Towards Automated Classification of Code Review Feedback to Support Analytics

Asif Kamal Turzo [♡], Fahim Faysal [♣], Ovi Poddar [♣] Jaydeb Sarker [♡]
Anindya Iqbal [♣] Amiangshu Bosu [♡]
[♡]*Wayne State University, Detroit, Michigan, USA*
[♣]*Bangladesh University of Engineering and Technology, Dhaka, Bangladesh*
asifkamal@wayne.edu, *ffalsoy@gmail.com*, *ovi.poddar2@gmail.com*, *jaydebsarker@wayne.edu*,
anindya@cse.buet.ac.bd, *amiangshu.bosu@wayne.edu*

Abstract—*Background:* **As improving code review (CR) effectiveness is a priority for many software development organizations, projects have deployed CR analytics platforms to identify potential improvement areas. The number of issues identified, which is a crucial metric to measure CR effectiveness, can be misleading if all issues are placed in the same bin. Therefore, a finer-grained classification of issues identified during CRs can provide actionable insights to improve CR effectiveness. Although a recent work by Fregnan *et* al. proposed automated models to classify CR-induced changes, we have noticed two potential improvement areas – i) classifying comments that do not induce changes and ii) using deep neural networks (DNN) in conjunction with code context to improve performances.**

Aims: **This study aims to develop an automated CR comment classifier that leverages DNN models to achieve a more reliable performance than Fregnan *et* al.**

Method: **Using a manually labeled dataset of 1,828 CR comments, we trained and evaluated supervised learning-based DNN models leveraging code context, comment text, and a set of code metrics to classify CR comments into one of the five high-level categories proposed by Turzo and Bosu.**

Results: **Based on our 10-fold cross-validation-based evaluations of multiple combinations of tokenization approaches, we found a model using CodeBERT achieving the best accuracy of 59.3%. Our approach outperforms Fregnan *et* al.'s approach by achieving 18.7% higher accuracy.**

Conclusion: **In addition to facilitating improved CR analytics, our proposed model can be useful for developers in prioritizing code review feedback and selecting reviewers.**

Index Terms—**code review, review comment, classification, open source software, analytics, OSS**

I. Introduction

Peer Code Review (CR) is ubiquitous among contemporary software development pipelines. CR acts as a quality assurance gateway among popular Open Source Software (OSS) and commercial organizations such as Microsoft, Google, and Facebook [1], [2] [1], [3]. Since many projects have made CR mandatory [4], [5], developers spend, on average, 10-15% of their time (i.e., more than an hour each day) on CR tasks [2]. While the primary expectation behind CR adoption is finding defects, most CRs do not meet this expectation [6]. Recent studies found four out of the five CR comments regarding style and nitpicking issues [7]–[9] that can also be identified using static analysis tools. Therefore, some project managers often wonder if their CRs are cost-effective and worth practicing [6], [10].On the other hand, some project managers, although they consider CRs crucial due to other benefits such as knowledge dissemination [3], [6], they seek to improve CR effectiveness, as even a minor improvement can incur significant savings for large organizations such as Microsoft and Google [3], [7].

To identify potential improvement areas, many organizations have developed dashboards detailing various CR analytics such as review completion time, best reviewers, and the number of issues identified [11], [12]. Although the number of issues identified during CRs is an important metric to measure CR effectiveness, this metric, as measured by existing analytics tools [11], can be misleading due to the coarse-grained analysis of identified issues. For example, a CR comment identifying a critical functional defect is placed in the same bin as one suggesting nitpicking issues such as fixing typos. Therefore, the ratio of functional defects as opposed to nitpicking ones among the identified issues can not be measured. If CRs mostly identify nitpicking issues, yet bugs are identified post-CR, there may be shortcomings in the current CR process that current CR analytics would fail to reveal. An automated classifier is necessary to facilitate more informative CR analytics as manual classification of CR comments on a large scale is infeasible.

On this need, Fregnan *et* al. [13] developed an automated model to classify CR-induced changes into four categories. The results of their user evaluation show a need for such a CR analytics platform among the developers. However, we noticed two potential improvement areas over their work. First, as their model focuses on classifying review-induced changes, it fails to account for CR comments that do not trigger changes (e.g., discussion or changes deferred as future works), even though that feedback was beneficial. Therefore, analytics based on Fregnan *et* al.'s [13] model fail to allocate credits to authors of such reviews. Second, Fregnan *et* al. [13] uses only classical Machine Learning (ML) algorithms such as Naive Bayes, Decision Tree, and Random Forest in their evaluation. We aim to explore deep neural network (DNN)-based algorithms with CodeBert [14], as these combinations have shown superior performances for code classification tasks [15], [16]. Therefore, this study aims to develop an automated CR comment classifier that leverages DNN models to achieve a more reliable performance than Fregnan *et* al. [13]

On this goal, we have manually labeled a dataset of 1,828 CR comments following the CR comment classification scheme proposed by Turzo and Bosu [8]. Using this dataset, we trained and evaluated supervised-learning-based DNN models leveraging various code and feedback characteristics to classify each CR comment into one of the five high-level categories proposed by Turzo and Bosu [8]. We empirically evaluated multiple combinations of tokenization approaches to identify the best-performing model. In summary, we answer the following two research questions:

(RQ1) Can we use an automated machine learning-based approach to classifying code review comments?

Motivation: An automatic CR comment classification model would provide three benefits. First, an author can prioritize high-priority issues to prepare revisions. Second, based on prior CR analytics, an author may identify reviewers who are more likely to identify their priority issues and select reviewers accordingly. Finally, it can provide project managers with additional CR analytics to access individuals and overall process performances.

Method: We develop a machine learning-based classification approach that considers code context, CR comments, and a set of code attributes for classifying CR comments.

Results: The proposed approach can classify CR comments with an overall accuracy of 59.3%. Our evaluation also suggests that all the features (i.e., code context, comment text, and code attributes) improve model performance.

(RQ2) Can our review comment classification tool perform better than the current state-of-the-art?

Motivation: Review comment classification task is slightly different from change classification because some comments might not induce change. However, by retraining, the existing change classification approach of Fregnan et al. [13] can be used for classifying review comments. They considered code attributes only for classifying a change. Our hypothesis is *if we incorporate code context, review comment, and additional code attributes, we may achieve better performance to classify review comments*. RQ2 tests this hypothesis.

Method: We replicated the study of Fregnan *et* al. [13] on our dataset of 1,828 CR comments from the OpenDev Nova project. We used precision, recall, F1-score, and model accuracy to compare the performance of both approaches.

Results: Experimental results found that our proposed approach achieved an 18.7% performance improvement in model accuracy over Fregnan *et* al.'s approach in the review comment classification task.

The primary contributions of this study are the following:

- A novel machine learning approach for classifying CR comments.
- A detailed investigation by varying the input attributes and experimenting with the effect of different review comment tokenization and vectorization approaches on the model's performance.
- Empirically validated that the proposed approach performs better than the existing comparable automatic change classification approach of Fregnan *et* al. [13].
- We have made our code and dataset publicly available at: https://github.com/WSU-SEAL/CR-classification-ESEM23

The remainder of the paper is organized as the following. Section II discusses prior related works. Section III presents the data collection and manual labeling approach. Section IV details our model construction and evaluation approach and answers RQ1. Section V answers RQ2 by comparing the performance of our approach with the approach of Fregnan et al. [13]. Section VI discusses the insight gathered from this experiment. Section VII and VIII address the threats to validity and concludes the paper, respectively.

II. Related Works

The following subsections briefly describe prior research relevant to the goals of this study.

Code review effectiveness : Due to the widespread adoption of code review, recent years have seen increased research focus on this area. This mismatch between developers' expectations from CRs and outcomes is due to code comprehension challenges with limited time and context [6]. Their finding that approximately 80% CRs do not find bugs is also supported by several subsequent studies [7]–[11]. Although most CRs do not find bugs, developers still consider CR as an essential practice due to other benefits such as knowledge dissemination, improving project maintainability, and community building [2], [3], [17]. Studies investigating CR effectiveness have identified several technical and non-technical factors having positive associations, which include the reviewer's experience [7], [8], [18], the author's reputation [19], and project tenure [7]. On the other hand, factors with negative associations with CR effectiveness include missing rationale, discussion of non-functional requirements of the solution, and lack of familiarity with existing code [20], co-working frequency of a reviewer with the patch author [8], [21], description length of a patch [22], changeset size [1], the number of files under review [7], [8] and the level of agreement among the reviewers [23].

Code review automation: To help the reviewers understand the code better, several studies have focused on developing the changeset decomposition tool [24]–[27]. By decomposing the changeset, these studies aim at helping reviewers understand the code better so that reviewers can provide useful functional feedback. A large number of prior studies have developed tools for selecting appropriate reviewers [28]–[32]. By finding appropriate reviewers, those tools assist code authors in obtaining useful review comments. Existing reviewer recommendation systems consider history as a benchmark for measuring performance. But a recent study suggests that history can overestimate or underestimate the capability of a reviewer recommendation system [33]. Several recent studies have proposed the complete automation of the whole code review process [34]–[38]. With the recent advancement of transformer model [39], automation of code review activity can be achieved. But, such complete automation of code review activity would deprive developers of the benefits of code review, such as knowledge sharing among team members. We are motivated by prior studies that found the need to develop tools for automating different aspects of the code review process and developers' interests in CR analytics platforms [11]–[13].

Code review classification: Several studies have manually classified CR comments to identify the ratios of various categories of identified issues. Mäntylä and Lassenius first studied industrial and student CRs and found that only one-fourth of CRs affect code functionality [40]. They found 12 types of CR-identified issues, which they further categorized into three higher-level groups. Beller *et* al. manually classified 1,400 CR changes and found 10-22% of changes unrelated to CR comments [9]. They also proposed a taxonomy that categorized CR-induced changes into two higher-level categories. As some of the CR comments do not induce any change, prior studies also focused on classifying CR comments. Bosu *et* al. conducted a study on Microsoft where they identified 13 types of comments that appear in the code review process [7]. Turzo and Bosu [8] recently adopted Beller *et* al.'s change classification scheme to create a CR comment taxonomy consists of 17 subcategories grouped into five higher-level categories. Using a labeled dataset of 800 CR comments, Bosu *et* el. [7] trained an automated classifier to automatically identify useful CR comments. Building on their work, two later studies have developed classifiers with the same goal at different settings [11], [41]. While earlier studies focused on a binary classification to predict whether a CR comment was useful, Fregnan *et* al. [13] is the first to propose an automated model to classify review-induced changes into four categories. As Fregnan *et* al.'s [13] model focuses on classifying review-induced changes, our work also differs from theirs by considering non-change inducing CR comments.

III. Dataset Preparation

This section details our project selection, data mining, CR comment classification rubric, and manual labeling approach. Figure 1 shows an overview of our dataset preparation steps.

A. Project Selection

For this study, we selected the OpenDev Nova project from the OpenDev community for the following reasons:

1) Developers in the OpenDev community practice tool-based CR [2], and OpenDev is one of the largest OSS communities with contributors from more than 700 organizations [1] worldwide.
2) OpenDev projects have been the subject of prior CR studies [8], [42], [43].

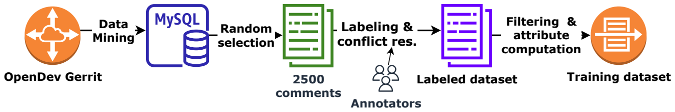


Fig. 1. An overview of our dataset preparations steps

B. Dataset Preparation

OpenDev community uses Gerrit[2] to manage CRs. We accessed the REST API provided by Gerrit to access and mine all the publicly available CRs from OpenDev's Gerrit instance[3]. Our dataset spans July 2011 to March 2022 and includes a total of 795,226 either 'Merged' or 'Abandoned' CRs. We stored the dataset in a local MySQL server.

C. Manual Labeling

We randomly selected 2,500 CR comments from the OpenDev Nova project for manual labeling using MySQL's `rand()` function. We only selected CR comments from OpenDev Nova as it has the highest number of posted CRs during our dataset span of July 2011 to March 2022. Prior studies have proposed several CR issues classification schemes [7]–[9], [11], [13], [40]. In this study, we select the classification scheme proposed by Turzo and Bosu [8], as their classification considers CR comments that did not induce any change, opposed to the one used by Fregnan *et* al. [13] Since we aim to classify CR comments, Turzo and Bosu's scheme is more suitable for our goal. Table I shows the 17 CR comment subcategories that are grouped into five higher-level categories based on this classification scheme. As building a reliable 17-category classifier is challenging, this study focuses on the five higher-level categories, which are: i) `Functional`, `Refactoring`, iii) `Documentation'` iv) `Discussion`, and v) `False Positive`.

Although our classifier requires labeling each CR comment into one of the five high-level categories, we manually labeled using the 17-category classification for future use cases of this dataset and fine-grained error analysis. Each of the CR comments was independently labeled by two annotators. During this process, they read the entire comment thread and its surrounding code context and induced changes among subsequent reviews (if any) to assign one of the 17 labels (Table I). We compared the assigned labels to identify conflicts. We computed Cohen's kappa (κ) [45] to measure the inter-rater reliability of this manual labeling process. The manual labeling achieved a κ value of 0.68 which indicates a substantial agreement[4]. To resolve the conflicts, a third annotator independently went through the conflicting ones to assign final labels. All three annotators are co-authors of this paper and have significant research experience in code reviews. After completing the manual labeling step, we assign

[1] https://openinfra.dev/annual-report/2022

[2] https://www.gerritcodereview.com/

[3] https://review.opendev.org

[4] Kappa (κ) value interpretation: 0.01–0.20 indicates 'none to slight', 0.21–0.40 indicates 'fair', 0.41– 0.60 indicates 'moderate', 0.61–0.80 indicates 'substantial', and 0.81–1.00 indicates 'almost perfect agreement' [46].

TABLE I

RUBRIC FOR CLASSIFYING CODE REVIEW COMMENTS WHICH WE ADOPTED FROM PRIOR STUDIES [7]–[9], [11], [40]

Group	Category	Description
Functional	Functional	Functional issues are defects where a code functionality is missing or implemented incorrectly. If requires large modification to resolve the issue.
	Logical	Logical issues are defects where there exist control flow problems or logical mistakes (wrong logic implementation, comparison issues).
	Validation	All types of user data sanitization issues or issues related to exception handling.
	Resource	Any kind of variable, memory, or file issues while handling or manipulating them.
	Timing	Any kind of synchronization issues while using the thread.
	Support issues	Any kind of support systems-related issues (e.g. configuration problem or version mismatch).
	Interface	Any types of interfacing issues such as issues in an import statement, issues while interacting with the database or internal system.
Refactoring	Solution approach	Review comments that suggest an alternative approach for problem-solving.
	Alternate Output	Review comments that address issues within the alert message, toast message, or error message.
	Code Organization	Code organization or refactoring issues presented in the catalog of Martin Fowler [44].
	Variable Naming	Review comments that address the violation of the variable naming convention.
	Visual Representation	Any kind of indentation, blank line, or code spacing-related issues.
Documentation	Documentation	Review comments that address issues related to code comments or documentation files for aiding code comprehension.
Discussion	Design discussion	Comments that discuss design pattern or sourcecode architecture.
	Question	If reviewers ask anything to the code author for clarification.
	Praise	Review comments that praise or complement the developer.
False positive	False positive	An invalid concern. A CR comment is considered 'False Positive' if (a) the code owner explicitly mentions the comment as an invalid concern or (b) no subsequent change occurred nor the code owner agrees to a future change [8].

the target label for each CR comment according to our schema from Table I . For example, if a category comment belongs to 'Design discussion', 'Question', or 'Praise', we assign the comment to the `Discussion` group. We found that 672 among the 2,500 CR comments were unrelated to source code files (e.g., configuration, build, resources, and commit message). As many of the code attributes selected for our classifiers (Table II) cannot be computed for non-source code files, we exclude those 672 CR comments at this step. After this exclusion, we use the remaining 1,828 CR comments to train and evaluate our classifiers. In this dataset, 8.64% comments belong to `False Positive`, 24.34% belong to `Discussion`, 32.71% belongs to `Refactoring`, 21.17% belongs to `Documentation`, and 13.13% comments belong to `Functional` group.

IV. (RQ1) MACHINE LEARNING MODEL FOR CLASSIFYING CODE REVIEW COMMENTS

To classify CR comments, we consider three types of input attributes for our machine-learning model. We consider the code context where the review comment was made, the CR comment in the form of natural language, and several code attributes. This section presents our attribute selection, model training approach, and results of our evaluations.

A. Attribute Selection

We first consider the code context where the review comment was made. Bosu *et* al. [7] found that a code change occurs within a limited proximity (+-10 lines) of code, which they define as the 'change trigger'. So, we consider the code context of +-10 lines from the occurrence of a CR comment and define that code segment as Review Comment Range (RCR). Code context can be a crucial feature as CR comments are made to pinpoint the existing code mistake. So, the code context can provide significant insight for classifying review comments. Several recent studies have also focused on utilizing source code context by extracting source code feature vector [14], [48]. We utilize the existing source code vectorization technique and use the code vector as a feature of the machine learning model. The second attribute that we consider is the review comment in the form of natural language. As our goal is to classify CR comments, comment text can be a crucial feature for the classifier. Hence, we also use comment texts as inputs to our models.

We additionally selected 27 code attributes. Those 27 code attributes were calculated from the *source* and the *destination* files. The *source* file is where the CR comment was made, and the *destination* file is the file that finally merged into the main codebase. If *destination* file is non-existent, no change occurred after the CR comment was made. We selected several Abstract Syntax tree-based (AST) based attributes, as recent studies have found that AST can better represent source code [49], [50]. Although AST difference-based code attributes were not explored by Fregnan *et* al. [13], those could be crucial to identify the type of recommendation made in a CR comment. For example, if a CR comment suggests no code change, only minor fixes such as updating documentation, there would be no difference between source and destination. If two ASTs have identical structures with different token names, their associated CR comment may be related to refactoring. Moreover, if two ASTs have the same number of tokens and nodes but differ in comparison operators, their CR comment may be related to a logical issue (i.e., functional). As machine learning features, we have incorporated nine AST-based attributes (e.g., anyDeleted, AnythingInLineMoved, getMovedSrcs). We also used 14 change-based attributes and four file-based attributes. The change-based and file-based attributes are selected considering our classification goal and the insight we gather from prior studies

TABLE II
CODE ATTRIBUTES SELECTION FOR MACHINE LEARNING MODELING

Attributes	**Description**	**Rationale**
AST-based attributes		
anyInserted	Count of AST nodes inserted in the *destination* file.	Some code changes involve the addition of a few AST nodes. For example, in case of variable value assignment or logical changes.
anyDeleted	Count of AST nodes deleted in the *destination* file.	Some code changes involve the deletion of a few AST nodes. In case of variable value assignment or logical changes, few AST node deletions might occur.
getMovedSrcs	How many AST nodes moved from the *source* file.	If the *source* code chunk is moved to the *destination*, it might indicate a refactoring change.
updatedSrcs	How many *sources* AST nodes are updated in the *destination* file?	Update of a few numbers of AST nodes may indicate logical changes.
anythingInLineMoved	Count of *source* AST nodes that were within the Review Comment Range (RCR) but moved elsewhere in the *destination* file.	Prior studies suggest most changes occur in close proximity to the review comment. The number of AST nodes moved within close proximity can provide valuable information about the change.
anythingInLineUpdated	Count of *source* AST nodes that were within the RCR and were updated in the *destination* file.	Similarly, the number of AST nodes updated within close proximity of the review comment can provide valuable information about the change.
anythingInLineDeleted	Count of *source* AST nodes that were within the RCR and were deleted.	Similarly, we argue that the number of AST nodes deleted within close proximity of the review comment can provide valuable information about the change.
anythingMovedIntoLine	If an AST node is in the RCR of the *source*, then the number of child nodes moved within that node in the *destination* file.	Number of nodes moved within RCR can provide crucial information for review comment classification, it can be indicative of refactoring changes.
anythingInsertedIntoLine	If an AST node is in the RCR of the *source*, then the number of child nodes that are newly inserted within that node in the *destination* file.	Number of nodes newly inserted within RCR can provide crucial information for review comment classification, it can be indicative of larger changes.
Change-based attributes		
insertedIfConditions	How many if statements in the *destination* file were inserted?	Insertion of ‘if’ condition indicates logical Functional change.
deletedIfConditions	How many if statements in the *source* file were deleted?	Similarly, deletion of the ‘if’ condition indicates logical Functional change.
elseInserted	How many else inserted in the *destination* file?	Insertion of ‘else’ condition indicates control flow change, which is a Functional change.
elseDeleted	How many else statements in the *source* file were deleted?	Similarly, deletion of the ‘else’ condition indicates logical Functional change.
entireLineMoved	How many full lines in the RCR were moved in the *destination* file?	Number of lines moved within close proximity can be valuable, for example, a large number of moved lines can indicate a structural rearrangement.
entireLineDeleted	How many full lines in the RCR were deleted from the *source* file?	Similarly, a large number of deleted lines can indicate a structural change or a larger change.
stringsUpdated	Number of strings in the *source* file that was updated.	Higher number of strings updated in the source file can indicate an ‘Alternate Output’ change.
magicStringsReplaced	Number of Magic strings replaced with variables.	Magic Strings are strings that are specified internally and impact the code behavior. Magic string replacement may indicate control flow-related change.
movedBlocksInIfConditions	Number of blocks within the if conditions were moved in the *destination* file.	Number of moved blocks within the ‘if’ condition indicates control flow-related issues within code.
insertedAssertConditions	Number of asserts inserted in *destination* file	As assert check for condition, insertion of assert condition might indicate functional change.
insertedTryCatch	Number of try-catch nodes inserted in the *destination* file.	Inserted try/catch statement may indicate Validation issues within code.
removedTryCatch	Number of try-catch nodes in the *source* file that was removed in the *destination* file.	Removed try/catch statement may indicate Validation issues or even a Structural change within the code.
updatedValueAssignments	Number of updated statements in the *destination* file where any variables’ value was updated.	Variables’ value update can indicate logical functional issues.
updatedFunctionArguments	Number of function arguments that were updated in the *destination* file.	Number of updated arguments within a function can provide valuable information about the change.
File-based attributes		
hasNewFile	A binary variable indicating *destination* file existence.	If 0, then it indicates no change occurred, so the comment might be false positive.
hasOldFile	A binary variable indicating *source* files’ existence.	Similarly to the previous one, this variable’s value can provide information about a comment.
cyclomaticComplexity	The Cyclomatic Complexity proposed by Mccabe [47] of the *source* file.	Cyclomatic complexity can provide vital information regarding the issue of the code or the types of review comments.
commentLOC	The line number of the *source* file where the review comment was made.	Comment line number can provide crucial information, such as if the commentLOC is small, then most probably the review comment is targeted for import statements.

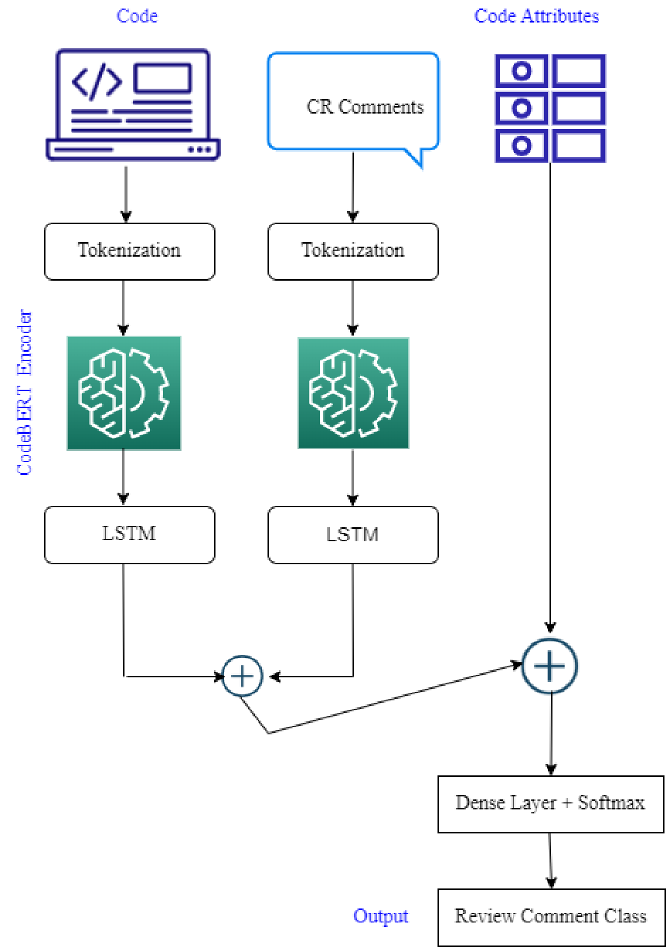


Fig. 2. Architecture of our proposed Machine Learning model. ⊕ indicates a concatenation unit in the image. CR Comments ⟶ Code Review Comments.

[13], [47], [51]. Table II presents the 27 selected attributes, a short description for each, and a brief rationale for selection.

B. Machine Learning Modeling

Figure 2 shows the architecture of our proposed model. The proposed machine-learning model inputs code, CR comments, and code attributes. Text data must be tokenized and converted into vectors to use as inputs for machine learning models. We tokenized the code and CR comments using CodeBERT tokenizer [14] separately. CodeBERT tokenizer provides *input_ids* and *attention_mask* for both source code and review comments. For implementing the CodeBERT tokenizer, we used transformers from Hugging Face [52].

For source code and review comment vectorization, we use the CodeBERT encoder [14]. We separately pass the tokenized source code and tokenized review comments to the CodeBERT encoder. CodeBERT is a hybrid pre-trained vectorizer that has been trained with both source code and natural language text. Also, CodeBERT achieved state-of-the-art performance for source code and natural language [14]. Since we use both source code and CR comments together, CodeBERT may be an excellent option for generating contextualized embedding vectors. After getting the embedding vector for code and CR comments, we pass it through separate LSTM layers.

Each embedded source code and review comment vectors are then separately passed through an LSTM [53] layer. The LSTM layer consists of 50 LSTM cells. The code output and comment output from both the LSTM layers are concatenated together. This concatenated vector is further concatenated with the calculated code attributes of Table II. The concatenated vector is then passed through a Dense layer with *softmax* activation function. The Dense layer produces the final class that a review comment belongs to. We used the Categorical Cross Entropy [54] as the loss function and used Adam optimizer [55] to optimize the loss function. For training the model, we used *batch size* 4 and *epoch* 8. As the code context and review comment generate a 512-dimensional vector, we had to choose a small batch size for training. We also implemented *EarlyStopping* method from the TensorFlow[5] library with 10% validation data. *EarlyStopping* was performed to avoid overfitting. We performed 10-fold cross-validation, and in each fold, 80% data were used for training, 10% were used for validation, and the remaining 10% were used for testing.

C. Evaluation Metrics

To assess the performance of the proposed model, we compute Precision, Recall, F1-Score, Matthew's Correlation Coefficient (MCC) [56], and overall Model Accuracy.

D. Experiments and Results

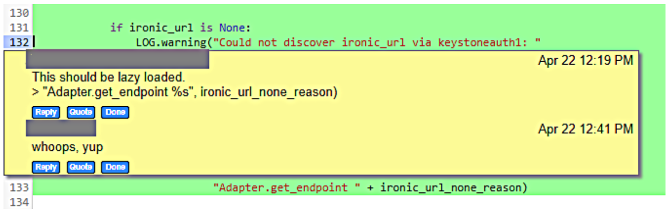
We conducted two types of experiments to evaluate the performance of our proposed approach. First, we experimented on the contribution of code context, CR comment, code features separately, and the contribution while the attributes are combined. We also experimented with different tokenization and embedding on the review comment. The result then leads us to perform an error analysis to understand the model's bias.

1) Effect of code context, review comment, and code attributes separately for classifying review comments: First, we experimented with the impact of each feature separately on the performance of the review comment classification task. Table III presents the result when we experimented with code context, CR comment, and code attributes separately. While Code context was used only, we achieved F1-Score of 0.232, 0.565, 0.019, 0.056, and 0.523 for the `Discussion`, `Documentation`, `False Positive`, `Functional`, and `Refactoring` classes, respectively. While the Review comment was used only, we achieved F1-Score of 0.637, 0.673, 0.257, 0.337, and 0.604 for the `Discussion`, `Documentation`, `False Positive`, `Functional`, and `Refactoring` classes respectively. While we used Code attributes only, we achieved F1-Score of 0.062, 0.056, 0.014, 0.093, and 0.263 for the `Discussion`, `Documentation`, `False Positive`, `Functional`, and `Refactoring` classes respectively. So,

[5] https://www.tensorflow.org/

TABLE III
PERFORMANCE OF OUR PROPOSED TOOL WHILE CODE, REVIEW COMMENT, AND CODE ATTRIBUTES ARE CONSIDERED SEPARATELY

Comment class	**Code context only**				**Review comment only**				**Code attributes only**			
	Precision	**Recall**	**F1-Score**	**MCC**	**Precision**	**Recall**	**F1-Score**	**MCC**	**Precision**	**Recall**	**F1-Score**	**MCC**
Discussion	0.326	0.208	0.232	0.396	0.586	0.706	0.637	0.543	0.049	0.121	0.062	0.183
Documentation	0.557	0.592	0.565	0.381	0.672	0.685	0.673	0.552	0.067	0.064	0.056	0.212
False Positive	0.033	0.013	0.019	0.414	0.441	0.194	0.257	0.571	0.009	0.036	0.014	0.235
Functional	0.176	0.055	0.056	0.414	0.416	0.327	0.337	0.568	0.068	0.297	0.093	0.224
Refactoring	0.415	0.719	0.523	0.283	0.602	0.624	0.604	0.520	0.255	0.452	0.263	0.145
Model Accuracy				0.418				0.579				0.230



```
        if ironic_url is None:
            LOG.warning("Could not discover ironic_url via keystoneauth1: "
```

[redacted] Apr 22 12:19 PM

This should be lazy loaded.
> "Adapter.get_endpoint %s", ironic_url_none_reason)

[redacted] Apr 22 12:41 PM

whoops, yup

```
                    "Adapter.get_endpoint " + ironic_url_none_reason)
```

Fig. 3. Example of a review comment where the reviewer presents a code line to pinpoint the issue better. This example has been taken from the OpenDev Nova project.

we can argue that Code context, Review comment, and Code attributes all contribute to the model performance.

2) Effect of different tokenization and vectorization on code review comment: CR comment is provided in natural language, so initially we selected pre-trained BERT [57] for review comment tokenization and vectorization task and CodeBERT [16] for the same for code context. We also experimented using CodeBERT as the $tokenizer+vectorizer$ for review comments as well. The processing of the other two input features, code and code attributes, are kept unchanged, i.e., code context features are always vectorized with CodeBERT. Table IV compares the results between two combinations, i) BERT for CR comments + CodeBERT for code context, ii) CodeBERT for CR Comments + CodeBERT for code context. Table IV suggests that with BERT for CR + CodeBERT for Code context combination, we achieved an overall model accuracy of 0.502. However, with CodeBERT for CR comments + CodeBET for code context combination, the model achieves an accuracy of 0.593, which is almost 10 percent higher than the other combination.

From Table IV, the result suggests that CodeBERT performs much better than the BERT for review comment tokenization and vectorization task. We were initially surprised by the findings and decided to manually inspect the CR comments. We selected 200 review comments randomly from our dataset for manual inspection. We found that reviewers present code chunks in many CR comments for pinpointing the issue to the code author. Figure 3 presents an example from the OpenDev Nova project. CR comments may contain natural language as well as programming language segments. Similarly, CodeBERT, is trained on both natural language and source code. Therefore, CodeBERT performs better than BERT for review comment tokenization and vectorization tasks.

3) Error Analysis: We analyzed the misclassifications qualitatively and quantitatively to better understand our model's performances. Table V shows the confusion matrix from 10-fold cross-validation for the best model. The confusion matrix shows how many samples of each class are correctly predicted and how many samples are classified into other classes. Since we took the classified instances for the confusion matrix from 10 folds, we combined all the samples for the test set and put their sums on the confusion matrix. The confusion matrix of Table V also provides us insights into the model's biases.

False Positive: According to the confusion matrix, out of 158 `False Positive` samples, 24 cases are accurately identified, whereas 43 samples are misclassified as `Discussion`, 29 samples are misclassified as `Documentation`, 36 samples are detected as `Refactoring`, and 26 samples as `Functional` class. So, our classifier is biased toward the other four classes in a similar ratio for the' `False Positive` comments. We manually analyzed those cases, and an example of a `False Positive` review comment that is classified as a `Functional`, "*A simple unit test would assert that we don't call service_get_all if enabled_services is passed in.*" This review comment addresses a Functional comment, but the concern was not valid for that circumstance. Therefore, our labelers marked it as a `False Positive`.

Discussion: We observe biases towards the `Refactoring` class while predicting the `Discussion` class comments. For example, "*We should make this a public method now yeah?*" belongs to the `Discussion` class but our model has misclassified it as a `Refactoring`.

Documentation: We found biases of the `Documentation` class towards the `Refactoring` class. For example, "*nit: add a TODO to remove this proxy and just have callers hit the necessary clients directly.*" is a `Documentation` comment that our classifier misclassified as a `Refactoring`.

Refactoring: From our evaluation, we found that many `Refactoring` samples are misclassified as a `Discussions`. For instance, "*Is anything else using this method now or can we remove it also?*" is misidentified in the `Discussion` class category.

Functional: In cases, where `Functional` samples were misclassified, they were more likely to be predicted as the `Refactoring` category. An example where a `Functional` comment is misclassified as a `Refactoring` comment, "*I would lower this too to be safe, but I guess it's not going to change.*"

TABLE IV
PERFORMANCE COMPARISON WHILE BERT AND CODEBERT ARE USED SEPARATELY FOR REVIEW COMMENT TOKENIZATION AND VECTORIZATION TASK

Comment class	BERT $(tokenizer + vectorizer)$				CodeBERT $(tokenizer + vectorizer)$			
	Precision	**Recall**	**F1-Score**	**MCC**	**Precision**	**Recall**	**F1-Score**	**MCC**
Discussion	0.516	0.570	0.522	0.449	**0.611**	**0.685**	**0.638**	**0.547**
Documentation	0.670	0.634	0.644	0.463	**0.704**	**0.762**	**0.723**	**0.556**
False Positive	0.124	0.041	0.056	0.487	**0.376**	**0.158**	**0.207**	**0.576**
Functional	0.380	0.116	0.162	0.483	**0.408**	**0.358**	**0.365**	**0.572**
Refactoring	0.461	0.632	0.523	0.409	**0.628**	**0.633**	**0.616**	**0.527**
Model Accuracy	0.502				**0.593**			

TABLE V
CONFUSION MATRIX FOR ERROR ANALYSIS

Predicted Ground truth	**Discussion**	**Documentation**	**False Positive**	**Functional**	**Refactoring**
Discussion	306	29	18	28	64
Documentation	32	292	3	8	52
False Positive	43	29	24	26	36
Functional	37	15	13	85	90
Refactoring	94	57	13	57	377

Key takeaway 1: *The proposed machine learning model can classify CR comments with 59.3% accuracy. The results of our evaluation also suggest that all three categories of input features (i.e., code attributes, code context, and comment text) contribute to improving model performance.*

Key takeaway 2: *Although review comments are provided in natural language, they often contain programming language lines and keywords. So, for the tokenization and vectorization of CR comments, we recommend using a hybrid model that is trained both in natural language and programming language, such as CodeBERT [14].*

V. (RQ2) COMPARISON AGAINST CURRENT STATE-OF-THE-ART FOR REVIEW COMMENT CLASSIFICATION TASK

This section presents the replication of Fregnan et al. [13] work and the performance comparison between the two approaches.

A. Replication

The work of Fregnan et al. [13] proposed models to classify review-induced changes into four categories. Although most CR comment categories trigger changes, some categories, such as false positive, discussion, or praise, do not. Even though our classifier has a slightly different goal, we can train a model using their approach and our dataset for CR comment classification. Therefore, we retrained Fregnan et al.'s [13] model to compare the performance of our approach against theirs. We followed their approach as closely as possible for replication. Despite the attributes of Fregnan *et* al. [13] being calculated for Java code, and our selected project being written in Python, we were able to calculate the same set of attributes

Since Fregnan *et* al. [13] provide adequate descriptions of the selected attributes and have made their replication package publicly available [58], we were able to write a script to extract their selected attributes for our dataset. We use our five-class category label for each CR comment as the dependent variable, which we computed in the data preparation step. Thus, we could utilize their approach in our context and obtain results as a five-class classification. As the Random Forest algorithm performed the best on classifying review-induced changes during their evaluation, we used this algorithm with 10-fold cross-validation for comparison.

B. Result Comparison

Table VI presents the result of the Fregnan *et* al. [13] approach and our proposed approach. The values presented in Bold font represent the best value between the two models. The results from Table VI indicate that for all the measures, our proposed model performs better than the approach of Fregnan *et* al. [13].

From Table VI, we can see that, for the `Discussion`, `Documentation`, `False Positive`, `Functional`, and `Refactoring` classes, Fregnan *et* al. [13] approach achieved an F1-Score of '0.362', '0.483', '0.068', '0.139', and '0.486' respectively. Whereas, for the `Discussion`, `Documentation`, `False Positive`, `Functional`, and `Refactoring` classes, we have achieved an F1-Score of '0.638', '0.723', '0.207', '0.365', and '0.616' respectively. Fregnan *et* al. [13] approach achieved an overall model accuracy of 0.406, whereas we achieved an overall model accuracy of 0.593. Our proposed approach has achieved a significant performance improvement for all five comment classes and overall model accuracy.

TABLE VI
COMPARISON BETWEEN OUR PROPOSED APPROACH WITH THE APPROACH OF FREGNAN *et* AL. [13]

Comment class	Fregnan *et* al.'s Approach				Proposed Approach			
	Precision	**Recall**	**F1-Score**	**MCC**	**Precision**	**Recall**	**F1-Score**	**MCC**
Discussion	0.355	0.373	0.362	0.368	**0.611**	**0.685**	**0.638**	**0.547**
Documentation	0.477	0.497	0.483	0.382	**0.704**	**0.762**	**0.723**	**0.556**
False Positive	0.283	0.040	0.068	0.410	**0.376**	**0.158**	**0.207**	**0.576**
Functional	0.297	0.093	0.139	0.408	**0.408**	**0.358**	**0.365**	**0.572**
Refactoring	0.415	0.592	0.486	0.307	**0.628**	**0.633**	**0.616**	**0.527**
Model Accuracy				0.406				**0.593**

Key takeaway 3: *Our proposed approach acheives better performance than Fregnan* et *al. [13] for the review comment classification task across all measures. Fregnan* et *al. [13] approach shows lower performance for the* False Positive *class. Although our proposed approach achieves a better performance than Fregnan et al. [13] for the* False Positive *class; still, the performance needs significant improvements. Further study is required for better analyzing and improving the detection of* False Positive *review comments.*

VI. IMPLICATIONS

This section presents the insights obtained from this study, potential future research directions, and recommendations for CR practitioners.

1. Improving classification performance: Our proposed model result shows that the model performs better for the Discussion, Documentation, Refactoring, and Functional classes than for the False Positive class. The approach of Fregnan et al. [13] shows an even worse performance for the False Positive class than our model. This result leads us to further investigate the reasons behind such poor showings for the False Positive class. We found that only 8.64% comments in our dataset belong to the False Positive class. This underrepresentation of the False positive in our sample may be a possible cause. While we did not explore class-balancing techniques such as oversampling, that may be a possible direction to improve performances. After examining false positive comments closely, we found that sometimes review comments pointing to issues belonging to other classes can become False Positive if refuted. For example, a reviewer has alluded to a Functional issue, but the concern is not valid for the current scenario. Therefore, while the same comment may have been Functional for another context, it is a False positive under the current scenario. This observation also shows the need to consider code context characteristics to classify CR comments. Moreover, the results of our error analysis also reveal cross miss-categorizations among the five classes. This analysis can also help identify additional features to distinguish between such pairs.

2. Selection of pre-trained vectorizers: From our experiment with two different combinations of $tokenizer + vectorizer$, we found that tokenization and vectorization can affect the performance of the review comment classifier. Code review comments contain both natural language and programming language tokens. So, a multipurpose model, such as CodeBERT [16] that is trained both in natural language and programming language performs better for review comment tokenization and classification task. Therefore, we recommend evaluating multipurpose pre-trained models in conjunction with general-purpose models for SE domain-specific NLP pipelines. *3. Recommendations for Practitioners:* To improve CR effectiveness, an organization needs to define a set of metrics and measure those to identify potential improvement areas [11]. Projects can leverage our model to track CR performances, such as counting the number of issues belonging to various categories, issue category-wise contributions from each reviewer for a project, and the ratios of defects escaping. These insights can help managers to identify potential improvement areas and adopt new initiatives [11]. Besides analytic supports, our model can also help authors prioritize CR comments. For example, a CR may receive ten comments from a reviewer, where eight are minor nitpicking issues, and the two remaining ones are critical defects. Using the code context and CR comment text, our model can automatically identify functional issues and help authors prioritize those accordingly. Authors can also leverage an analytic platform built on our model to identify reviewers who have more frequently identified the type of issues (e.g., Refactoring vs. Functional) that they are seeking for the current CR

4. Recommendations for researchers: Although our model improves the current state-of-the-art (SOTA) for CR comment classification by almost 20% in terms of accuracy, our best model has only 59.3% accuracy. While a five-category automated classification is more difficult than binary classifications, we believe there are significant improvement opportunities in this direction. Potential directions for improvements include training on larger-scale datasets, including balanced ratios of issues representing various categories and evaluating with oversampling techniques.

In addition, to improving this classifier, we believe our model can help build better review automation tools. As most of the CR comments belong to trivial issues, CR automation tools [34]–[38] trained on randomly curated datasets are more likely to amplify such issues and miss the rare yet more crucial ones. Future CR automation tools can leverage our model to build a more balanced sample, as manual labeling is time-consuming. Finally, existing history-based automated reviewer recommendation systems [28]–[31] consider all prior review interactions equally, which may not be the best approach. Our automated model can analyze reviewers' prior feedback history and provide higher priority to reviewers who previously provided more useful (e.g., `Functional`) feedback.

VII. Threats to Validity

The following subsections detail potential threats to the validity of this study and our countermeasures to mitigate those threats.

A. Internal Validity: To compare the performance of our approach with Fregnan *et* al. [13], we retrained their model on our dataset. A possible threat might appear during this retraining. To avoid any bias, we followed their approach as described in their paper. Although their code is publicly available, we could not directly use it because their attributes were calculated for the Java project, while our subject (i.e., OpenDev Nova) is written in Python. However, this threat may be minimal, as we used standard libraries to compute the attributes and followed their definitions. We also performed 10-fold cross-validation to mitigate biases introduced by using fixed training and testing samples.

B. Construct Validity: A significant threat to construct validity is the manual categorization process. For categorizing the code review comments, we performed a manual categorization. Using a rubric adopted from prior studies, we categorized each CR comment into one of the 17 categories. To mitigate the effect of human error, two authors did the categorization independently. We measured the agreement between the two annotators using Cohen's kappa [45] value. The manual labeling task achieved a kappa value of 0.68, which shows a 'Substantial' agreement. A third author then resolved the conflicts between the two authors' labeling. Therefore, this threat to validity remains minimal. Another potential threat to the construct validity is the features that were selected for the machine learning approach. Sometimes, used features might affect the performance of the model negatively. To mitigate the effect, we experimented separately with each type of feature we used in machine learning modeling. The result validates that each feature category contributes to the model's performance. Also, to mitigate the effect of improper tokenization and vectorization of review comments, we experimented with both BERT and CodeBERT.

C. External Validity: The main threat to external validity is the generalizability of the approach. Our approach requires manual labeling, which is extremely time-consuming. As a result, we could not incorporate multiple projects into this experiment. Even so, as characteristics vary from project to project, one cannot claim the generalizability of an approach with multiple projects. Regardless, this study has proposed an approach that can be utilized flexibly to develop project-specific models. For example, if, for some projects, the code context does not contribute to the overall performance, then the processing channel of code context can be discarded, and the remaining portion of our architecture can be utilized for CR comment classification tasks. To facilitate the replication of our work, we have made the dataset and the code publicly available.

D. Conclusion Validity First, selecting metrics to evaluate a model's performance can also threaten conclusion validity. We have used five different, widely used performance measuring metrics to mitigate this threat: precision, recall, F1-score, MCC, and model accuracy. Therefore, we do not anticipate any biases arising from our metrics. Second, machine learning models can show biased results when a fixed portion of data is selected for testing and training. To mitigate the effect of such bias, we performed a 10-fold cross-validation. Ten-fold cross-validation ensured that every portion of the dataset was utilized separately for the training and testing process. Finally, overfitting can be a significant threat to machine learning modeling. Overfitting occurs when a machine learning model performs well on training data but poorly on test data. We implemented the *EarlyStopping* method to avoid overfitting with 10% validation data. This method monitors the model performance on validation data and stops the training process once the convergence of the training process halts.

VIII. Conclusion

We have proposed a machine-learning approach to classify code review comments in this work. We collected 1,828 code review comments from the OpenDev Nova project and manually classified the code review comments using the prior review comment classification scheme. We experimented with different features that might be used for the review comment classification task and reported the contribution of those features separately. We then combined those features and developed a machine-learning model that uses those features for classifying code review comments.

Our proposed approach takes code context, code review comments, and a set of code attributes for classifying code review comments. We achieved an overall model accuracy of 59.3%. Experimental results suggest our proposed approach can classify code review comments better than the existing change classification approach proposed by Fregnan *et* al. [13]. We also experimented with how different tokenization and vectorization approaches can influence the performance of a review comment classifier. Finally, we presented the insight gathered by conducting this study, opportunities for researchers, and draw recommendations for practitioners.

Acknowledgement

Work conducted for this research is partially supported by the US National Science Foundation under Grant No. 1850475. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We thank Md. Toufikuzzaman for his assistance during the data mining process.

References

[1] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, "Peer review on open-source software projects: Parameters, statistical models, and theory," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–33, 2014.

[2] 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, 2016.

[3] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, pp. 181–190.

[4] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 202–212.

[5] P. Wurzel Gonçalves, G. Calikli, A. Serebrenik, and A. Bacchelli, "Competencies for code review," *Proceedings of the ACM on Human-Computer Interaction*, vol. 7, no. CSCW1, pp. 1–33, 2023.

[6] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.

[7] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at microsoft," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 146–156.

[8] A. K. Turzo and A. Bosu, "What makes a code review useful to opendev developers? an empirical investigation," *Empirical Software Engineering*, vol. 28, p. TBD, 2023.

[9] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 202–211.

[10] J. Czerwonka, M. Greiler, and J. Tilford, "Code reviews do not find bugs. how the current code review best practice slows us down," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 27–28.

[11] M. Hasan, A. Iqbal, M. R. U. Islam, A. I. Rahman, and A. Bosu, "Using a balanced scorecard to identify opportunities to improve code review effectiveness: An industrial experience report," *Empirical Software Engineering*, vol. 26, pp. 1–34, 2021.

[12] C. Bird, T. Carnahan, and M. Greiler, "Lessons learned from building and deploying a code review analytics platform," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 191–201.

[13] 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, 2022.

[14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[15] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 425–436.

[16] K. Liu, G. Yang, X. Chen, and Y. Zhou, "El-codebert: Better exploiting codebert to support source code-related classification tasks," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, 2022, pp. 147–155.

[17] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1028–1038.

[18] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1039–1050.

[19] A. Bosu and J. C. Carver, "Impact of developer reputation on code review outcomes in oss projects: An empirical investigation," in *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, 2014, pp. 1–10.

[20] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "An exploratory study on confusion in code reviews," *Empirical Software Engineering*, vol. 26, pp. 1–48, 2021.

[21] P. Thongtanunam and A. E. Hassan, "Review dynamics and their impact on software quality," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2698–2712, 2020.

[22] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Review participation in modern code review," *Empirical Software Engineering*, vol. 22, no. 2, pp. 768–817, 2017.

[23] T. Hirao, A. Ihara, Y. Ueda, P. Phannachitta, and K.-i. Matsumoto, "The impact of a low level of agreement among reviewers in a code review process," in *IFIP International Conference on Open Source Systems*. Springer, 2016, pp. 97–110.

[24] 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 International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 134–144.

[25] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 341–350.

[26] V. U. Gómez, S. Ducasse, and T. d'Hondt, "Visually characterizing source code changes," *Science of Computer Programming*, vol. 98, pp. 376–393, 2015.

[27] Y. Huang, N. Jia, X. Chen, K. Hong, and Z. Zheng, "Code review knowledge perception: Fusing multi-features for salient-class location," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1463–1479, 2020.

[28] M. B. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 530–543, 2015.

[29] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 141–150.

[30] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: code reviewer recommendation in github based on cross-project and technology experience," in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 222–231.

[31] P. Pandya and S. Tiwari, "Corms: a github and gerrit based hybrid code reviewer recommendation approach for modern code review," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 546–557.

[32] G. Rong, Y. Zhang, L. Yang, F. Zhang, H. Kuang, and H. Zhang, "Modeling review history for reviewer recommendation: a hypergraph approach," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1381–1392.

[33] I. X. Gauthier, M. Lamothe, G. Mussbacher, and S. McIntosh, "Is historical data an appropriate benchmark for reviewer recommendation systems?: A case study of the gerrit community," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 30–41.

[34] P. Thongtanunam, C. Pornprasit, and C. Tantithamthavorn, "Autotransform: automated code transformation to support modern code review process," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 237–248.

[35] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 163–174.

[36] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, and N. Sundaresan, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, 2022, p. 1035–1047.

[37] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2291–2302.

[38] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[40] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430–448, 2008.

[41] M. M. Rahman, C. K. Roy, and R. G. Kula, "Predicting usefulness of code review comments using textual features and developer experience," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 215–226.

[42] F. E. Zanaty, T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "An empirical study of design discussions in code review," in *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, 2018, pp. 1–10.

[43] T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "The review linkage graph for code review analytics: a recovery approach and empirical study," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 578–589.

[44] M. Fowler, "Refactoring catalog," *Refactoring Home Page, URL: http://www. refactoring. com/catalog/index. html (letzter Abruf: 09.02. 2006)*, 2012.

[45] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[46] J. R. Landis and G. G. Koch, "An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers," *Biometrics*, pp. 363–374, 1977.

[47] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[48] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[49] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[50] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 87–98.

[51] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[52] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6

[53] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[54] R. Y. Rubinstein and D. P. Kroese, *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning*. Springer, 2004, vol. 133.

[55] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization 3rd int," in *Conf. for Learning Representations, San*, 2014.

[56] Y. Reich and S. Barai, "Evaluating machine learning models for engineering problems," *Artificial Intelligence in Engineering*, vol. 13, no. 3, pp. 257–272, 1999.

[57] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of naacL-HLT*, vol. 1, 2019, p. 2.

[58] E. Fregnan, F. Petrulio, L. D. Geronimo, and A. Bacchelli, "What happens in my code reviews? - Replication package," Oct. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5592254