Classification of Code Review Comments

Team 11
Jia Xi Ong (25%), Ang Xuan Lee (25%), Kai Rong Lee (25%), Yu Xuan Low (25%),

Introduction

Code review is a fundamental practice in software development, serving as a crucial mechanism to maintain code quality and facilitate knowledge sharing among developers. The automation of code review comment classification can significantly streamline the development process by helping developers focus on the most relevant feedback.

Efforts to improve code review classification have led to advancements such as:

- Leveraging pre-trained models like BERT and CodeBERT for better contextual understanding, enhancing classification accuracy (Feng et al. 2020).
- Automating classification with machine learning to streamline workflows and reduce developer workload.
- Using predictive analytics to prioritize high-risk reviews, reducing time-to-merge and focusing on critical segments.

However, these approaches face challenges, including:

- Limited interpretability in complex models, reducing trust in automated classifications.
- Need for human oversight in complex or high-risk reviews due to insufficient accuracy (Feng et al. 2020).
- Lack of high quality labelled data for supervised learning.

Our study aims to develop an effective classification system for code review comments, categorizing them into three essential operations: **delete, replace, and insert.**

This project is relevant to our module as it lets us apply classical machine learning models like Logistic Regression with Softmax and extend our understanding of RNNs while exploring complex Neural Network models like CodeBERT in code review classification.

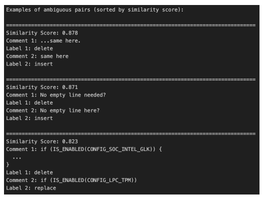
Dataset

The dataset comprises two main components: training and test sets in XLSX format. Each set contains two columns: `Review Comment` and their corresponding `Expected Operation by Developer`. Several notable characteristics and challenges were identified:

Firstly, many comments lacked contextual information, making it difficult to understand their intent in isolation (e.g., "same here" as shown in Figure 1 has two contradictory outputs). The dataset also contained typos and abbreviations, which complicated the text processing.

Additionally, the code-related language in the comment presents unique challenges where programming keywords (e.g., "if," "for") carry significant meaning within code review contexts. This highlights how we cannot process the data using the traditional stop-word removal method.

We also determined that cross-validation is unsuitable due to the limited size of our dataset.



```
Examples of ambiguous pairs (sorted by similarity score):

================================================================================
Similarity Score: 0.878
Comment 1: ...same here.
Label 1: delete
Comment 2: same here
Label 2: insert

================================================================================
Similarity Score: 0.871
Comment 1: No empty line needed?
Label 1: delete
Comment 2: No empty line here?
Label 2: insert

================================================================================
Similarity Score: 0.823
Comment 1: if (IS_ENABLED(CONFIG_SOC_INTEL_GLK)) {
  ...
}
Label 1: delete
Comment 2: if (IS_ENABLED(CONFIG_LPC_TPM))
Label 2: replace
```

Figure 1: Example of bad datasets

To address these challenges, we developed several methods to handle inputs with typos and abbreviations, alongside several filters to eliminate ambiguous and conflicting data.

Handling Typos and Abbreviations

- **Typos**: Each comment was processed through `TextBlob` – a library that uses context to correct spelling errors – which replaced misspelled words with their most likely correct forms based on context.
- **Abbreviations**: We sourced a programming-specific abbreviations dictionary from GitHub. We checked each input against this dictionary, replacing any programming abbreviations with their full forms. In addition, we manually created a dictionary for common informal abbreviations (e.g., "ur" for "your", "u" for "you") and applied similar replacements.

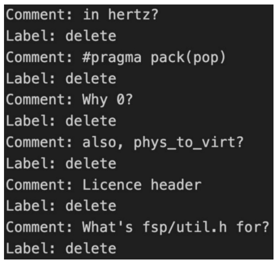
Data Filtering Methods

To further refine the dataset, we developed three filters to eliminate ambiguous and conflicting data points.

- **Identify Similar Comments with different labels**: The `similar_comments_w_diff_label` function, applied *only to training data*, identifies pairs of similar comments (similarity score > 0.6) with differing labels. The similarity score can be adjusted for future needs, but for this purpose, a threshold of 0.6 effectively removed 20 rows (2.1% of train data) with conflicting labels despite nearly identical comments (see Figure 1).
- **Identify Comments with 3 Keyword Labels**: For *both training and test data*. The function `has_all_keywords` flags comments as contradictory if they contain keywords (or their synonyms) from all three categories (“insert,” “delete,” “replace”), suggesting conflicting actions in a comment classified under only one category.
- **Identify Vague or Contextless Comments**: For *both training and test data*. The function `is_vague_or_contextless` identifies short and vague comments that lacks sufficient context. It uses a set of words that signal ambiguous references, such as “same as” or “similar to,” or indicate a lack of specificity (“this,” “that”). We enforce a length threshold of fewer than 4 words combined with one of these patterns to target brief but unclear statements. For very short comments (under 3 words), at least one keyword (or their synonyms) from all three categories (e.g. “insert,” “delete,” “replace”) is required; otherwise, the comment is flagged as too vague to classify reliably.

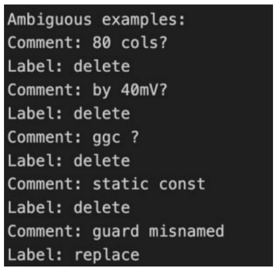
The two filter functions `has_all_keywords` and `is_vague_or_contextless` removed an additional 36 rows (4.0% of cleaned train data) and 13 rows (5.6% of test data) of ambiguous or contradicting data (see Figure 2a and 2b for examples of data rows removed).

IMPORTANT: While we chose to filter out ambiguous data for automation, in practice, a more nuanced approach would involve flagging these comments for human review, especially to retain high-risk or significant comments in the dataset.



```
Comment: in hertz?
Label: delete
Comment: #pragma pack(pop)
Label: delete
Comment: Why 0?
Label: delete
Comment: also, phys_to_virt?
Label: delete
Comment: Licence header
Label: delete
Comment: What's fsp/util.h for?
Label: delete
```

(a) Ambiguous or Contradicting Train Data removed



```
Ambiguous examples:
Comment: 80 cols?
Label: delete
Comment: by 40mV?
Label: delete
Comment: ggc ?
Label: delete
Comment: static const
Label: delete
Comment: guard misnamed
Label: replace
```

(b) Ambiguous or Contradicting Test Data removed

Figure 2: Data Filtering Results

Methods

With the filtered dataset, we will proceed with feature extraction and model training. We will explore three feature extraction methods – TF-IDF, CountVectorizer, and CodeBERT embeddings. Followed by feature selection for the selected feature extraction method. Lastly, we will explore Softmax Regression, Support Vector Machine (SVM), and fine-tuned CodeBERT as our classification models.

Feature Extraction

We leveraged two primary feature extraction strategies:

1. Bag-of-Words approaches (BoW);
2. Pre-Trained Language Models (PTLM);

catering to our dataset which comprises mixed natural language and programming language inputs.

Bag-of-Words Approaches Due to the keyword-driven nature of code review comments after our filter functions, we believe that models like Term Frequency-Inverse Document Frequency (TF-IDF) and CountVectorizer are suitable for capturing specific terms. We configured both with the parameter $ngram_range = (1, 3)$, enabling the capture of unigrams, bigrams, and trigrams to identify nuanced operational intents. The parameter $max_features = 15000$ was set to include frequent tokens while filtering out rare, noise-inducing terms. Given the small-sized dataset (about 900 entries), the BoW approach provided a direct method to quantify term significance related to intended operations without relying on complex contextual embeddings (Sahel 2023b).

Pre-trained Language Model To evaluate contextually-rich representations, we decided to use CodeBERT – a BERT variant pre-trained on both natural language (NL) and programming language (PL). Code review comments often blend NL and PL, thus necessitating a method capable of capturing both semantics and syntax concurrently. CodeBERT’s pre-training on Masked Language Modeling (MLM) and Masked Identifier Modeling (MIM) made it ideal for our dataset (Wang et al. 2022). We experimented with three different techniques for extracting embeddings:

- **Token-Level Embedding**: Embeds each token individually, providing a fine-grained representation for detailed analysis of specific terms.
- **Mean-Pooling**: Averages token embeddings to represent the overall context of the comment, useful for understanding broad operational patterns.
- **Max-Pooling**: Takes the maximum value across token embeddings to emphasize the most significant features, reducing noise from irrelevant tokens.

Through these feature extraction strategies, we aimed to determine whether a context-sensitive representation (CodeBERT) would outperform a frequency-driven representation (BoW) for capturing the operational intent of code review comments.

Model Training

Once features were extracted, we experimented with three models—Softmax Regression, Support Vector Machine (SVM), and fine-tuned CodeBERT. The selection of models and tuning parameters was driven by a desire to achieve the best balance between performance, computational efficiency, and interpretability given our small dataset.

Softmax Regression We start with Softmax Regression as a baseline for each feature extraction method, due to its simplicity and effectiveness for multi-class classification. The `lbfgs` solver was selected for its efficiency with smaller datasets, while `max_iter=300000` provided sufficient iterations to ensure convergence. `lbfgs` solver is good for small datasets as it allows for $O(n^2)$ convergence as compared to steepest descent methods. The cost for each iteration however, is computationally expensive as you have to calculate the inverse of the Hessian matrix which is $O(n^3)$. The regularization parameter `C=1` was set to control overfitting, balancing model generalization and stability for our limited dataset size.

Support Vector Machine (SVM) We also used SVM due to its ability to model non-linear decision boundaries, which are crucial for our mixed NL+PL data. We used the `rbf` (Radial Basis Function) kernel, which performs well with non-linear, high-dimensional data. The kernel's non-parametric nature made it ideal for capturing complex relationships present in our data (Sahel 2023a). We set `C=20` for low misclassification tolerance, focusing the model's attention on important distinctions, and `gamma=0.01` to allow a larger expand of influence of individual data points.

Fine-Tuning CodeBERT We will only experiment Fine-tune CodeBERT with CodeBERT feature extraction, regardless of the feature extraction chosen. Fine-tuning allowed us to adapt the pre-trained model to the specific structure and language used in code review comments, leveraging its contextual strengths. Fine-tuning inherently reduces interpretability and risks overfitting, especially with a small dataset like ours (Feng et al. 2020). However, we decided to experiment with CodeBERT and fine-tune CodeBERT regardless, as we understand that given a larger dataset, CodeBERT has significant potential to classify code review categories accurately due to its ability to understand both semantic meaning and code-specific contexts.

Model Selection Strategy

- If the chosen feature extraction method is TF-IDF or CountVectorizer, we will experiment with Softmax Regression and SVM. Fine-tuning CodeBERT using BoW features is not considered appropriate, as BoW lacks the contextual depth required for this approach. However, we will still experiment with the CodeBERT model using its native contextual embeddings to explore its performance, independent of the BoW features
- If CodeBERT embedding is selected, we will proceed with Softmax Regression, SVM, and fine-tuned CodeBERT.

Results and Discussions

Firstly, let us discuss the findings on the best feature extraction technique. To evaluate the performance of different feature extraction methods, we conducted comparative analyses based on accuracy, precision, recall, and F1-score. We will utilize **Softmax Regression as our baseline model** to compare the feature extraction methods, as discussed in the previous section. This section presents the results of selecting the best feature extraction method, followed by detailed discussions.

Feature Extraction Results

Performance Overview We chose accuracy as our primary evaluation metric since our dataset is balanced across the `insert`, `delete`, and `replace` classes, making it a reliable indicator of overall performance. Additionally, we used the F1-score to gain detailed insights into class-level performance, especially for the `replace` class, which is more ambiguous as it overlaps characteristics of both `insert` and `delete` operations.

Feature Extraction Method	Accuracy
CountVectorizer (n-grams up to 3)	69%
TF-IDF Vectorizer (n-grams up to 3)	64%
CodeBERT Embeddings (Mean Pooling)	58%
CodeBERT Embeddings (Token-Level)	54%
CodeBERT Embeddings (Max Pooling)	55%

Table 1: Accuracy for Different Feature Extraction Methods

CountVectorizer CountVectorizer performed well by effectively capturing specific n-grams (unigrams, bigrams, trigrams) that represent operational intent in code review comments. Its direct approach was well-suited to identifying key terms like "insert" and "delete," which were crucial for classification.

TF-IDF Vectorizer TF-IDF Vectorizer slightly underperformed compared to CountVectorizer likely because it downweights frequent terms, which may have included important keywords for classification. While TF-IDF helps reduce noise, some frequent terms carried essential signals that were necessary for differentiating operations.

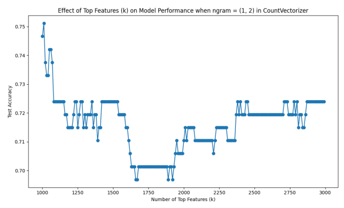
CodeBERT CodeBERT's lower accuracy in this task likely stems from the challenge of capturing task-specific nuances with a limited dataset. Unlike simpler models, CodeBERT relies on **large datasets** to capture both language and code context effectively. Max pooling, in particular, performed the worst likely because it emphasizes only the most significant features across tokens, which can overlook important contextual details by focusing too heavily on specific terms.

We decided to choose CountVectorizer due to its high accuracy (69%), ease of implementation, and relatively low computational cost compared to other more complex feature extraction methods.

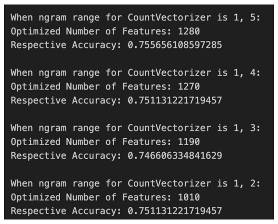
Feature Selection

After deciding on CountVectorizer as our feature extraction method, we focused on fine-tuning its parameters to identify the most effective configuration. This included selecting the best n-gram range as well as optimizing the number of top features (k).

N-gram Range Selection We experimented with various n-gram ranges and found that n-gram range (1, 5) provided slightly higher accuracy, however, we opted for n-gram range (1, 2) due to its better interpretability as its accuracy score is not far off from n-gram range (1, 5). The smaller range captures enough meaningful context for our code review classification task without overfitting or introducing excessive noise, which is particularly advantageous given our limited dataset size. The best top k features for n-gram range (1, 2) was chosen to be 1010, as seen in Figure 3a.
Next, we will proceed with model selection using the already chosen and tuned CountVectorizer feature extraction.



(a) Effect of Top Features (k) on Model Performance when n-gram = (1, 2)



```
When ngram range for CountVectorizer is 1, 5:
Optimized Number of Features: 1280
Respective Accuracy: 0.755656108597285

When ngram range for CountVectorizer is 1, 4:
Optimized Number of Features: 1270
Respective Accuracy: 0.751131221719457

When ngram range for CountVectorizer is 1, 3:
Optimized Number of Features: 1190
Respective Accuracy: 0.746606334841629

When ngram range for CountVectorizer is 1, 2:
Optimized Number of Features: 1010
Respective Accuracy: 0.751131221719457
```

(b) Effect of Different N-gram Ranges and Corresponding Optimized Number of Features (k)

Figure 3: Comparing Optimal Number of Features (k) and N-gram Ranges

Model Selection Results

CountVectorizer + Softmax Regression Using the optimized CountVectorizer parameters (`n-gram range` of (1, 2) and `k=1010` features), we achieved an accuracy of **0.75**. This result is promising not only because of the high accuracy but also due to the high interpretability of this combination. The Softmax Regression model is straightforward and offers insights into the feature importance for each class, making this combination ideal for practical use.

CountVectorizer + SVM SVM achieved an accuracy of **0.68** on the cleaned test data. Although the RBF kernel is powerful and well-suited for high-dimensional, non-linear data, SVM's performance was not as strong as Softmax Regression. The RBF kernel attempts to increase the dimensionality of the data to find a clear separation between classes. However, this approach may have lead to excessive complexity, which might have negatively impacted the performance due to overfitting on limited data.

Fine-Tuned CodeBERT We also experimented with fine-tuning CodeBERT as a Neural Network model, using CodeBERT for feature extraction. The results showed an accuracy of **0.67**, which is still lower compared to the classic ML models. This lower accuracy could be attributed to the complexity of CodeBERT and its higher risk of overfitting, especially given the relatively small size of our dataset, which CodeBERT might have struggled to adapt effectively without sufficient data.

Model	**Accuracy**
CountVectorizer + Softmax Regression	0.75
CountVectorizer + SVM	0.68
Fine-Tuned CodeBERT	0.67

Table 2: Accuracy Summary for Different Models

Based on these results, **CountVectorizer + Softmax Regression** emerged as the best combination due to its high accuracy and interpretability. While SVM and CodeBERT offered competitive performance, they lacked the accuracy and simplicity offered by the Softmax model with CountVectorizer feature extraction.

Ensemble Method (Future Work)

After analyzing the results, we considered that the full potential of CodeBERT might not have been fully demonstrated due to the small size of our dataset. This led us to explore an ensemble approach, combining 60% of the features from CountVectorizer+Softmax and 40% from CodeBERT. The ensemble approach resulted in an accuracy of **0.75**, which is comparable to our Softmax model.

We believe this approach has substantial potential if applied to a larger dataset. However, we chose not to expand the dataset due to the inherent ambiguity and inconsistency in labeling code review comments. Different reviewers may interpret and define code review categories differently, making it challenging to find accurately labeled data that fully represents the ambiguity of real-world data without introducing overfitting. Thus, we propose this ensemble approach as an open opportunity for future research, particularly when more comprehensive and well-labeled code review datasets become available.

References

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155.

Sahel, E. 2023. Introduction to RBF SVM: A Powerful Machine Learning Algorithm for Non-linear Data. Medium.

Sahel, E. 2023. Exploring Feature Extraction Techniques for Natural Language Processing. Medium.

Wang, H.; Zhang, T.; Wu, Z.; and Chen, X. 2022. Pre-Trained Language Models and Their Applications. Engineering 2022(4): 249-261.