Pre-trained Language Models for Multi-Label Text Classification of Competitive Programming Problems

Bruno Vargas de Souza[1], **Ana Sofia S. Silvestre**[1], **Victor Hugo F. Lisboa**[1]
Vinicius R. P. Borges[1]

[1]Departamento de Ciência da Computação
Universidade de Brasília (UnB) – Brasília – DF – Brazil

{bruno.vargas,ana.silvestre,victor.lisboa}@aluno.unb.br,viniciusrpb@unb.br

Abstract. *This paper explores the use of pre-trained language models for classifying programming problems from online judges based on topics commonly addressed in competitive programming. State-of-the-art language models were employed as text classifiers, including Long Short-Term Memory (LSTM) and Bidirectional LSTM with pre-trained Word2Vec embeddings, Bidirectional Encoder Representations from Transformers (BERT), and Llama3.1-8B. Experiments were conducted using two different representations of the programming problems: standalone statement and statement with source code. The results showed that Llama3.1-8B achieved the best overall Macro F1-Score, outperforming the other models by a significant margin.*

1. Introduction

Competitive programming is a mental sport in which participants solve a set of problems within a specified time limit. A key aspect of competitive programming is the use of web platforms known as Online Judges (OJs), which incorporate an automated mechanism that evaluate submitted solutions, providing immediate feedback to the user. Additionally, many OJs include a repository of programming problems covering various topics in Computer Science, such as graphs, number theory, dynamic programming, and data structures. These problems are typically created by the authors to reflect the appropriate strategies for solving them. While most OJs, such as Codeforces[1], are more appropriate for mature developers and contestants, others are designed for different level of developers, such as Beecrowd[2] and AtCoder[3].

For learning computer programming, the labels (tags) assigned to problems in OJs are important for several reasons. They aid students to select topics of interest and identify their performance in solving problems related to specific topics and difficulty levels. Although some OJs already have predefined labels for their programming problems, these labels can sometimes be too broad or inconsistent. For instance, the "graphs" tag might encompass problems that involve solving shortest path algorithms or detecting cycles. Additionally, some problems in OJs or those customized by faculties may lack labels, requiring manual annotation, which is an unfeasible task when dealing with a large number of problems.

[1]`https://codeforces.com/`
[2]`https://judge.beecrowd.com/`
[3]`https://atcoder.jp/`

Natural Language Processing (NLP) and Machine Learning (ML) have been employed to automatically analyze programming problems for various purposes. Some studies have explored predicting the difficulty level [Zhou and Tao 2020], generating hints to aid students in solving problems [Suciu et al. 2021], and identifying the subject of the problem [Fonseca et al. 2020]. Other research has focused on categorizing programming problems as a multi-label text classification task, often considering Recurrent Neural Network language models [Iancu et al. 2019] [Pinnow et al. 2021]. Additionally, transformer-based pre-trained language models with millions of parameters, such as Bidirectional Encoder Representations from Transformers (BERT) [Devlin et al. 2019] and the Generative Pretrained Transformer (GPT) [Brown 2020], have also been considered for this task [Kim et al. 2023] [Lobanov et al. 2023]. These studies reported the challenging nature of programming problem categorization due to the diverse strategies that can be used to solve a problem and the complexity of some source code-based solutions, which have proven difficult for models available at the time to capture the patterns for accurate tag predictions.

Recently, Large Language Models (LLMs) with billions of parameters have demonstrated great performance in various text generation tasks, such as summarization [Zhang et al. 2024a], machine translation [Zhang et al. 2023] and, question answering [Shao et al. 2023]. In the context of analyzing programming problems, LLMs have proven useful in providing feedback for programming activities [Zhang et al. 2024b], automated source code generation [Huang et al. 2024], and in assisting the development of computational thinking [Yilmaz and Yilmaz 2023]. In this sense, there is an opportunity to explore the powerful capability of LLMs to the programming problem categorization.

This paper introduces a methodology for predicting the labels of programming problems using multi-label classifiers based on language models. Four state-of-the-art language models were considered: Long Short-Term Memory (LSTM), Bidirectional LSTM (BiLSTM), Bidirectional Encoder Representations from Transformers (BERT), and Llama3.1-8B. Experiments were conducted on a dataset of programming problems from Codeforces and aimed to compare the performances of the underlying classifiers. Two different strategies were used to represent the programming problems: problem statements and statements alongside their respective source code solutions.

The main contribution described in this paper is the use of a large language model with billions of parameters to determine the categories of competitive programming problems. We expect that the powerful capabilities of these LLMs can predict more reliable labels than classical ML and NLP techniques. The proposed multi-label classification scheme can support the automatic tagging of programming problems, allowing students to focus on more targeted topics and assisting professors in selecting problems for teaching purposes, especially when solutions are not available to them.

This paper is organized as follows. Section 2 discusses on studies related with classification of programming problems. Section 3 describes the proposed methodology. The experimental setup and the obtained results are reported and analysed in Section 4. Lastly, the insights obtained from the experiments as well as our thoughts for future works are reported in Section 5.

2. Related Works

Some research in the literature has explored the prediction of labels for programming problems for various purposes. We selected papers that addressed this task using NLP approaches, including text mining, language models, and transformers for analyzing the statements and source code of programming problems.

In [Moreira et al. 2024] it is presented a method to infer topics from code solutions for programming exercises based on an unsupervised-learning approach. The data was collected from previous editions of the Brazilian Olympiad in Informatics and submitted to BERT for word representation. The selected clustering model was K-means, and the topics, according to the formed clusters, were named and inferred by an expert who also made the model validation by qualitative analysis. Unlike the presented method, we had already predefined the tags for classification.

The article [Lobanov et al. 2023] presents an innovative approach to tag prediction, combining statements and source code. BERT fine-tuned was used to process the statements and a Gated Graph Neural Network (GNN) to analyze the solutions. The representation of the code in graph format was performed through the Abstract Syntax Tree (AST) using a specific parser. Models such as LSTM and CNN were tested, but BERT stood out in the statements, achieving an F1-score of 0.308, while GGNN got an F1-score of 0.430 for the source code. The combination of both models resulted in an F1-score of 0.532, demonstrating a considerable result. This article served as the basis for considering an alternative approach, involving the analysis of combining statements and descriptive solutions using LLM to explore potential improvements.

The study presented in [Kim et al. 2023] proposes an approach to efficiently guide programmers in developing solutions for programming problems from Codeforces based on the problem's tag and difficulty. Both were previously set up on the Codeforces platform, so the authors did not consider the scenario of wrong tags and unfair difficulty scores. Addressing the classification based on two features, the authors established a multi-task architecture combining the multi-class problem of inferring the difficulty and the multi-label problem of predicting the tag from the statement. Beyond the multi-tasks approach, the authors also highlighted that BERT-based methods can be instrumental in the context of text classification and tag inference.

In [Shalaby et al. 2017] addresses the challenge of recognizing algorithms within source code to help understand programs. The article proposes methods to classify algorithms in the code, extracting some relevant characteristics for the analysis, such as: number of variables, number of operations, number of lines, and constructs. The study attempts to differentiate between similar algorithms, such as dynamic programming (DP) and greedy algorithms, or DP and brute force. The results demonstrate high accuracy in binary classification tasks and acceptable performance in multilabel classification. The authors suggest future work that could extract more complex data and explore the application of different datasets.

These previous works, motivated us to present a methodology for label prediction of programming problems using pre-trained language models, including a LLM with billions of parameters. Using a public dataset from Codeforces, three corpora are next created to compare which one produces the more accurate results: considering only the

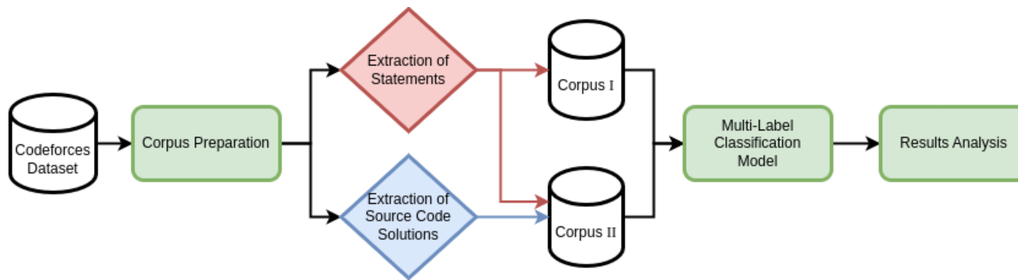

Figure 1. Flowchart illustrating the steps of the proposed methodology.

statements, and the statements combined with their code solution.

3. Methodology

In this section we describe the methodology for categorizing programming problems from OJs which consists of using pre-trained Language Models and by considering two distinct strategies for the representation of the available problems: using the problem statements and the statements combined with their respective source code solutions. The methodology steps are depicted in Figure 1.

3.1. Dataset

The dataset is composed of problem statements and solutions from Codeforces. A statement comprises the problem description, input and output formats, constraints, and sample cases, providing sufficient details so that contests can understand and formulate a correct solution.

The code solutions were obtained from a public dataset available online at Kaggle[4] that contains a significant range of programming problems from various difficulty levels. Each problem has multiple solutions in different programming languages, being 14 solutions in average per problem. The solutions were joined with the statements from another public dataset also available at Kaggle[5], as each contest - a programming competition in which a set of problems to be solved in a limited period - has an unique identifier number, allowing their combination. The solutions from contests not available in the statement dataset were discharged. In total, the final dataset is composed of 1300 problems and 18096 solutions.

The labels were derived from the same dataset used to collect the problem statements, where experienced competitors on Codeforces can openly categorize problems based on their topics. Although the dataset includes 37 different labels, only the 10 most frequent were selected for the method experiments. We decided to group the labels that rarely appear in the dataset compared to the rest of the labels within “others” tag. These tags may not be essential to the analysis as they are covered by a more comprehensive tag related to them. For example, the tags “number theory”, “combinatorics”, and “probabilities” fell under “Math” label, as well as “dfs and similar”, “graph matchings”, and “trees” in the “graphs” topic. This decision simplifies the analysis and reduces the complexity of the classification models learning processing.

[4] https://www.kaggle.com/datasets/yeoyunsianggeremie/codeforces-code-dataset

[5] https://www.kaggle.com/datasets/immortal3/codeforces-dataset

Table 1. Categories of the programming problems sorted in ascending order according to their frequency.

Labels	Frequency
Greedy	7336
Math	6471
Constructive Algorithms	4775
DP	4596
Implementation	4569
Data Structures	3853
Brute Force	3443
Sortings	2687
Binary Search	2430
Graphs	2294
Others	16637

3.2. Corpus Preparation

We assembled two corpora based on the nature of each experiment for comparing which one yields the best results:

- **Corpus I:** consists of using only the statements for each problem;
- **Corpus II:** contains the statements and their respective code solutions with no processing besides.

Our strategy for representing each sample in Corpus II assigns multiple solutions to each statement and these solutions were considered in only one programming language. The chosen programming language was C++, as it is currently the most used language in competitive programming, and this is because C++ is an efficient language and its standard library contains many algorithms and data structures. Unlike in Corpus I, in which each statement is submitted to the model once, the Corpus II is composed by unique combinations of statement with the respective source code separately, as each problem have many solutions, there are many occurrences of each statement among the corpus. Although the repetition of statements, each associated source code is different, which distinguishes those instances. The Corpus I is formed by 1300 samples and the Corpus II is assembled by 18906 combinations.

3.3. Selected Models' Setting

Four state-of-the-art language models were chosen for the experiments to predict programming problem labels. In this subsection, we specify their architecture and the steps implemented to process the corpus and submit it as input to the language models. The only pre-processing method applied in the source code was comment removal, and the pre-processing techniques listed below are associated with the statement processing.

3.3.1. LSTM/BiLSTM

- **Pre-processing**: The punctuation, latex notations, and stop words in English were removed. In Natural Language Processing, stop words denote a group of words

that do not carry much semantic value to a sentence and have a high frequency of occurrence, such as prepositions and articles.

- **Word representation**: The Word2Vec technique was utilized to represent words and capture their meaning and semantic resemblance. This method represents words as vectors in a multidimensional space based on their surrounding words. The model used the CBOW approach, which predicts the central word given neighboring words. As a result, words with semantic resemblances end up having similar vector representations.
- **Model description**: Long Short Term Memory (LSTM) [Hochreiter and Schmidhuber 1997] is a subclass of Recurrent Neural Network (RNN) models. The LSTM model overcomes the Vanish Gradient Descent problem that affected RNN models. The LSTM model is known for performing considerably well in sequential and time-series data classification tasks, in particular, Natural Language Processing (NLP) tasks. We also considered a BiLSTM model in our experiments, which learns through bidirectional sequential data concatenating two LSTM components, and can possibly increase learning performance.

3.3.2. BERT

- **Pre-processing**: Only the punctuation and latex notations were removed. For transformers-based model, stop words do not affect the performance of the models and for many cases it is not necessary to remove them [Qiao et al. 2019].
- **Word representation**: the words are represented by subword tokenization, in which the words are divided into smaller subwords that provide information to the model, called tokens.
- **Model description**: BERT [Devlin et al. 2019] is a pre-trained open model based on bidirectional Transformers architecture [Vaswani et al. 2023]. It surpassed many models in performance when it was released, and can be applied in many NLP tasks, such as text classification [Wilkho et al. 2024], text generation [Zhao et al. 2024], and language understanding [Mountantonakis et al. 2024]. For the experiments we chose the standard BERT deployed by Google.

3.3.3. Llama

- **Pre-processing**: Due to memory limitations, punctuation and stop words in English were removed to make better use of the fixed token size.
- **Word representation**: Convert text into tokens, which are then used to analyze their relationships in similar contexts, such as the approach used in BERT.
- **Model description**: [Llama team 2024] is a next-generation language model available in 8 billion, 70 billion, and 405 billion parameters, using a standard decoder-only transformer architecture, the 8 billion version was chosen for the experiments executed in this paper. Improvements over Llama 2 [GenAI Meta 2023] include more efficient 128K tokens and grouped query attention (QGA) [Ainslie et al. 2023] for better inference. Llama 3.1 was measured across more than 150 benchmarks, competes with leading models, and is used for

a variety of tasks in AI while also being open-source. Additionally, fine-tuning with Llama was used for the experiments presented in this paper.

4. Experimental Results

Experiments were conducted to validate the proposed methodology and compare the performance of classification models. The proposed methodology was developed in Python 3.10 environment with the assistance of the following libraries: Gensim [6], Keras [7], NLTK [8], Numpy [9], Pandas [10], Peft [11], PyTorch [12], Scikit-Learn [13], Tensorflow [14], and Transformers [15]. The experiments were conducted on a machine equipped with an Intel Core i5 9th generation processor, a GeForce RTX 3060 graphics card, and 16 GB of VRAM. The source code and dataset are available in a public repository [16].

The selected metric to evaluate the performance of the classification models was the F1-score, as it combines precision and recall into a single value. Moreover, other papers in literature also use this metric, allowing their results to serve as a baseline. We opted for the macro F1-score since this metric is suitable for unbalanced datasets. For the four derived classification models, the final score was measured by comparing the predicted labels for each test sample with their respective expected labels.

K-Fold Cross Validation (KCV) was used to evaluate the performance of the training models, which involves splitting the dataset into K consecutive folds to test the model with a more diverse dataset. To prevent data overlapping, where the same statement - potentially associated with multiple solutions - could appear in both the training, validation or test datasets, we used the GroupKFold [17] strategy. This method ensures that all instances with the same statement are kept together in the same fold. The data was split into 5 folds, allocating 80% for training, 10% for validation, and 10% for test, this amount was chosen for test due to the dataset being relatively large. The validation set was used to search for the best hyperparameter values for the models. It was employed with the Early Stopping strategy, set with a patience of 5 epochs, meaning it will stop after 5 epochs if the validation loss has not decreased.

Table 2 shows the optimal hyperparameter values for each model obtained for each round of KCV. Table 3 presents the average macro F1-Scores across the K-Folds. Finally, tables 4 and 5 presents the average F1-Scores across the K-Folds for each label. For the matter of baseline, [Kim et al. 2023] obtained a Macro F1-Score of 0.51 for tag prediction, however, the authors considered the most frequent 20 categories while in our strategy we grouped them according to labels' similarity. Furthermore, [Lobanov et al. 2023] accomplished an F1-Score of 0.53.

[6] https://radimrehurek.com/gensim/
[7] https://keras.io/
[8] https://www.nltk.org/
[9] https://numpy.org/
[10] https://pandas.pydata.org/
[11] https://huggingface.co/docs/peft/index
[12] https://pytorch.org/
[13] https://scikit-learn.org/
[14] https://www.tensorflow.org/
[15] https://huggingface.co/docs/transformers/index
[16] https://gitlab.com/gvic-unb/cp-problems-pre-trained-models-classification
[17] https://scikit-learn.org/stable/api/index.html

Table 2. Hyperparameter values determined in each fold.

Model	Hyperparameter	Optimal Values per Fold
LSTM/BiLSTM	Learning Rate	$[10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}]$
LSTM/BiLSTM	Units	[64, 64, 64, 64, 64]
LSTM/BiLSTM	Batch Size	[32, 32, 32, 32, 32]
BERT	Learning Rate	$[10^{-5}, 10^{-5}, 10^{-4}, 10^{-4}, 10^{-5}]$

Table 3. Comparison of macro F1-scores and standard deviation between models for each corpus.

Input	LSTM	BiLSTM	BERT	LLaMA 3.1
Corpus I	0.36 ± 0.01	0.36 ± 0.01	0.34 ± 0.02	$\mathbf{0.79 \pm 0.21}$
Corpus II	0.36 ± 0.01	0.36 ± 0.01	0.33 ± 0.01	$\mathbf{0.75 \pm 0.20}$

Table 3 shows that the Llama outperformed the other language models regarding both corpora, especially in Corpus I (Statements), where it achieved an F1-Score of 0.79, with a standard deviation of 0.21. Additionally, the LLM also had difficulty with source code, as it processes natural language more effectively. Tables 4 and 5 reveal that some labels are easier for the models to predict. Moreover, it suggests that using only the statements of programming problems can lead to better prediction performance, especially with Llama3.1-8B, while also leveraging a reduced length of the problem representations.

The ‘graphs’ tag yielded the highest F1-score, obtained by the Llama-based classification model using Corpus I and Corpus II, demonstrating that LLMs can identify statements and input features related to graph-based strategies for solving problems, involving shortest paths, max flow, connected components, cycle detection, etc. Despite the overall low scores for the LSTM, BiLSTM, and BERT models, the ‘greedy’ and ‘math’ labels achieved higher F1-scores compared to the other labels. These results can be explained by the fact that the statements contain distinctive information related to these labels, such as geometric figures, direct mathematical descriptions, and the presence of equations.

We can verify that the ‘brute force’, ‘binary search’, ‘implementation’, and ‘sortings’ labels were more challenging for the language models to predict in both Corpus. The statements of these problems typically ask for optimizing a search process, which can be approached using different strategies, including breadth-first search (BFS), depth-first search (DFS), and dynamic programming, for instance. In the case of the ‘sortings’ label, other search-based strategies may require rearranging the input data without explicitly using a sorting procedure, such as in a prefix sum approach. Although the ‘implementation’ label suggests a straightforward problem-solving strategy, it is often challenging to develop and reason, frequently resulting in long source codes.

The number of labels grouped to the ‘others’ label is an important factor that might affect model training. It is well-known that datasets with a large number of class labels can make the learning process more difficult due to ambiguity between class labels and imbalance. Therefore, it is important to keep some labels separate from the ‘others’ label based on their relevance regarding the fundamental problem-solving strategies in competitive programming. Less representative labels, such as ‘fft’ (Fast Fourier Transform) and ‘ternary search’, can be grouped into the ‘others’ label since they are very specific

Table 4. Classification results: average and standard deviation of macro F1-Score per label - Corpus I.

Label	LSTM	BiLSTM	BERT	Llama 3.1
Greedy	0.57 ± 0.04	0.57 ± 0.04	0.57 ± 0.03	$\mathbf{0.85 \pm 0.16}$
Math	0.54 ± 0.03	0.54 ± 0.03	0.52 ± 0.03	$\mathbf{0.87 \pm 0.19}$
Constructive Algorithms	0.37 ± 0.03	0.37 ± 0.03	0.42 ± 0.05	$\mathbf{0.83 \pm 0.18}$
DP	0.42 ± 0.05	0.42 ± 0.05	0.39 ± 0.06	$\mathbf{0.82 \pm 0.21}$
Implementation	0.40 ± 0.04	0.40 ± 0.04	0.40 ± 0.04	$\mathbf{0.80 \pm 0.29}$
Data Structures	0.36 ± 0.04	0.36 ± 0.04	0.34 ± 0.03	$\mathbf{0.85 \pm 0.15}$
Brute Force	0.29 ± 0.05	0.29 ± 0.05	0.31 ± 0.07	$\mathbf{0.76 \pm 0.28}$
Sortings	0.26 ± 0.04	0.26 ± 0.04	0.27 ± 0.05	$\mathbf{0.80 \pm 0.30}$
Binary Search	0.24 ± 0.05	0.24 ± 0.05	0.23 ± 0.05	$\mathbf{0.76 \pm 0.25}$
Graphs	0.24 ± 0.02	0.24 ± 0.02	0.19 ± 0.04	$\mathbf{0.91 \pm 0.11}$
Others	0.04 ± 0.00	0.04 ± 0.00	0.07 ± 0.11	$\mathbf{0.40 \pm 0.34}$

Table 5. Classification results: average and standard deviation of macro F1-Score per label - Corpus II.

Label	LSTM	BiLSTM	BERT	Llama 3.1
Greedy	0.57 ± 0.04	0.57 ± 0.04	0.57 ± 0.03	$\mathbf{0.82 \pm 0.19}$
Math	0.54 ± 0.03	0.54 ± 0.03	0.52 ± 0.04	$\mathbf{0.88 \pm 0.12}$
Constructive Algorithms	0.37 ± 0.03	0.37 ± 0.03	0.42 ± 0.05	$\mathbf{0.78 \pm 0.20}$
DP	0.42 ± 0.05	0.42 ± 0.05	0.39 ± 0.07	$\mathbf{0.81 \pm 0.19}$
Implementation	0.40 ± 0.04	0.40 ± 0.04	0.40 ± 0.05	$\mathbf{0.77 \pm 0.26}$
Data Structures	0.36 ± 0.04	0.36 ± 0.04	0.34 ± 0.03	$\mathbf{0.81 \pm 0.14}$
Brute Force	0.29 ± 0.05	0.29 ± 0.05	0.31 ± 0.07	$\mathbf{0.68 \pm 0.30}$
Sortings	0.26 ± 0.04	0.26 ± 0.04	0.27 ± 0.06	$\mathbf{0.80 \pm 0.19}$
Binary Search	0.24 ± 0.05	0.24 ± 0.05	0.23 ± 0.05	$\mathbf{0.70 \pm 0.25}$
Graphs	0.24 ± 0.02	0.24 ± 0.02	0.19 ± 0.04	$\mathbf{0.86 \pm 0.14}$
Others	0.24 ± 0.02	0.24 ± 0.02	0.04 ± 0.02	$\mathbf{0.40 \pm 0.34}$

techniques that would be challenging to the models to capture patterns during training.

Our research showed that LLMs with billion of parameters can be used to label programming problems according to predefined labels, which can reduce inconsistencies in the labels. However, some limitations were identified in the classification models considered in our methodology. Although Llama achieved the best overall results, the large standard deviation observed in predictions across the folds is a concern that emphasizes the complex nature of the task. This is also demonstrated by the poor performance of other language models with millions of parameters, even when the source code was provided as input with the statement for training the models. Therefore, predicting labels for competitive programming problems remains a challenging topic, which may require the use of LLMs with more than 8 billion parameters, compared to those used in our study.

5. Conclusion

This paper presents a methodology for the multi-label classification of programming problems using pre-defined labels associated to fundamental topics in competitive program-

ming. For that purpose, we curated two corpora based on problems collected from Codeforces: the first corpus consists only of problem statements while the second includes both the statements and their respective source code solutions.

For the multi-label text classification, four state-of-the-art pre-trained language models were selected: LSTM, BiLSTM, BERT, and Llama 3.1-8B. The experiments conducted on the created corpora showed that Corpus I presented the best results with the Llama 3.1-8B-based classifier, achieving a global average F1-score of 0.79. We obtained satisfactory results for individual labels with Llama 3.1-8B, which outperformed the other language models.

The proposed study demonstrated that LLMs with billions of parameters can be used in automated labeling of programming problems. Knowing that some problems might present inconsistent labels, these approaches ensure that problems are categorized according to the most suitable and reliable strategies for solving them, enabling students to focus on specific programming skills and topics. This eliminates the time-consuming and error-prone process of manual labeling, which is increasingly impractical as the volume of problems in OJs is constantly growing. Such models can also be applied to unlabeled datasets or those requiring relabeling by using fine-tuned LLMs on other programming problem corpora.

For future work, we plan to explore alternative approaches for representing the source codes of programming problem solutions, such as Abstract Syntax Trees [Zhang et al. 2019]. It would be interesting to study other strategies to group problems to the label 'others'. Finally, other open LLMs specialized in the analysis and generation of source code could also be investigated to obtain more reliable predictions for less representative labels.

References

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebron, F., and Sanghai, S. (2023). Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Stroudsburg, PA, USA. Association for Computational Linguistics.

Brown, T. B. (2020). Language models are few-shot learners. *arXiv preprint ArXiv:2005.14165*.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Fonseca, S. C., Pereira, F. D., Oliveira, E. H., Oliveira, D. B., Carvalho, L. S., and Cristea, A. I. (2020). Automatic subject-based contextualisation of programming assignment lists. *International Educational Data Mining Society*.

GenAI Meta (2023). Llama 2: Open foundation and fine-tuned chat models.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780. [Online; accessed 2024-08-18].

Huang, T., Sun, Z., Jin, Z., Li, G., and Lyu, C. (2024). Knowledge-aware code generation with large language models. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 52–63.

Iancu, B., Mazzola, G., Psarakis, K., and Soilis, P. (2019). Multi-label classification for automatic tag prediction in the context of programming challenges. *arXiv preprint arXiv:1911.12224*.

Kim, J., Cho, E., Kim, D., and Na, D. (2023). Problem-solving guide: Predicting the algorithm tags and difficulty for competitive programming problems.

Llama team (2024). The llama 3 herd of models.

Lobanov, A., Bogomolov, E., Golubev, Y., Mirzayanov, M., and Bryksin, T. (2023). Predicting tags for programming tasks by combining textual and source code data.

Moreira, J., Silva, C., Santos, A., Ferreira, L., and Reis, J. (2024). Abordagem não-supervisionada para inferência do tópico de um exercício de programação a partir do código solução. In *Anais do XXXII Workshop sobre Educação em Computação*, pages 842–853, Porto Alegre, RS, Brasil. SBC.

Mountantonakis, M., Mertzanis, L., Bastakis, M., and Tzitzikas, Y. (2024). A comparative evaluation for question answering over Greek texts by using machine translation and BERT. *Language Resources and Evaluation*.

Pinnow, N., Ramadan, T., Islam, T. Z., Phelps, C., and Thiagarajan, J. J. (2021). Comparative code structure analysis using deep learning for performance prediction. *arXiv preprint arXiv:2102.07660*.

Qiao, Y., Xiong, C., Liu, Z., and Liu, Z. (2019). Understanding the behaviors of bert in ranking.

Shalaby, M., Mehrez, T., El Mougy, A., Abdulnasser, K., and Al-Safty, A. (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.

Shao, Z., Yu, Z., Wang, M., and Yu, J. (2023). Prompting large language models with answer heuristics for knowledge-based visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14974–14983.

Suciu, V., Giang, I., Zhao, B., Runandy, J., and Dang, M. (2021). Generating hints for programming problems without a solution. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 1382–1382.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention is all you need.

Wilkho, R. S., Chang, S., and Gharaibeh, N. G. (2024). Ff-bert: A bert-based ensemble for automated classification of web-based text on flash flood events. *Advanced Engineering Informatics*, 59:102293.

Yilmaz, R. and Yilmaz, F. G. K. (2023). The effect of generative artificial intelligence (ai)-based tool use on students' computational thinking skills, programming self-efficacy and motivation. *Computers and Education: Artificial Intelligence*, 4:100147.

Zhang, B., Haddow, B., and Birch, A. (2023). Prompting large language model for machine translation: A case study. In *International Conference on Machine Learning*, pages 41092–41110. PMLR.

Zhang, H., Yu, P. S., and Zhang, J. (2024a). A systematic survey of text summarization: From statistical methods to large language models. *arXiv preprint arXiv:2406.11289*.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X. (2019). A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE.

Zhang, Z., Dong, Z., Shi, Y., Price, T., Matsuda, N., and Xu, D. (2024b). Students' perceptions and preferences of generative artificial intelligence feedback for programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 23250–23258.

Zhao, C., Feng, R., Sun, X., Shen, L., Gao, J., and Wang, Y. (2024). Enhancing aspect-based sentiment analysis with bert-driven context generation and quality filtering. *Natural Language Processing Journal*, 7:100077.

Zhou, Y. and Tao, C. (2020). Multi-task bert for problem difficulty prediction. In *2020 International Conference on Communications, Information System and Computer Engineering (CISCE)*, pages 213–216. IEEE.