On the Classification of Refactoring Code Reviews

Josef Sieber*, Tim Schwirtlich*, John Melwin Richard John Paul Raj*, Yeshwant Santhanakrishnan Premanand*, Eman Abdullah AlOmar[†], AbdElRahman Ahmed ElSaid[‡], Mohamed Wiem Mkaouer[§]
* Rochester Institute of Technology, NY, USA
[†] Stevens Institute of Technology, NJ, USA
[‡] University of North Carolina Wilmington, NC, USA
[§] University of Michigan-Flint, MI, USA

Abstract—**Code review has become standard practice in both business and open source projects with the objective of improving software quality, promoting knowledge exchange, and ensuring adherence to coding rules and guidelines. It involves developer discussions on potential refactoring operations before incorporating changes into the code base. Despite extensive investigations into the general challenges, best practices, and socio-technical elements of code review, there is limited understanding about the assessment of refactoring and what developers prioritize when examining refactored code. Therefore, our study aims to comprehend the primary factors developers consider when reviewing refactored code. To do so, we design a model that takes as input a given code review conversation, and classifies it into one refactoring category, e.g., *quality*, *integration*, *testing*, etc. Multiple machine learning approaches are used and evaluated, and the experimental results are examined qualitatively and quantitatively at various granularities. Refactoring-related code review category prediction is found to be possible with high consistency, achieving an average scores of 0.8 on accepted universal classification metrics, through the utilization of contemporary and traditional machine learning approaches. These findings suggest that refactoring code review categorization can be embedded in traditional workflows to increase efficiency and decrease the time to production in software development.**

Index Terms—**code review, refactoring, quality**

I. Introduction

Code reviews are essential for the modern software development lifecycle. They are necessary to ensure that modified code fulfills claims for correctness, integrity and compliance with standards [1]–[6]. Software engineers perform code reviews based on the modifications in a pull request made by their peers. Reviewers evaluate not only whether the committed code performs the task that it claims to, but also how it interacts with the larger system as a whole. Bacchelli et al. [7] performed an observational review of the practice via survey, and determined that the outcome of this process was the creation of awareness of system interactions along with gaining more understanding of the code base as a whole. This was confirmed by Macleod et al. [8], who performed an observational study and interviewed multiple teams at Microsoft, who concluded that code reviews serve the purpose of finding defects, improving code, and increasing knowledge transfer.

Although code reviews are a necessary part of the software development life cycle, refactoring-related code reviews take more time than other types of code reviews, and are linked to longer times to deployment. AlOmar et al. [9] found in their review of a large, active open-source project that refactoring-related reviews had statistically significant differences in code review metrics. These code reviews, on average, involved more reviewers, more review comments, more inline comments, more revisions, more file changes, lengthier review time, lengthier discussions, lengthier descriptions, and more added or deleted lines between revisions when compared to non-refactoring-related code reviews.

As part of the examination of code reviews related to refactoring, AlOmar et al. [9] came up with six overarching review criteria that reviewers use for reviews related to refactoring. These categories were manually derived from a qualitative analysis of review discussions tracked in the code review tool. The results were verified in an industrial setting. The categories are summarized in Table I.

As an illustrative example, here is a statement extracted from a review where the reviewers point out the need to refactor a given patch, as there are two classes (base and common) that contain common instructions. According to the refactoring taxonomy, this review belongs to the *Code smell* subcategory, under the *Quality* category.

> I do understand the desire to **refactor** some code to **eliminate duplicate code**. The purpose of the common class was to contain all of the duplicate code between the 2 drivers. This seems like a half-baked approach to refactoring to accomplish that goal, when the common class should have been used as a new base (...).

Given how the review of refactoring-related changes takes more effort and other types of changes [9], if developers are aware of what category of refactoring their changes belong to (i.e., Table I), they can properly document their changes to facilitate their review, yet, there is no automated technique that automatically classifies them. Therefore, the goal of this paper is to develop an automated method to classify to a given refactoring category, the refactoring-related code change descriptions for review (referred to in the remainder of the paper as *refactoring reviews*) based on their subject and description. The predicted category shall indicate the criteria that will be the focus of the discussion between the reviewer and the developer when clarifying the correctness, quality, and purpose of the code proposed code change. We hypothesize that such a classification of reviews reveals which frequent criteria are the

TABLE I: Refactoring Review Categories, as defined by AlOmar et al. [9]

Category	Description
Quality	Reviewers enforce the adherence to coding convention, optimization of internal quality attributes, external quality attributes, the avoidance of (i.e., code smell, resolution of technical debt, correctness of design pattern implementations), and lack of documentation
Refactoring	Reviews discuss refactoring correctness, behavior preservation, refactoring co-occurrence with changes, and domain constraints
Objective	Reviewers eventually ask to clearly document the goal, benefit, side effects, scope, feature-related, and bug fix-related activities to better understand the rationale of the submitted code changes
Testing	Refactoring reviews in which current testing did not adequately reflect the changed behavior after refactoring.
Integration	Reviews highlighting how refactoring has complicated the merging process, or triggered configuration issues
Management	Review took longer than normal due to a lack of reviewer attention, with little prompt discussion about proposed changes

focus of developers when reviewing refactorings. Additionally, those indications could also be useful for developers in order to preemptively examine their own code during the review to fix critiques their peers may have. These two benefits together should speed code review and help the smoothing of refactoring reviews.

We tackle the problem of refactoring-related reviews, as a multi-label classification problem, since reviews may belong to more than one category, as we will detail later in Section II-A. For embedding our input, we leverage fastText algorithm [10] which is a text classification and representation learning machine learning algorithm developed by Meta. It is an extension of the traditional word2vec algorithm that represents text documents using character n-grams (sub-words) as well as word embeddings [11]. For training, the algorithm also utilizes hierarchical softmax and negative sampling, making it computationally efficient and scalable to large datasets. The fastText algorithm can be used in a refactor-aware code review process to automatically classify and group code changes based on their underlying functionality or purpose.

Our experiments have designed multiple combinations of models, leveraging mainly FastText as the main pre-trained word vectorizer for the input descriptions, and Gradient Boosted Machines as the main classifier. Our approach achieved an efficient classification performance, reaching an F-Measure of 80%, despite the class imbalance that exists between the categories.

Our model is publicly available for researchers and practitioners to replicate and extend[1], and it has been designed and implemented to be easily exported and deployed as a standalone plugin for GitHub. Although recent advances in natural language processing (NLP) and deep learning (DL) have been using large language models, these solutions remain limited in their input window [12], and newer models remain less cost-effective for extensive usage [13].

II. Study Design

We now describe the experiments we conducted to analyze the classification task of the refactoring review categories and their feasibility.

A. Data

The data set consists of 3837 non-unique samples of refactoring reviews, each having subject, description, and a singular associated assigned category. This was reduced to 3579 values after the removal of 258 duplicate samples. Cleaning the data further required removing the subject from the description column, where it was duplicated. The breakdown of label counts is visualized in Figure 2.

[1]https://github.com/smilevo/RefactoringReviewClassification

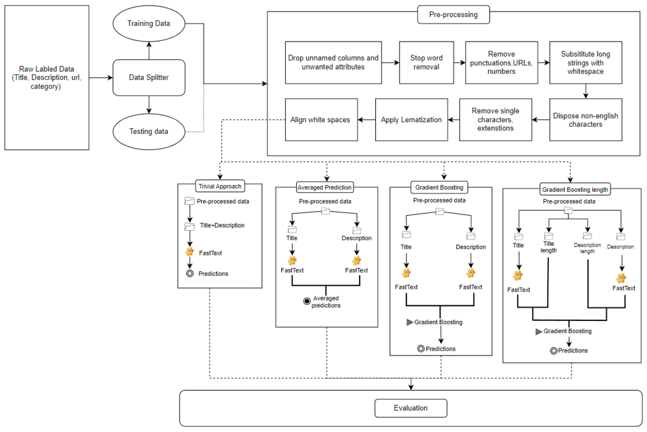

Fig. 1: Approach Overview

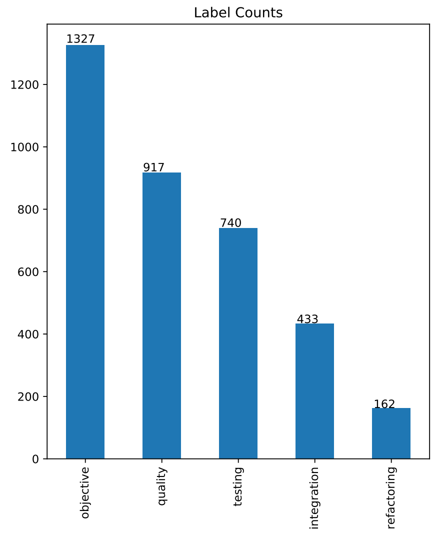

Fig. 2: Number of Reviews per Category

B. Preprocessing

Preprocessing is a crucial step in Natural Language Processing (NLP) tasks. Raw text data contains noise and irrelevant information, making it challenging to extract meaningful insights. Preprocessing involves applying a series of techniques to clean and transform the raw text data into a format that machine learning algorithms can analyze.

Preprocessing is necessary in NLP tasks for several reasons. First, it helps reduce the data's dimensionality by removing stop words and non-relevant information. This, in turn, makes it easier for machine learning algorithms to identify patterns and extract meaningful features from the data.

Secondly, preprocessing helps to standardize the text data by converting all characters to lowercase, removing punctuation, and other non-alphabetic characters. This standardization ensures that the machine learning algorithms do not treat similar words differently, which could result in inaccuracy.

Thirdly, preprocessing techniques such as stemming and lemmatization help to reduce words to their base form. This can help to group together words that have the same root but different suffixes, which can improve the accuracy of the machine learning models.

The code review natural language text fields required preprocessing to be suitable for classification tasks. Although the fastText algorithm is designed to accommodate generally unprocessed data, it is advised by the algorithm developers to still perform preprocessing on the input data [14]. For the acquired data in this classification task, common preprocessing steps referenced above were applied. Numbers, non-english words, web addresses, and single characters were removed. Casing was standardized across the input set. Description-less code review requests were modified to have a description of ”No Description”. Finally, the words were lemmatized using the WordNet Lemmatizer from NLTK, a leading platform for natural language processing [15].

C. Classification Model

The fastText algorithm python API was utilized [10]. The model was trained with a ”one-vs-all” loss function, creating independent binary classifiers for all labels. For direct prediction, each label was predicted by each binary model, resulting in a probability in [0,1] for each label. The learning rate was scaled lower, to 0.05, as suggested in the official documentation [16].

Given that each code review contains a *subject* text and a *description* text of various lengths, it is important to calibrate the model to give them appropriate weights. To do so, the outputs of fastText models trained on the subject and description separately are concatenated and used to train gradient boosted machines for binary prediction of each category. To examine whether the added fields of *subject* and *description* length positively contribute to model performance, the length of the subject and description fields are engineered into variables and concatenated with the input fields for gradient boosted machines.

D. Modeling Approaches

As shown in Figure 1, from the different combinations of approaches and models, we arrive at 3 final models alongside the trivial approach, for the classification task that we evaluate. For the trivial approach as a baseline comparison, we will consider the fastText algorithm trained on the concatenated subject and description in the data, as it is a proven standard for classification tasks utilizing natural language. This will be referred to as the trivial model moving forward. We compare this approach to the averaged prediction of two fastText models built separately on the subject and description, referred to as the averaged model moving forward, the prediction of a gradient boosted machine built off the outputs of the two fastText models built separately on the subject and description, referred to as the GBM model moving forward, and finally to the prediction of a gradient boosted machine built off the outputs of the two fastText model previously mentioned along with added variables for the length of the subject and description, referred to as the GBM-Length model moving forward. When referenced together, the GBM and GBM-Length models will be referred to as GBM variants.

E. Model Workflow

a) fastText Algorithm

The FastText algorithm is based on the bag-of-words approach, which represents each text sample as a bag of individual words. The bag of words approach works by first tokenizing the input text into individual words or tokens, which is in this case, done prior to modeling in the pre-processing steps. Then, it creates a dictionary of all unique words that appear in the text corpus. This dictionary is used as the basis for feature extraction. Next, the approach creates a feature vector for each piece of text by counting the occurrences of each word in the dictionary. The resulting feature vector represents the frequency of each word in the text and can be used as input to a machine learning algorithm. In addition, the fastText algorithm also uses subword information to capture more abstract similarities between words. The algorithm learns a set of n-gram character embeddings for all possible n-gram lengths up to a maximum length k, which in this case is 1. These embeddings are then used to represent the subwords of words in the input text.

During training, the FastText algorithm learns a logistic regression classifier using the word and subword embeddings as input features, learning weights for each word and subword individually. The classifier is trained using stochastic gradient descent (SGD) with the negative log-likelihood loss function. The log-likelihood is the logarithm of the likelihood function, which is the probability of observing the data given the model parameters. For optimization purposes in machine learning, this is made negative to find a minima rather than a maxima.

Hierarchical softmax, based on Huffman Coding, is utilized for label prediction. This algorithm represents the output labels as a binary tree, such that each internal node corresponds to a binary decision, and each leaf node corresponds to a specific class. The probability of a particular class is computed by traversing the tree from the root to the corresponding leaf node, using the binary decisions at each internal node to determine the path to follow. The hierarchical softmax reduces the computational complexity of computing the output layer of the logistic regression, by reducing the number of computations required to compute the probability distribution over the output classes. This is particularly useful in cases where the number of output classes is very large.

b) Gradient Boosted Machine

Gradient Boosted Machines (GBMs) work by iteratively improving an initial weak learner model by minimizing a loss function with respect to the model parameters using gradient descent optimization.

The weak learner used in the GBMs for this task is a decision tree. In the first iteration of a GBM, a single decision tree is trained on the training data. In each subsequent iteration, a new decision tree is trained on the residuals of the previous iteration's predictions, with the goal of improving the overall model performance. We chose to utilize 100 estimators per GBM model.

The residual is the difference between the actual target value and the predicted value of the previous iteration's model. By training a new model on the residuals from the previous iteration, GBMs focus on the data points that were not correctly predicted in the previous iteration, attempting to correct the errors of the previous model.

To prevent overfitting, GBMs use regularization parameters. The maximum depth of the decision trees was set at 2 after experimentation, and the learning rate was set at 0.1, as recommended by the official documentation. The regularization parameters control the contribution of each new tree to the overall model. The learning rate determines how much each new tree improves the previous model, and the maximum depth controls the complexity of the decision trees.

The loss function utilized in the model selected was the log-likelihood loss function, referenced in II-E0a.

The modeling of a GBM based on the output of the fastText algorithm is an instance of stacking. Stacking is an ensemble learning technique that combines the predictions of multiple machine learning models to improve the overall accuracy of the final prediction. In stacking, a meta-model is trained on the outputs of several base models, rather than on the original features.The stacking approach can help to improve the accuracy of the final prediction by combining the strengths of several models. By using a diverse set of base models, the stacking approach can also help to reduce the risk of overfitting, as each base model may capture different aspects of the data. The stacking performed in this instance aims to learn the weights of the subject and description fields separately through the modeling.

F. Performance Metrics

We choose to utilize the metrics utilized by Herbold et al. [17] in their fastText algorithm based research, with a modification to suit our multi-label task. We measure the performance of our classifiers on the overall dataset with *precision*, *recall*, and F_1 *score*.

To adjust for multi-label performance, true positives are defined as all correctly identified labels across all categories, rather than in only one binary category. False positives are defined as all labels that are incorrectly applied through prediction across all categories. Precision in this context measures the percentage of labels that were correct among all labels predicted. Recall in this context measures the percentage of correctly predicted labels out of all true labels. F_1 score is the harmonic mean of recall and precision. Therefore, a high F_1 score indicates that the model is effective at predicting labels correctly and mostly predicts the correct labels.

For examination of individual label performance, *accuracy* is examined along with the traditional *precision*, *recall*, and F_1 *score*. The direct breakdown of true positives, true negatives, false positives, and false negatives is directly available in the relevant table. True positives are defined as the correctly identified presence of a label on an example within one category. False positives are defined as label incorrectly applied to an example within a category. Precision in this context measures the percentage of labels predicted correctly within a category. Recall in this context measures the percentage of labels that were correctly predicted out of the true labels within a category. F_1 score is the harmonic mean of recall and precision within one category. A high F_1 score indicates that the model is effective at predicting the labels within a category correctly and mostly predicts the presence of a label correctly within a specific category.

III. Experiments

A. RQ1: What level of performance can be achieved when using fastText based machine learning models to predict associated categories for refactoring reviews?

TABLE II: 10-Fold Cross-Validated Metrics

Model	Precision-Mean	Recall-Mean	F_1 Score-Mean
Trivial	0.804	0.809	0.806
Averaged	0.802	0.802	0.801
GBM	0.806	0.797	0.801
GBM-Length	0.807	0.797	0.802

To address **RQ**$_1$, the trivial, averaged, GBM, and GBM-Length models were built and 10-fold cross-validated. The selected performance metrics were averaged from the 10 iterations and presented in Table II. An inherent limitation in these models is the random nature of their training. The fastText features revealed through the Python API do not include a random seed, and the actual training of fastText models is randomized every run. The results discussed around **RQ**$_1$ are therefore non-deterministic, but generally accurate, as verified by repeated runs of the cross-validation functions.

Referring to Table II, the performance level achievable utilizing machine learning models built off the state-of-the-art fastText algorithm in regards to precision exceeds 80%

among all classifiers. Recall performance is between 79% and 80%, with the trivial model performing the strongest. The F_1 score exceeds 80% in all models, with the trivial model again performing the strongest.

At this level of detail, the averaged model underperforms the trivial model in all categories. This may be due to the equal weights it inherently places on the separate subject and description fields. The trivial, GBM, and GBM-Length models do not have static weights applied to the input fields and are allowed to learn the relationships during training. When comparing the GBM and GBM-Length models to the trivial and averaged, the slightly higher precision they afford comes at the price of slightly lower recall. The labels they predict are slightly more reliable, at the cost of less label predictions.

The addition of subject and description lengths to the GBM-Length model did not increase any performance metric meaningfully. The precision score for the GBM-Length model is slightly higher, but within one standard deviation of the base GBM model. From the similar performance, it appears that the GBM model variants are provided with all necessary prediction information from the outputs of the separate fastText algorithms without the need for additional length information.

RQ_1 Summary: *Utilizing the fastText algorithm allows us to predict the category of refactoring reviews, achieving approximately 80% in common machine learning performance metrics (Precision, Recall, and F_1 Score). The trivial model performs better than the averaged model. The GBM and GBM-Length variants may have a slightly higher precision metric than the trivial model in exchange for a slightly lower recall metric.*

TABLE III: Category Level Performance

Category	Model	FN	FP	TN	TP	Precision	Recall	F_1
Integration	Trivial	51.8	9.3	306.0	59.9	0.87	0.54	0.66
Integration	Averaged	56.9	7.5	307.8	54.8	0.88	0.49	0.63
Integration	GBM	38.2	17.7	297.6	73.5	0.81	0.66	0.72
Integration	GBM-Length	37.2	18.0	297.3	74.5	0.81	0.67	0.73
Objective	Trivial	17.1	58.9	36.9	314.1	0.84	0.95	0.89
Objective	Averaged	8.0	74.8	21.0	323.3	0.81	0.98	0.89
Objective	GBM	28.3	55.2	40.6	302.9	0.85	0.91	0.88
Objective	GBM-Length	26.1	57.1	38.7	305.1	0.84	0.92	0.88
Quality	Trivial	37.0	88.8	111.1	190.1	0.68	0.84	0.75
Quality	Averaged	39.9	84.0	115.9	187.2	0.69	0.82	0.75
Quality	GBM	54.1	70.6	129.3	173.0	0.71	0.76	0.74
Quality	GBM-Length	53.8	71.8	128.1	173.3	0.71	0.76	0.73
Testing	Trivial	32.6	16.9	222.3	155.2	0.90	0.83	0.86
Testing	Averaged	36.3	12.8	226.4	151.5	0.92	0.81	0.86
Testing	GBM	35.5	17.5	221.7	152.3	0.90	0.81	0.85
Testing	GBM-Length	35.6	16.2	223.0	152.2	0.90	0.81	0.85
Refactoring	Trivial	31.9	4.7	390.6	9.8	0.68	0.24	0.35
Refactoring	Averaged	36.7	0.9	384.4	5.0	0.85	0.12	0.21
Refactoring	GBM	28.3	11.3	374.0	13.4	0.54	0.32	0.40
Refactoring	GBM-Length	29.1	11.0	374.3	12.6	0.54	0.30	0.39

B. RQ2: Do certain fastText model variations exhibit better performance over specific categories, compared to the other fastText models examined?

To address **RQ_2**, the trivial, averaged, GBM, and GBM-Length models built for **RQ_1** had their average true positive, true negative, false positive, and false negative rates calculated through a 10-fold average, in the same method as the performance metrics calculated previously. The results are presented in Table III.

Quality. The quality category label is applied when ”reviewers enforce the adherence to coding convention, optimization of internal quality attribute, external quality attribute, the avoidance of (i.e., code smell, resolution of technical debt, correctness of design pattern implementations), and lack of documentation” [9]. False negatives from this category are short and non-descriptive, providing little information for the prediction task. The false positive also seems to not convey much information, and prediction may have been swayed by the keyword ”Selenium” occurring twice. The true labels of the false negative were ”Objective” and ”Quality”. The true labels of the false positive were ”Objective” and ”Testing”. Within the quality category, the trivial and averaged models predicted more positives, both true and false, when compared to the GBM variants. This resulted in a slightly lower precision rate along with an increased recall rate. The F_1 score is higher in the trivial and averaged models than the GBM variants, due to the gains in recall being larger than the losses in precision. The GBM variants were more effective in learning rules to dictate when a refactoring review was not quality related when compared to the trivial and averaged models, as indicated by the lower false positive rate and higher true negative rate. All models struggled when classifying this category, having the second-lowest precision scores within the categories.

Integration. The integration category buckets together refactoring reviews where ”refactoring has complicated the merging process, or triggered configuration issues” [9]. The false negatives (Integration cases missed) did contain the keyword *API*, which often comes with the label, but the subject and description are short and do not provide much information, which would make the model have difficulty predicting in this context. This false negative occurred with the true labels ”Integration” and ”Objective” in the data set. The false positive has the word API in it as well, which is indicative of integration refactoring reviews. This false positive occurred with the true labels ”Objective” and ”Testing” in the data set.

Objective. The objective category reflects refactoring reviews where ”reviewers eventually ask to clearly document the goal, benefit, side effects, scope, feature-related, and bug fix-related activities to better understand the rationale of the submitted code changes” [9]. This task is abstract, being based on the lack of certain information, rather than the presence of information. False negatives belonging to this category, appear to clearly document the purpose, and it would require knowledge of the code base to determine it to be inadequate objective documentation. The false positive appears to provide no objective for the refactoring, and again would require knowledge of the code base to determine it did not need to provide objective. The false negative’s true labels were ”Objective” and ”Quality”. The false positive’s true label was ”Quality”.

Testing. The testing category is applicable to refactoring reviews in which current testing did not adequately reflect the changed behavior after refactoring. False negatives belonging to this category do not discuss refactoring activities, although they mention keywords like *refactor* in the subject. It only refers to adding a test helper function, which would not qualify as a refactoring in and of itself. It is difficult to deduce the category of the false negative from only the subject and description, and prediction may not have been possible on this example. The false positive directly states in the subject and description that the author refactors a test case, which causes deduction of the ”Testing” category incorrectly. The true labels of the false negative were ”Objective” and ”Testing”. The true label of the false positive was ”Quality”.

Refactoring. The refactoring category applies to refactoring reviews in which there is a “focus on evaluating the correctness of the code transformation and checking whether or not the submitted changes lead to a safe and trustworthy refactoring“ [9]. False negatives belonging to this category, portray multiple code refactorings in one commit. On basic intuition, the refactoring category is reasonable to apply. The incorrect prediction on this example is attributed to the minority class status of the ”Refactoring” category, and the inability of the model to learn rules for predicting. The false positive samples lack a description and are a non-feasible prediction task in any case. The true labels of the false negative were ”Testing” and ”Refactoring”. The true label of the false positive was ”Refactoring”.

RQ$_2$ Summary: *The qualitative examination of example misclassifications supports the efficacy of the model. The incorrectly categorized refactoring reviews seem to be either ambiguous to human intuition, counter-intuitive, or lacking in information to the point where no conclusion could be drawn from just the subject and description supplied in these cases. Those conclusions can be derived given the assumptions that all true labels are representing the ground truth. However, given the fact that all case classifications have been made by manual assessments of single engineers [9], they are not necessarily representative of a common understanding across all members of this domain. For meaningful evaluation of the models performance, a validation with feedback from a higher number of knowledgeable people would become necessary. Model performance varied significantly more at the category level than at the aggregate level addressed in* ***RQ***$_1$*. The GBM and GBM-Length models correctly predicted more integration and refactoring labels than the trivial and averaged models, achieving higher recall rates, while the reverse occurred in the quality and objective categories. The GBM and GBM-Length models predicted more positive labels on minority classes than the trivial and averaged models. There was no meaningful difference between the GBM and GBM-Length model in any category. All category level improvements in precision or recall between model variations were the result of a performance trade-off, and accuracy was similar across all models across all categories. Across all models, the refactoring category could not be reliably predicted, whereas Objective and Testing were able to be consistently predicted.*

IV. Conclusion

Predicting the category of refactoring code reviews in their initial push stands to benefit the code review workflow and decrease average time to merge in code bases. In this research, we tested the feasibility of utilizing computationally-economical state-of-the-art models for this prediction task, as well as alternative traditional methods. The results were evaluated quantitatively and qualitatively. Our findings reveal that this prediction task shows strong potential, and we were able to create models achieving high levels of traditional performance measures.

References

[1] Johnatan Oliveira, Markos Viggiato, Mateus F Santos, Eduardo Figueiredo, and Humberto Marques-Neto. An empirical study on the impact of android code smells on resource usage. In *SEKE*, pages 314–313, 2018.

[2] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 171–180, 2015.

[3] Eman Abdullah AlOmar. Deciphering refactoring branch dynamics in modern code review: An empirical study on qt. *Information and Software Technology*, 177:107596, 2025.

[4] Ishan Aryendu, Ying Wang, Farah Elkourdi, and Eman Abdullah Alomar. Intelligent code review assignment for large scale open source software stacks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2022.

[5] Moataz Chouchen, Ali Ouni, and Mohamed Wiem Mkaouer. Multicr: Predicting merged and abandoned code changes in modern code review using multi-objective search. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–44, 2024.

[6] Moataz Chouchen, Ali Ouni, Jefferson Olongo, and Mohamed Wiem Mkaouer. Learning to predict code review completion time in modern code review. *Empirical Software Engineering*, 28(4):82, 2023.

[7] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE ’13, page 712–721. IEEE Press, 2013.

[8] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, 35(4):34–42, 2018.

[9] Eman Abdullah AlOmar, Moataz Chouchen, Mohamed Wiem Mkaouer, and Ali Ouni. Code review practices for refactoring changes: An empirical study on openstack. In *Proceedings of the 19th international conference on mining software repositories*, pages 689–701, 2022.

[10] Facebook AI Research. Fasttext, 2015.

[11] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information, Jun 2017.

[12] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[13] Jianyang Deng and Yijia Lin. The benefits and challenges of chatgpt: An overview. *Frontiers in Computing and Intelligent Systems*, 2(2):81–83, 2022.

[14] Dwaipayan Roy, Debasis Ganguly, Sumit Bhatia, Srikanta Bedathur, and Mandar Mitra. Using word embeddings for information retrieval: How collection and term normalization choices affect performance. In *Proceedings of the 27th ACM international conference on information and knowledge management*, pages 1835–1838, 2018.

[15] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with python*. O'Reilly Media Inc., 2009.

[16] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[17] Steffen Herbold, Alexander Trautsch, and Fabian Trautsch. On the feasibility of automated prediction of bug and non-bug issues. *Empirical Software Engineering*, 25(6):5333–5369, 2020.