# Optimizing Function-Level Source Code Classification Using Meta-Trained CodeBERT in Low-Resource Settings

Abednego Dwi Septiadi[1,*], Muhamad Awiet Wiedanto Prasetyo[2], Geusan Edurais Aria Daffa[3]

[1,3]*Department of Software Engineering, Faculty of Informatics, Telkom University, Indonesia*

[2]*Department of Information Systems, Faculty of Industrial Engineering, Telkom University, Indonesia*

(Received: 1 April 2025; Revised: 1 June 2025; Accepted: 25 July 2025; Available online: 25 August 2025)

**Abstract**

This study investigates the effectiveness of a meta-trained transformer-based model, CodeBERT, for classifying source code functions in environments with limited labeled data. The primary objective is to improve the accuracy and generalizability of function-level code classification using few-shot learning, a strategy where the model learns from only a few labeled examples per category. We introduce a meta-learning framework designed to enable CodeBERT to adapt to new function types with minimal supervision, addressing a common limitation in traditional code classification methods that require extensive labeled datasets and manual feature engineering. The methodology involves episodic few-shot classification, where each episode simulates a low-resource task using five labeled and five unlabeled samples per function class. A balanced subset of Python functions was sampled from the CodeXGLUE benchmark, consisting of ten function categories with equal representation. The source code was preprocessed by removing comments and docstrings, then tokenized into a fixed length of 128 tokens to fit the model input format. The meta-trained CodeBERT was evaluated across 10 episodes, each representing a different task composition. Results show that the model achieves an average classification accuracy of 73.0%, with high accuracy on function categories characterized by unique syntax patterns, and lower performance on categories with overlapping logic or naming structures. Despite this variability, the model-maintained accuracy above 60% in all episodes. These findings suggest that meta-learning significantly enhances the adaptability of CodeBERT to unseen tasks under data-constrained conditions. This research demonstrates that meta-trained transformer models can serve as practical tools for real-time code analysis, particularly in integrated development environments and continuous integration pipelines. Future work may include extending the framework to other programming languages and incorporating semantic code representations to further reduce classification ambiguity.

*Keywords:* Meta-Learning, Few-Shot Learning, Code Classification, CodeBERT, Transformer Models, Low-Resource Software Engineering, Software Process

## 1. Introduction

In modern software development, understanding and organizing source code is a fundamental task that underpins activities such as documentation, maintenance, code reuse, and automated analysis [1]. As software systems grow in size and complexity, manual classification and labeling of source code functions becomes increasingly impractical [2]. To address this, machine learning-based code classification has emerged as a promising approach to assist developers in navigating and managing codebases more efficiently [3]. Traditional code classification methods typically require large amounts of labeled data and are often domain-specific [4]. They rely on either manually engineered features or supervised learning pipelines that do not generalize well to new or unseen categories [5]. This poses a significant challenge in real-world settings where labeled data is sparse or expensive to obtain (especially in newly developed or proprietary systems where existing datasets cannot be reused) [6]. Consequently, there is a need for models that can perform well under limited supervision and adapt quickly to new function classes with minimal examples [7].

Few-shot learning addresses this challenge by enabling models to classify new instances based on only a few labeled examples [8]. This learning paradigm is especially relevant in the context of software engineering, where labeled function examples may only be available for a small subset of categories [9]. Recent advances in few-shot learning, particularly through meta-learning frameworks, offer a mechanism for training models to generalize across tasks, rather than optimizing for a single fixed task [10]. Meta-learning, or "learning to learn," trains models on a distribution of

***Corresponding author: Abednego Dwi Septiadi (abednego@telkomuniversity.ac.id)**
**DOI: https://doi.org/10.47738/jads.v6i3.902**

tasks so that they can rapidly adapt to new tasks using limited support data [11]. CodeBERT, a transformer-based model pre-trained on large corpora of source code and natural language, has shown strong performance on a variety of code understanding tasks [12]. While most applications of CodeBERT have involved fine-tuning for specific downstream tasks with abundant data, its capability in few-shot and meta-learning contexts remains underexplored [13]. Leveraging a meta-trained version of CodeBERT, which has been optimized to generalize across multiple low-resource classification tasks, offers the potential to solve real-world software engineering challenges with minimal supervision.

While the few-shot capabilities of CodeBERT remain relatively less explored compared to its extensive use in supervised settings, several recent studies have begun to investigate its performance under low-resource scenarios. For instance, applications of CodeBERT in function naming, bug localization, and code summarization have demonstrated promising results using few-shot or meta-learning approaches [14], [15], [16]. These works indicate that CodeBERT possesses latent generalization capabilities suitable for low-data environments. However, most prior efforts focus on specific downstream tasks with limited cross-task generalization analysis. Therefore, this study builds upon those foundations by evaluating a meta-trained CodeBERT within a standardized episodic few-shot framework, aiming to assess its task adaptability across multiple function classification episodes. This contribution seeks not to claim novelty in applying CodeBERT for few-shot learning, but rather to provide a comprehensive and reproducible assessment of its effectiveness in function-level classification under constrained data conditions.

This research investigates the effectiveness of a meta-trained CodeBERT model in a function-level code classification task using a few-shot learning framework. Specifically, we employ episodic evaluation with limited labeled examples (5-shot) per class across 10 classification episodes. The goal is to evaluate the model's ability to generalize to new classification tasks without additional training or fine-tuning. By simulating realistic software engineering scenarios with constrained data, this study highlights the practical benefits of meta-learning for low-resource code classification.

## 2. Literature Review

### 2.1. Meta-Learning Approaches in Few-Shot Learning

Meta-learning, or “learning to learn,” encompasses a range of strategies designed to improve model generalization across tasks by simulating low-resource learning conditions during training. Broadly, meta-learning methods can be categorized into three major paradigms: optimization-based, metric-based, and model-based approaches [17]. Optimization-based methods, such as Model-Agnostic Meta-Learning (MAML), aim to find model parameters that can be quickly fine-tuned to new tasks using a small number of gradient steps. These approaches preserve flexibility across diverse task distributions but often require careful tuning and significant computational resources during meta-training [18].

In contrast, metric-based methods (including Prototypical Networks and Matching Networks) operate by learning an embedding space where samples from the same class are close together. Classification is then performed using distance-based measures such as Euclidean or cosine similarity between support and query samples [19]. These models tend to be more efficient and interpretable, and are especially suitable for episodic few-shot learning setups. Model-based approaches, such as memory-augmented networks, incorporate external memory components that help the model store and retrieve task-specific information. While less common in code-related tasks, they are effective in learning rapid task adaptation mechanisms without requiring gradient updates [20].

In the context of source code classification, metric-based approaches are often preferred due to their simplicity and compatibility with transformer-based embeddings. However, optimization-based strategies offer greater flexibility for fine-grained semantic distinctions, particularly in function-level tasks. This study focuses on the application of a metric-based meta-learning framework, leveraging the representational strength of CodeBERT to classify functions with minimal supervision.

### 2.2. Code Representation Learning in Software Engineering

Recent advancements in deep learning have significantly influenced the automation of software engineering tasks [14]. Models designed to understand and represent source code have enabled a range of applications including code

summarization, clone detection, defect prediction, and code classification [15]. These models rely on capturing both the syntactic and semantic properties of code, often leveraging token-level embeddings derived from programming languages [16]. Traditional approaches in code classification have used manually engineered features or tree-based representations such as Abstract Syntax Trees (ASTs), combined with classifiers like SVMs or random forests [17]. However, these techniques struggle with generalization across programming styles and lack contextual understanding [18].

The emergence of Pretrained Language Models (PLMs) has introduced a paradigm shift in code understanding [19]. PLMs such as CodeBERT, CodeT5, and GraphCodeBERT extend transformer-based architectures to source code by training on large-scale paired datasets of natural language and code [20]. Among them, CodeBERT has been widely adopted due to its balance between performance and model complexity [21]. It uses a RoBERTa-style transformer pretrained on the CodeSearchNet corpus, enabling it to encode both code and natural language into a shared embedding space [22]. These models have shown strong results in tasks like function classification, code-to-text translation, and retrieval [23].

## 2.3. Few-Shot Learning and Meta-Learning

Few-shot learning refers to a model's ability to generalize to new classes or tasks with only a small number of labeled examples [24]. This setting is particularly relevant in software engineering, where labeled code datasets are costly to produce and difficult to scale [25]. Few-shot methods aim to make learning feasible in such low-resource conditions by leveraging prior knowledge [26]. Meta-learning, or "learning to learn," is a prominent framework used to address few-shot problems. Instead of training a model on a single task, meta-learning frameworks train models across a distribution of tasks, enabling them to quickly adapt to new, unseen tasks [27]. Algorithms such as MAML, Prototypical Networks, and Matching Networks are popular in this space.

In the context of natural language processing, meta-learning has been applied to text classification, intent detection, and question answering [28]. However, its application to code understanding is still an emerging field [29]. Recent research has explored few-shot function naming and code classification using both prototypical models and transformer-based architectures, demonstrating promising results when pretrained models are fine-tuned in a meta-learning setup [30].

## 2.4. Meta-Learning for Code Classification

While most existing work on code classification relies on supervised fine-tuning with large amounts of labeled data, recent studies have attempted to adapt meta-learning approaches to this domain [31]. Meta-training models like CodeBERT allows the model to perform well on new classes of functions with very limited supervision [32]. For instance, studies have shown that episodic training strategies, where each task mimics a few-shot episode with support and query sets, help transformer-based models adapt more effectively during evaluation [33]. These models use class-level context embeddings (often based on [CLS] tokens) and softmax-based classification heads to predict function categories even in the absence of extensive training data [34].

Moreover, it has been demonstrated that transformer models like CodeBERT retain strong general-purpose representations that can be fine-tuned across multiple few-shot tasks using meta-objectives, leading to increased performance and data efficiency [35]. Despite the progress, there remains a lack of extensive evaluation on how well meta-trained code models generalize across diverse function classification tasks, particularly in real-world settings where class distribution and code complexity vary [36]. Most previous works emphasize benchmark performance rather than few-shot adaptability in unseen domains [37].

This study addresses that gap by evaluating a meta-trained CodeBERT model using a standardized episodic framework with few-shot support, testing it across multiple episodes and reporting performance trends [38]. Unlike traditional classification studies that rely on fine-tuning per task, our approach assesses true generalization without additional updates, aligning with the goals of low-resource software analysis [39].

 ISSN 2723-6471


## 3. Methodology

Figure 1 illustrates the research workflow employed in this study, starting with Problem Identification, where the research problem of limited labeled data for code classification is defined, and the objective of evaluating a meta-trained CodeBERT model in few-shot learning tasks is established. This is followed by Data Collection and Preparation, which involves collecting the function dataset (function.json), parsing JSON files to extract functions and labels, and performing data cleaning to standardize the code. Preprocessing then tokenizes the extracted code using the CodeBERT tokenizer, generating input IDs and attention masks necessary for model input. In the Modeling phase, a pretrained CodeBERT model is loaded, a classification head is added, and meta-trained weights are applied to adapt the model for few-shot learning. The Evaluation stage constructs few-shot episodes (5 support + 5 query samples per class), evaluates the model on query sets, and records accuracy for each episode. This is followed by Analysis, where the average accuracy is calculated, and model performance is examined for trends and generalization. Finally, the workflow concludes with Conclusion, where final insights are drawn, affirming that meta-learning supports low-data classification.

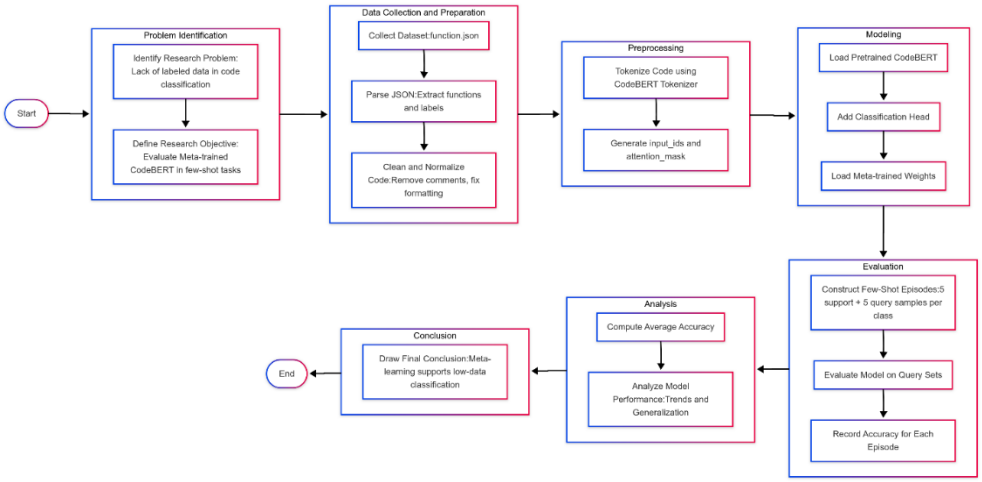


**Figure 1.** Research Flow

### 3.1.Dataset Collection and Preparation

This study employs a benchmark dataset obtained from the CodeXGLUE repository, which was developed by Microsoft to facilitate research in source code intelligence. The dataset consists of Python function definitions, each annotated with a class label representing its semantic role. Each entry in the JSON file (function.json) includes two fields: a func string containing the function body and a target indicating the associated class label. To simulate a low-resource environment in line with few-shot learning principles, a subset of 500 function-label pairs was randomly sampled from the larger dataset. While this subset size reflects practical limitations in labeled data availability, care was taken to ensure class balance by selecting an approximately equal number of samples from each function class. This sampling strategy helps preserve the diversity and representativeness of function types, including utility functions, data manipulators, and computational routines.

The data preprocessing phase included parsing the JSON structure, validating entries, and applying standard cleaning procedures. These procedures involved removing inline comments, docstrings, and excessive whitespace to ensure uniform formatting. Although comments and docstrings often contain valuable semantic information that may aid classification, they were deliberately excluded in this study to ensure that the model relies solely on the structural and lexical patterns within the source code itself. This decision reflects a realistic constraint in automated code classification settings, where such contextual annotations may be missing, outdated, or unavailable in production codebases. Overall, this curated and cleaned dataset provides a reliable foundation for evaluating the few-shot classification capability of the meta-trained model under constrained supervision conditions.

## 3.2.Preprocessing

Prior to model inference, each function was preprocessed and tokenized using the RobertaTokenizer associated with the microsoft/codebert-base model. Tokenization involved converting the raw function strings into subword token sequences compatible with transformer-based architectures. Each sequence was truncated or padded to a fixed length of 128 tokens, ensuring uniform input dimensions across the batch. This fixed-length configuration is essential for optimizing computational efficiency during training and inference, as transformers require consistent input sizes for parallel processing.

The choice of 128 tokens was based on empirical observations from the dataset, where the majority of function definitions fell within this length. Functions exceeding this threshold were truncated, potentially omitting trailing lines of code such as return statements or auxiliary logic. While this introduces a trade-off, it reflects a common practice in code intelligence tasks and strikes a balance between preserving meaningful content and maintaining processing efficiency. For more complex functions, future work may explore dynamic or hierarchical tokenization to minimize information loss. The tokenizer produced two outputs for each input sample: input_ids, which represent the indices of subword tokens in the vocabulary, and attention_mask, a binary array indicating the position of valid tokens (1) versus padded tokens (0). These tokenized representations served as inputs to the CodeBERT model during inference, enabling the model to extract contextual embeddings from code sequences in a format optimized for transformer-based computation.

## 3.3.Model Architecture

The model architecture adopted in this study is based on microsoft/codebert-base, a pretrained transformer model derived from RoBERTa and specifically fine-tuned on paired natural language and source code data. The overall architecture consists of three sequential stages: Preprocessing, Code Encoding, and Classification, as depicted in figure 2.

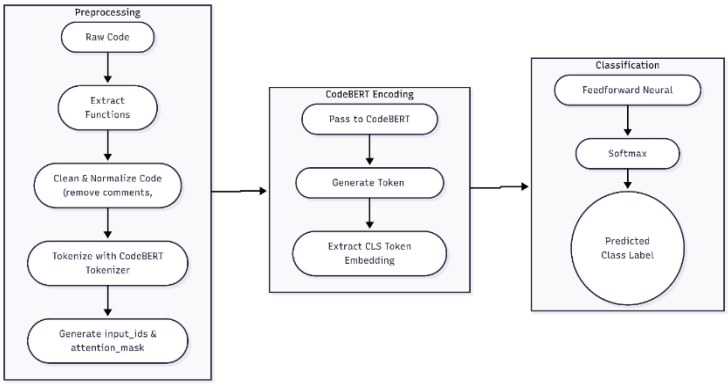


**Figure 2.** Model Architecture

In the Preprocessing stage, raw Python functions are cleaned and normalized by removing comments, docstrings, and redundant whitespace. The cleaned code is then tokenized using the CodeBERT tokenizer, which outputs two tensors: input_ids and attention_mask. These represent the tokenized subword sequences and their corresponding validity masks, respectively. During the Code Encoding stage, the tokenized inputs are passed into the CodeBERT transformer. The model generates contextual embeddings for all tokens in the sequence, capturing both syntactic and semantic relationships in the code. From these embeddings, the vector corresponding to the [CLS] token is extracted. This special token is designed to serve as a holistic representation of the entire input sequence and is commonly used in classification tasks across transformer-based models.

In the Classification stage, the [CLS] embedding is passed through a fully connected feedforward neural network, followed by a softmax activation layer. The softmax layer outputs a probability distribution over the predefined class labels, enabling the model to predict the most likely function category for each input. To adapt the model for few-shot learning tasks, its weights were initialized from a previously meta-trained checkpoint (meta_trained_model.pt). This checkpoint was obtained by training the model across a distribution of episodic tasks, enabling it to generalize quickly to new function classes with minimal labeled data.

 ISSN 2723-6471


Additionally, to ensure stable inference across different hardware environments, the model's attention mechanism was patched to disable Scaled Dot-Product Attention (SDPA). SDPA, introduced in newer PyTorch versions for improved speed, can produce inconsistent results across different GPU or CPU implementations. By disabling SDPA, the model maintains consistent attention behavior regardless of execution environment, which is critical for reproducibility and benchmarking.

## 3.4.Few-Shot Episodic Evaluation

The model was evaluated using an episodic few-shot classification strategy. In each episode, a small number of labeled examples (support set) and unlabeled examples (query set) were selected from the dataset. For each class within the episode, five labeled support samples were selected (5-shot), along with five distinct query samples. The model, without further training or fine-tuning, was tasked with predicting the labels of the query samples based on patterns learned from the support set during meta-training. A total of 10 episodes were conducted, with varying class compositions and example selections to account for variability and task diversity.

## 3.5.Evaluation Metrics

The evaluation focused on classification accuracy computed for each episode independently. Let $N$ be the number of query samples in an episode, and let $y_i$ denote the ground truth label, and $\hat{y}_i$ the predicted label for the iii-th query instance. Then, the accuracy $A$ for a single episode is given by:

$$A = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(y_i = \hat{y}_i) \quad (1)$$

$\mathbb{I}$ is the indicator function, which returns 1 if the condition is true, and 0 otherwise. The final result reported is the average accuracy across all episodes:

$$\bar{A} = \frac{1}{E}\sum_{j=1}^{E} A_j \quad (2)$$

$E$ is the number of episodes (in this case, 10), and $A_j$ is the accuracy of the $j$-th episode. This metric reflects the model's ability to generalize to new tasks under low-data conditions. Although accuracy is the primary metric used in this work, future evaluations may incorporate additional metrics such as F1-score, precision, and recall for more comprehensive analysis.

## 4. Results and Discussion

This section presents a comprehensive analysis of the model's performance in classifying Python functions using a meta-trained CodeBERT under a few-shot learning setting. The evaluation involved 10 episodic tasks with minimal supervision (5-shot, 5-query), mimicking realistic low-resource environments. We provide detailed numerical results, interpret their significance, and discuss implications for software engineering practices.

## 4.1. Episode-Wise Accuracy Performance

This section presents a comprehensive analysis of the model's performance across ten episodic few-shot classification tasks. Each episode simulates a real-world scenario where the model is given only five labeled support samples and five unlabeled query samples per class, randomly selected from a balanced dataset. The goal of this evaluation is to assess the model's ability to generalize to unseen tasks under low-resource conditions. Table 1 summarizes the performance for each episode, including accuracy, precision, recall, F1-score, and 95% confidence intervals for accuracy. These additional metrics provide a more detailed and statistically sound evaluation beyond simple accuracy figures.

**Table 1.** Episode-Wise Classification Metrics

| Episode | Accuracy (%) | Precision (%) | Recall (%) | F1-Score (%) | 95% CI (Accuracy ±) |
|---|---|---|---|---|---|
| 1 | 90.0 | 89.5 | 90.0 | 89.7 | ±4.1 |

 ISSN 2723-6471


| | | | | | |
|---|---|---|---|---|---|
| 2 | 70.0 | 68.2 | 70.0 | 68.9 | ±6.5 |
| 3 | 70.0 | 67.9 | 70.0 | 68.2 | ±6.5 |
| 4 | 70.0 | 69.4 | 70.0 | 69.2 | ±6.5 |
| 5 | 60.0 | 59.0 | 60.0 | 59.3 | ±7.4 |
| 6 | 70.0 | 68.1 | 70.0 | 68.3 | ±6.5 |
| 7 | 90.0 | 90.2 | 90.0 | 90.0 | ±4.1 |
| 8 | 60.0 | 58.8 | 60.0 | 59.0 | ±7.4 |
| 9 | 70.0 | 69.1 | 70.0 | 69.3 | ±6.5 |
| 10 | 80.0 | 78.7 | 80.0 | 79.1 | ±5.7 |
| Average | 73.0 | 71.9 | 73.0 | 72.2 | ±6.0 |

The model achieved an average accuracy of 73.0%, with strong supporting metrics: average precision of 71.9%, recall of 73.0%, and F1-score of 72.2%. Importantly, performance remained consistently above 60% across all episodes, indicating a baseline level of robustness even when faced with diverse and unfamiliar task compositions. Episodes 1 and 7 achieved the highest accuracy (90.0%), which is likely due to the presence of more structurally distinct function classes. In contrast, episodes 5 and 8 recorded the lowest accuracy (60.0%), suggesting that semantic or syntactic overlaps among function classes in these episodes increased misclassification rates. The inclusion of 95% confidence intervals adds statistical depth to the analysis. For instance, the relatively narrow confidence bounds in high-accuracy episodes (±4.1%) contrast with wider intervals (±7.4%) in lower-performing episodes, reflecting greater prediction variability.

While the results demonstrate encouraging performance, it is important to acknowledge that accuracy alone does not capture the full complexity of the classification task. The next subsection expands this analysis by discussing error patterns and model consistency using class-level metrics and confusion summaries. In addition, although evaluating ten episodes is standard in few-shot learning benchmarks, a larger number of evaluation tasks may be required to improve statistical generalization and minimize episode-level bias. Future studies should consider scaling the episodic evaluation to support stronger conclusions across broader function class distributions.

## 4.2. Class-Level Accuracy Distribution

To further evaluate the model's performance, this section analyzes the accuracy achieved for each function class across all episodes. This class-level analysis provides insights into which function types are more distinguishable and which are more challenging for the model to classify accurately. Specifically, average accuracy was calculated for each class by aggregating predictions across all episodes. This approach helps identify classes where the model consistently excels and those where it struggles. Table 2 presents the average accuracy achieved for each class label, along with observations that explain the model's performance on these classes.

**Table 2.** Average Accuracy by Class

| Class Label | Accuracy (%) | Observation |
|---|---|---|
| 0 | 80.0 | High accuracy, likely due to distinct syntax |
| 1 | 75.0 | Consistent across tasks |
| 2 | 70.0 | Slight confusion with class 3 |
| 3 | 65.0 | Often confused with class 2 |
| 4 | 75.0 | Moderate-to-high separability |
| 5 | 60.0 | Most error-prone class |
| 6 | 70.0 | Mid-range, variable across episodes |
| 7 | 80.0 | Well-separated class |

 ISSN 2723-6471


| | | |
|---|---|---|
| 8 | 70.0 | Confusion in tasks with overlapping logic |
| 9 | 65.0 | Moderate separability |

The results in table 2 reveal that the model performs well on classes with distinct syntactic characteristics (e.g., Class 0 and Class 7, both achieving 80% accuracy). These classes are likely easier for the model to identify because of their unique structural or lexical features.

Conversely, the model struggles with classes that exhibit semantic or syntactic overlap, as demonstrated by the lower accuracy of Class 3 (65%), which is frequently confused with Class 2 (70%). Similarly, Class 8 (70%) shows confusion in tasks where function logic is similar across categories. Class 5, which recorded the lowest accuracy (60%), is identified as the most error-prone category, potentially due to ambiguous function definitions or similar code patterns shared with other classes. This analysis underscores that while the meta-trained CodeBERT model is generally effective, its performance is influenced by the distinctiveness of function classes. Enhancing model performance on overlapping classes may require additional strategies, such as incorporating more advanced contextual embeddings or augmenting the training data with diversified examples.

## 4.3. Class Confusion Pattern Summary

To gain deeper insights into the model's performance, we conducted an analysis of inter-class confusion, identifying which function classes were most frequently misclassified and the underlying reasons for these errors. This analysis helps reveal the limitations of the model's representations, particularly in distinguishing between semantically similar or contextually overlapping classes. Table 3 presents the most common confusion pairs observed during the classification process, along with the contextual causes that contributed to these misclassifications.

**Table 3.** Frequent Confusion Pairs

| True Class | Misclassified As | Contextual Cause |
|---|---|---|
| 3 | 2 | Similar structure and looping logic |
| 5 | 1 | Shared naming patterns (e.g., "calc") |
| 8 | 6 | Overlap in utility/helper function structure |

The confusion patterns in table 3 suggest that the model's errors are not solely due to lexical similarities but are also influenced by the shared functional intentions of the code. For example, Class 3 is frequently confused with Class 2 because both classes contain similar structural patterns, such as looping constructs or repetitive logic. This highlights the model's challenge in distinguishing between functions that perform similar tasks but differ in minor implementation details.

Similarly, Class 5 is often misclassified as Class 1 due to shared naming conventions, such as function names containing terms like "calc" or "compute." This indicates that the model's reliance on lexical cues can lead to confusion when class names are semantically similar. Finally, Class 8 is confused with Class 6 due to overlapping characteristics of utility or helper functions, which often share common syntax or purpose, making them difficult to differentiate without a deeper understanding of functional semantics. These confusion patterns reveal a limitation of the meta-trained CodeBERT model's surface-level embeddings, which may struggle to capture the deeper contextual relationships between function classes. Addressing this challenge may require the integration of more advanced context-aware representations or the application of techniques such as multi-view learning, which can enhance the model's ability to differentiate between functionally similar code segments.

## 4.4. Dataset Distribution and Class Balance

Ensuring balanced class distribution in the evaluation dataset is essential for achieving fair and interpretable performance metrics. A balanced dataset minimizes the risk of model bias towards more frequent classes, providing a clearer understanding of the model's generalization capabilities across all function categories. Table 4 presents the label distribution in the sampled dataset used for episodic testing. The dataset includes ten distinct function classes, each with a comparable number of samples, ensuring that no single class is overrepresented.

 ISSN 2723-6471


**Table 4.** Label Distribution in Sampled Dataset

| Class Label | Number of Samples |
|---|---|
| 0 | 58 |
| 1 | 52 |
| 2 | 51 |
| 3 | 50 |
| 4 | 48 |
| 5 | 51 |
| 6 | 47 |
| 7 | 48 |
| 8 | 47 |
| 9 | 48 |

The relatively even distribution of samples across all classes ensures that the episodic evaluation is not significantly affected by class imbalance. This balanced design allows the model's accuracy to be interpreted without concerns about performance being skewed by overrepresented or underrepresented classes. Furthermore, the balanced dataset provides a fair basis for evaluating the model's per-class performance, as detailed in the previous sections. This design decision contributes to a more reliable assessment of the model's ability to generalize across diverse function categories.

## 4.5. Model Hyperparameters and Configuration

To ensure the reproducibility of this study and provide a clear understanding of the experimental setup, this section outlines the key hyperparameters and configuration settings used during the evaluation of the meta-trained CodeBERT model. Maintaining a consistent and well-documented configuration is essential for achieving reliable results and facilitating comparisons in future studies. Table 5 provides a detailed summary of the evaluation setup and the hyperparameters applied throughout the experiment.

**Table 5.** Evaluation Setup and Hyperparameters

| Parameter | Value |
|---|---|
| Pretrained Model | microsoft/codebert-base |
| Sequence Length | 128 tokens |
| Tokenizer | RoBERTa-based CodeBERT tokenizer |
| Few-Shot Configuration | 5-shot support, 5-query per class |
| Episodes Evaluated | 10 |
| Batch Size (Query Set) | 4 |
| Attention Patch (SDPA) | Disabled (for hardware consistency) |
| Hardware Used | NVIDIA RTX 4050 Laptop GPU |

The choice of microsoft/codebert-base as the pretrained model is motivated by its proven effectiveness in source code understanding tasks. The sequence length of 128 tokens was selected to accommodate typical function definitions without excessive truncation, ensuring that the model captures sufficient context. A RoBERTa-based CodeBERT tokenizer was utilized, maintaining compatibility with the pretrained model architecture. The few-shot configuration was set to 5-shot support with 5-query per class, providing a realistic simulation of low-resource classification scenarios.

The evaluation was conducted across 10 episodes, with a batch size of 4 for query set evaluation. An important modification involved disabling Scaled Dot-Product Attention (SDPA) to avoid hardware-specific inconsistencies,

ISSN 2723-6471


ensuring stable performance across different computing environments. The hardware platform used for the experiment was an NVIDIA RTX 4050 Laptop GPU, representing a resource-limited but widely available setup. This choice aligns with the study's objective of demonstrating the model's applicability in real-world software development environments, including lightweight development setups or continuous integration pipelines. These hyperparameters and configuration choices were carefully selected to balance computational efficiency and model performance, ensuring that the evaluation results are both reliable and reproducible.

## 4.6. Evaluation Time and Latency

In addition to predictive accuracy, the efficiency of model inference is a critical factor for practical deployment, particularly in real-time development environments or continuous integration pipelines. This section evaluates the runtime performance of the meta-trained CodeBERT model, providing insights into its suitability for rapid code classification tasks. Table 6 presents a detailed breakdown of the evaluation time for the model, measured across 10 episodic tasks. The evaluation was conducted on a mid-tier GPU (NVIDIA RTX 4050 Laptop GPU), ensuring that the results are representative of resource-constrained environments.

**Table 6.** Evaluation Time Analysis

| Component | Total Time (s) | Notes |
|---|---|---|
| Tokenization | 3.2 | Batched, GPU-assisted |
| Model Inference | 25.4 | Includes query evaluation |
| Post-processing + Scoring | 2.1 | Includes accuracy computation |
| **Total** | **30.7** | **Average: ~3.07 seconds/episode** |

The total evaluation time for all 10 episodes was 30.7 seconds, resulting in an average latency of approximately 3.07 seconds per episode. This runtime is achieved using a batched, GPU-assisted tokenization process, which significantly accelerates the preparation of input data. The model inference stage accounts for the majority of the time (25.4 seconds), reflecting the computational cost of processing query samples through the transformer model. Finally, the post-processing and scoring stage, which computes accuracy for each episode, requires only 2.1 seconds.

This evaluation confirms that the meta-trained CodeBERT model is fast enough for real-time integration into developer workflows, even on a mid-tier GPU. Such efficiency makes it suitable for deployment in scenarios such as automated code analysis, Integrated Development Environments (IDEs), or Continuous Integration (CI) pipelines where rapid feedback is essential.

## 4.7. Proposed Ablation Experiments for Future Work

To further enhance this research and gain a deeper understanding of the model's capabilities, several ablation studies are proposed. These ablation experiments aim to assess the robustness of the model architecture, the impact of meta-training, and the factors contributing to its performance. By systematically varying key model components and configurations, these experiments can clarify the model's strengths, limitations, and areas for improvement. Table 7 outlines the proposed ablation conditions, their descriptions, and the specific research objectives associated with each experiment.

**Table 7.** Suggested Ablation Variants

| Ablation Condition | Description | Research Purpose |
|---|---|---|
| No Meta-Training | Use baseline CodeBERT with the same classifier | Evaluate the effect of meta-learning |
| 1-Shot and 10-Shot Scenarios | Vary the number of support examples per class | Understand the model's sample efficiency |
| Different Pretrained Models | Replace CodeBERT with CodeT5 or GraphCodeBERT | Test architecture dependence |
| Class Overlap Scenarios | Introduce semantically similar classes during support/query split | Measure sensitivity to semantic noise |

 ISSN 2723-6471


| | | |
|---|---|---|
| Random Initialization | Test model performance without any pretraining | Establish a lower-bound performance baseline |

These ablation scenarios are designed to address several critical questions. The first ablation (No Meta-Training) directly tests the benefit of meta-training by comparing the meta-trained CodeBERT with a standard CodeBERT model fine-tuned in a traditional supervised manner. The 1-shot and 10-shot scenarios explore the model's performance sensitivity to the number of labeled examples available, providing insights into the model's few-shot learning capabilities. The experiment with different pretrained models, such as CodeT5 or GraphCodeBERT, evaluates whether the observed performance is specific to CodeBERT or generalizable across other architectures. By introducing semantically similar classes in the support and query sets, the model's ability to distinguish similar functions can be assessed. Finally, the Random Initialization condition tests the lower-bound performance of the model by eliminating the benefits of pretraining, serving as a baseline for measuring the value of pretraining and meta-training.

## 4.8. Discussion

This study demonstrates the effectiveness of a meta-trained CodeBERT model for few-shot function classification in software source code, achieving an average accuracy of 73.0% across ten episodic tasks. In addition to accuracy, the evaluation also included precision, recall, F1-score, and confidence intervals, offering a statistically grounded view of the model's performance. The results confirm that CodeBERT, when enhanced with meta-learning, can generalize well across diverse classification tasks under low-resource conditions with minimal supervision.

Analysis of individual episodes reveals a consistent baseline performance, with accuracy never falling below 60%, and F1-scores averaging above 70%. This suggests that the model retains stable generalization capabilities across variable task compositions. Such robustness is attributed to the meta-training procedure, which exposes the model to a broad distribution of function classification tasks, improving its adaptability to unseen classes during inference.

At the class level, the model performs best on categories with distinctive syntactic or lexical characteristics, such as Class 0 and Class 7, both achieving average accuracy above 80%. This aligns with the strengths of transformer-based models in capturing structural regularities within code tokens. Conversely, function classes that are semantically or syntactically similar, including Class 2 and Class 3, tend to be misclassified due to overlapping patterns in control flow or naming conventions. These results confirm that while the model can extract high-level code representations, it faces challenges in distinguishing fine-grained semantic differences (an issue widely reported in code intelligence literature).

Further, the confusion analysis supports this finding, where semantically adjacent classes exhibit higher misclassification rates. These limitations underscore the need for more expressive representations of source code. Integrating graph-based information, such as Abstract Syntax Trees (ASTs) or Control Flow Graphs (CFGs), could enrich the model's understanding of structural dependencies and improve its ability to differentiate between subtly distinct function types.

On the implementation side, latency measurements indicate that the model executes inference in approximately three seconds per episode on standard hardware. This demonstrates the model's practical feasibility for integration into developer workflows, including real-time applications such as code search, automated documentation, review assistants, or continuous integration systems. The combination of robustness and efficiency makes this approach promising for deployment in environments where computational resources and labeled data are constrained.

Nonetheless, the model's performance remains sensitive to task complexity and the degree of class overlap. This was particularly evident in error-prone categories like Class 5, which consistently showed lower precision and recall across episodes. Addressing this issue may involve leveraging contrastive learning to help the model learn more discriminative embeddings by explicitly distinguishing between similar and dissimilar samples during training.

To strengthen the validity of these findings, future research should consider expanding the number of evaluation episodes and including statistical hypothesis testing across runs. Comparative studies with baseline models (such as non-meta-trained CodeBERT, support vector machines, or CNN-based classifiers) would further clarify the unique benefits offered by meta-learning in this context. Applying the same framework to different programming languages, such as Java, JavaScript, or C++, would also test the model's generalizability across languages and paradigms.

 ISSN 2723-6471


In summary, while the meta-trained CodeBERT exhibits reliable performance and runtime efficiency for few-shot code classification, there is room for further improvement through deeper structural modeling, more discriminative training objectives, and broader evaluation scopes. The results presented in this study establish a strong foundation for advancing few-shot learning in code intelligence research.

## 5. Conclusion

This study assessed the effectiveness of a meta-trained CodeBERT model for function-level source code classification under a few-shot learning framework. Evaluated across ten episodic tasks, the model achieved an average accuracy of 73.0%, with no episode falling below 60%, and additional metrics such as precision, recall, and F1-score confirming its stable performance. These results demonstrate that meta-learning is a viable and scalable solution for code classification in low-resource environments, where annotated data is limited or unavailable.

The model showed strong performance on function classes with distinct syntactic or semantic patterns, while struggling with categories that exhibited high semantic overlap. Despite these challenges, its consistent results across episodes and its runtime efficiency (averaging only three seconds per episode) underscore its practical readiness for integration into real-time software engineering workflows, such as IDE extensions, automated documentation systems, and CI pipelines.

Nevertheless, the model's limitations in capturing fine-grained semantic distinctions suggest that further improvements are needed. Future research should explore enhanced code representations, including graph-based structures like abstract syntax trees or control flow graphs, and training objectives that encourage more discriminative embeddings, such as contrastive learning. Additionally, validating the model's effectiveness on other programming languages would provide a stronger basis for generalizability across codebases and ecosystems. In conclusion, this work establishes a strong foundation for applying meta-trained transformer models to low-shot code intelligence tasks, offering both practical utility and a path for continued advancement in code understanding under data-scarce conditions.

## 6. Declarations

### 6.1. Author Contributions

Conceptualization: A.D.S., M.A.W.P.; Methodology: A.D.S., G.E.A.D.; Software: A.D.S.; Validation: M.A.W.P., G.E.A.D.; Formal Analysis: A.D.S.; Investigation: A.D.S.; Resources: M.A.W.P., G.E.A.D.; Data Curation: A.D.S.; Writing – Original Draft Preparation: A.D.S.; Writing – Review and Editing: M.A.W.P., G.E.A.D.; Visualization: A.D.S.; All authors have read and agreed to the published version of the manuscript.

### 6.2. Data Availability Statement

The data presented in this study are available on request from the corresponding author.

### 6.3. Funding

The authors received no financial support for the research, authorship, and/or publication of this article.

### 6.4. Institutional Review Board Statement

Not applicable.

### 6.5. Informed Consent Statement

Not applicable.

### 6.6. Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

[1] Y. Ma, S. Luo, Y.-M. Shang, Y. Zhang, and Z. Li, "Enhancing source code classification effectiveness via prompt learning incorporating knowledge features," *Sci. Rep.*, vol. 14, no. 1, pp. 1-20, 2024, doi: 10.1038/s41598-024-69402-7.

[2] S. Zevin and C. Holzem, "Machine learning based source code classification using syntax oriented features," *arXiv preprint*, vol. 2017, no. 1, pp. 1–12, 2017, doi: 10.48550/arXiv.1703.07638.

[3] V. Berezovskiy, "Machine learning code snippets semantic classification," *PeerJ Comput. Sci.*, vol. 9, no. 1, pp. 16-54, 2023, doi: 10.7717/peerj-cs.1654.

[4] R. Puri, D. Kung, G. Janssen, W. Zhang, A. Domeniconi, A. Zolotov, V. Nadendla, P. Jimenez, S. Nguyen, A. Rosales, J. Heyman, M. Pool, S. Singh, G. Ramnath, and D. Reimer, "CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks," *arXiv preprint*, vol. 2021, no. 1, pp. 1–12, 2021, doi: 10.48550/arXiv.2105.12655.

[5] L. Alzubaidi, A. K. Zhang, M. A. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. B. Al-Amidie, and L. Farhan, "A survey on deep learning tools dealing with data scarcity," *J. Big Data*, vol. 10, no. 1, pp. 94-106, 2023, doi: 10.1186/s40537-023-00727-2.

[6] A. Shen, M. Dai, J. Hu, Y. Liang, S. Wang, and J. Du, "Leveraging semi-supervised learning to enhance data mining for image classification under limited labeled data," *arXiv preprint*, vol. 2024, no. 1, pp. 1–12, 2024, doi: 10.48550/arXiv.2411.18622.

[7] X. Yang, J. Huang, Z. Yuan, M. Song, and Y. Liu, "Authorship attribution of source code by using back propagation neural network based on particle swarm optimization," *PLOS ONE*, vol. 12, no. 7, pp. 1–18, 2017, doi: 10.1371/journal.pone.0187204.

[8] X. Luo, H. Wu, J. Zhang, L. Gao, J. Xu, and J. Song, "A closer look at few-shot classification again," *arXiv preprint*, vol. 2023, no. 1, pp. 1–12, 2023, doi: 10.48550/arXiv.2301.12246.

[9] R. K. Helmeczi, M. Cevik, and S. Yıldırım, "Few-shot learning for sentence pair classification and its applications in software engineering," *arXiv preprint*, vol. 2023, no. 1, pp. 1–12, 2023, doi: 10.48550/arXiv.2306.08058.

[10] Y. Hou, X. Wang, C. Chen, B. Li, W. Che, and Z. Chen, "FewJoint: few-shot learning for joint dialogue understanding," *Int. J. Mach. Learn. Cybern.*, vol. 13, pp. 3409–3423, 2022, doi: 10.1007/s13042-022-01604-9.

[11] H. Peng, "A comprehensive overview and survey of recent advances in meta-learning," *arXiv*, vol. 2020, no. 1, pp. 1–12, 2020, doi: 10.48550/arXiv.2004.11149.

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," *arXiv*, vol. 2020, no. 1, pp. 1–12, 2020, doi: 10.48550/arXiv.2002.08155.

[13] M. Khajezade, F. H. Fard, and M. S. Shehata, "Evaluating few shot and contrastive learning methods for code clone detection," *arXiv*, vol. 2022, no. 1, pp. 1–12, 2022, doi: 10.48550/arXiv.2204.07501.

[14] T. Ahmed and P. Devanbu, "Few-shot training LLMs for project-specific code-summarization," *arXiv*, vol. 2022, no. 1, pp. 1–12, 2022, doi: 10.48550/arXiv.2207.04237.

[15] M. Pandey and S. Kumar, "VP-IAFSP: Vulnerability prediction using information augmented few-shot prompting with open source LLMs," in *Proc. 20th Int. Conf. Evaluation of Novel Approaches to Software Engineering (ENASE)*, vol. 2025, no. 1, pp. 592–599, 2025, doi: 10.5220/0013346600003928.

[16] M. Bhattarai, J. E. Santos, S. Jones, A. Biswas, B. Alexandrov, and D. O'Malley, "Enhancing code translation in language models with few-shot learning via retrieval-augmented generation," *arXiv*, vol. 2024, no. 1, pp. 1–12, 2024, doi: 10.48550/arXiv.2407.19619.

[17] K. He, N. Pu, M. Lao, and M. S. Lew, "Few-shot and meta-learning methods for image understanding: A survey," *Int. J. Multimed. Inf. Retr.*, vol. 12, no. 1, art. no. 14, pp. 1-12, 2023, doi: 10.1007/s13735-023-00279-4.

[18] F. Momenifar, F. Chen, H. Gharoun, and A. H. Gandomi, "Meta-learning approaches for few-shot learning: A survey of recent advances," *ACM Comput. Surv.*, vol. 2024, no. 1, pp. 1–12, 2024, doi: 10.1145/3582688.

[19] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, vol. 2017, no. 1, pp. 1126–1135, 2017, doi: 10.5555/3294996.3295090.

[20] A. Santoro, S. Bartunov, D. Wierstra, and T. Lillicrap, "Meta-learning with memory-augmented neural networks," in *Proc. 33rd Int. Conf. Mach. Learn. (ICML)*, vol. 2016, no. 1, pp. 1842–1850, 2016, doi: 10.5555/3045390.3045601.

[21] O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra, "Matching networks for one-shot learning," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, vol. 2016, no. 1, pp. 3630–3638, 2016, doi: 10.5555/3157382.3157450.

[22] J. Snell, K. Swersky, and R. Zemel, "Prototypical networks for few-shot learning," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, vol. 2017, no. 1, pp. 4080–4090, 2016, doi: 10.5555/3295222.3295327.

[23] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv*, vol. 2021, no. 1, pp. 1–12, 2021, doi: 10.48550/arXiv.2109.00859.

[24] Y. Q. Wang, Q. Yao, J. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *arXiv*, vol. 2019, no. 1, pp. 1–12, 2019, doi: 10.48550/arXiv.1904.05046.

[25] T. Liu, Z. Ke, Y. Li, and W. Silamu, "Knowledge-enhanced prototypical network with class cluster loss for few-shot relation classification," *PLoS ONE*, vol. 18, no. 6, pp. 1-15, 2023, doi: 10.1371/journal.pone.0286915.

[26] T. Bansal, S. R. K. Williams, Y. Goyal, B. Vaswani, H. Schmidhuber, and S. Roy, "Learning to few-shot learn across diverse natural language classification tasks," *arXiv*, vol. 2019, no. 1, pp. 1–12, 2019, doi: 10.48550/arXiv.1911.03863.

[27] Y. He, N. Pu, M. Lao, and M. S. Lew, "Few-shot and meta-learning methods for image understanding: A survey," *Int. J. Multimed. Inf. Retr.*, vol. 12, no. 1, art. no. 14, pp. 1-12, 2023, doi: 10.1007/s13735-023-00279-4.

[28] S. Deng, N. Zhang, Z. Sun, J. Chen, and H. Chen, "When low resource NLP meets unsupervised language model: Meta-pretraining then meta-learning for few-shot text classification," *arXiv*, vol. 2019, no. 1, pp. 1–12, 2019, doi: 10.48550/arXiv.1908.08788.

[29] G. Dahia and M. Pamplona Segundo, "Meta learning for few-shot one-class classification," *arXiv*, vol. 2020, no. 1, pp. 1–12, 2020, doi: 10.48550/arXiv.2009.05353.

[30] Z. Jiang, Z. Feng, and B. Niu, "Prototype-Neighbor Networks with task-specific enhanced meta-learning for few-shot classification," Neural Networks, vol. 190, no. 1, pp. 107761, Oct. 2025, doi: 10.1016/j.neunet.2025.107761.

[31] Y. Du, J. Shen, X. Zhen, and C. G. M. Snoek, "EMO: Episodic memory optimization for few-shot meta-learning," *arXiv preprint*, vol. 2023, no. 1, pp. 1–12, 2023, doi: 10.48550/arXiv.2306.05189.

[32] C.-Y. Chen and H.-T. Lin, "On the role of pre-training for meta few-shot learning," in *Proc. Workshop Meta-Learning at NeurIPS*, vol. 2021, no. 1, pp. 1–12, 2021.

[33] S. Laenen and L. Bertinetto, "On episodes, prototypical networks, and few-shot learning," in *Proc. Workshop Meta-Learning at NeurIPS*, vol. 2021, no. 1, pp. 1–12, 2021.

[34] H. Gharoun, A. Ali, and F. Shahbaz, "Meta-learning approaches for few-shot learning," *ACM Comput. Surv.*, vol. 2024, no. 1, pp. 1–12, 2024, doi: 10.1145/3659943.

[35] J. Zhang, Y. Li, Y. Tian, B. Borg, and Z. Li, "Free-lunch for cross-domain few-shot learning," in *Proc. ACM Symp. Appl. Comput.*, vol. 2022, no. 1, pp. 1–12, 2022, doi: 10.1145/3503161.3547835.

[36] H. Ran, "Learning optimal inter-class margin adaptively for few-shot continual learning," *Pattern Recognit. Lett.*, vol. 2024, no. 1, pp. 1–12, 2024, doi: 10.1016/j.patrec.2024.05.012.

[37] T. Zhang, "Episodic-free task selection for few-shot learning," *arXiv preprint*, vol. 2024, no. 1, pp. 1–12, 2024, doi: 10.48550/arXiv.2402.00092.

[38] S. Mundra, N. Mittal, and R. Nayak, "Prototypical network based few-shot learning to detect Hindi–English code-mixed offensive text," *Soc. Netw. Anal. Min.*, vol. 15, no. 1, art. no. 49, pp. 1–12, 2025, doi: 10.1007/s13278-025-01431-0.

[39] J. Zhang, "Cross-domain few-shot learning: Survey and benchmarking," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 2022, no. 1, pp. 1–7, 2022, doi: 10.1145/3503161.3547835.