A Multi-label Classification Approach for Categorizing Beginner Programming Problems from Online Judges

Ana Sofia S. Silvestre, Bruno Vargas de Souza, Victor Hugo F. Lisboa, Vinicius R. P. Borges
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, DF, Brazil
{ana.silvestre,bruno.vargas,victor.lisboa}@aluno.unb.br, viniciusrpb@unb.br

***Abstract*—This is a full paper, and it belongs to the research category. Online Judges (OJs) have been used by students and programming practitioners to solve problems by submitting solutions as source code and receiving immediate feedback. This important characteristic of OJs makes the learning and teaching of computer programming more productive and efficient, and they are considered valuable in several programming courses as well as by self-study students aiming to improve their programming skills. Moreover, OJs store a repository of problems that are mostly categorized according to topics related to competitive programming (such as graphs, dynamic programming, etc.). Although these problems are inappropriate for introductory programming courses, few OJs cover beginner-level programming problems categorized into basic topics, such as conditionals, loops, arrays, etc. Providing such categories is time-consuming and prone to error when done manually by faculties that formulate the proper categories and code solutions for all problems in the repository. This scenario motivated us to explore natural language processing (NLP) approaches to automatically classify beginner programming problems by analyzing their statements. For modeling this multi-label text classification approach, a corpus of beginner problems from a Brazilian OJ was manually curated using predefined class labels related to an introductory programming course. Experiments were performed using well-known classification models: Support Vector Machines (SVM), Gradient Boosting, Long Short-Term Memory (LSTM), Bidirectional LSTM, and BERT. The results showed that SVM yielded the best overall macro F1-Score, though class imbalance affected performance on individual topics.**

***Index Terms*—online judges, problem solving, multi-label text classification, language models**

I. Introduction

Developing computer programming skills requires regular problem-solving across a wide diversity of computer science topics and at different difficulty levels. Recently, Online Judge (OJ) platforms have emerged as a powerful platform [1], allowing users to access a repository of programming problems to be solved for practicing purposes. Programmers can submit their own solutions to the platform and receive immediate feedback from the automatic correction mechanism, which can return correct answer, wrong answer, or very time-consuming verdicts.

These OJs have become very popular tools in programming classes, as faculties can select specific programming problems and guide students through programming learning. This can positively affect students' academic performance, as some Brazilian undergraduate computer-related programs have shown significant failure rates in introductory programming courses [2] [3]. Furthermore, self-learning students can also benefit from these platforms, as they provide automatic feedback and additional support through community forums and regular active users to resolve doubts. OJs are also the main platforms for competitive programming [4], a mental sport that is very popular among students and programming enthusiasts, as it encourages participants to think of optimized solutions for problems of different difficulty levels [5].

OJs typically have a very extensive repository of programming problems, with each problem belonging to multiple categories that indicate the strategies suggested for solving them. In most OJs, these labels are associated with specific problem-solving strategies for competitive programming topics, such as dynamic programming, graphs, and data structures. Additionally, these platforms allow users to build customized lists of problems, though users must manually search for the appropriate problems according to the class labels. However, many problems are poorly labeled or mislabeled, which would require a considerable amount of time to analyze and relabel, among other issues.

Most OJ problems have tags related to fundamental computer science topics, such as graphs, dynamic programming, and number theory, which require a deeper understanding of math and algorithm concepts. Despite the importance of these topics, they are considered advanced subjects for beginners, including many students from introductory programming courses. Since programming problems from OJs are not categorized according to basic programming topics such as loops, conditionals, and arrays, their use may be more suitable for advanced students who generally seek challenging problems, thereby reducing the applicability of OJs in introductory programming classes. Furthermore, the literature has presented some strategies for labeling programming problems from OJs [6] [7] [8], which generally demand solutions based on Natural Language Processing (NLP) and machine learning. However, these strategies focus only on the automatic labeling of non-

introductory problems. Furthermore, these works also emphasized the challenging nature of this task when considering the analysis of statements or alongside the solution as source code.

In this sense, this research investigates the multi-label classification of beginner problems since it would benefit faculties to concentrate efforts on the main educational activities that cannot be automatized. That means reducing efforts for categorizing beginner problems since teaching introductory programming is challenging [9] and would require the creation of customized repositories of programming problems, in which manual labeling is time-consuming and prone to error. Moreover, the profile of students in programming courses can vary from semester to semester or between different classes, requiring regular adjustments in the problem labels to meet these students' educational and pedagogical needs.

This paper presents a methodology for the multi-label classification of beginner programming problems from OJs since they can belong to one or more categories. To validate this methodology, we created a corpus of problem statements from the beginner section of the repository from Beecrowd Online Judge [1]. In this sense, we conducted an annotation process that involved expert developers and followed the major topics covered by the introductory programming course at Universidade de Brasília in C Language. Moreover, we selected some models for text classification in literature based on traditional machine learning, such as Support Vector Machines (SVM) and Gradient Boosting, as well as state-of-the-art language models, such as Long Short-Term Memory (LSTM) and Bidirectional Encoder Representations from Transformers (BERT) [10]. We performed experiments to validate the proposed methodology and evaluate those classifiers' performance in the constructed corpus.

We expect that the proposed methodology can be employed or adjusted to label raw programming problem datasets or provide additional labels in already labeled datasets. Thus, professors, teaching assistants, and instructors can prepare instructional material associated with problem sets for programming practice with the support of an automatic approach based on NLP.

The main contributions of this paper are:

- An annotated corpus of beginner problem statements from the Beecrowd OJ, in which the class labels correspond to topics covered in an introductory programming course taught in C language at Universidade de Brasília.
- a benchmark of experiments for multi-label classification using the created corpus.

This paper is organized as follows. Section II described the related work in the literature concerning the classification of programming problems from OJs. Section III describes the steps of the proposed methodology, including creating a corpus of problem statements and modeling a multi-label classification approach for predicting its categories. Section IV reports the experiments comparing the classification models and discloses the results. Finally, Section V discusses the advantages and limitations of the considered techniques in the proposed method, and suggests possibilities for future work.

[1]https://judge.beecrowd.com/

II. Related Work

To the best of our knowledge, there is limited research in the literature on methods for classifying OJ problems based on problem statements and other features, such as submission statistics or source code. Therefore, we selected several related studies that explore the use of NLP for classifying and labeling OJ problems, focusing on methods that demonstrated satisfactory results given the challenging nature of this task.

Sudha et al. [11] aimed to predict suitable labels for OJ problems based on the approach of using their respective code solutions as the corpus, combined with a Convolutional Neural Network (CNN) model to address the unreliability of tags in OJ problems. The proposed CNN-based system processes submissions in C++, predicting appropriate tags for each problem to improve competitors' skills in specific domains related to competitive programming. The proposed method involves dataset preparation, pre-processing, classification, and recommendation steps, including data scraping, manual data selection, and code pre-processing. The classification model architecture comprises two convolutional layers, pooling layers, and a fully connected layer for the final prediction. Experiments were performed using a hold-out strategy by splitting the data into 80% for training and 20% for testing, yielding 80.5% and 85% accuracy values in both classification and recommendation tasks. These results suggest potential extensions, such as increasing the number of tags and incorporating problem statements into the CNN model.

Majid et al. [12] created a Benchmark called Code4Bench, which extracts user information from the Codeforces OJ to serve as a benchmark due to this platform's large amount of available data (statements, tags, and submission statistics). Codeforces is a rich source of information related to the solutions submitted as attempts to solve problems since real-world failures can be collected for analysis and evaluation purposes. In addition to using a Web Crawler (a tool for collecting data from web pages), the authors also collected data related to specific social and demographic aspects through an online survey filled out by the OJ participants. This diverse data is useful for dealing with bias, allowing researchers to adjust their techniques to obtain better results. This diverse data helps deal with bias

Lobanov et al. [6] approached the problem tag's prediction by considering both problem statements and source code as input to the underlying classification models. Each input instance is represented by two vectors: the BERT vectors from the problem statements and the vectors associated with the source code, which is obtained from a Gated Graph Neural Network (GGNN) based on its Abstract Syntax Tree. The authors performed experiments by comparing them with similar models in the literature and an ablation study [13] to demonstrate the effectiveness of the proposed model. The results showed that the proposed method surpassed the performances obtained by the literature, even considering the standalone BERT and

GGNN vectors, proving that the combination of statement and source code brings better results. The authors emphasized that even with positive results, the approach was not enough to replace manual labeling.

Lokat et al. [14] present a text classification approach for competitive programming problems extracted from Codeforces. The dataset was collected, focusing on extracting problem tags using BeautifulSoup. The pre-processing step involved removing unwanted characters, stop words, converting text to lowercase, and tokenization followed by lemmatization [15]. The study emphasized three main tags: Greedy, Graph, and Implementation. Three deep learning models were employed: Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), and Multi-Layer Perceptron (MLP), each with specific parameters. The MLP model achieved the highest accuracy of approximately 73%, while the other models achieved around 59%. One identified limitation was the dataset size, suggesting potential improvement by including data from additional platforms like TopCoder and HackerRank. Although the authors showed that combining source code and textual data can improve metrics, many online judges do not allow users to access submitted codes, limiting this solution to specific platforms.

In this article, we will present the creation of a corpus of online judge problems for beginners to assist student's learning and guidance. The problems are collected from the Beecrowd platform, a Brazilian Online Judge, and the corpus is validated by NLP to ensure correctness and automation. The source code from problems will not be used for validation, as we are focusing on solving a problem for a platform that does not allow access to them.

III. Proposed Methodology

The proposed methodology is divided into two major phases: data collection and statement classification, as depicted in Figure 1. During the data collection stage, web scraping is performed to obtain problem statements and additional information, followed by the manual labeling of problem statements. The statement classification stage is divided into three steps: first, the statements are prepared as inputs for classification models; next, different techniques are compared to ensure optimal classification performance; and finally, the best model is applied to the corpus to obtain the new classification.

A. Beecrowd Judge

A sample page illustrating a Beecrowd problem is depicted in Figure 2. Each problem is constituted by four main parts: header, statement, input and output description, and test samples (input and correct output). The header includes the problem's title along with the specification of a time limit in seconds, i.e., the maximum time that the proposed solution by the student should run for each test case. If the running time exceeds that limit, the automatic judge returns the 'Time Limit Exceeded' verdict. Although this verdict is not commonly returned on beginner's problems, it can be helpful to encourage students to write efficient solutions from the beginning.

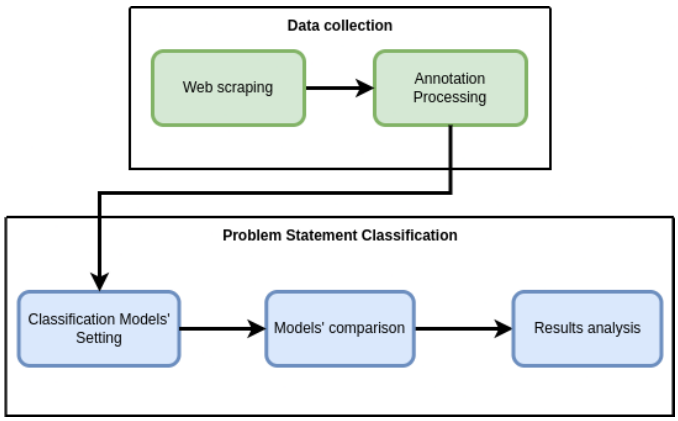

Figure 1. Steps comprising the proposed methodology.

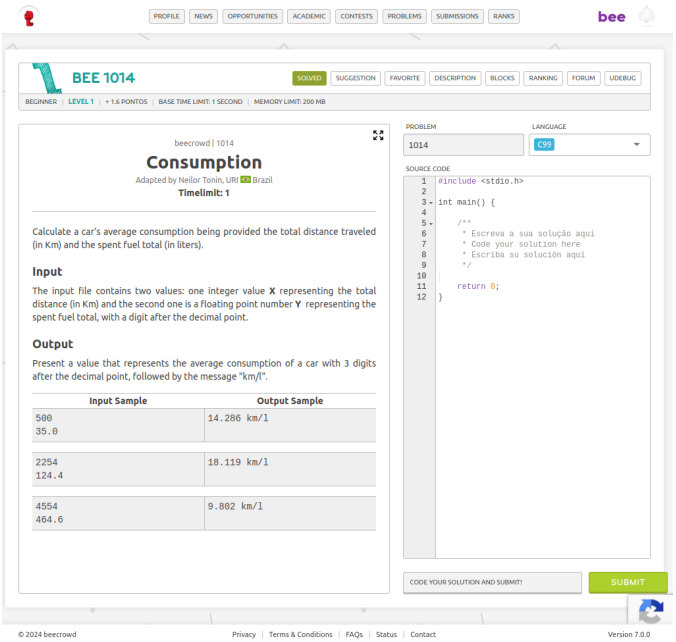

Figure 2. Sample page presenting Problem 1014 of Beecrowd Judge.

The statement can be written defining objectively the problem to be solved or can describe a history (fictitious or real) so that the student must interpret, understand, and identify the underlying problem and the proper strategy to solve it. The Input section defines the constraints over all variables described in the statement, which should be taken into account by the student when developing the algorithmic solution. Moreover, this section details the formatting of input values that should be read by the student's solution from the Standard Input. The Output section presents the guidelines for printing the answer in the Standard Output, which can be strictly followed to allow the comparison with the correct solution. Finally, there are input and output sample correspondences that the student can analyze to formulate their own solution. It is worth noting that the automatic judge also evaluates the submitted solution using hidden test cases, which can

result in an incorrect verdict ("Wrong Answer", "Time Limit Exceeded", "Memory Limit Exceeded", "Runtime Error"). A proposed solution that fails in hidden test cases demands the student to reason the corner and harder cases that led to the wrong answer verdict since they are not explicitly indicated in the judge's feedback.

B. Data Collection

Beecrowd OJ does not have a tool, such as an API or a specific web interface, that extracts contests (programming competitions) and problems' data. That motivated us to develop an automated extraction script using Selenium[2], an open-source tool commonly used for web automation and web scraping, in order to extract the problems' IDs. After each problem ID was collected in the associated web page, the problem statements were extracted using BeautifulSoup[3], a Python HTML parsing library, to navigate through and remove HTML tags.

C. Annotation Process

The labels of the Beecrowd Judge problems fall into 8 major categories: beginner, data structures, ad hoc, graphs, strings, mathematics, paradigms, and computational geometry. As this research aims to investigate the automatic tagging of online judge problems related to introductory programming courses, we focus on the Beginner section. The problems within this category are originally labeled according to the difficulty level, but to accomplish the goals of this research, additional class labels related to the fundamental topics of introductory programming are required.

Therefore, an annotation process is proposed to provide customized class labels related to topics commonly studied in introductory programming courses. The class labels were defined according to the content approached by C Programming Language [16] alongside the syllabus currently adopted in the introductory programming course at the Department of Computer Science at the Universidade de Brasília. We detail them below:

- **simple math**: problems involving simple arithmetic operations;
- **simple geometry**: problems involving trigonometry operations;
- **conditionals**: problems that require if-else structures;
- **loops**: involves the use of for, while and do-while structures;
- **array1d**: data storage on homogeneous and unidimensional arrays;
- **strings**: unidimensional arrays storing a finite sequence of characters;
- **array2d**: data storage on homogeneous bidimensional arrays;
- **structures**: heterogeneous data storage;
- **io**: problems that are solved only using formatted input and output functions (`printf` and `scanf`). Thus, the process of reading and writing data to the Standard Input/Output to feed the algorithm are not considered.
- **recursion**: a strategy to solve problems that require using a function that calls itself in its scope.
- **sorting**: problems that involve organizing data into a specific order.

The proposed annotation process was performed by four experienced developers on the Beginner section of the Beecrowd repository. The complete set of problems to be labeled within the Beginner Section was randomly assigned to the annotators. The proposed algorithmic solutions for each problem were coded in the C language and validated on the automatic judge of Beecrowd. The annotation process is described as follows:

1) The annotator takes a problem and interprets the statement along with the input and output restrictions.
2) The annotator then develops an algorithm in C language, obtains a source code, and submits it to the Beecrowd Judge until receiving the "Accepted" verdict, i.e., ensuring the solution is correct.
3) The annotator writes down the class labels associated with the topics employed in the source code for the problem. The annotator can inquire for a second opinion in case of uncertainty regarding the labeling.

Table I
DISTRIBUTION OF CLASS LABELS IN THE FINAL CORPUS.

Labels	Occurrences
loops	120
conditionals	107
simple math	84
strings	19
array1d	18
array2d	15
simple geometry	12
io	9
sorting	3
recursion	2
structs	1

After the annotation process, only problems with the predefined class labels are retained in the final corpus. That means that problems with complex solving strategies that are not covered nor considered to be introductory programming level are disregarded. Therefore, the final corpus consists of 188 problems, in which the class labels present the distribution shown in Table I.

D. Classification Models' Setting

We selected some well-known models based on machine learning, recurrent neural networks, and transformers that have been employed for text classification in the literature. As each classification model requires a specific structured representation of the texts, we designed a specific pipeline for each one to allow the processing of the problem statements by the underlying models. The pipeline associated with the classification model aims to prepare the input text (problem statement), originally in natural language, and then transform

[2]https://www.selenium.dev/
[3]https://www.crummy.com/software/BeautifulSoup/

its tokens (word, subwords) to numerical structured representations. These pipelines and the brief descriptions of each classification model are given below:

1) Support Vector Machines and Gradient Boosting:

- **Text pre-processing**: Stop words and punctuation removal were applied, taking into account the Portuguese language. In NLP, stop words are terms that carry very few semantic value, such as preposition, articles, and conjunctions. They frequently present high occurrence in texts, which can affect the performance of NLP models. Furthermore, high-frequency words in statements were also removed.
- **Word representation**: considers Term Frequency-Inverse Document Frequency (TF-IDF), which is obtained by calculating the frequency of each term included in the text and multiplying it by the inverse frequency of texts that contain the term. This process results in a numerical representation reflecting the importance of each word based on its occurrence in the corpus.
- **Model description**: Support Vector Machines [17] are classical models for NLP solutions, which are configured as supervised models for linear classification and regression analysis. Gradient Boosting Tree [18] is an ensemble learning approach based on decision trees for classification and regression tasks. The initial classification model is constituted by a single decision tree, which makes preliminary predictions regarding the categories of the problem statements. After that, a new decision tree is added to the model, forming an ensemble model that aims to correct the predictions of the first decision tree. This process is repeated until the convergence of the classification model. Thus, each new decision tree added to the ensemble aims to improve the prediction of the remaining trees, allowing the learning of complex patterns.

2) LSTM/BiLSTM:

- **Text pre-processing**: stop words and punctuation removal were performed as in the SVM/GDB models.
- **Word representation**: dense vectors represent the words and capture their semantics, with words in similar contexts presenting closer vector representations. That is achieved using the Word2Vec technique with the CBOW-300 (Continuous Bag of Words) model, which was pretrained in massive Portuguese corpora [19]. The CBOW model predicts a target word based on its surrounding context words, effectively learning the relationships and similarities between words. Consequently, words frequently appearing in similar contexts in the training data are mapped to nearby points in the vector space, enabling embeddings to reflect semantic similarities.
- **Model description**: The LSTM (Long Short-Term Memory) is a class of Recurrent Neural Network (RNN) that handles sequential data, specifically focusing on the temporal aspect. The key difference lies in its ability to retain long-term information, enabling it to effectively deal with temporal dependencies in sequential data. Another considered model is the Bidirectional LSTM (BiLSTM), which processes the input text in both directions (from the start of the sentence until the end of the sentence and vice versa), allowing the network to capture information from both preceding and subsequent positions in a sequence, thus providing more comprehensive results.

3) BERT:

- **Text pre-processing**: Only punctuation removal was applied as in LSTM and BiLSTM models.
- **Word representation**: Unlike the other models listed above, the word representation used in BERT is based on subword tokenization, in which each word is split into smaller parts. This allows the model to capture detailed information from the input text.
- **Model description**: BERT [10] is a pre-trained language model based on bidirectional transformer architecture [20]. This model overcame many state-of-the-art models at the time it was presented. Its architecture allows the model to be an approach for different kinds of problems, such as multi-label [21], topic modeling [22], and sentence similarity [23].
  A pre-trained BERT model trained with Portuguese text data, Bertimbau [24], was selected for the experiments in this paper, considering that the collected corpus only contains statements in Portuguese.

E. Evaluation Metrics

Every experiment with learning algorithms requires an evaluation step to assess the model performance. In this sense, the whole dataset is traditionally split into training and testing data. The training samples serve as the knowledge base so that the classification models can learn their patterns during the training process. The testing samples are unknown to the model, being used to evaluate the model's performance in a real test scenario without biases.

Let a binary classification task in which the class labels stand for positive and negative labels. Considering the predictions on the test examples, we compute the true positives (TP), the true negatives (TN), the false positives (FP), and the false negatives (FN). We then derive the precision (Pr) to assess the proportion of positive examples correctly predicted by the classification model out of all examples predicted as positive, as described by Eq. (1):

$$\mathrm{Pr} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \tag{1}$$

where the closer the resulting value gets to 1, the better the model's performance. We also denote the recall (R), shown by Eq. (2), as the proportion of real examples correctly classified (*TP*) by the underlying model in relation to all real examples (*TP + FN*).

$$\mathrm{R} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}. \tag{2}$$

The F1-Score, described in Eq. (3), is computed by the harmonic mean between the precision and recall values, which are complementary metrics:

$$F_1 = 2\frac{\text{Pr} \cdot \text{R}}{\text{Pr} + \text{R}}. \quad (3)$$

As the proposed methodology is related to a multi-label classification task, we compute the F1-score by considering each label individually. Then, the final score is determined by taking the weighted arithmetic mean for each label. As the obtained corpus is naturally imbalanced in relation to its labels, the Macro F1-score is used, so when computing the final average, every label has the same weight.

IV. Experimental Results

We performed experiments to validate the proposed methodology and to compare the performance of the considered multi-label classification models. The proposed methodology was developed in Python 3.10 environment with the following auxiliary libraries: Gensim[4], Transformers[5], Tensorflow[6], Keras[7], Scikit-Learn[8], XGBoost[9], Numpy[10], and Pandas[11]. The experiments were performed on an Intel Core i5 9th generation CPU, GeForce RTX 3060 GPU, and 16 GB RAM memory computer. We made the source code and the corpus available in a public repository in GitLab[12].

Stratified K-Fold Cross Validation was used to evaluate the performances of the classification models SVM, Gradient Boosting, LSTM, BiLSTM, and BERT. For each fold, 20% of the training set was used as a validation set, which was used to optimize the hyperparameters for each model. The early stopping strategy was employed to halt the training when the validation loss did not increase in 5 epochs. After obtaining the best hyperparameter values, a new classifier was trained using the entire training set prior to the predictions using the testing set. Table II presents the optimal hyperparameter values for each model in each fold.

After the experiments were performed, each classifier was compared using the weighted and macro F1-Scores, and the results are presented in Tables III and IV. Table IV presents the metric applied to each label individually, where it can be noticed that the class labels with a small number of samples presented lower F1-Score values. This is explained by the low representativeness of those class labels in the training set, which affected the learning of patterns by the classification models. Therefore, we decided to remove the instances associated with the minor class labels from further experiments: Simple Geometry, Sorting, Recursion, IO, and Structs.

[4]https://radimrehurek.com/gensim/
[5]https://huggingface.co/docs/transformers/index
[6]https://www.tensorflow.org
[7]https://keras.io
[8]https://scikit-learn.org
[9]https://xgboost.readthedocs.io/en/stable
[10]https://numpy.org
[11]https://pandas.pydata.org
[12]https://gitlab.com/gvic-unb/oj-problem-classification

In terms of an educational context, the removal of those class labels is not crucial, considering the goals of our research. The label "sorting" is not covered in the introductory programming course at the Universidade de Brasília, being contemplated in the following programming course (Data Structures). Although the label "recursion" is taught in introductory programming courses, there are a few problems under this category due to a limitation in Beecrowd to ensure that the solutions submitted by the students make explicit use of recursion. Problems with the label "structs" can also be implemented using arrays, but most of the problems of this category belong to the major category of Beecrowd "Data Structures". The problems from the class label "IO" only focus on the formatting of input and output information, which is a common step for every problem from an OJ. Thus, this category is not very relevant for lecturing or learning computational thinking by means of computational algorithms. The "Simple Geometry" class label is a subtopic from the "Simple Math" class label, which embraces solutions that involve trigonometry.

Despite the related works on literature employed other datasets in the same task, we brought other papers' results for the matter of baseline. In Lobanov et al. [6], which used both problem statement and source code as input for the final model, the final F1-Score obtained was 0.532. In Iacob et al. [7], which proposed an architecture for text and code classification for competitive programming problems, obtained an F1-Score of 0.62 for text classification. In Iancu et al. [8], many models were compared for tag prediction of problem statements; among various metrics collected, the best F1-Score presented was 0.502, although the authors highlighted the best model as the one that presented 0.762 weighted Hamming Score and 0.489 F1-Score.

The obtained results allow us to verify that the built corpus presents a few problem statements, which affected the overall performance of the classification models, as shown previously in the metrics of labels with few samples in Table IV. The size of the corpus is justified by the need for more available datasets with beginner-level problems due to most of OJs' repositories of problems being made for mature competitors and advanced programming topics, as studied in competitive programming. However, the multi-label text classification of programming problems has proven to be challenging even in those kinds of programming problems with more statements available. As our research focused on beginner programming problems, we could obtain satisfactory results by considering only the statements and assuming they are simpler to interpret than those associated with competitive programming problems.

From an educational point of view, this paper brings important benefits to faculties and students. For teachers, the availability of a set of programming questions correctly labeled searches for relevant exercise topics is easier, allowing them to direct their students to questions that address precisely the discussed topics. In this way, it reduces the professors' efforts when spending time reviewing each question individually to ensure its relevance to the planned content. In addition, the

Table II
HYPERPARAMETER VALUES DETERMINED IN EACH FOLD.

Model	Hyperparameter	Optimal Values per Fold
LSTM	Learning Rate	[10^{-2}, 10^{-2}, 10^{-2}, 10^{-2}, 10^{-2}]
LSTM	Units	[64, 64, 64, 64, 64]
LSTM	Batch Size	[64, 64, 64, 64, 64]
BiLSTM	Learning Rate	[10^{-3}, 10^{-3}, 10^{-3}, 10^{-3}, 10^{-3}]
BiLSTM	Units	[32, 32, 32, 32, 32]
BiLSTM	Batch Size	[32, 32, 32, 32, 32]
BERT	Learning Rate	[10^{-5}, 10^{-5}, 10^{-4}, 10^{-4}, 10^{-5}]

Table III
WEIGHTED AND MACRO F1-SCORE

	SVM	Gradient Boosting	LSTM	BiLSTM	BERT
Weighted F1-Score	0.62	0.63	0.61	0.59	0.60
Macro F1-Score	0.39	0.38	0.25	0.23	0.25

Table IV
CLASSIFICATION RESULTS: AVERAGE AND STANDARD DEVIATION OF MACRO F1-SCORE PER LABEL.

Label	SVM	Gradient Boosting	LSTM	BiLSTM	BERT
Loops	0.82 ± 0.07	0.82 ± 0.03	0.78 ± 0.03	0.78 ± 0.03	0.78 ± 0.03
Conditionals	0.61 ± 0.06	0.70 ± 0.07	0.72 ± 0.03	0.72 ± 0.03	0.72 ± 0.03
Simple Math	0.63 ± 0.09	0.59 ± 0.14	0.62 ± 0.04	0.62 ± 0.04	0.62 ± 0.04
Arrays 1d	0.53 ± 0.09	0.23 ± 0.29	0.17 ± 0.07	0.17 ± 0.07	0.18 ± 0.08
Strings	0.20 ± 0.27	0.10 ± 0.20	0.05 ± 0.10	0.18 ± 0.09	0.13 ± 0.11
Arrays 2d	0.74 ± 0.10	0.72 ± 0.21	0.15 ± 0.06	0.15 ± 0.06	0.15 ± 0.05
Simple Geometry	0.08 ± 0.16	0.00 ± 0.00	0.07 ± 0.07	0.12 ± 0.05	0.12 ± 0.05
Sorting	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.01	0.01 ± 0.02
Recursion	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
IO	0.67 ± 0.37	0.97 ± 0.06	0.00 ± 0.00	0.00 ± 0.00	0.01 ± 0.02
Structs	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
Macro F1-Score	0.39 ± 0.04	0.38 ± 0.04	0.23 ± 0.01	0.25 ± 0.01	0.25 ± 0.02

Table V
CLASSIFICATION RESULTS: AVERAGE AND STANDARD DEVIATION OF MACRO F1-SCORE ACROSS RELEVANT LABELS

Label	SVM	Gradient Boosting	LSTM	BiLSTM	BERT
Loops	0.87 ± 0.04	0.82 ± 0.03	0.78 ± 0.03	0.78 ± 0.03	0.78 ± 0.03
Conditionals	0.75 ± 0.04	0.70 ± 0.07	0.72 ± 0.03	0.72 ± 0.03	0.72 ± 0.03
Simple Math	0.60 ± 0.09	0.59 ± 0.14	0.62 ± 0.04	0.62 ± 0.04	0.62 ± 0.04
Strings	0.13 ± 0.17	0.10 ± 0.20	0.18 ± 0.09	0.18 ± 0.09	0.19 ± 0.10
Array1D	0.33 ± 0.28	0.23 ± 0.29	0.17 ± 0.07	0.17 ± 0.07	0.17 ± 0.07
Array2D	0.62 ± 0.07	0.62 ± 0.17	0.26 ± 0.05	0.26 ± 0.05	0.26 ± 0.05
Macro F1-Score	0.55 ± 0.08	0.51 ± 0.06	0.45 ± 0.02	0.45 ± 0.02	0.46 ± 0.02

proposed classification approach allows faculties to focus more on other activities that cannot be so benefited by automatic approaches, such as working on didactic and preparing suitable content to make the teaching process more efficient.

For students, especially those seeking autonomy in their studies, the detailed labeling of OJ problems is equally valuable. Therefore, the solution proposed in this paper allows students to find problems that correspond directly to the topic they are studying without wasting time on questions they cannot solve or that require knowledge beyond their current level. For example, topics such as "loops", which are often not identified in online judge systems, can be specifically highlighted by the model, facilitating more focused and efficient study. As a result, this approach not only optimizes the time of faculties and students but also promotes more targeted and effective learning, aligning the educational resources available with the specific needs of teaching and learning.

V. CONCLUSION

This paper introduced a methodology for predicting the categories of computer programming problems based on their statements. The focus was on analyzing the statements of beginner problems, which were collected from Beecrowd, a well-known OJ that is suitable to be employed in introductory programming courses. We modeled this problem as a multi-label classification task, which required the manual annotation of the problems' statements by taking into account the topics commonly covered in introductory programming, such as conditionals, loops, arrays, etc. After that, we selected different classification models and designed specific pipelines to enable the processing of statements that were originally in natural language by the underlying classifiers.

We performed experiments to compare the performance of the classifiers based on SVM, Gradient Boosting, LSTM, BiLSTM, and BERT. The results showed that, at the current state of available data, the model that presented the best results was the SVM, with the F1-Score of 0.55. As mentioned in Section IV, the low number of samples of problem statements affected the models' performance. For the state-of-the-art models, such as LSTM and BERT, that rely on pre-trained models, the lack of training data results in poor performances; thus, a larger corpus could improve the representativeness over the class labels, leading to improving their metrics and broadening the debate over results.

In the context of teaching programming to beginner students, practice and problem solving are essential for improving programming skills, with the proposed methodology, the educational landscape undergoes a significant transformation, yielding benefits for both educators and learners. By facilitating the selection of relevant and proper programming problems, faculties can focus on didactic, while students, especially those who study independently, can focus their efforts on the desired question. Therefore, the methodology presented in this paper not only aims to spare time of faculties and students, but also to improve the quality of learning, assuring that the studied subject is aligned with the students' necessities and levels of knowledge, fostering targeted and effective teaching.

Next steps as future work relies on analyzing the performance of the proposed method when the class labels were manually annotated based on Python source code since Python is also used in introductory programming courses. Moreover, we plan to extend this study to problems related to data structures and competitive programming, involving the proposal of deep learning models as the statements are more challenging since the underlying algorithmic solutions are generally more complex.

REFERENCES

[1] H. Zhang, M. Zhang, F. chao Meng, X. quan Zhou, and D. hui Chu, "Application of the online judge technology in programming experimental teaching," in *Proceedings of the 2020 5th International Conference on Modern Management and Education Technology (MMET 2020)*. Atlantis Press, 2020, pp. 296–299. [Online]. Available: https://doi.org/10.2991/assehr.k.201023.059

[2] J. L. Oliveira, A. P. Ambrósio, U. Silva, J. Brancher, and J. J. Franco, "Undergraduate students' effectiveness in an institution with high dropout index," in *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2020, pp. 1–7.

[3] M. G. Carneiro, B. L. Dutra, J. G. S. Paiva, P. H. R. Gabriel, and R. D. Araújo, "Educational data mining to support identification and prevention of academic retention and dropout: a case study in introductory programming," *Revista Brasileira de Informática na Educação*, vol. 30, pp. 379–395, 2022.

[4] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–34, 2018.

[5] I. N. Bandeira, T. V. Machado, V. F. Dullens, and E. D. Canedo, "Competitive programming: A teaching methodology analysis applied to first-year programming classes," in *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2019, pp. 1–8.

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

[7] R. C. A. Iacob, V. C. Monea, D. Radulescu, A.-F. Ceapa, T. Rebedea, and S. Trausan-Matu, "Algolabel: A large dataset for multi-label classification of algorithmic challenges," *Mathematics*, vol. 8, no. 11, 2020. [Online]. Available: https://www.mdpi.com/2227-7390/8/11/1995

[8] B. Iancu, G. Mazzola, K. Psarakis, and P. Soilis, "Multi-label classification for automatic tag prediction in the context of programming challenges," 2019.

[9] J. Figueiredo and F. García-Peñalvo, "Teaching and learning tools for introductory programming in university courses," in *2021 International Symposium on Computers in Education (SIIE)*. IEEE, 2021, pp. 1–6.

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

[11] S. Sudha, A. Arun Kumar, M. Muthu Nagappan, and R. Suresh, "Classification and recommendation of competitive programming problems using cnn," *Communications in Computer and Information Science*, p. 262–272, Dec 2017.

[12] A. Majd, M. Vahidi-Asla, A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani, "Code4bench: A multidimensional benchmark of codeforces data for different program analysis techniques," *Journal of Computer Languages*, vol. 53, pp. 38–52, 2019.

[13] R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen, "Ablation studies in artificial neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1901.08644

[14] T. Lokat, D. Prajapati, and S. Labde, "Tag prediction of competitive programming problems using deep learning techniques," *Cornell University*, 2023.

[15] V. Balakrishnan and E. Lloyd-Yemoh, "Stemming and lemmatization: A comparison of retrieval performances," vol. 2, no. 3, Apr 2014.

[16] B. Kernighan and D. Ritchie, *The C Programming Language*, ser. Prentice-Hall Software Series. Prentice Hall, 1988.

[17] C. Cortes and V. Vapnik, "Support-vector networks," vol. 20, no. 3, Sep 1995.

[18] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," vol. 29, no. 5, Oct 2001.

[19] N. Hartmann, E. Fonseca, C. Shulby, M. Treviso, J. Rodrigues, and S. Aluisio, "Portuguese word embeddings: Evaluating on word analogies and natural language tasks," 2017.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[21] S. Amin, G. Neumann, K. A. Dunfield, A. Vechkaeva, K. A. Chapman, and M. K. Wixted, "Mlt-dfki at clef ehealth 2019: Multi-label classification of icd-10 codes with bert," in *Proceedings of the 20th Conference and Labs of the Evaluation Forum (Working Notes)*, 2019, pp. 1–15. [Online]. Available: http://ceur-ws.org/Vol-2380/paper_67.pdf

[22] M. Grootendorst, "Bertopic: Neural topic modeling with a class-based tf-idf procedure," 2022.

[23] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," 2019.

[24] F. Souza, R. Nogueira, and R. Lotufo, "Portuguese named entity recognition using bert-crf," 2020.