PIPE: Predicting Logical Programming Errors in Programming Exercises

Dezhuang Miao
School of Data Science and Engineering
East China Normal University
Shanghai, China
{51185100025,51195100031}@stu.ecnu.edu.cn

Yu Dong
School of Data Science and Engineering
East China Normal University
Shanghai, China

Xuesong Lu*
School of Data Science and Engineering
East China Normal University
Shanghai, China
xslu@dase.ecnu.edu.cn

ABSTRACT

In colleges, programming is increasingly becoming a general education course of almost all STEM majors as well as some art majors, resulting in an emerging demand for scalable programming education. To support scalable education, teaching activities such as grading and feedback have to be automated. Recently, online judge systems have been extensively used for programming training, because they are able to automatically evaluate the correctness of programs in real time and thereby make grading work scalable. However, existing online judge systems lack of the ability to give effective feedback on logical programming errors. As such, instructors and teaching assistants are still overwhelmed by the work of helping students fix programs, especially for those novice students. To tackle the challenge, we develop **PIPE**, a deep learning model that is able to **P**redict log**I**cal **P**rogramming **E**rrors in student programs. The model seamlessly integrates a representation learning model for obtaining the latent feature of a program and a multi-label classification model for predicting the error types in the program, thereby allowing end-to-end learning and prediction. We use the C programs submitted in our online judge system to train PIPE, and demonstrate its superior performance over the baseline models. We use PIPE to implement the error-feedback feature in our online judge system and enable automated feedback on logical programming errors to the students.

Keywords

Online Judge System, Scalable Programming Training, Logical Programming Error, Automated Error Feedback, Deep Learning

1. INTRODUCTION

The evolution of big data and AI technologies has made programming a ubiquitous skill in almost all industries and thereby led to a massive demand for programming professionals. In colleges and MOOC platforms, programming is no longer a professional course of ICT-related majors and becoming a general education course for all STEM majors and even some art majors. As such there is an urgent need for scalable programming teaching methodologies and learning tools to cater for the increasingly overwhelmed teaching workload. One of the most important mechanisms to achieve scalable teaching is automation. For example, online judge (OJ) systems [22], which are originally used for competitive programming contests, have now been extensively used in programming training mainly due to their ability of automated program evaluation. Given a programming exercise and a set of predefined input, the judge system evaluates a submitted program[1] by comparing the expected output with the actual output obtained from the execution of the program. Such a pair of predefined input and output is called a *test case*. This feature can largely reduce the grading workload of instructors and teaching assistants, and thus make class sizes scalable to some extent.

Despite the ability of automated program evaluation, existing OJ systems often provide to students next-to-zero feedback on programming errors when they submit an incorrect program. We refer to an "incorrect program" as a piece of code that is compilable but generates wrong output for the test cases. The errors in such a program are often termed as "logical errors", as opposed to "common errors" that are related to the use of incorrect syntax. In our teaching, we observe that the students can easily fix common errors with the help of an IDE, but are quite struggling when dealing with logical errors. In the latter case, existing OJ systems only show to students feedback such as "Wrong Answer" and "Runtime Error", and cannot provide any information on detailed types of errors. The problem is even severer in case of a quiz, where students are not allowed to check the test cases[2]. As such, students, especially novices, rely heavily on instructors and teaching assistants to help them fix logical errors, which prevents programming training from becoming more scalable. This has motivated us to develop an automated tool for logical error feedback.

*Xuesong Lu is the corresponding author.

Dezhuang Miao, Yu Dong and Xuesong Lu "PIPE: Predicting Logical Programming Errors in Programming Exercises" In: *Proceedings of The 13th International Conference on Educational Data Mining (EDM 2020)*, Anna N. Rafferty, Jacob Whitehill, Violetta Cavalli-Sforza, and Cristobal Romero (eds.) 2020, pp. 473 - 479

[1]Below we use the term 'program' and 'code' interchangeably.
[2]Otherwise, students may fake the output.

In this work, we develop a deep learning model, PIPE, that is able to predict the detailed types of logical errors and therefore can be deployed in OJ systems to enable automated error feedback. We collect the C programs that are compilable but fail to pass the test cases, submitted by the students in our OJ system. We manually label the programs with a set of predefined types of logical errors. Since each program may contain multiple types of logical errors, we regard the prediction task as a multi-label classification problem. Therefore, the architecture of PIPE is inspired by the work of *code2vec* [2] and *C2AE* [24], which are originally developed to predict semantic properties of code snippets and boost the performance of multi-label classification tasks, respectively. In particular, we first use the idea of code2vec to obtain the latent representations of C programs. For each program, in addition to just embedding the code itself, we embed two more types of information in the model input, namely, the corresponding exercise identity and the evaluation results returned by the judge system. Then following the idea in C2AE, we also transform the corresponding error types into latent representations with an encoder, and jointly learn deep latent spaces together with the representations of the C programs. The error types are finally reconstructed from the deep latent spaces using a decoder, which are then used to compute the loss function with the true error labels for backpropagation. Thanks to the seamless integration of code2vec and C2AE, PIPE allows end-to-end training and prediction. We then conduct extensive experiments to demonstrate PIPE' superior performance over the baseline models. We deploy PIPE in our OJ system and show the usage of the automated error-feedback feature.

The rest of the paper is organized as follows. Section 2 presents the detailed architecture of PIPE. Then Section 3 describes the real dataset used in our experiments and presents the performance evaluation of the proposed model. Section 4 gives a brief literature review of related work, and finally Section 5 concludes the work and points out some future work to improve the feature of automated error feedback.

2. THE PIPE MODEL

We describe the architecture and the optimization method of PIPE in this section.

2.1 Architecture Overview of PIPE

Since each program may contain more than one logical error, we regard the error prediction task as a multi-label classification problem. We use the structure of the C2AE model as the backbone of PIPE. The C2AE model performs joint input and output embedding which correlates the features and the labels, and hence achieves the new state-of-the-art performance on multi-label classification tasks. In particular, PIPE uses a feature mapping $\mathbf{F}_x$ to transform the programs $\mathbf{X}$ and uses a encoding function $\mathbf{F}_e$ to transform the corresponding labels $\mathbf{Y}$ of the logical errors into deep latent spaces $\mathbf{L}$. Then it utilizes Deep Canonical Correlation Analysis [3] (DCCA) to learn $\mathbf{L}$ for joint program and label embedding. Finally, PIPE uses a decoding function $\mathbf{F}_d$ to recover the label outputs from $\mathbf{L}$, where $\mathbf{F}_e$ and $\mathbf{F}_d$ thus compose an autoencoder for the reconstruction of the labels. The objective function of PIPE is formulated as follows:

$$\Theta = \min_{\mathbf{F}_x, \mathbf{F}_e, \mathbf{F}_d} \Phi(\mathbf{F}_x, \mathbf{F}_e) + \alpha\Gamma(\mathbf{F}_e, \mathbf{F}_d) \tag{1}$$

where Θ represents the total loss of PIPE, $\Phi(\mathbf{F}_x, \mathbf{F}_e)$ and $\Gamma(\mathbf{F}_e, \mathbf{F}_d)$ denote the loss at the latent space layer for associating features and labels, and the loss at the output layer for reconstructing the labels, respectively. The hyperparameter α balances the two components of the objective function. Once the training is completed, PIPE can throw away the component pertaining to $\mathbf{F}_e$ and use $\mathbf{F}_d(\mathbf{F}_x)$ to predict the logical errors in each program.

In PIPE, we simply use a fully-connected network to implement the functions $\mathbf{F}_e$ and $\mathbf{F}_d$, respectively. We further leverage the idea in code2vec to implement the feature mapping $\mathbf{F}_x$ for program representation learning, as shown in the part surrounded by the red dotted line in Figure 1. Rather than directly embed the source code, code2vec first decomposes the program into a collection of paths in its abstract syntax tree (AST) and then learns to aggregate the paths into a single program vector. The method is proved to better capture the regularities that reflect common program patterns and lower the learning effort, compared to learning over original program text. To capture more information about the logical errors pertaining to each particular exercise, we embed the exercise identity and the evaluation results on the test cases returned by the judge system, and concatenate them with the program vector to form a unified feature vector, which we call *program embedding*. Then the program embeddings are transformed into the latent space $\mathbf{L}$ using a fully-connected layer. The architecture overview of PIPE is shown in Figure 1.

2.2 Program Embedding

Firstly, we need to transform the programs into vectorized representations. Following the method in previous work [2, 19], we compile each C program X and parse it to construct an AST. An AST is a tree representation of the abstract syntactic structure of source code, where the nodes denote the various elements appearing in the original source code. By traversing between the AST leaves, we can obtain multiple syntactic paths that represent the context of the corresponding C program. Then the syntactic paths are converted into context vectors, which are used as one type of input to learn the values of program embedding. Each context vector $\mathbf{c}_i \in \mathbb{R}^{3d}$ is concatenated to using three individual vectors, as depicted in Equation 2,

$$\mathbf{c}_i = [\mathbf{s}_i, \mathbf{p}_i, \mathbf{t}_i] \tag{2}$$

where $\mathbf{s}_i \in \mathbb{R}^d$, $\mathbf{p}_i \in \mathbb{R}^d$ and $\mathbf{t}_i \in \mathbb{R}^d$ are the vectorized representation of the source node, the path and the target node of the corresponding syntactic path, respectively. Then each context vector $\mathbf{c}_i \in \mathbb{R}^{3d}$ is transformed into a combined context vector $\hat{\mathbf{c}}_i \in \mathbb{R}^d$ using a shared fully-connected layer, and finally all the combined context vectors are aggregated into a single program vector $\boldsymbol{v}_p \in \mathbb{R}^d$ using the following attention mechanism,

$$\begin{aligned} \boldsymbol{v}_p &= \sum_{i=1}^{n} \alpha_i \cdot \hat{\mathbf{c}}_i \\ s.t. \quad \alpha_i &= \frac{exp(\hat{\mathbf{c}}_i^T \cdot \boldsymbol{a})}{\sum_{j=1}^{n} exp(\hat{\mathbf{c}}_j^T \cdot \boldsymbol{a})} \end{aligned} \tag{3}$$

where $\boldsymbol{a}$ is the attention vector, α_i is the attention weight and n is the number of combined context vectors. The attention vector learns the importance of each combined context

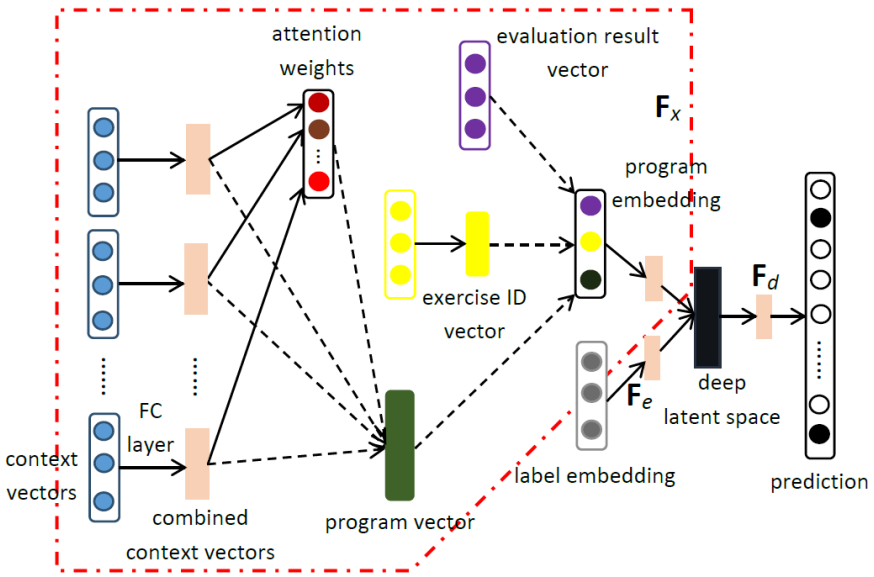

Figure 1: The overview of PIPE's architecture.

vector. To capture more information pertaining to the errors in the program w.r.t a particular exercise, we further embed the exercise identity and the evaluation results on the test cases into two vectors $\boldsymbol{v}_e \in \mathbb{R}^{d_e}$ and $\boldsymbol{v}_r \in \{0,1\}^{d_r}$, respectively. $\boldsymbol{v}_r$ is a bit vector where 1 indicates a correct output on the test case and 0 otherwise. The vector $\boldsymbol{v}_e$ restricts the exercise-related characteristics such as functions and algorithmic logic, and the vector $\boldsymbol{v}_r$ captures specific types of errors since similar logical errors should result in wrong output on similar test cases. Eventually the program embedding $\boldsymbol{v}_X$ is obtained by concatenating $\boldsymbol{v}_p$, $\boldsymbol{v}_e$ and $\boldsymbol{v}_r$, as formulated in Equation 4.

$$\boldsymbol{v}_X = [\boldsymbol{v}_p, \boldsymbol{v}_e, \boldsymbol{v}_r] \tag{4}$$

2.3 Learning Deep Latent Spaces for Joint Program & Label Embedding

Following the idea in the work [24], we learn deep latent spaces $\mathbf{L}$ to associate program embedding and label embedding, using Deep Canonical Correlation Analysis [7, 3] (DCCA). For each C program X, we simply represent its label Y as a bit vector $\boldsymbol{v}_Y \in \{0,1\}^N$, where N is the number of logical error types. The vector $\boldsymbol{v}_Y$ may contain multiple 1s since each program may have multiple types of logical errors. Then both the program embedding $\boldsymbol{v}_X$ and the label embedding $\boldsymbol{v}_Y$ are transformed into a latent vector of size l using a fully-connected layer with the tanh activation function. The holistic functions that mapping X and Y are refer to as $\mathbf{F}_x$ and $\mathbf{F}_e$, respectively, as depicted in Section 2.1. Then the objective function for correlating the latent representations are formulated as Equation 5,

$$\begin{aligned} &\Phi(\mathbf{F}_x, \mathbf{F}_e) = ||\mathbf{F}_x(\mathbf{X}) - \mathbf{F}_e(\mathbf{Y})||_F^2 \\ s.t. \quad &\mathbf{F}_x(\mathbf{X})\mathbf{F}_x(\mathbf{X})^T = \mathbf{F}_e(\mathbf{Y})\mathbf{F}_e(\mathbf{Y})^T = \mathbf{I}, \end{aligned} \tag{5}$$

where $\mathbf{I} \in \mathbb{R}^{l \times l}$ is the identity matrix. By solving the objective function, we enforce the deep latent space $\mathbf{L}$ to associate the programs $\mathbf{X}$ and the labels $\mathbf{Y}$, and hence $\mathbf{F}_x(X)$ can be used as the input to predicting the label Y.

2.4 Recovering Label Outputs from the Deep Latent Space

In the training phase, the output label $\hat{Y}$ is reconstructed from the latent representation $\mathbf{F}_e(Y)$ using a decoder $\mathbf{F}_d$, which is simply implemented as a fully-connected layer in this work. In the original work [24], the model uses a label-correlation aware function to calculate the label reconstruction loss $\Gamma(\mathbf{F}_e, \mathbf{F}_d)$ at the output layer, in order to better preserve the label co-occurrence information for multi-label classification task. However, we notice that there is no strong correlation between the labels of the logical error types in our dataset. Hence, we insteadly use the multi-label cross-entropy function to calculate the label reconstruction loss, as depicted in Equation 6,

$$\begin{aligned} \Gamma(\mathbf{F}_e, \mathbf{F}_d) &= \frac{1}{|\mathbf{Y}|} \sum_{j=1}^{|\mathbf{Y}|} E_j \\ E_j &= -\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \end{aligned} \tag{6}$$

where $|\mathbf{Y}|$ is the number of training instances, N is the number of logical error types and y_i equals to 1 if the program contains the corresponding error and equals to 0 otherwise. We use the Sigmoid activation function in the output layer. By solving the loss function, we enforce the autoencoder $\mathbf{F}_d(\mathbf{F}_e(Y))$ to reconstruct the label of the logical error types. Since the latent representation $\mathbf{F}_e(Y)$ and $\mathbf{F}_x(X)$ are highly correlated after the training is completed, $\mathbf{F}_d(\mathbf{F}_x(X))$ can be used to predict the error types of a given C program X.

2.5 Optimization

The gradient of the label-reconstruction loss $\Gamma(\mathbf{F}_e, \mathbf{F}_d)$ can be easily calculated since it is a cross-entropy function. Following the method in [24], the gradient of the association aware loss $\Phi(\mathbf{F}_x, \mathbf{F}_e)$ in the latent space can be calculated with the help of Lagrange multipliers [20]. In particular,

$\Phi(\mathbf{F}_x, \mathbf{F}_e)$ is first reformulated as

$$\Phi(\mathbf{F}_x, \mathbf{F}_e) = \mathrm{Tr}(\mathbf{C}_1^T \mathbf{C}_1) + \lambda \mathrm{Tr}(\mathbf{C}_2^T \mathbf{C}_2 + \mathbf{C}_3^T \mathbf{C}_3), \quad (7)$$

where

$$\begin{aligned} \mathbf{C}_1 &= \mathbf{F}_x(\mathbf{X}) - \mathbf{F}_e(\mathbf{Y}) \\ \mathbf{C}_2 &= \mathbf{F}_x(\mathbf{X})\mathbf{F}_x(\mathbf{X})^T - \mathbf{I} \\ \mathbf{C}_3 &= \mathbf{F}_e(\mathbf{Y})\mathbf{F}_e(\mathbf{Y})^T - \mathbf{I}. \end{aligned}$$

We fix λ to 0.5 in accordance with [24]. Then the gradient w.r.t $\mathbf{F}_x(\mathbf{X})$ and $\mathbf{F}_e(\mathbf{Y})$ can be calculated as

$$\begin{aligned} \frac{\partial \Phi(\mathbf{F}_x, \mathbf{F}_e)}{\partial \mathbf{F}_x(\mathbf{X})} &= 2\mathbf{C}_1 + 4\lambda \mathbf{F}_x(\mathbf{X})\mathbf{C}_2 \\ \frac{\partial \Phi(\mathbf{F}_x, \mathbf{F}_e)}{\partial \mathbf{F}_e(\mathbf{Y})} &= 2\mathbf{C}_1 + 4\lambda \mathbf{F}_e(\mathbf{Y})\mathbf{C}_3. \end{aligned} \quad (8)$$

3. PERFORMANCE EVALUATION

In this section, we evaluate the performance of PIPE on a real dataset collected in our OJ system, and report the experimental results by comparing PIPE with other baseline methods. In the end, we demonstrate an example of the error-feedback feature implemented with PIPE in our OJ system.

3.1 The Dataset and Settings

The real dataset is collected from an introductory C programming course for undergraduate students in our school. The course uses heavily an OJ system to train the students, and we collect all the programs with logical errors submitted by the 29 enrolled students throughout one entire semester. Most programs have less than 50 lines. After cleaning work such as removing repeated submissions of programs with minor changes, we obtain 5196 C programs pertaining to 200 programming exercises. We have carefully designed for each exercise 10 test cases. Then we randomly disseminate the URLs of these programs to 17 senior students and ask them to annotate the labels of logical errors. In order to guarantee the correctness of annotation, they are allowed to freely run the programs and check the output of the test cases. Also, each program is annotated and cross validated by three students. The annotation work takes roughly two months.

After annotation, we observe that 5125 out of the 5196 programs fall into 10 major types of logical errors. The remaining 71 programs have very uncommon errors and are thus discarded from the dataset. The 10 types of logical errors are summarized and explained as follows. The distribution of the numbers of the errors is plotted in Figure 2.

1. **Incorrect input variables** - mainly due to misuse of the '&' operator in the `scanf()` function.
2. **No output** - forgetting to write output.
3. **Incorrect output format** - output format not complying with the exercise requirements.
4. **Incorrect initialization** - errors related to incorrect initialization of variables.
5. **Incorrect data types** - mainly due to undesired type conversions.
6. **Incorrect data precision** - mainly due to loss of precision during calculation.

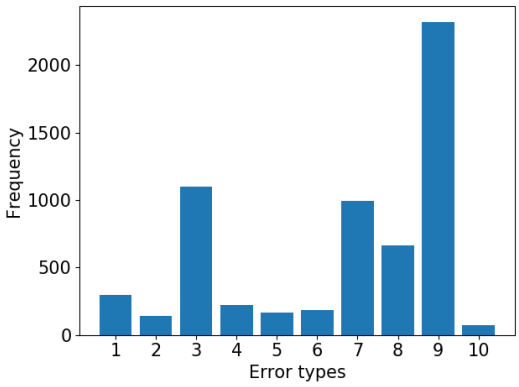

Figure 2: Distribution of the numbers of error types.

7. **Incorrect loops** - loop-related errors such as incorrect termination condition and incorrect step size of iteration.
8. **Incorrect branches** - errors due to incorrect conditional statements.
9. **Incorrect logic** - program's logic not complying with the exercise.
10. **Incorrect operators** - misuse of operators.

The dataset is randomly splitted into training, validation and testing set with proportion $6:2:2$ in the experiments. We implement PIPE and four baseline models for comparison using Python 3.6 and TensorFlow 1.13. The first three baseline models are the original code2vec model, code2vec plus exercise identity embedding, and code2vec plus exercise identity and evaluation result embeddings. The fourth model is the same as PIPE except that we use the original label-reconstruction loss in the C2AE model. At the input we randomly choose 200 context vectors for each program. At the output all the models are modified to cater for the multi-label classification task accordingly. The batch size is 64 and the learning rate is 0.001. We use the Adam algorithm for optimization. All other optimal hyperparameter settings are determined via the validation process, including the thresholds for rounding to the predicted labels. The metrics of interest are therefore precision, recall and F1 score, in accordance with [2, 24]. We also measure the averaged percentage of exact match, which means that the predicted types of errors are exactly the same as the ground truth for a given program. All experiments are conducted using a normal PC installed with an Intel Core i7-8550U CPU and 8GB RAM.

3.2 Main Results

The main results are presented in Table 1. For PIPE, the program embedding size is set to 138[3], the size of the latent space is 69, and the balancing factor $\alpha = 0.1$. For each model, we calculate seven metrics on the testing set, which are the averaged percentage of exact match, per-class precision (C-P), per-class recall (C-R), macro F1 score (Ma-F1), overall precision (O-P), overall recall (O-R) and micro

[3]program embedding(138)=program vector(64)+exercise ID vector(64)+evaluation result vector(10).

Model	Exact Match	C-P	C-R	Ma-F1	O-P	O-R	Mi-F1
code2vec	0.5735	0.4037	0.3671	0.3822	0.7013	0.6472	0.6714
code2vec + exercise ID	0.5643	0.3696	0.3427	0.3543	0.6978	0.6437	0.6687
code2vec + exercise ID + evaluation	0.5809	0.3798	0.3438	0.3592	0.7088	0.6441	0.6735
PIPE + C2AE loss	0.0	0.09817	0.3679	0.1528	0.2443	**0.8497**	0.3792
PIPE	**0.6259**	**0.4386**	**0.3984**	**0.4151**	**0.7527**	**0.7037**	**0.7255**

Table 1: Comparison between PIPE and baseline models.

F1 score (Mi-F1). We observe that PIPE performs constantly much better than the other models on all metrics. Although PIPE using original C2AE loss achieves the best overall recall, it has poor results on all other metrics. This is because the label-reconstruction loss in C2AE attempts to preserve the correlation between the labels and hence more error types are predicted. However, this also drastically reduces the precision and causes no case of exact match is predicted. The results prove the effectiveness of the seamless integration of code2vec and C2AE, as well as the use of cross-entropy for the label-reconstruction loss.

3.3 Sensitivity Analysis

We perform sensitivity analysis for three most important hyperparameters, that is, the size of program embedding $\boldsymbol{v}_X$, the size of the latent space l and the loss balancing factor α. For each of them, we fix the values of all other hyperparameters and vary it in the corresponding ranges.

The size of program embedding. The size of $\boldsymbol{v}_X$ equals to the sum of the size of program vector $\boldsymbol{v}_p$, the size of exercise identity vector $\boldsymbol{v}_e$ and the size of evaluation result vector $\boldsymbol{v}_r$. The size of $\boldsymbol{v}_r$ is fixed to 10 since each exercise has 10 test cases, and the size of $\boldsymbol{v}_e$ is fixed to 64 for the sake of simplicity. We then vary the size of $\boldsymbol{v}_p$ in $(64, 128, 192, 256)$, following the setting in [2]. Therefore, the size of $\boldsymbol{v}_X$ varies in $(138, 202, 266, 330)$. The results are presented in Figure 3. We observe that PIPE prefers smaller program embedding size on all metrics.

The size of the latent space. Following [24], we measure the size of the latent space $\mathbf{L}$ as its ratio to the size of program embedding, i.e., $l/|\boldsymbol{v}_X|$. We vary the ratio in the range $[0.1, 1]$ with increments 0.1, and report the results in Figure 4. We observe that roughly all the metrics first increase and then decrease as the size of latent space increases. Overall taking half of the size of program embedding achieves the best performance.

The balancing factor α. We vary α in the range $[0.1, 1]$ with increments 0.1. We also set $\alpha = 0.05$ to show the performance on the very small value. The results are presented in Figure 5. We observe that $\alpha = 0.1$ achieves the best overall performance. Further increasing α would break the balance between the losses of the two parts.

3.4 Demonstration

We have implemented the error-feedback feature in our OJ system using PIPE. Figure 6 shows the usage of the feature. In case of an incorrect submission, a student may check the possible errors predicted by the system and modify the program accordingly, where each type of error is associated with a probability. For example in Figure 6, the student may have 99.04% chance to write incorrect loops, and may also have 93.21% chance to lose precision during calculation, etc.

4. RELATED WORK

Code error prediction (or detection) is a branch of automatic software repair (ASR) [13], which is a long and active research area of software engineering. ASR is with respect to an oracle that is able to determine whether the execution of a given program is correct. Among various types of oracles, test suites or test cases are mostly used in recent ASR researches, which are also used as an important input feature in our PIPE model. Traditionally, test-suite-based methods can be broadly classified into two categories, i.e., search-based methodology [9, 8, 11] and semantics-based methodology [14, 5, 12]. The former category of methods explore a search space of programs to find the most suitable repair candidate that can pass the test cases; the latter category of methods synthesize a repair candidate using semantic information via symbolic execution and constraint solving. All these algorithms are specifically designed for repairing software with thousands to hundreds of thousands lines of code, and often cannot be directly applied in the setting of programming education. For instance, search-based methodologies typically rely on redundancy presented in other parts of the program to limit the search space, whereas redundant code is hardly observed in a students' program. Moreover, rather than directly correcting the bugs in students' programs, providing hints for students to find the errors is more preferable for education purpose. Therefore, the methods for ASR are somehow too heavy to cater for our prediction requirements.

In the past few years, research at the intersection of deep learning and programming languages has been driven by the availability of "big code". Massive source code obtained from the sites such as GitHub as well as some MOOC courses facilitates the design of learnable probabilistic models that exploit abundant patterns of code. These models are then applied to various applications, including program repair [1, 21], clone detection [10] and code synthesis [18], etc.

Training deep learning models to provide feedback to student code has recently drawn attention of both researchers and programming educators. For example, the work of [4] trains recurrent neural networks to automatically detect and correct syntax errors in programming assignments. The models are first trained on syntactically correct student programs and then are used to predict the correct token sequences given the prefix token sequence of a student program with syntax errors. Similarly, the work of [6] trains a multi-layered sequence-to-sequence neural network with attention to predict erroneous locations in student programs and attempts to fix the errors with correct statements. The

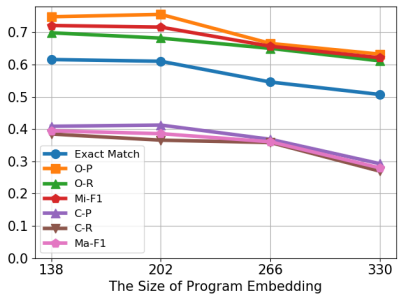

Figure 3: Varying the size of program embedding.

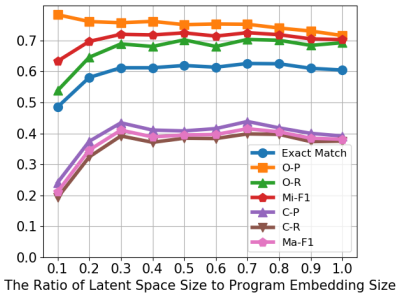

Figure 4: Varying the ratio of the size of latent space to the size of program embedding.

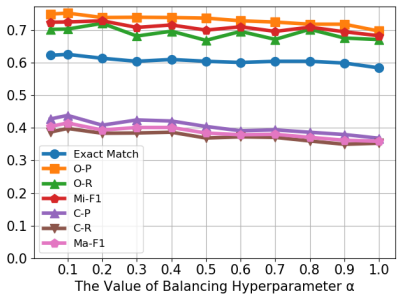

Figure 5: Varying the balancing factor α.

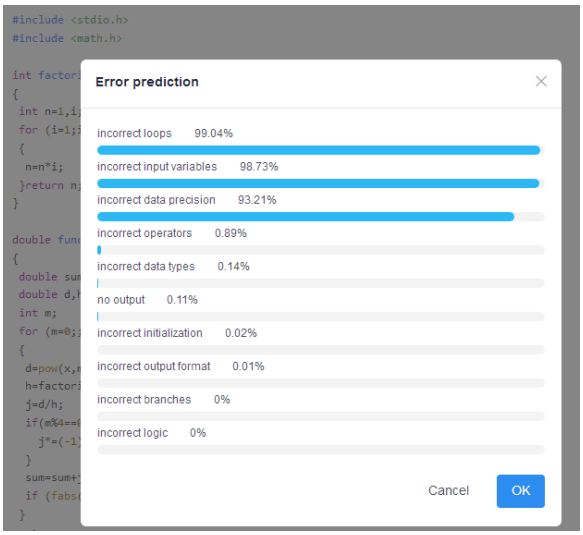

Figure 6: The usage of the error-feedback feature.

model requires to construct training pairs of syntactically incorrect program and the corresponding syntactically correct program. Since both works are focused on detecting and fixing syntax errors, they cannot generate abstract syntax trees for program embedding and thus directly use the language tokens in the original program text. The work in [16] trains an autoencoder to learn joint embedding of program states and programs. The embedding are then used as the input to train an RNN-based model, which can automatically propagate teacher feedback to similar programs. While the focus of their work is representation learning of program state, our model allows end-to-end learning and prediction of logical errors in programs. Other work pertaining to program feedback in the educational setting include [15, 17, 23].

5. CONCLUSIONS

To automate the feedback on logical programming errors in OJ systems, we develop PIPE, a deep learning model that is able to predict the types of errors in students' programs. PIPE seamlessly integrates program representation learning into a multi-label classification model, and thereby can perform end-to-end learning and prediction. To boost the prediction performance, PIPE also incorporates the exercise identity and the evaluation results on the test cases into the program representation, with the hope that the error information w.r.t each particular exercise and each particular evaluation pattern could be captured. Experimental results on a real dataset show PIPE's superior performance over the baseline models. We have used PIPE to implement the error-feedback feature in our OJ system, and will further evaluate its impact on programming education.

In future, we plan to improve PIPE so that it may not only predict but also localize the errors, i.e., telling the students which lines of the program may contain logical errors and what are the potential types of the errors. Such feedback would further promote students' learning efficiency and help us to achieve higher scalability in programming education.

Acknowledgement

This work was partially supported by the grant from the National Natural Science Foundation of China (Grant No. U1811264).

6. REFERENCES

[1] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *Proceedings of the International Conference on Learning Representations*, 2018.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[3] G. Andrew, R. Arora, J. Bilmes, and K. Livescu. Deep canonical correlation analysis. In *International conference on machine learning*, 2013.

[4] S. Bhatia and R. Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. In *International Conference on Software Engineering*, 2018.

[5] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*, pages 30–39, 2014.

[6] R. Gupta, S. Pal, A. Kanade, and S. Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[7] J. R. Kettenring. Canonical analysis of several sets of variables. *Biometrika*, 58(3):433–451, 1971.

[8] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

[9] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[10] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE, 2017.

[11] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178, 2015.

[12] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE, 2015.

[13] M. Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[14] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[15] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-symbolic program synthesis. In *Proceedings of the International Conference on Learning Representations*, 2017.

[16] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *Proceedings of the International Conference on Machine Learning*, 2015.

[17] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay. sk_p: a neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 39–40, 2016.

[18] M. Rabinovich, M. Stern, and D. Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2017.

[19] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from" big code". *ACM SIGPLAN Notices*, 50(1):111–124, 2015.

[20] R. T. Rockafellar. Lagrange multipliers and optimality. *SIAM review*, 35(2):183–238, 1993.

[21] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh. Neural program repair by jointly learning to localize and repair. In *Proceedings of the International Conference on Learning Representations*, 2019.

[22] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal. A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)*, 51(1):1–34, 2018.

[23] M. Wu, M. Mosse, N. Goodman, and C. Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 782–790, 2019.

[24] C.-K. Yeh, W.-C. Wu, W.-J. Ko, and Y.-C. F. Wang. Learning deep latent space for multi-label classification. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.