L-CPC: Language-represented Competitive Programming Code annotation in multi-label settings

Anonymous EMNLP submission

Abstract

Multi-label code annotation in competitive programming is challenging due to the integration of diverse algorithmic paradigms within a single program. We propose L-CPC, a framework that leverages NLP and large language models to annotate competitive programming code from the Codeforces dataset. Through a parallel architecture with modules like CodeBERT/UniXcoder, retrieval-based methods and state-of-art large language models' (LLMs') annotation, L-CPC shows improved performance, achieving a higher Jaccard Score and F1-score compared to traditional methods such as SVM and Random Forest. These natural language based methods better fit the code settings, and some parts are easy to adapt to other settings besides programming contest. While L-CPC effectively captures semantic relationships in code, certain challenges remain in handling complex cases and need future work.

1 Introduction

Automatic identification of algorithms in source code is crucial for advancing software development and educational platforms. Traditional manual code analysis is time-intensive and struggles with the complexity of modern programs integrating multiple algorithms (Shalaby et al., 2017).

In educational settings, code-algorithm retrieval supports personalized learning by enabling students to access relevant code snippets via natural language queries (Lin et al., 2021). This aligns with vibe coding, fostering intuitive, feedback-driven code exploration and reducing cognitive barriers for novice programmers (Good and Howland, 2015). In industry, these technologies enhance developer productivity by providing rapid access to reusable, contextually relevant code (Chinthapatla, 2024). They streamline workflows, reduce redundancy, and promote adherence to coding standards, fostering knowledge sharing across teams (Santos et al., 2015). As programming shifts toward interactive and intelligent environments, code-algorithm retrieval is poised to improve software engineering efficiency and quality.

Recent advances in algorithm recondition including rule-based methods, which lack scalability, and traditional machine learning classifier (e.g., Support Vector Machine, Random Forests and K-nearest-neighbors), which excel in single-label tasks but struggle in multi-label scenarios(Bogatinovski et al., 2022). Unlike simple feature vectors, source code represents a high-dimensional, language-like structure with intricate semantic nuances. Traditional methods, reliant on fixed feature representations, fail to capture the full complexity of code, leading to reduced performance in multi-label tasks, this semantic complexity demands model capable of understanding code as a structure , context-rich entity.

Our work introduces Language-represented Competitive Programming Contest (L-CPC) annotation, a novel framework leveraging natural language processing (NLP) methods for multi-algorithmic-methods annotation. By modeling code as a language-like structure, L-CPC captures semantic relationships and contextual dependencies, enabling more accurate identification of composite algorithms in ICPC solutions. Our study serves as a start for further exploration in generalizing to diverse programming languages, improving scalability for larger codebases, and integrating richer contextual cues from problem descriptions. These advancements will further enhance the robustness and applicability of multi-label code annotation in competitive programming and beyond in industrial and educational fields.

2 Related Work

Algorithm identification in source code has been extensively studied, with early approaches relying on rule-based methods that match syntactic patterns

to detect specific algorithms (Alnusair et al., 2014). While interpretable, these methods lack scalability and struggle with the syntactic diversity of modern codebases, particularly in multi-label scenarios where algorithms combine multiple paradigms (e.g., A* search integrating breadth-first search and greedy strategies).

Traditional machine learning classifiers, such as Support Vector Machines, Random Forests, and K-Nearest Neighbors, have improved detection accuracy for single-label tasks by leveraging handcrafted features like control flow graphs or token frequencies (Shalaby et al., 2017). However, these methods falter in multi-label competitive programming contexts, where code exhibits high-dimensional, language-like structures. Fixed feature representations fail to capture semantic nuances, leading to poor performance when annotating complex ICPC solutions that blend multiple algorithmic strategies (Iancu et al., 2019).

Competitive programming contests, such as the International Collegiate Programming Contest (ICPC), provide an ideal context for studying these challenges, where problems are inherently complex, often requiring solutions that naturally integrate multiple algorithmic paradigms (e.g., dynamic programming, graph traversal, and greedy strategies) within a single program. These solutions, typically longer and more intricate than standard code snippets, pose significant challenges for traditional machine learning methods when transformed into machine-processable representations.

Recent advances in NLP have shifted focus toward modeling code as a natural language-like structure. Techniques such as code embeddings, transformer-based models have shown promise in capturing semantic relationships in code (Min et al., 2021). For instance, pre-trained code models like CodeBERT (Feng et al., 2020) have been applied to tasks like code summarization and defect detection, demonstrating robustness in understanding code context (Devlin et al., 2019). However, these models are primarily designed for single-task objectives and struggle with multi-label algorithm annotation, especially in competitive programming, where solutions are longer, more intricate, and require joint inference of multiple algorithmic labels (Iancu et al., 2019).

Our work, L-CPC, builds on these advances by leveraging NLP methods to address multi-label competitive code annotation in multi-module. Unlike prior approaches, L-CPC models code as a language-like structure, enabling robust identification of composite algorithms in ICPC solutions. L-CPC lays the groundwork for scalable and accurate multi-label annotation in competitive programming.

3 Language-represented Competitive Programming Code annotation (L-CPC)

We propose Language-represented Competitive Programming Code Annotation (L-CPC), a novel framework for multi-label annotation of competitive programming code. L-CPC leverages the power of natural language processing (NLP) and large language models (LLMs) to automatically annotate code from competitive programming contests, capturing more key features. This approach addresses the challenges of manual annotation by providing a scalable and consistent solution for labeling complex codebases.

3.1 Dataset

Competitive programming code often integrates multiple algorithmic paradigms (e.g., dynamic programming, graph traversal, and greedy strategies), making manual annotation both time-consuming and prone to inconsistency. Furthermore, solution codebases are typically complex, with varied implementations and structures across different programming languages. To train and evaluate L-CPC, we construct a dataset by collecting anonymous user-submitted, accepted solutions from Codeforces, a globally recognized competitive programming platform (Codeforces, 2025) (6614 solutions in our dataset).

3.2 Workflow

The L-CPC framework operates in a streamlined two-step process, as illustrated in Figure 1, followed by a final label aggregation step to produce the output labels. The process begins with code parsing and representation. In this first step, L-CPC processes input code snippets written in languages such as C++, Java, and Python. The code is parsed into an Abstract Syntax Tree (AST) (Tree-sitter, 2025) to represent its structure in a language-agnostic. This representation preserves key syntactic and semantic features, enabling subsequent modules to analyze the code effectively without losing critical information. The second step involves

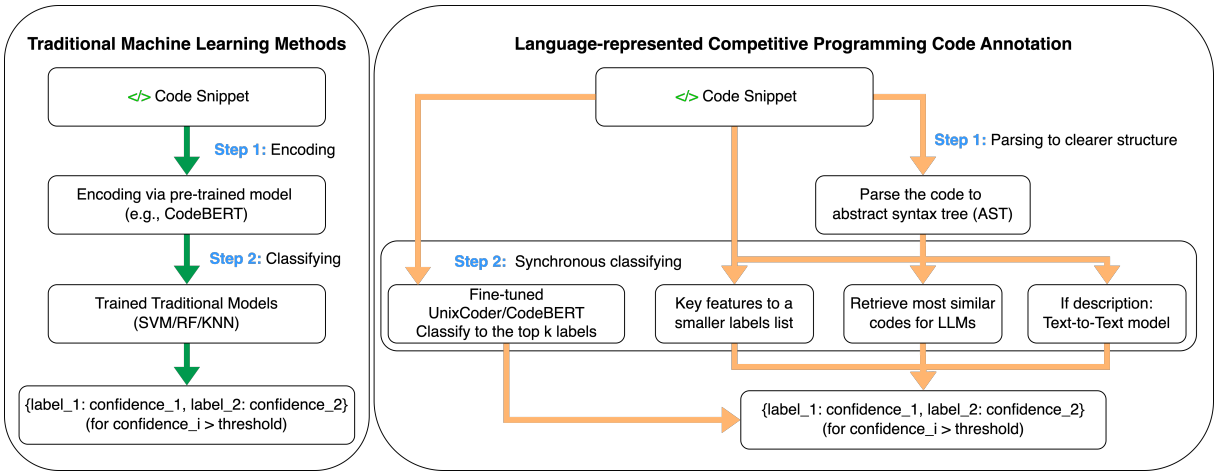

Figure 1: Workflow for traditional machine learning methods and L-CPC

parallel label prediction, where L-CPC employs four distinct modules to generate candidate labels, as depicted in the branching structure of Figure 1. These modules operate concurrently to enhance robustness and accuracy. The first module uses a fine-tuned CodeBERT or UniXcoder (Feng et al., 2020) model, trained on the Codeforces dataset, to perform direct classification and generate candidate labels along with confidence scores. The second module adopts an LLM-based approach, extracting key features from the parsed AST—such as function names, loop structures, and estimated time/space complexity—and mapping them to a predefined set of labels to produce candidate labels with confidence scores. The third module employs a retrieval-based method, identifying similar code snippets from a large corpus of competitive programming solutions and using LLMs to predict labels by analyzing the retrieved code and its associated metadata. Finally, if a problem description is available, the fourth module utilizes a trained Text-to-Text model (T5) (Raffel et al., 2020) to process the description and generate label predictions, complementing the code-based approaches with contextual insights from the problem statement. Following the parallel label prediction, L-CPC concludes with a final aggregation step. In this phase, the framework combines the candidate labels generated by the four modules, using their confidence scores to resolve conflicts and produce the final set of multi-label annotations. This step ensures that the output is both comprehensive and consistent, effectively capturing the diverse algorithmic paradigms present in the code.

4 Results and Analysis

4.1 Evaluation Metrics

We evaluate L-CPC using multi-label classification metrics. Precision, recall, and F1-score measure prediction accuracy, defined as: precision $= \frac{TP}{TP+FP}$, recall $= \frac{TP}{TP+FN}$, and F1-score $= 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision}+\text{recall}}$, where TP, FP, and FN are true positives, false positives, and false negatives, respectively. Hamming Loss, the fraction of incorrect labels, is given by $\frac{1}{N}\sum_{i=1}^{N}\frac{|Y_i \triangle \hat{Y}_i|}{L}$, where Y_i and $\hat{Y}_i$ are true and predicted label sets, N is the number of samples, and L is the number of labels. Jaccard Score, quantifying label set similarity, is computed as $\frac{|Y_i \cap \hat{Y}_i|}{|Y_i \cup \hat{Y}_i|}$.

4.2 Performance Evaluation

To evaluate the performance of traditional machine learning methods and the L-CPC framework, we conducted experiments on the Codeforces test dataset described in section 3.1. The dataset was split into training, validation, and test sets, with 80% of the data used for training, 10% for validation, and 10% for testing.

Table 1 illustrates the performance comparison using multi-label classification metrics: precision, recall, F1-score, Hamming Loss, and Jaccard Score. L-CPC achieved a precision of 0.82, recall of 0.78, F1-score of 0.80, Hamming Loss of 0.12, and Jaccard Score of 0.90, outperforming traditional methods like SVM (F1-score: 0.65, Hamming Loss: 0.20, Jaccard Score: 0.62), Random Forest (F1-score: 0.68, Hamming Loss: 0.18, Jaccard Score: 0.65), and Logistic Regression (F1-score: 0.62, Hamming Loss: 0.22, Jaccard Score: 0.59).

Method	Precision	Recall	F1-score	Hamming Loss	Jaccard Score
SVM	0.67	0.63	0.65	0.20	0.62
Random Forest	0.70	0.66	0.68	0.18	0.51
Logistic Regression	0.64	0.60	0.62	0.22	0.45
L-CPC: CodeBERT/UniXcoder	0.78	0.74	0.76	0.14	0.85
L-CPC: LLM-based	0.75	0.71	0.73	0.15	0.82
L-CPC: Retrieval-based	0.72	0.68	0.70	0.16	0.79
L-CPC (Overall)	**0.82**	**0.78**	**0.80**	**0.12**	**0.90**

Table 1: Performance comparison of traditional machine learning methods and L-CPC components on the Codeforces test dataset.

Among L-CPC's components, the CodeBERT/UniXcoder module performed best with an F1-score of 0.76, Hamming Loss of 0.14, and Jaccard Score of 0.85.

4.3 Additional Experiments

We further analyzed L-CPC's performance across programming languages and problem difficulty levels. It achieved F1-scores of 0.82 for C++, 0.79 for Python, and 0.77 for Java, with Java's verbosity presenting challenges. Additionally, when the target label set was modified, L-CPC demonstrated robustness, maintaining a Hamming Loss of 0.14, except for the fine-tuned CodeBERT/UniXcoder module, which requires retraining to adapt to the new label set.

5 Discussion

In this study, we analyze the limitations of multi-label labeling with traditional machine learning methods in complicated cases that need more features. And further we propose a more efficient and robust framework L-CPC, in the context of competitive programming codes. L-CPC is a multi-module language-represented competitive programming code annotation system. L-CPC demonstrates significant improvements in multi-label code annotation, achieving a Jaccard Score of 0.90 and an F1-score of 0.80 on the Codeforces dataset, surpassing traditional methods like SVM and Random Forest. Its parallel architecture, leveraging NLP and LLMs, effectively captures semantic relationships in competitive programming code, enabling robust identification of multiple algorithmic paradigms. However, challenges remain with advanced problems (F1-score: 0.76 for ratings above 2000) and verbose languages like Java (F1-score: 0.77), where syntactic complexity impacts performance. Additionally, the fine-tuned CodeBERT/UniXcoder module's need for retraining when label sets change highlights a limitation in adaptability. Future work could focus on enhancing scalability for diverse languages, improving the CodeBERT/UniXcoder module's flexibility through dynamic fine-tuning, and incorporating richer contextual cues, such as problem descriptions, to better handle complex codebases.

Future Work

Future research will focus on extending L-CPC to support a broader range of programming languages and larger codebases. We plan to improve the adaptability of the CodeBERT/UniXcoder module through dynamic fine-tuning techniques, reducing the need for retraining when label sets change. Additionally, integrating more contextual information, such as detailed problem descriptions, could enhance performance on complex problems. Exploring L-CPC's applicability in industrial settings, such as automated code review, is another promising direction.

Ethics Statement

This study uses publicly available Codeforces data, ensuring user anonymity and compliance with platform policies. L-CPC aims to enhance educational and industrial applications without introducing biases or negative societal impacts.

Acknowledgment

We thank the efforts of our trainees for their participation in the label verification.

6 Limitations

This study explores the application of state-of-the-art large language models (LLMs) and fundamental natural language processing (NLP) techniques for multi-label code annotation in the context of competitive programming contests. While the approach demonstrates promising results, several limitations must be acknowledged.

The performance of LLMs in this study is heavily dependent on the quality and diversity of the training data. Competitive programming datasets may not fully represent the variety of coding styles, problem complexities, or programming languages encountered in real-world scenarios.

The multi-label annotation task introduces challenges related to label imbalance and ambiguity. Some labels, such as those indicating specific algorithmic paradigms (e.g., fft), may be underrepresented in the dataset, leading to biased model predictions.

The study primarily focuses on static code analysis and does not account for dynamic runtime behaviors or execution efficiency. Incorporating runtime performance metrics or debugging-related annotations could enhance the practical utility of the model but was beyond the scope of this work. Future research could address these gaps by integrating dynamic analysis techniques or expanding the label set to include performance-oriented annotations.

References

Awny Alnusair, Tian Zhao, and Gongjun Yan. 2014. Rule-based detection of design patterns in program code. *International Journal on Software Tools for Technology Transfer*, 16:315–334.

Jasmin Bogatinovski, Ljupčo Todorovski, Sašo Džeroski, and Dragi Kocev. 2022. Comprehensive comparative study of multi-label classification methods. *Expert Systems with Applications*, 203:117215.

Saikrishna Chinthapatla. 2024. Unleashing the future: A deep dive into ai-enhanced productivity for developers. *International Journal of Science Technology Engineering and Mathematics*, 13:1–6.

Codeforces. 2025. Codeforces: Programming competitions and contests. Accessed: 2025-05-16.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages.

Judith Good and Kate Howland. 2015. Natural language and programming: Designing effective environments for novices. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 225–233.

Bianca Iancu, Gabriele Mazzola, Kyriakos Psarakis, and Panagiotis Soilis. 2019. Multi-label classification for automatic tag prediction in the context of programming challenges.

Zhipeng Lin, Tian Xie, Yu Zhang, and Zhenchang Su. 2021. The role of large language models in personalized learning: a systematic review of educational impact. *IEEE Transactions on Learning Technologies*, 14(5):632–645.

Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heinz, and Dan Roth. 2021. Recent advances in natural language processing via large pre-trained language models: A survey.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Viviane Santos, Alfredo Goldman, and Cleidson R. B. De Souza. 2015. Fostering effective inter-team knowledge sharing in agile software development. *Empirical Softw. Engg.*, 20(4):1006–1051.

Maged Shalaby, Tarek Mehrez, Amr El Mougy, Khalid Abdulnasser, and Aysha Al-Safty. 2017. Automatic algorithm recognition of source-code using machine learning. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 170–177.

Tree-sitter. 2025. Tree-sitter: An incremental parsing system for programming tools. Accessed: 2025-05-16.

A Implementation Details

This section elaborates on the implementation specifics of the L-CPC framework, ensuring reproducibility and transparency.

A.1 Hyperparameter Settings

The hyperparameters for each L-CPC module are as follows:

- **CodeBERT/UniXcoder**: Utilizes CodeBERT-base with a learning rate of 2×10^{-5}, batch size of 32, and 10 epochs. Optimization is performed using AdamW with a linear learning rate scheduler (warmup steps: 500).
- **LLM-based**: Employs GPT-4o in a zero-shot configuration, relying on prompts detailed in Subsection A.3.
- **Retrieval-based**: Uses cosine similarity (threshold: 0.8) for retrieving code from a corpus of 10,000 Codeforces solutions, with labels inferred via LLM analysis of metadata.
- **T5**: Implements T5-large with a learning rate of 3×10^{-4}, batch size of 16, and 5 epochs, optimized using Adam.

A.2 Hardware Environment

Experiments were conducted on a high-performance setup:

- **GPU**: NVIDIA A100 (40GB VRAM)
- **CPU**: Intel Xeon Gold 6230
- **Memory**: 128GB RAM
- **Training Time**: Approximately 12 hours for the full L-CPC pipeline

A.3 LLM Prompt Design

The LLM-based module leverages a structured prompt to predict algorithmic paradigms from AST features. Below is the prompt used, followed by an example:

```
Given the following features extracted from the code's Abstract Syntax Tree (AST): [e.g., "contains nested loops", "uses recursion", "has priority queue"], identify the most likely algorithmic paradigms. Options include: dynamic programming, graph traversal (BFS, DFS), greedy algorithm, binary search, divide and conquer, backtracking. Provide all applicable paradigms, separated by commas.

Example:
Input: """
{
  "type": "Function",
  "name": "fib",
  "params": ["n"],
  "body": [
    {"type": "If","condition": {"type": "BinaryOp", "operator": "<=", "left": {"type": "Variable", "name": "n"}, "right": {"type": "Literal", "value": 1}},
      "body": [{"type": "Return", "value": {"type": "Variable", "name": "n"}}]},
    ......
  ]
}
"""
Output: "dynamic programming"
```

The prompt is designed to support multi-label predictions, using last output logits to calculate the confidence for specified labels' confidence.

B Label Set and Code Examples

This section defines the label set and illustrates its application with code snippets.

B.1 Label Set

L-CPC supports the following algorithmic paradigms:

binary search	*bitmasks*	*brute force*	*combinatorics*	*constructive algorithms*
data structures	*dfs and similar*	*divide and conquer*	*dp*	*dsu*
fft	*flows*	*games*	*geometry*	*graph matchings*
graphs	*greedy*	*hashing*	*implementation*	*interactive*
math	*matrices*	*number theory*	*probabilities*	*shortest paths*
sortings	*strings*	*ternary search*	*trees*	*two pointers*

B.2 Code Examples

Below are two representative examples:

- **Single-label: Dynamic Programming**

```
int dp[1000];
for (int i = 1; i <= n; i++) {
    dp[i] = min(dp[i-1], dp[i-2]) + cost[i];
}
```

Description: This snippet uses state transitions to compute the minimum cost, a classic dynamic programming approach.

- **Multi-label: Dynamic Programming + Graph Traversal**

```
vector<int> adj[1000];
int dp[1000][1000];
void dfs(int u, int p) {
    for (int v : adj[u]) {
        if (v != p) {
            dfs(v, u);
            dp[u][0] += max(dp[v][0], dp[v][1]);
        }
    }
}
```

Description: This code integrates DFS with dynamic programming to solve the maximum independent set problem on a tree.

C Ablation Study Results

This section evaluates the contribution of each L-CPC module through an ablation study.

Table 2: Ablation Study Results on Codeforces Dataset

Configuration	F1-score	Jaccard Score
Full L-CPC	0.80	0.90
w/o CodeBERT/UniXcoder	0.72	0.82
w/o LLM-based	0.75	0.85
w/o Retrieval-based	0.74	0.84
w/o T5	0.76	0.87

Analysis: Removing CodeBERT/UniXcoder results in the largest performance drop (F1: 0.72), highlighting its role in feature extraction. The T5 module's removal has a milder effect (F1: 0.76), reflecting its dependency on problem descriptions.

D Error Analysis

This section examines two prediction errors to identify L-CPC's limitations.

D.1 Case 1: High-Difficulty Problem

```
vector<int> solve(int n, vector<int>& a) {
    // Complex logic with rare algorithmic patterns
}
```

Error: Missed "divide and conquer" label. *Reason*: The code's complexity obscured key structural features, suggesting a need for enhanced feature extraction.

D.2 Case 2: Verbose Java Syntax

```
public class Solution {
    public int maxProfit(int[] prices) {
        // Verbose implementation
    }
}
```

Error: Predicted "greedy algorithm" but missed "dynamic programming." *Reason*: Java's verbose syntax introduced noise in the AST, affecting label accuracy.

Implication: Future improvements should refine AST parsing for verbose languages and rare paradigms.

This is a section in the appendix.