Machine Learning for Model Driven Engineering

Code Comment Classification Project Report

Luca Francesco Macera, Calogero Carlino

January 2026

Contents

1 Introduction to the Problem

Software maintenance and evolution are highly dependent on understanding existing codebases. Code comments are a primary source of documentation, but they are often unstructured and heterogeneous. Automatically classifying code comments into functional categories (e.g., "Summary", "Usage", "Parameters") can significantly enhance automated software engineering tools, such as documentation generators and code understanding systems.

The goal of this project is to build a machine learning pipeline to automatically classify source code comments into predefined categories. This task presents several challenges, including short and ambiguous text, semantic overlap between categories, and class imbalance.

2 Explanation of the Dataset

The project utilizes the `code-comment-classification.csv` dataset from the NLBSE'23 tool competition on code comment classification[1], where each row represent a sentence (aka an instance) and each sentence contains six columns as follow:

1. `comment_sentence_id` is the unique sentence ID;
2. `class` is the class name referring to the source code file where the sentence comes from;
3. `comment_sentence` is the actual sentence string, which is a part of a (multi-line) class comment;
4. `partition` is the dataset split in training and testing, `0` identifies training instances and `1` identifies testing instances, respectively;
5. `instance_type` specifies if an instance actually belongs to the given category or not: `0` for negative and `1` for positive instances;
6. `category` is the ground-truth or oracle category.

3 Implementation

The project is implemented in Python using a Jupyter Notebook environment. The solution emphasizes modularity and data integrity, ensuring no data leakage during the training process. All the relevant code is available in the project repository[2].

[1] https://nlbse2023.github.io/tools/
[2] https://github.com/LucaFraMacera/ML4SE-2025-Project

3.1 Initial Dataset Manipulation

3.1.1 Data Augmentation and Scaling

During the initial experimental phase, it was observed that the models trained solely on the original dataset yielded suboptimal performance, with accuracy metrics plateauing around 60%. This limitation was primarily attributed to the small sample size (approximately 2,800 usable instances after preprocessing), which provided insufficient variance for the models to generalize effectively over the high-dimensional BERT embedding space.

To address this data scarcity, a data augmentation strategy was implemented involving the following steps:

1. **Repository Scraping**: We extracted additional source code comments from various public GitHub repositories to create a supplemental corpus of real-world data.

2. **LLM-Based Weak Supervision**: As the scraped comments lacked ground truth labels, we utilized a Large Language Model (LLM) to perform zero-shot classification. The LLM automatically annotated the raw comments into the project's target categories ("Summary", "Usage", "Parameters", etc.), effectively generating "silver standard" labels.

3. **Integration and Hardware Constraints**: The augmented data was merged with the original dataset, significantly increasing the volume of training instances. However, processing the dense vector embeddings for this expanded dataset exceeded the memory (RAM) limitations of the available local hardware.

4. **Strategic Downsampling**: To resolve the computational bottleneck, we applied a randomized downsampling technique. The dataset was iteratively reduced, preserving the class distribution, until the volume of data was compatible with our hardware resources while still remaining significantly larger and more diverse than the original baseline.

3.1.2 Cleaning

The preprocessing phase focused on ensuring data integrity and optimizing the label space for supervised learning. The raw dataset contained negative samples (`instance_type=0`) which were removed to isolate the true labels (`instance_type=1`). To facilitate a rigorous experimental design, redundant metadata columns (such as the original `partition` attribute) were removed. This allowed for the implementation of an independent, stratified k-fold cross-validation strategy, ensuring that the model's performance was not biased by pre-existing data splits.
Furthermore, an exhaustive deduplication process was performed to eliminate identical entries, a critical step to prevent data leakage between the training and evaluation subsets.

A semantic similarity analysis was conducted using BERT embeddings to calculate the cosine similarity between category centroids. The analysis revealed a high similarity (0.82) between the "Usage" and "Expand" categories. Consequently, these two categories were automatically merged to reduce classifier confusion and improve label distinctness.

3.1.3 Encoding

The classification framework utilizes a multifaceted encoding pipeline to project code comments into a unified high-dimensional feature space. This hybrid strategy is designed to capture semantic depth, structural context, and explicit technical heuristics, resulting in a comprehensive and complete representational vector. By combining these diverse feature sets, the model follows the industry-standard practice of multi-modal feature fusion to achieve higher discriminative power than single-source models.

3.1.4 Text and Categorical Encoding

In alignment with current state-of-the-art NLP practices, the primary representation layer utilizes a transformer-based model to handle the inherent complexity of natural language. Specifically, the `all-MiniLM-L6-v2 model` is employed to generate dense embeddings, a technique widely recognized as superior to traditional bag-of-words or TF-IDF approaches due to its ability to capture latent contextual semantics. To provide the necessary structural context, this semantic vector is augmented with categorical data through the **One-Hot** encoding of the source `class` attribute. This transformation results in 306 additional features, representing a standard solution for incorporating non-textual environment variables into a machine learning pipeline. By merging these two components, the framework ensures the classifier evaluates both the linguistic intent and the specific source code origin of each comment.

3.1.5 Metadata Encoding

To complement the high-dimensional semantic representations provided by the BERT encoder, we incorporate a suite of handcrafted metadata features. This hybrid approach is predicated on the observation that structural and stylistic attributes of source code comments often provide orthogonal signals that latent embeddings may not fully prioritize.

- **Lexical Complexity**: We calculate the `comment_length` as a proxy for information density, distinguishing between concise operational notes and verbose documentation.
- **Domain-specific Indicators**: The features `has_params` and `has_default` serve as boolean markers for API-related specifications, identifying comments that describe method signatures or initialization states.

- **Structural Syntax**: Through the detection of `code_symbols`, we quantify the technical orientation of the comment, identifying the presence of formal logic within the natural language.
- **Action-Oriented Linguistics**: The `starts_with_verb` feature evaluates the presence of imperative phrasing, a common convention in documentation aimed at describing functional behaviour.

This augmented dataset allows for a multi-modal approach to classification, where the model simultaneously evaluates the latent semantic content of the text and the explicit structural metadata associated with it.

3.2 Model Training

The training process was designed to handle high-dimensional sparse data and significant class imbalance. The implementation relies on the `imbalanced-learn` library to ensure that resampling strategies are correctly applied within cross-validation loops, preventing data leakage.

3.3 Feature Representation and Data Loading

The classification models ingest the data in the form of sparse matrices to efficiently handle the high dimensionality produced by the embedding process. The training features are loaded from compressed `.npz` files, combining the dense BERT embeddings with the encoded metadata features. The target variables are read from CSV files and subsequently flattened into one-dimensional arrays using the ravel method, ensuring compatibility with the input requirements of Scikit-learn estimators.

3.4 Pipeline Architecture and Class Imbalance

To address the imbalanced distribution of the dataset, a specialized `ImbPipeline` was utilized rather than a standard Scikit-learn pipeline. This distinction is critical for the validity of the experiment as it allows for resampling steps to occur strictly during the fitting phase. A `RandomOverSampler` with a fixed random state of `42` is applied to the training folds only. This ensures that synthetic samples are generated exclusively from the training portion of the fold, leaving the validation fold in its original distribution and preventing data leakage.

3.5 Classification Architectures

Four distinct classification architectures were implemented to evaluate performance across different mathematical approaches. All models utilized a consistent random state to ensure comparable results.

3.5.1 Logistic Regression

This model served as the primary baseline for linear separability. It was configured with a high maximum iteration count of 2000 to ensure convergence on the high-dimensional feature space.

3.5.2 SGD Classifier

The Stochastic Gradient Descent Classifier was utilized with the hinge loss function. This configuration effectively functions as a linear Support Vector Machine optimized via gradient descent, making it particularly suitable for efficient processing of large sparse datasets.

3.5.3 Linear SVC

A standard Linear Support Vector Classification implementation was included to evaluate the effectiveness of finding a maximum-margin hyperplane in the embedding space. Unlike the SGD approximation, this model uses a dedicated linear kernel solver.

3.5.4 Random Forest Classifier

To test if a non-linear approach could capture complex relationships better than linear models, a Random Forest Classifier was employed. This ensemble method consisted of `100` decision trees.

3.6 Model Selection

To maximize the performance of the most promising architecture, an extensive Grid Search was performed on the Logistic Regression model using 5-fold cross-validation. The optimization targeted the **F1-Macro** score to ensure balanced performance across categories.

The search space included:

- **Regularization Strength**: Multiple values were tested to control overfitting.
- **Solvers**: Comparison between `liblinear` (high variance) and lbfgs (multiclass).
- **Penalties**: Both L1 (Lasso) and L2 (Ridge) regularization were evaluated.
- **Balancing Strategy**: The model's native `class_weight='balanced'` parameter was contrasted against the pipeline's oversampling strategy to determine the optimal method for handling class imbalance.

The best estimator identified by the Grid Search was then evaluated on the hold-out test set to produce the final accuracy and classification report.

4 Results

The empirical evaluation, summarized in Table ??, demonstrates the effectiveness of the hybrid feature set. The results indicate that linear architectures are particularly well-suited for high-dimensional semantic spaces, significantly outperforming ensemble tree-based methods in this specific classification task.

Model	**Accuracy**	**Avg. Std. Deviation**	**Result**
Logistic Regression	**0.8323**	**0.0039**	✓**Best**
Linear SVC	0.8210	0.0075	Close second
SGD Classifier	0.8139	0.0197	Moderate
Random Forest	0.7944	0.0050	× Worst

Table 1: Performance Comparison of Machine Learning Models

Results have also shown a training accuracy of 90% compared to an 82% testing accuracy. While a gap is expected in complex classification tasks involving natural language, this margin suggests that the model has memorized some specific noise or patterns in the training data that do not generalize to the test set. However, since the testing accuracy remains high ($> 80\%$), the model is not suffering from significant overfitting, nor is it underfitting (which would be characterized by poor scores on both sets).

In conclusion, the model maintains a symmetrical performance profile. For most classes, `Precision` and `Recall` are very close (e.g., Class 3: `0.80 Precision` vs `0.82 Recall`). This consistency suggests the decision boundary is well-calibrated; the model is neither overly conservative (which would spike `Precision` but drop `Recall`) nor overly aggressive (which would spike `Recall` but drop `Precision`).

4.1 Key findings

The performance across models reveals several critical insights into the nature of code comment classification:

- **Best Model**: The superior performance of **Logistic Regression** (83.2% accuracy; $F_1 = 0.83$) suggests that the combination of BERT-derived latent semantics and engineered metadata creates a feature space that is largely linearly separable. While **Random Forest** is typically robust, its relative underperformance (79.4%) may be attributed to the high dimensionality and sparsity of the one-hot encoded class context, which can dilute the splitting criteria in decision trees.

- **Effectiveness of Merging**: The automated merging of the Usage and Expand categories, driven by our initial cosine similarity analysis (0.82), proved instrumental. By resolving this specific semantic overlap, we effectively reduced label noise, allowing the classifier to establish more distinct decision boundaries.

- **Robustness and Generalization**: The low standard deviation observed in the winning model (0.0039) confirms that the performance is consistent across the `5-fold cross-validation`. This indicates that the feature enrichment strategy (integrating structural heuristics like `has_params` and `starts_with_verb`) provided stable signals that generalize well with new and unseen data.

4.2 Conclusion

This project demonstrates an effective pipeline for classifying source code comments by fusing deep semantic embeddings with structural metadata. By augmenting the dataset to 30,000 rows and optimizing the label space through similarity-based merging, we achieved a final accuracy of 83.23%. The results confirm that for short-text software documentation, a large-scale augmented dataset combined with a hybrid feature set provides a reliable foundation for automated classification.