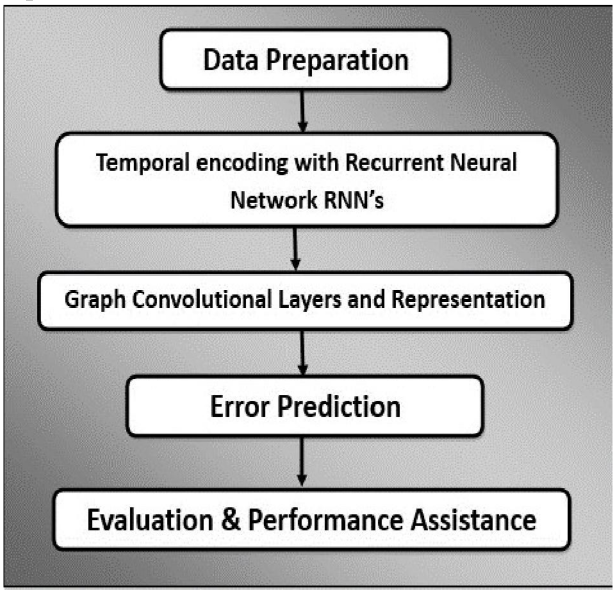
However, structural dependencies and interactions between code parts represented as graphs are well-captured by Graph Convolutional Networks (GCNs), which are another powerful tool. Code in software development frequently has intricate relationships, and GCNs may use the graph structure to efficiently determine the effects of mistakes on nearby code sections. GCNs can find complex correlations and patterns by comprehending the connections between code sections that may go unnoticed by conventional error prediction techniques. For the purpose of error prediction, software engineers can make use of the advantages of both approaches by integrating RNNs and GCNs in a hybrid model. The graph structure and temporal correlations may be successfully included in the hybrid GCN-RNN model to predict software development failures. This method attempts to get over the constraints of traditional error prediction techniques by using deep learning to find intricate patterns and relationships in software code and error data.

In order to capture structural relationships, the hybrid model's design would include supplying the GCN layers with software artefacts that are represented as graphs. The RNN layers would receive the output of the GCN layers, which now includes knowledge of the connections between code components, in a sequential fashion. In order to anticipate the occurrence of mistakes in the future, the RNN layers would learn from the sequence of errors and capture temporal relationships.

The hybrid GCN-RNN model may learn to correlate particular patterns in the graph structure and temporal sequences with error proneness by training on labelled data with mistakes recognised. The model's ability to reliably forecast mistakes may be measured using evaluation measures like accuracy, recall, or F1 score. A thorough and effective method to error prediction in software development is provided by the hybrid GCN-RNN model. This model can give software practitioners useful insights to spot potential problems early in the development process and take preventative measures to improve the quality and reliability of software systems by leveraging both the structural relationships and temporal dependencies in software artefacts.



Steps Followed:

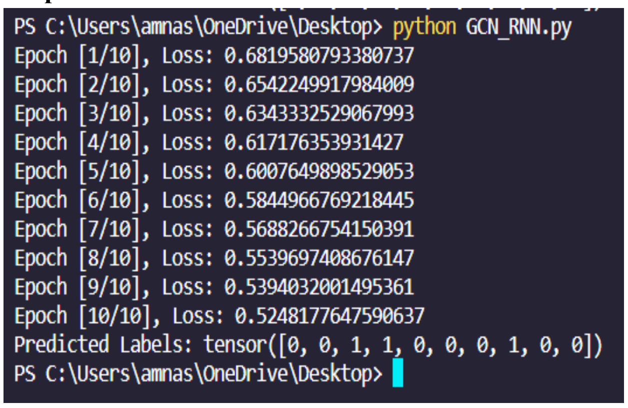

Data Preparation
Temporal encoding with Recurrent Neural Network RNN's
Graph Convolutional Layers and Representation
Error Prediction
Evaluation & Performance Assistance

- Gather and prepare data for software development.
- Divide the data into sets for training and testing.
- Get the data ready for the GCN and RNN models.
- Describe the RNN model's architecture.
- Use input sequences that are sequentially encoded.
- Set the RNN model's parameters appropriately.
- Use the training data to train the RNN model.
- Make use of the testing data to validate the RNN model.
- Predict mistakes on fresh data using the learned RNN model.
- Use a graph structure to represent software artefacts.
- Give each node in the graph a feature.
- Create the GCN model's architecture.
- Set the correct settings for the GCN model.
- Use the training set to train the GCN model.
- Assess the performance of the GCN model using the test data.
- Use the GCN model that has been taught to foretell faults in brand-new software artefacts.
- Combine the RNN and GCN models' forecasts.
- Specify an appropriate integration plan for the hybrid model.
- Use the training data to perfect the hybrid model.
- Assess the hybrid model's performance using the test data.
- Examine the performance indicators to find areas that need work.
- Improve the models by changing the architectures or hyperparameters.
- Use testing data to validate the improved models.
- Use actual software development situations to test the trained and improved models.
- Use the RNN model to predict mistakes based on sequential patterns and temporal relationships.
- Use the GCN model to look for probable faults and structural relationships.
- Take proactive steps to avoid software development problems.
- Continually review and incorporate fresh data into the error prediction models.

Example for GCN_RNN Model

```
import torch
import torch.nn as nn
import torch.optim as optim
# Define the combined GCN-RNN model
class GCN_RNN(nn.Module):
def __init__(self, gcn_input_size,
gcn_hidden_size, rnn_input_size,
rnn_hidden_size, output_size):
super(GCN_RNN, self).__init__()
self.gcn = nn.Linear(gcn_input_size,
gcn_hidden_size)
self.rnn = nn.GRU(rnn_input_size,
rnn_hidden_size)
self.fc = nn.Linear(gcn_hidden_size +
rnn_hidden_size, output_size)
def forward(self, gcn_inputs,
rnn_inputs):
gcn_output =
torch.relu(self.gcn(gcn_inputs))
rnn_output, _ =
self.rnn(rnn_inputs.unsqueeze(0))
rnn_output = rnn_output.squeeze(0)
combined = torch.cat((gcn_output,
rnn_output), dim=1)
output = self.fc(combined)
return output
# Generate random inputs and labels
(simplified for demonstration)
gcn_inputs = torch.randn(100, 10)
rnn_inputs = torch.randn(100, 5)
labels = torch.randint(2, (100,))
# Create an instance of the GCN-RNN model
gcn_input_size = gcn_inputs.size(1)
gcn_hidden_size = 32
rnn_input_size = rnn_inputs.size(1)
rnn_hidden_size = 64
output_size = 2  # Binary classification (has
error or not)
model = GCN_RNN(gcn_input_size,
gcn_hidden_size, rnn_input_size,
rnn_hidden_size, output_size)
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(),
lr=0.01)
# Training loop
num_epochs = 10
for epoch in range(num_epochs):
optimizer.zero_grad()
outputs = model(gcn_inputs, rnn_inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
print(f"Epoch [{epoch+1}/{num_epochs}],
Loss: {loss.item()}")
# Test the model
test_gcn_inputs = torch.randn(10, 10)
test_rnn_inputs = torch.randn(10, 5)
test_outputs = model(test_gcn_inputs,
test_rnn_inputs)
predicted_labels = torch.argmax(test_outputs,
dim=1)
print("Predicted Labels:", predicted_labels)
```

Output:



```
PS C:\Users\amnas\OneDrive\Desktop> python GCN_RNN.py
Epoch [1/10], Loss: 0.6819580793380737
Epoch [2/10], Loss: 0.6542249917984009
Epoch [3/10], Loss: 0.6343332529067993
Epoch [4/10], Loss: 0.617176353931427
Epoch [5/10], Loss: 0.6007649898529053
Epoch [6/10], Loss: 0.5844966769218445
Epoch [7/10], Loss: 0.5688266754150391
Epoch [8/10], Loss: 0.5539697408676147
Epoch [9/10], Loss: 0.5394032001495361
Epoch [10/10], Loss: 0.5248177647590637
Predicted Labels: tensor([0, 0, 1, 1, 0, 0, 0, 1, 0, 0])
PS C:\Users\amnas\OneDrive\Desktop>
```

V. CONCLUSION

This research study addresses the use of deep learning approaches for this purpose and concludes by highlighting the importance of mistake prediction in software development. The necessity of proactive mistake avoidance and the drawbacks of reactive bug-fixing strategies are emphasized in the study. It covers different deep learning models and explores their use in error prediction, including Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and Graph Convolutional Networks (GCNs).

According to the study's conclusions, RNNs can accurately capture temporal dependencies and sequential patterns in error data, allowing for the investigation of error progression over time. The capacity of CNNs to extract pertinent features and local patterns from software artefacts and treat them as image- like data is demonstrated. Software artefacts' graph structures are used by GCNs to capture structural dependencies and interactions between code elements.

By including both temporal and structural information, this study's hybrid model for error prediction combines RNNs with GCNs. The model's capability to make use of deep learning's ability to recognize patterns and model relationships offers promise for precise forecasting and preventing mistakes in software development Overall, the quality of software, resource efficiency, and user experience may all be improved by software development teams implementing proactive mistake prediction tactics made possible by deep learning.

REFERENCES

[1] Wang, H., Khoshgoftaar, T.M., Napolitano, A.: Software measurement data reduction using ensemble techniques. Neurocomputing 92 (2012) 124–132

[2] Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. Software Engineering, IEEE Transactions on 38 (2012) 1276–1304

[3] Ahmad Hasanpour, Pourya Farzi, Ali Tehrani, Reza Akbari, "Software Defect Prediction Based On Deep Learning Models: Performance Study" https://arxiv.org/ftp/arxiv/papers/2004/2004.02589.pdf

[4] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross- company software defect prediction" Information and Software Technology, vol. 54, no. 3, pp. 248–256, 2012.

[5] S. Das, R. K. Behera, S. K. Rath et al., "Real-time sentiment analysis of Twitter streaming data for stock prediction" Procedia computer science, vol. 132, pp. 956–964, 2018.

[6] 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):78-83

[7] Farid AB, Fathy EM, Sharaf Eldin A, Abd-Elmegid LA. 2021. Software defect prediction using hybrid model (CBIL) of convolutional neural network (CNN) and bidirectional long short- term memory (Bi-LSTM) PeerJ Computer Science 7:e739 https://doi.org/10.7717/peerj-cs.739

[8] Software fault prediction using data mining, machine learning and deep learning techniques: A systematic literature review Batool I., Khan T.A. (2022) Computers and Electrical Engineering, 100, art. no. 107886

[9] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction" Applied Soft Computing Journal (2015)