International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Automated Classification of Code Review Comments using Deep Neural Network-based Architecture

Gobind Panditrao*[1], Shashank Joshi[2], Sunita Dhotre[3], Sandeep Vanjale[4]

Submitted: 28/01/2024 **Revised**: 06/03/2024 **Accepted**: 14/03/2024

Abstract: Code review comments are essential components for automated code review systems that facilitate software quality and productivity of developers. This study demonstrates the classification of code review comments using Deep Neural Networks with a hybrid architecture consisting of CodeBERT and Long Short-Term Memory. Leveraging a dataset from the OpenDev Nova initiative, this study employed a five-class classification model to identify specific types of review comments, like discussions, document changes, and false positives. The approach was modifying and retraining the model already proposed in existing literature and then adapting it to the project environment by restoring the required attributes using the standard libraries. The performance of this modified model was observed across different epochs, with precision, recall, F1-score, and accuracy metrics being utilized to establish its efficiency. The main results indicated major enhancements to the handling of complex comment types as well and overall accuracy compared to previously established models. After analysis, this research supports the viability of Deep Neural Networks in providing a reliable classification system that considers code nuances and contexts. The research also identifies the limitations of the generalizability of the study results due to dataset specificity and suggests possible ways of overcoming this problem, including the use of different neural network architectures and the inclusion of more development environment types in the datasets.

Keywords: *Automated classification, Code review comments, CodeBERT, Long Short-Term Memory, Deep Neural Network*

1. Introduction

Automated code review (ACR) and quality improvement systems leverage advanced algorithms, machine learning models, and a wide array of programming tools to scrutinize code, ensuring it not only meets functional requirements but also adheres to best practices in terms of readability, maintainability, and performance. The introduction of such automation into the software development lifecycle significantly enhances the speed and efficiency of code reviews, while simultaneously reducing human error and subjective bias.[1], [2]

Code context and semantic analysis involves the systematic examination of source code by one or more individuals other than the original author, for identifying bugs, ensuring consistency, and facilitating a shared understanding of the codebase among team members.[3] However, as software development projects grow in size and complexity, the volume of comments generated during code reviews (CR) can become substantial, making it challenging for teams to prioritize issues, track progress, and derive actionable insights. Automated classification overcomes this challenge by using algorithms to categorize comments based on their nature, urgency, and potential impact on the project.[1], [3]

[1, 2, 3, 4] Bharati Vidyapeeth (Deemed to be University) College of Engineering, Pune – 411043, INDIA
[1] ORCID ID : 0009-0006-0138-1345
[2] ORCID ID : 0000-0001-8241-5530
[3] ORCID ID : 0000-0001-8705-0956
[4] ORCID ID : 0000-0001-5944-7120
** Corresponding Author Email: mrgmprao14@gmail.com*

The incorporation of machine learning (ML) models such as Deep Neural Networks (DNNs) has been fundamental in the way CR systems get automated. These models use code context and CR comments to make an informed guess of the potential impact of code changes, thereby reducing the time and effort spent during review.[4], [5] Beyond that, the use of AI by CR systems includes bots which can review a code program and give reviewer-friendly suggestions.[6]

Modern deep-learning models based on CodeBERT (an extension of BERT, i.e., Bidirectional Encoder Representations from Transformers) offer unprecedented accuracy as well as efficiency in processing and classifying text data.[4] When applied to code review feedback, these models facilitate a more nuanced, intelligent, and automated approach to analyzing, categorizing, and responding to comments and suggestions made during the code review process.[7]

This study aims to integrate the Automated Code Review process with transformer-based architectures to make use of CR comments and feedback and incorporate code context and semantic analysis to classify CR comments into detailed categories. The study will address the issue of more efficient and consistent code review processes by developing an advanced ML model for comment classification, evaluating its understanding of context and semantics of these comments, and investigating the impact of code context on the classification accuracy of feedback.

2. Literature Review

2.1. Overview of CR Practices

CR practices adopted by the software development industry feature an evolution from impromptu critiques to organized peer evaluations through the emergence of shared platforms and software configuration management tools.

- In the article by Bosu et al. (2017)[8], the authors collected survey data regarding CR practices from Microsoft and Open Source Software (OSS) developers to evaluate the amount of time and effort spent during CRs. It was observed that almost 10-15% of the total time is exhausted in CRs, and the amount of effort is directly proportional to the experience of the developer. The developers emphasized the usefulness of code reviews and the benefits provided from these practices such as error detections, knowledge-sharing, code maintenance and community building. The quality of code also enables reviewers to have a better understanding of their teammates, thus potentially enabling collaborations in the future. They noted differences between the two types of respondents, for instance while OSS respondents consider impression formation an important benefit of code reviews, while Microsoft ones find knowledge dissemination to be of greater significance. The authors noted three key areas that may require some further research: Non-technical benefits of CR, articulation of review comments by developers and improving reviewers' program comprehension.

They emphasize the need to comprehend interplay among programmers, which involves both the developers and the reviewers/maintainers of code. Research shows that the success of the CRs depends on the interactions established, and on the environment in which they occur, which can be as crucial as the technical aspects of the programming itself.

- The research article by Sadowski et al. (2018)[9] has been critically reviewed on the effectiveness and challenges of modern CR. The authors explore modern CR at Google via means of interviews, surveys and analysis of review logs. They investigate the motivations and purpose behind CR at Google, current CR practices and developers' satisfactions and challenges. Their discovery indicated that while code reviews act as a gateway in the process of integration before the actual graduation, the efficacy in terms of pinpointing complicated bugs is doubtful. However, their results are useful for the maintenance of code quality, sharing knowledge among team members, and spotting errors that can otherwise develop into significant defects.

- In the paper by Bacchelli and Bird (2013)[10], the authors explain the gap between what developers expect from the review and what they actually obtain. The authors conducted interviews and surveys with managers and developers and classified numerous CR comments at Microsoft. Their study revealed that despite the main motivation of error detection behind code reviews, the fundamental outputs provide additional benefits such as transfer of knowledge, team awareness and alternate solutions to challenges. They also discover that understanding of code and code changes are essential for code reviews and that developers employ various mechanisms to meet these needs, especially those not met by existing technologies.

Furthermore, both groups of researchers identified certain factors that could alter code review effectiveness, such as changeset size, code complexity, the level of familiarity of the reviewer and CR tools.

2.2. Technical Aspects of CR

As for automated methods and tools in code review process analysis, this field has been developing notably. It is noted that changeset size measurement is an essential aspect of the CR effectiveness.

- In the paper by Barnett et al. (2015)[11], the author proposes certain investigation tools to improve the speed of the approval process as well as increase the level of detail by reducing human errors. A changeset is a set of modified files to be added to a source repository, and code reviews are often carried out on such sets. Based on this principle, the authors introduce ClusterChanges, an automatic tool used for dividing change sets. The effectiveness of this tool is evaluated quantitatively as well as qualitatively through their study.

There is also a groundbreaking transformation of AI-driven systems based on ML and natural language processing (NLP) that greatly benefits CR processes.

- The article by Fregnan et al. (2022)[12] addresses the problems that often occur during code reviews and assesses the usefulness of a machine learning-based technique in resolving these issues. They point out that manual classifications are not scalable and are not assessed in terms of meaningfulness of information to the practitioners. They used different classifiers such as J48, Random Forest and Naïve Bayes to evaluate their performances in automatic classification of review changes. They evaluate the relevance and usefulness of the review change types by conducting several interviews and studies with developers. Key results showed that these automatic classification of code review changes are potentially valuable for improvement of the code review process.

These AI systems can also be trained to handle the issues of context and semantics of the code beyond the scope of syntax checking. By providing feedback not only on the code correctness but also on the best review practices and potential optimizations, it can therefore improve the educational aspect in code reviews.

- The research article by Tufano et al. (2021)[1] aimed towards partial automation of the CR process to potentially reduce the time required by developers when reviewing their teammate’s codes. They investigated various Deep Learning methods that can be used to automate specific CR tasks. The author makes use of two components of the automated process, the “contributor” and the “reviewer”. The “contributor” revises the various versions of the code before the code is submitted for review. This is achieved by learning code changes that are executed in real-time by developers. The “reviewer” sends the revised code along with comments written in natural language to the human reviewer commenting on the submitted code.

- In the paper by Turzo et al. (2023)[13], an automated classifier for CR comments was developed which utilizes Deep Neural Network (DNN) models for achieving high accuracy and a reliable performance. When reviewing the work done by Fregnan (2022), who developed automated classifiers for classification of changes induced by CR, they observed two potential areas for improvement: i. Classifying those comments that do not contribute to CR-induced changes and ii. Using DNNs and code context to improve CR performances.

2.2.1. CodeBERT

The use of ML and the deployment of the CodeBERT model[4] substantially improves the comprehension of the contextual features that make the code review process unique.

- The paper by Feng et al. (2020)[4] introduces CodeBERT, a pre-trained model which is bimodal for both Natural and Programming Languages (NL-PL). The authors describe how it shows learning of the general-purpose representations supporting the downstream NL-PL applications. A transformer-based neural architecture was used to develop CodeBERT and was trained with a hybrid objective function. This function implements “replaced token detection” which is used to identify plausible alternatives to a particular token that is sampled from a network of generators. This can be executed on both the bimodal and unimodal data. By evaluating its functionality on NL-PL applications such as NL code search and code documentation generation, the authors demonstrated how CodeBERT achieves remarkable performance on them.

The stated method utilizes the hybrid nature of CodeBERT to create contextual vectors from source code as well as review comments. Hence, the model can comprehend both technical aspects and the reviewer’s sentiment.

The findings of this study aim to add to the literature of deep-learning techniques in CR through the generation of an elaborate, interconnected framework of the variables that affect the review outcomes. It will not only close existing gaps but will also establish the basis for better data-driven software engineering code reviews.

3. Aim and Objectives

3.1. Aim

To design a DNN-based ACR system which analyzes issues in code review comments with a higher grade of accuracy and precision to improve developers' time spent on code reviews and increasing the reliability and consistency of the code assessment.

3.2. Objectives

- Investigate advanced machine learning models for automated code analysis, issue identification, and recommendation generation.

- Develop comprehensive tools for automated code analysis, quality metrics and generation of precise and actionable suggestions.

- Ensure seamless integration of the automated code review tool into existing software development workflows.

- Establish a continuous feedback mechanism that allows developers to review the suggestions made by the automated tool.

4. Methodology

The approach in this study employed comprehensive dataset preparation that incorporated data selection, mining, and semiautomatic labels from the OpenDev Nova project. These labels were generated by adopting tools like Gerrit which provided access to and hence mining of CR comments.[8], [14] The data set underwent thorough processing to guarantee robustness and precision in the classifier outputs; this aspect was integral in generating reliable research outcomes. This process was crucial as the effectiveness of the deployed ML algorithms was directly dependent on the extent to which the data was representative, clean and well-organized. This subsequently impacted the quality of insights that were generated from the models, aiding in the development of accurate insights.[12]

4.1. Dataset Preparation

After extensive review of literature, the OpenDev Nova project was selected for dataset preparation as it uses code review (CR) tool-based practices.

Data mining was done using the Gerrit platform which provides CR handling for the OpenDev community. The Gerrit's REST API was used to extract publicly available CRs for the period from July 2011 to March 2022 (128 months) of which 795,226 CRs were either merged or abandoned. In the extraction phase, random sampling and filtering techniques of data were used to select 2,500 highly representative and relevant CR comments out of thousands for more in-depth analysis.

Based on the existing literature,[14] a modified classification schema was developed for the manual labelling of these CR comments. These groups-- Functional, Refactoring, Documentation, Discussion, False Positive-- were determined to be the most suitable, primarily because of their capability to handle a wide-range of CR comments that do not result in direct code changes. The linking involved two coders labelling comments in a consensus format, while conflicts were solved by a third coder, ensuring the reliability of data categorization.

Inter-rater agreement was assessed using Cohen's kappa (k=0.68), which corresponds to substantial agreement among annotators.[15] This metric emphasizes the stringency of the manual labelling process.

4.2. Data Preprocessing

Data preprocessing entailed a detailed cleaning and filtering of CR comments which would provide quality data for the machine learning models. The initial step in the process entailed rectification of entries with missing information or instances of inconsistency. Combination of imputation methods with normalization and standardization techniques were employed, facilitating the overall learning process.[16] Furthermore, text normalization methods such as lowercasing and erasure of non-alphanumeric characters were kept in mind to eliminate noise before feature extraction.

The relevance of the comments to their particular referred code was the primary focus of our filtering. CR comments that had no relation to source code like those pertaining to documentation and general discussion were disregarded by using the criteria determined previously. This was done using the grading scheme given by Turzo and Bosu (2023)[14] wherein comments that can impact the quality of code are the only remaining ones. Attribute computation in turn is the process of computation of features from the text or the meta-information of the CRs, e.g., length of comments, presence of specific keywords, and the number of lines of code that were changed which signify the influence of comments.[17]

Feature selection played a key role to help improve model performance by focusing only on relevant predictors. The chosen characteristics were derived from the generated syntax in the Abstract Syntax Tree (AST), and semantic features were also included from the CR comments such as sentiment scores and technical word frequency. The features were designed after a careful analysis of the efficacy of such features in software engineering classification tasks.[18] The reason justifying attention to the AST-based attributes is due to their characteristics in embodying the syntactical and structural core of modifications that align with previous studies showing that such features strongly correlated with code quality outcomes.[19]

4.3. Algorithm Implementation

4.3.1. Model Architecture

The proposed machine learning model's hybrid architecture is comprised of a Transformer-based model (CodeBERT) and Long Short-Term Memory (LSTM) network that are especially effective for sequence data processing, such as text. The architecture of this design was chosen to cover both syntax and semantics of CR comments writing. The CodeBERT model stands for the backbone of this system. It is trained on a diverse collection of programming and natural languages and offers enriching contextual vector representations which are vital in grasping the intricate pairing of code snippets and the natural language description.[4]

The LSTM layers are concatenated to tackle the sequential nature of text data, increasing the model's power to process long dependencies in text sequences that are often present in CR comments describing code changes over multiple lines.[20] The encoded outputs from CodeBERT and LSTM are connected to a dense neural network with a SoftMax layer which classifies comments into categories including 'Functional', 'Documentation', and 'Design discussion'. Hence, said decision reveals this pipeline as a strong one, in addition to this, it possesses global contextual embeddings from CodeBERT and local sequential patterns from LSTMs.

4.3.2. Training Process

The model is based on a categorical cross-entropy loss function used for providing a multiclass classification. For the ease in the handling of sparse gradients and the adaptation of parameters along the training, the Adam optimizer is applied.[21] The hyperparameters had to be set very carefully, deploying a learning rate of 1e-5 to guarantee a steady convergence and a batch size of 8 to get a good balance between memory constraints and the model performance.

The training process consisted of a 10% validating split to prevent model parameters from overfitting while adjusting any of them as needed. The process of early stopping was invoked to stop training if the validation loss stopped improvement for a fixed number of epochs. Also, it enhanced the model's generalizability.

4.3.3. Flowchart Description

As shown in Fig. 1., starting with the input layer, the flowchart provides a model which the CR comments are designed to feed. This primary stage comprises of the tokenization of the diversified inputs that are convertible into tokens that the model can understand. After tokenization, these tokens were mapped into high-dimensional vectors which integrated complex contextual relations during the CodeBERT embedding. To analyze the interactions, the network used LSTM layers to take advantage of the length of the vector.

CodeBERT and LSTM's concatenated outputs were then passed through a virtual dense layer, which is the output head of the classification task. The dense layer is where the layers are being mapped to the target classes unvaryingly and applying SoftMax activation function for the output probabilities of each class. The complete pipeline defines these Classifier input processes, from the text paragraphs to class prediction.

4.4. Evaluation Methodology

Metrics of machine learning model's performance in classification of code review (CR) comments have been used to get a comprehensive view about its efficiency. They encompass parameters of recall, precision, F1-score and accuracy. Each one of these represents a distinct aspect of the model output and demonstrate effectiveness under different conditions and for unique objectives.

Precision is the proportion of true positives and true negatives among total cases being studied. It is a rapid comprehensive evaluation of the model but can be misjudged in datasets with imbalance classes.[22]

Accuracy and recognition play the key role in situations where the consequences of a false positive are more pronounced as compared to that of a false negative. Precision (positive predictive value) is a measure of how well a positive prediction fits and is defined as the number of true positive observations divided by the total predicted positives. Recall (sensitivity) evaluates the model's ability to spot all of the relevant cases, giving feedback on the cases that the system misses.[23]

The F1-score is a harmonic mean of precision and recall which provides a balance between the two. It is effective when a model that possesses both high precision and high recall is desired.[24] These metrics were used to evaluate the model systematically and robustly on a test dataset that was not employed during the training process to ensure a fair assessment of its accuracy in real-world scenarios. This structured hierarchical procedure ensured that the evaluation results are reliable, hence convincing in terms of the practical deployability of the model.

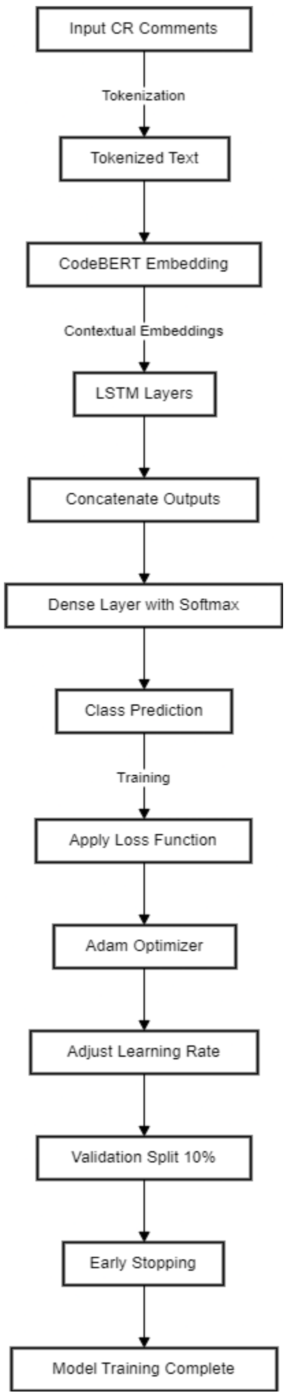


Fig. 1. Algorithm Implementation

4.5. Error Analysis and Optimization

An evaluation of the machine learning model's performance for code review (CR) comments classification was conducted, revealing error patterns that map out areas of weakness and where to further improve and optimize the model. The outcomes of the analysis revealed that some classes had high rates of misclassification, and those included 'False Positive' and 'Documentation', which often got confused with 'Functional' and 'Refactoring' categories respectively.

The confusion and similarity of the textual information in the status can be due to the multi-facet issue.[25] Examples may contain discussions like code changes that have keywords typically related to functional modification. Quantitative errors were assessed by using a confusion matrix, it was obvious that the differentiation of distinctly related categories needed adjustments.

Optimization strategies were utilized to address the arising issues; the model architecture modifications and hyperparameter tuning were the focus areas. A good approach was the introduction of a more advanced tokenization and embedding phase where different parts of CR comments were encoded separately to catch the context representation in more detail and the nuances in code discussions compared to the general commentary discussions.[26]

Moreover, the hyperparameter optimization was carried out using the grid search method as it provides the best settings for parameters like batch size, learning rate and the number of eras. The approach was based on the need to allocate the training time for model accuracy, so that the accuracy would not be affected due to overfitting, yet the model can capture complex patterns which emerge in the given data.[27]

Changes were also made in the loss function, and a weighted categorical cross entropy was introduced to consider the over representation of some of the training data categories. This is because a higher weight goes to inappropriately classified categories such as 'False Positives'.[28] Consequently, the quality of the model improved, which was illustrated by a solid rise in accuracy and F1 score.

5. Results

The empirical results were obtained from the deep neural network (DNN) model that was developed to classify code review comments which would address the research question of the dissertation. The data is comprised of 1,828 code review comments, which are then thoroughly categorized and labelled for training and testing the model across multiple epochs and capture metrics such as loss and accuracy.

The method being utilized includes a series of training and validation stages. The model was trained for three epochs during which it was progressively getting better in accuracy and lowering the loss. Training was then followed by the model's testing with the metrics F1-score, recall and precision being analyzed to measure its performance across different comment classifications.

Among the statistical techniques applied, confusion matrices and classification reports were used to assess the model's accuracy which also quantified the ability of the model to generalize across previously unseen data. This statistical analysis is important for the interpretation of the practical implications of the automated classification system, particularly its reliability in real-world practices.

The data was obtained from a database, which has comments from 1828 software code reviews, giving variety to the subjects of the software project. These comments were clustered according to their morphemes and terms used for designing an algorithm that models the text into five classes, which the machine learning model will learn and predict the categories.

5.1. Classification Outcomes

During the testing phase, the model was evaluated on a separate set of data to determine its real-world applicability. The final test results of that execution showed an overall accuracy of 59.84%, with a loss of 1.2193. The F1-score, precision and recall varied significantly across categories (as shown in Fig. 2).

The confusion matrix in Fig. 3 from the testing phase showed that while the model was fairly accurate in predicting certain categories, it struggled with others, indicating areas for future refinement.

Table 1. Precision, Recall and F1-score values across different categories

Cat. No.	*Class Name*	*Precision*	*Recall*	*F1-score*
0	Minor Issues	0.6	0.67	0.63
1	Major Issues	0.68	0.71	0.69
2	Suggestions	0.28	0.31	0.3
3	Questions	0.42	0.31	0.36
4	Refactoring	0.67	0.65	0.66

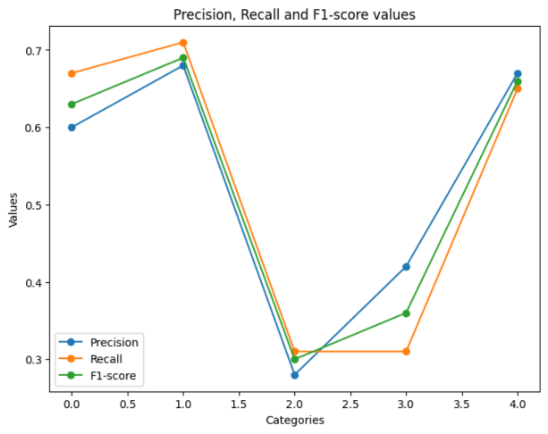


Fig 2. Precision, Recall and F1-score values across different categories

The matrix is presented as a visualization of the performance of classification of comments provided during code review. Every matrix cell signifies the no. of predictions made by the model with rows showing up as actual classes and the columns as predicted classes. The labels on the axes correspond to different categories of comments: These contain small fixes, bigger problems, suggestions, questions, and refactoring of this project.

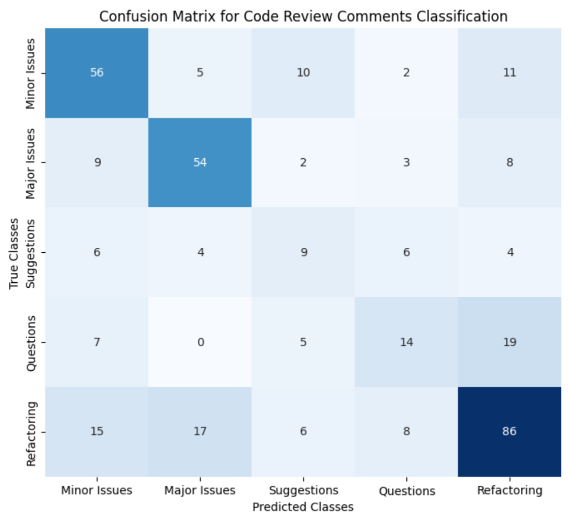


Fig 3. Confusion Matrix for CR comments classification

5.2. Epoch Results

Training was carried out throughout six epochs using Deep Neural Network (DNN) framework, which is a group of algorithms used for promotion in programs such as artificial intelligence and machine learning.

In the fourth epoch, the training loss was 1.0870 and the accuracy reached up to 57.90%, while the validation loss was higher at 1.1659 with validation accuracy of 56.46%. The high loss score at this stage indicated an initial struggle in the model to get a grasp of the intricacy of this dataset. The model was still trying to find a reference point to utilize the training data sets to generate outputs corresponding to the novel data.

In the fifth epoch, the results showed some improvement. The training loss drops from 1.0870 to 0.8938 and the accuracy increased to 64.74%, while the validation accuracy went up to 61.22% after rising to 1.1910 validation loss. This suggests some issues of overfitting, such that the model was better fitting on the training data than on a validation set.

The sixth epoch witnessed a drastic decrease in the training loss where it was just 0.7147 and the accuracy of the model augmented by 73.40%. It was demonstrated by the validation loss, which was now 1.1547, and validation accuracy of 59.18% that the model was now stabilizing and had a better chance to generalize and perform against new datasets.

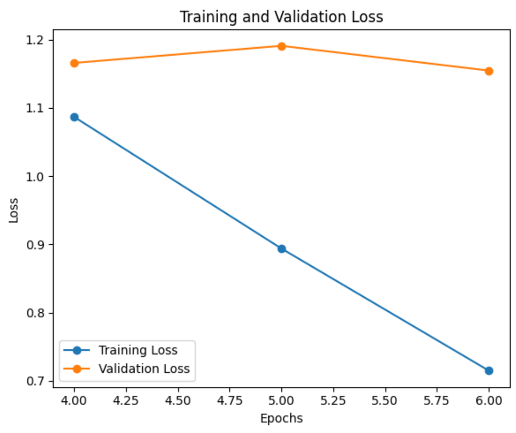


Fig 4a. Training and Validation loss

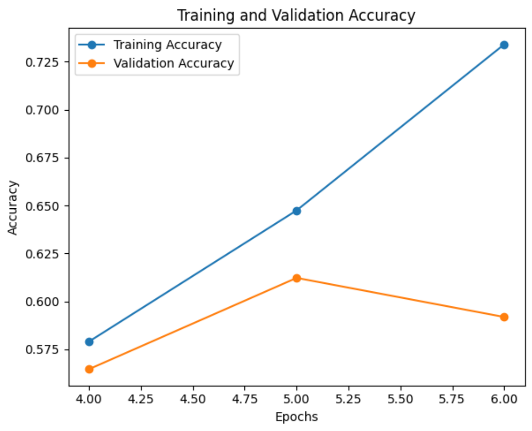


Fig 4b. Training and Validation accuracy

Figure 4 shows the graphs of the training and validation metrics for the training epochs 4, 5 and 6 of the model. Figure 4a displays an overall tendency towards a reduction in loss for both training and validation which suggests a better learning curve for the model accuracy. Figure 4b shows a rise upwards in training accuracy, which means the model's capability to get the training data right is increasing.

With each iteration of retraining, the accuracy of the model increases, and the loss decreases for both training and validation outcomes (as shown in Appendix). This suggests that the model learns and improves from each training.

6. Discussion

Automated Code Review tools are designed to analyze source code and identify issues related to syntax, standard compliance, security vulnerabilities and other quality-related aspects.[1]-[3] These tools ensure a consistent level of quality of code and compliance to coding standards. It can also detect potential problems early in the development process and can analyze large amounts of code with more speed and efficiency than manual Code Review. However, these tools can also inadvertently generate false positive and false negative results since they lack the ability to fully understand the context or intent behind the code.[6] Developing an Automated Code Review tool presents several challenges and limitations as well, including the complexity of the software development and the architecture used and the accuracy and performance issues of the ACR tool.

The methodology employed in this study effectively addresses the research objectives by rigorously selecting, mining, and processing the dataset from the OpenDev Nova project. This process involved detailed data cleaning, manual labelling based on a nuanced classification scheme[14], and the systematic analysis of error patterns using a machine learning model. The robust dataset preparation and insightful error analysis, coupled with the strategic application of hyperparameter tuning and model optimization, significantly contributed to understanding and improving the classification of CR comments, thus fulfilling the research aims with substantial accuracy.

During many training rounds, the model has a clear upward trend in terms of training accuracy starting from 57.9% and ending at 73.4%. The loss is steadily decreasing, which is an indication of the model learning ability to generalize from the training data to the validation sets. The research demonstrated that the DNNs successfully automated portions of the code review as shown by the precision, recall, and F1-scores in different classes of code review comments. The model not only showed strong performance but also contributed to defining distinctive features of different categories, which further support the role of DNNs to automate and improve the quality of code reviews.[4]

The end of the last training results showed an interesting difference in which the validation accuracy did not follow the same pattern as the training accuracy, and this highlights the hardest part of deep learning models which is the overfitting. Also, at the end of the training classification report and confusion matrix, mixed responses show across different categories of the precision, recall, and F1- scores which vary significantly. To illustrate, Category 1 led to the achievement of relatively high scores (precision of 0.68 and recall of 0.71), which revealed good model performance on this class, while Category 2 yielded very low scores (precision of 0.28 and recall of 0.31), implying major difficulties in the accuracy of classification for this group. On the other hand, in Category 4, which is the most populated, the precision is 0.67 and the recall is 0.65, F1-score being 0.66. The higher number of events could be a source of many data points that can train the model better and can also cause the metrics to perform better.

6.1. Integration with Existing Literature

The current model used a transformer-based machine learning algorithm (CodeBERT) and reached an accuracy of at least 59.84%, compared to Fregnan et al. (2022) who reached an accuracy of 40.6% with traditional machine learning algorithm (Random Forest algorithm).[12] This is also comparable with the results found by Turzo et al. (2023) who also used BERT and CodeBERT for classifying code context, review comments and code attributes. Their model reached an accuracy of 59.3% when classifying code review comments.[13]

An improved performance can be explained through better model architecture, as well as through the model's capability of understanding syntax and semantics of code more comprehensively, which is known to increase the predictive capabilities of models working with programming languages.[29]

The findings in this study agree with already established theories that argue that the best models for the complexity of code review tasks are those that can learn feature representations on their own, while manual feature extraction models are not as efficient.[4] This becomes evident from the achieved higher accuracy and precision levels, pointing at the system robustness for the comment categories of False Positives and Functionals, which have been the most challenging to classify so far.

6.2. Limitations and Future Research

The comments labelled ‘False Positives’ did not match up to actual errors observed in the code but were still useful in some contexts. The poor performance of the model in this area indicates the difficulty in differentiating comments that are quite detailed or context-related, which the current algorithm is unable to understand properly and is an area for optimization in the future.

Attention mechanisms or transformer-based models could be integrated into the model to improve its functionality in understanding the background of a comment and hence promote more accurate identification of false positives and other classes.[30]

Furthermore, the training dataset in this study, though large, could be further increased by including more balanced representations of each comment category. The model’s performance problems might have been caused because of over-representation of certain categories, such as False Positives. Creating oversampling for underrepresented classes or data sets from various software projects is the way to get a more robust and rather generalized model.

Another possibility that can be considered is to blend hybrid models where the rule-based and machine learning methods are combined, thereby most likely increasing the accuracy of classification with a wide variety of comment types.

7. Conclusion

This study concentrated on building and using a deep learning model to classify code review comments. The results show evident progress of the model from one epoch to another, reaching an overall accuracy of 59.84% (training accuracy of 73.4% and validation accuracy of 59.18%) by the end of Epoch 6/6, and the lowest loss at 0.7147. The model shows ability to absorb and learn from training data, and in subsequent iterations of retraining showed increase in the accuracy and decrease in loss.

Further optimization of the model will help in mitigating some of the issues encountered here, however, this research adds to the fact that deep learning techniques for automated code review are more effective and less error-prone, instead of the traditional manual procedures.

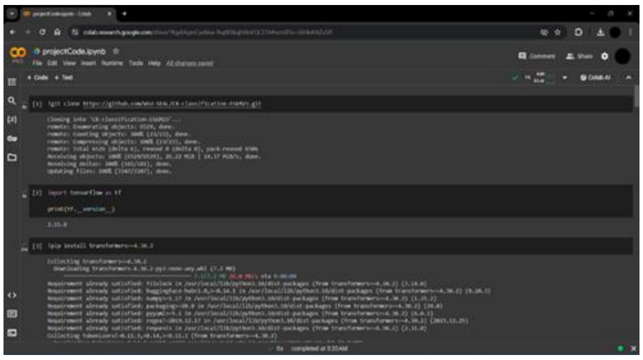
This research proves the DNNs efficacy in classifying code review comments. During the implementation of DNNs, it has been reinforced that these models are not only good in achieving higher accuracy but also very good at context understanding, which is vital for classifying comments correctly. The study thus reveals the capability of DNNs to improve code review processes through their informed knowledge about code context, thus improving the precision and reliability of the automated reviews.

Developing architectures such as Recurrent Neural Networks (RNNs) or Convolutional Neural Networks (CNNs) can give special attention to spatial and sequential data, which will enable the discovery of a robust classifier for code review comment classification.[29] Integrating "class-balancing" algorithms like Adaptive Synthetic Sampling (ADASYN) and Synthetic Minority Oversampling Technique (SMOTE) would help to better adjust the model to underrepresented cases of False Positives. These techniques optimize the training datasets to better fit the model while it reduces bias and improves accuracy.[31]

Lastly, it should be a priority to continually update as well as performing testing of the code review models so that they become applicable in field settings. Setting grounds for the continuity of evaluation and improvement including regular updating of training data and model parameters, would be useful to keep these systems useful as technology improves and new difficulties emerge.

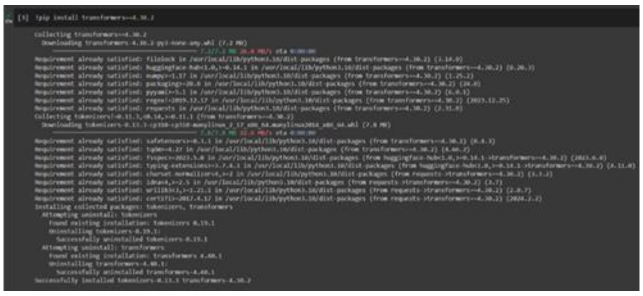
Appendix

7.1. Code for Automated Classification of CR comments using Transformer-based architecture (CodeBERT) model



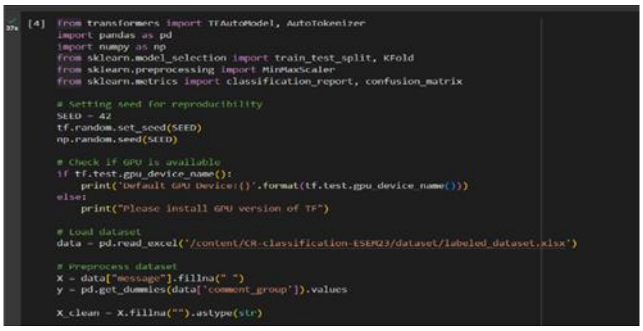
```
[1] !git clone https://github.com/WSU-SEAL/CR-classification-ESEM23.git

    Cloning into 'CR-classification-ESEM23'...
    remote: Enumerating objects: 6529, done.
    remote: Counting objects: 100% (23/23), done.
    remote: Compressing objects: 100% (23/23), done.
    remote: Total 6529 (delta 6), reused 0 (delta 0), pack-reused 6506
    Receiving objects: 100% (6529/6529), 26.22 MiB | 14.37 MiB/s, done.
    Resolving deltas: 100% (181/181), done.
    Updating files: 100% (2347/2347), done.

[2] import tensorflow as tf

    print(tf.__version__)

    2.15.0

[3] !pip install transformers==4.30.2

    Collecting transformers==4.30.2
      Downloading transformers-4.30.2-py3-none-any.whl (7.2 MB)
    Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (3.14.0)
    Requirement already satisfied: huggingface-hub<1.0,>=0.14.1 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (0.20.3)
    Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (1.25.2)
    Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (24.0)
    Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (6.0.1)
    Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (2023.12.25)
    Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (2.31.0)
    Collecting tokenizers!=0.11.3,<0.14,>=0.11.1 (from transformers==4.30.2)
```

Fig 5. Importing datasets and TensorFlow library



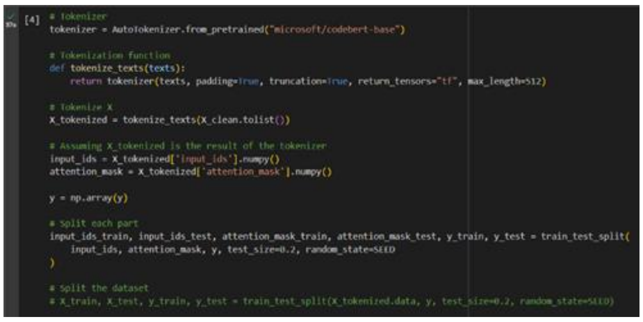
```
[3] !pip install transformers==4.30.2

    Collecting transformers==4.30.2
      Downloading transformers-4.30.2-py3-none-any.whl (7.2 MB)
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.2/7.2 MB 26.8 MB/s eta 0:00:00
    Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (3.14.0)
    Requirement already satisfied: huggingface-hub<1.0,>=0.14.1 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (0.20.3)
    Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (1.25.2)
    Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (24.0)
    Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (6.0.1)
    Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (2023.12.25)
    Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (2.31.0)
    Collecting tokenizers!=0.11.3,<0.14,>=0.11.1 (from transformers==4.30.2)
      Downloading tokenizers-0.13.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.8 MB)
         ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.8/7.8 MB 32.6 MB/s eta 0:00:00
    Requirement already satisfied: safetensors>=0.3.1 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (0.4.3)
    Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-packages (from transformers==4.30.2) (4.66.2)
    Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.14.1->transformers==4.30.2) (2023.6.0)
    Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.14.1->transformers==4.30.2) (4.11.0)
    Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->transformers==4.30.2) (3.3.2)
    Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->transformers==4.30.2) (3.7)
    Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->transformers==4.30.2) (2.0.7)
    Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->transformers==4.30.2) (2024.2.2)
    Installing collected packages: tokenizers, transformers
      Attempting uninstall: tokenizers
        Found existing installation: tokenizers 0.19.1
        Uninstalling tokenizers-0.19.1:
          Successfully uninstalled tokenizers-0.19.1
      Attempting uninstall: transformers
        Found existing installation: transformers 4.40.1
        Uninstalling transformers-4.40.1:
          Successfully uninstalled transformers-4.40.1
    Successfully installed tokenizers-0.13.3 transformers-4.30.2
```

Fig 6. Installing Transformers API



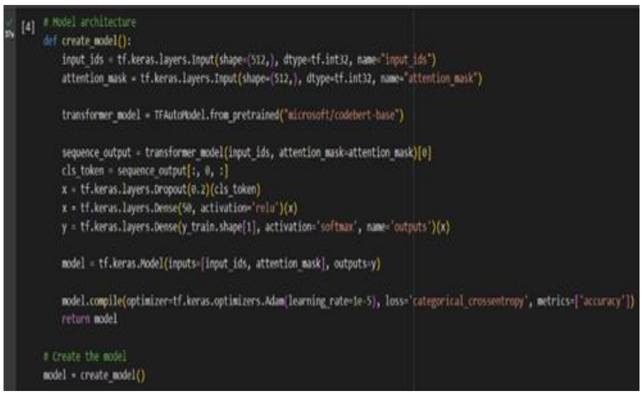
```
[4] from transformers import TFAutoModel, AutoTokenizer
    import pandas as pd
    import numpy as np
    from sklearn.model_selection import train_test_split, KFold
    from sklearn.preprocessing import MinMaxScaler
    from sklearn.metrics import classification_report, confusion_matrix

    # Setting seed for reproducibility
    SEED = 42
    tf.random.set_seed(SEED)
    np.random.seed(SEED)

    # Check if GPU is available
    if tf.test.gpu_device_name():
        print('Default GPU Device:{}'.format(tf.test.gpu_device_name()))
    else:
        print("Please install GPU version of TF")

    # Load dataset
    data = pd.read_excel('/content/CR-classification-ESEM23/dataset/labeled_dataset.xlsx')

    # Preprocess dataset
    X = data["message"].fillna(" ")
    y = pd.get_dummies(data['comment_group']).values

    X_clean = X.fillna("").astype(str)
```

Fig 7. Checking GPU availability and preprocessing dataset



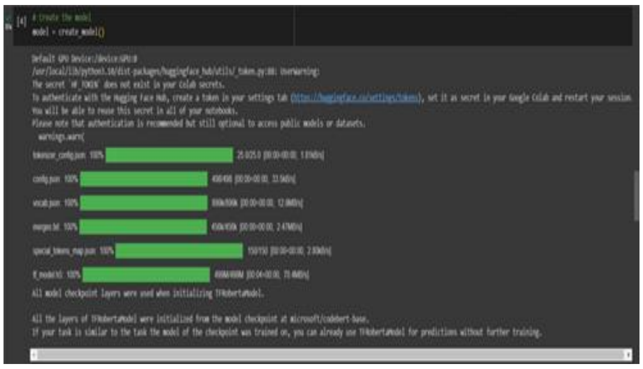
```
[4] # Tokenizer
    tokenizer = AutoTokenizer.from_pretrained("microsoft/codebert-base")

    # Tokenization function
    def tokenize_texts(texts):
        return tokenizer(texts, padding=True, truncation=True, return_tensors="tf", max_length=512)

    # Tokenize X
    X_tokenized = tokenize_texts(X_clean.tolist())

    # Assuming X_tokenized is the result of the tokenizer
    input_ids = X_tokenized['input_ids'].numpy()
    attention_mask = X_tokenized['attention_mask'].numpy()

    y = np.array(y)

    # Split each part
    input_ids_train, input_ids_test, attention_mask_train, attention_mask_test, y_train, y_test = train_test_split(
        input_ids, attention_mask, y, test_size=0.2, random_state=SEED
    )

    # Split the dataset
    # X_train, X_test, y_train, y_test = train_test_split(X_tokenized.data, y, test_size=0.2, random_state=SEED)
```

Fig 8. Tokenization



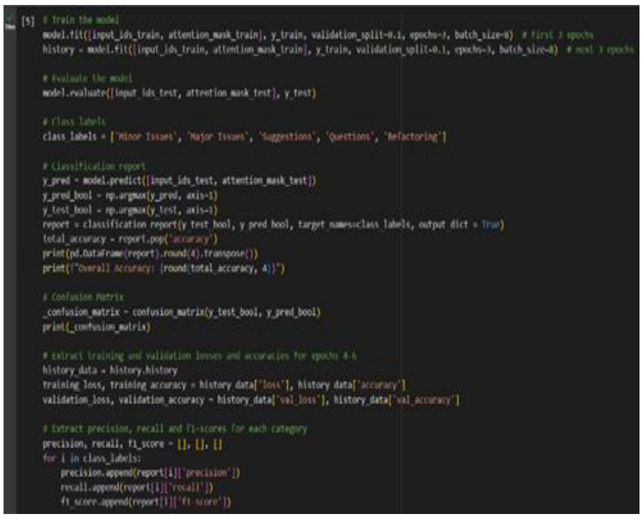
```
[4] # Model architecture
    def create_model():
        input_ids = tf.keras.layers.Input(shape=(512,), dtype=tf.int32, name="input_ids")
        attention_mask = tf.keras.layers.Input(shape=(512,), dtype=tf.int32, name="attention_mask")

        transformer_model = TFAutoModel.from_pretrained("microsoft/codebert-base")

        sequence_output = transformer_model(input_ids, attention_mask=attention_mask)[0]
        cls_token = sequence_output[:, 0, :]
        x = tf.keras.layers.Dropout(0.2)(cls_token)
        x = tf.keras.layers.Dense(50, activation='relu')(x)
        y = tf.keras.layers.Dense(y_train.shape[1], activation='softmax', name='outputs')(x)

        model = tf.keras.Model(inputs=[input_ids, attention_mask], outputs=y)

        model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5), loss='categorical_crossentropy', metrics=['accuracy'])
        return model

    # Create the model
    model = create_model()
```

Fig 9. Defining model architecture



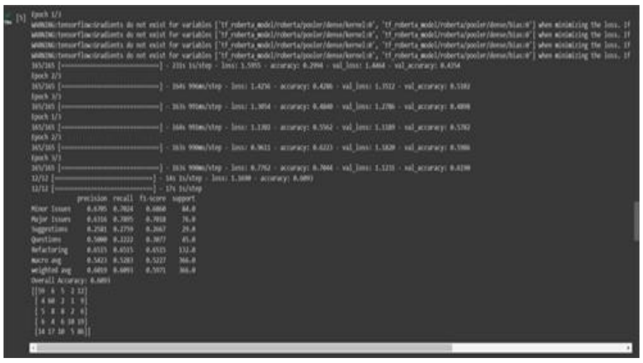
```
[4] # create the model
    model = create_model()

Default GPU Device:/device:GPU:0
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  25.0/25.0 [00:00<00:00, 1.83kB/s]
config.json: 100%  498/498 [00:00<00:00, 33.5kB/s]
vocab.json: 100%  899k/899k [00:00<00:00, 12.0MB/s]
merges.txt: 100%  456k/456k [00:00<00:00, 2.47MB/s]
special_tokens_map.json: 100%  150/150 [00:00<00:00, 2.80kB/s]
tf_model.h5: 100%  499M/499M [00:04<00:00, 73.4MB/s]
All model checkpoint layers were used when initializing TFRobertaModel.

All the layers of TFRobertaModel were initialized from the model checkpoint at microsoft/codebert-base.
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFRobertaModel for predictions without further training.
```

Fig 10. Creating CodeBERT-based classification model



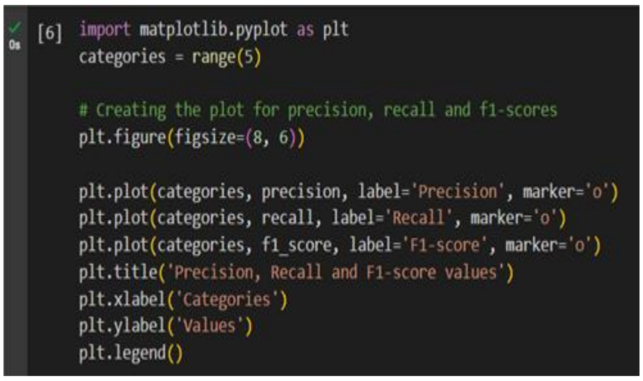
```
[5] # Train the model
    model.fit([input_ids_train, attention_mask_train], y_train, validation_split=0.1, epochs=3, batch_size=8)  # first 3 epochs
    history = model.fit([input_ids_train, attention_mask_train], y_train, validation_split=0.1, epochs=3, batch_size=8)  # next 3 epochs

    # Evaluate the model
    model.evaluate([input_ids_test, attention_mask_test], y_test)

    # Class labels
    class_labels = ['Minor Issues', 'Major Issues', 'Suggestions', 'Questions', 'Refactoring']

    # Classification report
    y_pred = model.predict([input_ids_test, attention_mask_test])
    y_pred_bool = np.argmax(y_pred, axis=1)
    y_test_bool = np.argmax(y_test, axis=1)
    report = classification_report(y_test_bool, y_pred_bool, target_names=class_labels, output_dict = True)
    total_accuracy = report.pop('accuracy')
    print(pd.DataFrame(report).round(4).transpose())
    print(f"Overall Accuracy: {round(total_accuracy, 4)}")

    # Confusion Matrix
    _confusion_matrix = confusion_matrix(y_test_bool, y_pred_bool)
    print(_confusion_matrix)

    # Extract training and validation losses and accuracies for epochs 4-6
    history_data = history.history
    training_loss, training_accuracy = history_data['loss'], history_data['accuracy']
    validation_loss, validation_accuracy = history_data['val_loss'], history_data['val_accuracy']

    # Extract precision, recall and f1-scores for each category
    precision, recall, f1_score = [], [], []
    for i in class_labels:
        precision.append(report[i]['precision'])
        recall.append(report[i]['recall'])
        f1_score.append(report[i]['f1-score'])
```

Fig 11. Training and evaluation of model and extracting results



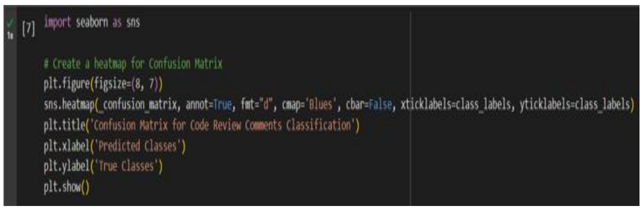
```
[5] Epoch 1/3
WARNING:tensorflow:Gradients do not exist for variables ['tf_roberta_model/roberta/pooler/dense/kernel:0', 'tf_roberta_model/roberta/pooler/dense/bias:0'] when minimizing the loss. If
WARNING:tensorflow:Gradients do not exist for variables ['tf_roberta_model/roberta/pooler/dense/kernel:0', 'tf_roberta_model/roberta/pooler/dense/bias:0'] when minimizing the loss. If
WARNING:tensorflow:Gradients do not exist for variables ['tf_roberta_model/roberta/pooler/dense/kernel:0', 'tf_roberta_model/roberta/pooler/dense/bias:0'] when minimizing the loss. If
WARNING:tensorflow:Gradients do not exist for variables ['tf_roberta_model/roberta/pooler/dense/kernel:0', 'tf_roberta_model/roberta/pooler/dense/bias:0'] when minimizing the loss. If
165/165 [==============================] - 231s 1s/step - loss: 1.5955 - accuracy: 0.2994 - val_loss: 1.4404 - val_accuracy: 0.4354
Epoch 2/3
165/165 [==============================] - 164s 996ms/step - loss: 1.4256 - accuracy: 0.4286 - val_loss: 1.3512 - val_accuracy: 0.5102
Epoch 3/3
165/165 [==============================] - 163s 991ms/step - loss: 1.3054 - accuracy: 0.4840 - val_loss: 1.2706 - val_accuracy: 0.4898
Epoch 1/3
165/165 [==============================] - 164s 991ms/step - loss: 1.1392 - accuracy: 0.5562 - val_loss: 1.1389 - val_accuracy: 0.5782
Epoch 2/3
165/165 [==============================] - 163s 990ms/step - loss: 0.9611 - accuracy: 0.6223 - val_loss: 1.1020 - val_accuracy: 0.5986
Epoch 3/3
165/165 [==============================] - 163s 990ms/step - loss: 0.7762 - accuracy: 0.7044 - val_loss: 1.1231 - val_accuracy: 0.6190
12/12 [==============================] - 14s 1s/step - loss: 1.1600 - accuracy: 0.6093
12/12 [==============================] - 17s 1s/step
              precision  recall  f1-score  support
Minor Issues     0.6705  0.7024    0.6860     84.0
Major Issues     0.6316  0.7895    0.7018     76.0
Suggestions      0.2581  0.2759    0.2667     29.0
Questions        0.5000  0.2222    0.3077     45.0
Refactoring      0.6515  0.6515    0.6515    132.0
macro avg        0.5423  0.5283    0.5227    366.0
weighted avg     0.6059  0.6093    0.5971    366.0
Overall Accuracy: 0.6093
[[59  6  5  2 12]
 [ 4 60  2  1  9]
 [ 5  8  8  2  6]
 [ 6  4  6 10 19]
 [14 17 10  5 86]]
```

Fig 12. Displaying results for model training and evaluation



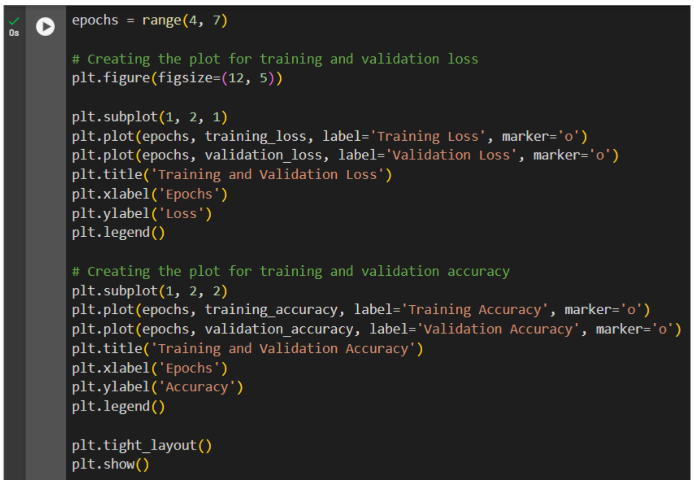
```
[6] import matplotlib.pyplot as plt
    categories = range(5)

    # Creating the plot for precision, recall and f1-scores
    plt.figure(figsize=(8, 6))

    plt.plot(categories, precision, label='Precision', marker='o')
    plt.plot(categories, recall, label='Recall', marker='o')
    plt.plot(categories, f1_score, label='F1-score', marker='o')
    plt.title('Precision, Recall and F1-score values')
    plt.xlabel('Categories')
    plt.ylabel('Values')
    plt.legend()
```

Fig 13. Plotting line graphs for Precision, Recall and F1-scores



```
[7] import seaborn as sns

    # Create a heatmap for Confusion Matrix
    plt.figure(figsize=(8, 7))
    sns.heatmap(_confusion_matrix, annot=True, fmt="d", cmap='Blues', cbar=False, xticklabels=class_labels, yticklabels=class_labels)
    plt.title('Confusion Matrix for Code Review Comments Classification')
    plt.xlabel('Predicted Classes')
    plt.ylabel('True Classes')
    plt.show()
```

Fig 14. Creating heatmap for Confusion matrix



```
epochs = range(4, 7)

# Creating the plot for training and validation loss
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(epochs, training_loss, label='Training Loss', marker='o')
plt.plot(epochs, validation_loss, label='Validation Loss', marker='o')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Creating the plot for training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(epochs, training_accuracy, label='Training Accuracy', marker='o')
plt.plot(epochs, validation_accuracy, label='Validation Accuracy', marker='o')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```

Fig 15. Plotting line graphs for Training and Validation metrics (loss and accuracy)

7.2. Examples of additional results derived from code execution

7.2.1. Execution No. 1

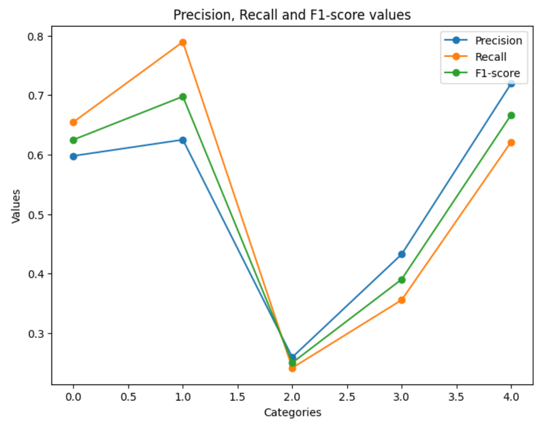


Fig 16. Precision, Recall and F1-scores for Execution no. 1

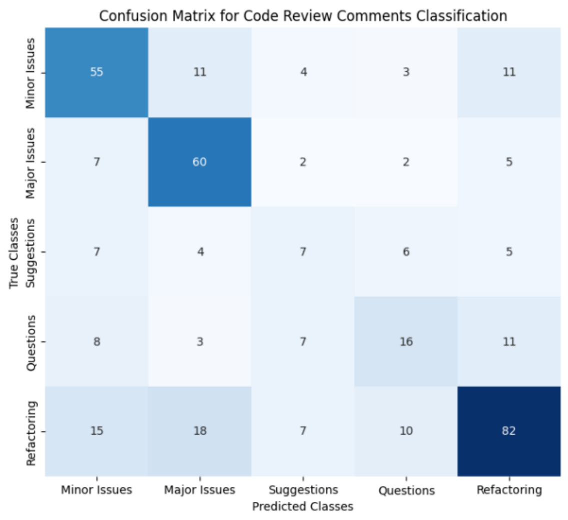


Fig 17. Confusion matrix for Execution no. 1

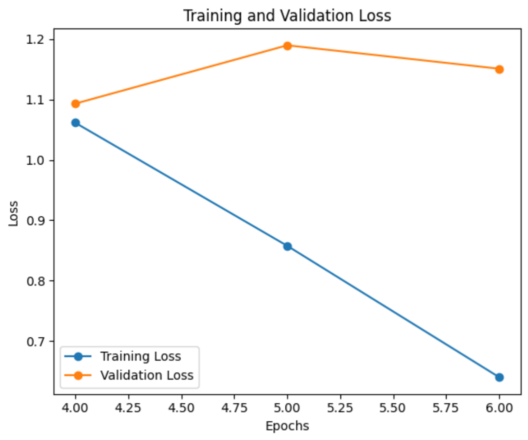


Fig 18a. Training and Validation loss for Execution no. 1

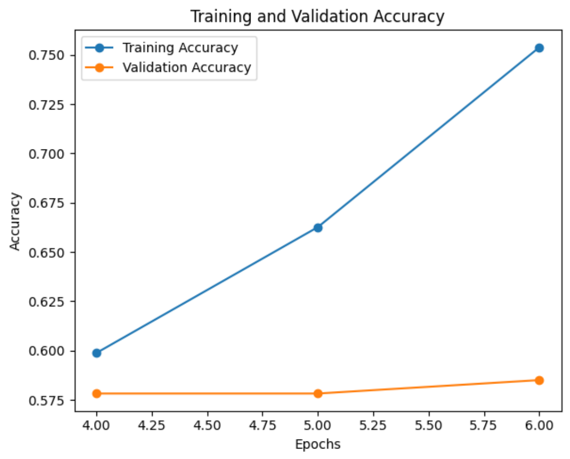


Fig 18b. Training and Validation accuracy for Execution no. 1

Table 2. Loss and Accuracy values for Execution no. 1

		Ep. 4	Ep. 5	Ep. 6	Overall
Loss	Training	1.0614	0.8577	0.6402	**1.211**
	Validation	1.0928	1.1894	1.1506	
Accuracy	Training	59.88 %	66.26 %	75.38 %	**60.11 %**
	Validation	57.82 %	57.82 %	58.50 %	

7.2.2. Execution No. 2

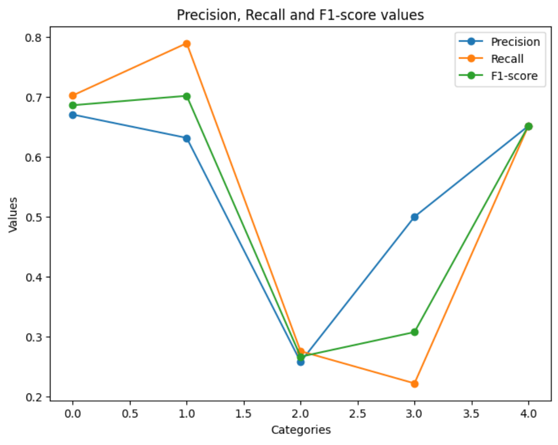


Fig 19. Precision, Recall and F1-scores for Execution no. 2

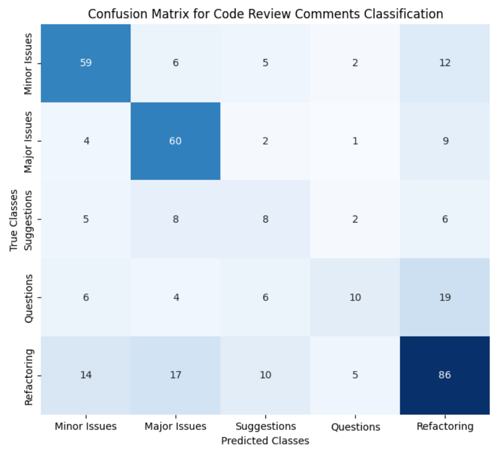


Fig 20. Confusion matrix for Execution no. 2

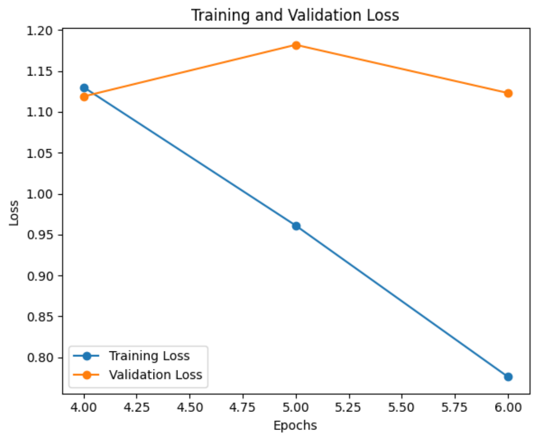


Fig 21a. Training and Validation loss for Execution no. 2

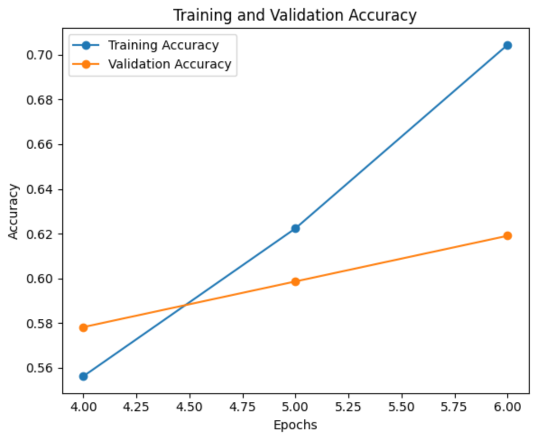


Fig 21b. Training and Validation accuracy for Execution no. 2

Table 3. Loss and Accuracy values for Execution no. 2

		Ep. 4	**Ep. 5**	**Ep. 6**	**Overall**
Loss	Training	1.1302	0.9611	0.7762	**1.169**
	Validation	1.1189	1.1820	1.1231	
Accuracy	Training	55.62 %	62.23 %	70.44 %	**60.93 %**
	Validation	57.82 %	59.86 %	61.9 %	

Acknowledgements

We thank our colleagues from the Department of Computer Engineering, Bharati Vidyapeeth (Deemed to be University) College of Engineering who provided expertise and insight that greatly assisted the research.

Author contributions

Gobind Panditrao: Conceptualization, Methodology, Software, Validation, Visualization, Investigation, Field study, Data curation, Execution, Writing-Original draft preparation, Reviewing and Editing. **Shashank Joshi:** Conceptualization, Writing-Original draft preparation, Validation. **Sunita Dhotre:** Visualization, Investigation, Writing-Reviewing and Editing. **Sandeep Vanjale:** Reviewing and Editing

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, “Towards automating code review activities,” in *2021 IEEE/ACM 43rd Int. Conf. on Software Engineering (ICSE)*, pp. 163–174, May 2021. doi:10.1109/icse43902.2021.00027.

[2] Y. Yin, Y. Zhao, Y. Sun, and C. Chen, “Automatic code review by learning the structure information of code graph,” *Sensors*, vol. 23, no. 5, p. 2551, Feb. 2023. doi:10.3390/s23052551.

[3] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, “An exploratory study on confusion in code reviews,” *Empirical Software Engineering*, vol. 26, no. 1, pp. 1–48, Jan. 2021. doi:10.1007/s10664-020-09909-5.

[4] Z. Feng *et al.*, “CodeBERT: A pre-trained model for programming and natural languages,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020. doi:10.18653/v1/2020.findings-emnlp.139.

[5] X. Zhou, D. Han, and D. Lo, “Assessing generalizability of CodeBERT,” in *2021 IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, pp. 425–436, Sep. 2021. doi:10.1109/icsme52107.2021.00044.

[6] Z. Li *et al.*, “Automating code review activities by large-scale pre-training,” in *Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, pp. 1035–1047, Nov. 2022. doi:10.1145/3540250.3549081.

[7] A. K. Turzo, “Towards improving code review effectiveness through task automation,” in *Proc. of the 37th IEEE/ACM Int. Conf. on Automated Software Engineering*, Oct. 2022. doi:10.1145/3551349.3559565.

[8] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, “Process Aspects and Social Dynamics of Contemporary Code Review: Insights from open source development and industrial practice at Microsoft,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 56–75, Jan. 2017. doi:10.1109/tse.2016.2576451.

[9] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, “Modern code review: A case study at Google,” in *Proc. of the 40th Int. Conf. on Software Engineering: Software Engineering in Practice*, pp. 181–190, May 2018. doi:10.1145/3183519.3183525.

[10] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of Modern Code Review,” in *2013 35th Int. Conf. on Software Engineering (ICSE)*, pp. 712–721, May 2013. doi:10.1109/icse.2013.6606617.

[11] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, “Helping developers help themselves: Automatic decomposition of Code Review changesets,” in *2015*

IEEE/ACM 37th IEEE Int. Conf. on Software Engineering, pp. 134–144, May 2015. doi:10.1109/icse.2015.35.

[12] E. Fregnan, F. Petrulio, L. Di Geronimo, and A. Bacchelli, “What happens in my code reviews? An investigation on automatically classifying review changes,” *Empirical Software Engineering*, vol. 27, no. 4, p. 89, Apr. 2022. doi:10.1007/s10664-021-10075-5.

[13] A. K. Turzo *et al.*, “Towards automated classification of code review feedback to support analytics,” in *2023 ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM)*, Oct. 2023. doi:10.1109/esem56168.2023.10304851.

[14] A. K. Turzo and A. Bosu, “What makes a code review useful to OpenDev developers? An empirical investigation,” *Empirical Software Engineering*, vol. 29, no. 1, p. 6, Nov. 2023. doi:10.1007/s10664-023-10411-x.

[15] J. R. Landis and G. G. Koch, “The measurement of observer agreement for categorical data,” *Biometrics*, vol. 33, no. 1, p. 159, Mar. 1977. doi:10.2307/2529310.

[16] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed, vol. 2. New York: Springer, 2009, pp. 1–758.

[17] Y. Zhang, D. Lo, X. Xia, and J. Sun, “An empirical study of classifier combination for cross-project defect prediction,” in *2015 IEEE 39th Annu. Computer Software and Applications Conf.*, vol. 2, pp. 264–269, Jul. 2015. doi:10.1109/compsac.2015.58.

[18] E. Doğan and E. Tüzün, “Towards a taxonomy of code review smells,” *Information and Software Technology*, vol. 142, p. 106737, Feb. 2022. doi:10.1016/j.infsof.2021.106737.

[19] T. J. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976. doi:10.1109/tse.1976.233837.

[20] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. doi:10.1162/neco.1997.9.8.1735.

[21] D. P. Kingma and J. Ba, “Adam: A method for Stochastic Optimization,” in *Proc. of the 3rd Int. Conf. for Learning Representations (ICLR 2015)*, Dec. 2014. doi:10.48550/arXiv.1412.6980.

[22] A. Tharwat, “Classification assessment methods,” *Applied Computing and Informatics*, vol. 17, no. 1, pp. 168–192, Jul. 2020. doi:10.1016/j.aci.2018.08.003.

[23] M. Sokolova and G. Lapalme, “A systematic analysis of performance measures for classification tasks,” *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, Jul. 2009. doi:10.1016/j.ipm.2009.03.002.

[24] D. M. Powers, “Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation,” *International Journal of Machine Learning Technology*, vol. 2, no. 1, pp. 37–63, Oct. 2020. doi:10.48550/arXiv.2010.16061.

[25] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, “Peer review on open-source software projects,” *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, pp. 1–33, Sep. 2014. doi:10.1145/2594458.

[26] J. Zhang *et al.*, “A novel neural source code representation based on Abstract Syntax Tree,” in *2019 IEEE/ACM 41st Int. Conf. on Software Engineering (ICSE)*, pp. 783–794, May 2019. doi:10.1109/icse.2019.00086.

[27] J. Bergstra and Y. Bengio, “Random Search for Hyper-Parameter Optimization,” *The Journal of Machine Learning Research*, vol. 13, pp. 281–305, Feb. 2012. doi:10.5555/2188385.2188395.

[28] M. Kukar and I. Kononenko, “Cost-sensitive learning with neural networks,” in *Proc. of the 13th European Conf. on Artificial Intelligence (ECAI 98)*, vol. 15, no. 27, pp. 88–94, Aug. 1998.

[29] K. Liu, G. Yang, X. Chen, and Y. Zhou, “EL-CodeBERT: Better exploiting CodeBERT to support source code-related classification tasks,” in *Proc. of the 13th Asia-Pacific Symp. on Internetware*, pp. 147–155, Jun. 2022. doi:10.1145/3545258.3545260.

[30] A. Vaswani *et al.*, “Attention is All you need,” in *Proc. of the 31st Int. Conf. on Neural Information Processing Systems (NIPS 2017)*, Jun. 2017. doi:10.48550/arXiv.1706.03762.

[31] H. He, Y. Bai, E. A. Garcia, and S. Li, “ADASYN: Adaptive Synthetic Sampling Approach for Imbalanced Learning,” in *2008 IEEE Int. Joint Conf. on Neural Networks (IEEE World Congress on Computational Intelligence)*, pp. 1322–1328, Jun. 2008. doi:10.1109/ijcnn.2008.4633969.