Implement of Code Timeout Detection System Based on Tree-LSTM

Shao Mingyuan(Informatics)23320221154299, Zhang Haoran(AI)36920221153147, Yan Fei(Informatics)23320221154304, Qi Pengyu(Informatics)23020221154105, Lin Hong(Informatics)23320221154296

[1]School of Informatics
Xiamen University
Xiamen, China

Abstract

Codeforces is an online evaluation website for programming competitions. There are many results on this website: AC (Accepted), WA (Wrong answer),RE(Runtime error),CE(Compilation error) and TLE (Time limit exceeded) and so on. Among them, the time required for TLE is relatively long, which is easy to reduce the operating efficiency of the website. In this project, the source codes of AC and TLE are mainly involved.

The main purpose of this project is to design a self-developed classification model based on whether the source code collected from the program competition website Codeforces will predict the time limit exceed of the code without running the test case, and then implement the entire code time limit exceed exception detection system. The system includes the following:

(1) The pyppeteer headless browser was used to crawl 54848 codes from 1,513 program competitions on Codeforces, including 27424 AC and 27424 TLE code pairs, achieving balanced distribution of training samples.

(2) Parse the crawled code into AST (Abstract Syntax Tree), and extract the syntax dependency tree from AST based on the node name of AST. Then use one-hot vectors to achieve vectorized representation of grammatical relational dependency trees.

(3) The model is built with the Dependency-TreeLSTM[1] as the classifier and the prediction results are analyzed and studied. The final results show that the system can save 94% of the running time when the recall rate reaches 78%, the F1 score reaches 74% and the accuracy rate reaches 73%.

Introduction

With the rapid development of information society and computer technology, the program design course is becoming one of the required courses for students of various majors. And the improvement and practice of programming ability need a lot of programming practice, so it needs a lot of problems of appropriate difficulty and timely information feedback. It is time-consuming and laborious for teachers to correct by themselves, and the reliability of evaluation results is not stable. Therefore, a series of websites based on source code automatic assessment system have emerged as auxiliary teaching resources for such courses.

Usually, the main indicator of software quality is functionality. However, today's software users are not only satisfied with correctness, performance and user experience increasingly become the factors affecting user selection and loyalty. Performance errors are defined as software defects that do not affect functionality but can significantly increase operating efficiency when improved. As the maturity of system functional testing increases, system problems are usually related to system functional performance, and the problems related to system performance defects require more time to fix than the functional errors. The detection and optimization of performance defects depends more on the developer's experience. Different programmers may write completely different "correct" code for the same problem, and its execution efficiency may be completely different due to the different organization/structure of the code. This problem is especially acute for novice programmers. For example, on the program competition site codeforce, students solve the same problem by submitting their own code. A commit is marked Accept(AC) only if the code runs successfully through all test cases in the time and memory indicated in the problem statement. Through the analysis of historical data, it is found that 5% of code submissions lead to timeout, named TLE, and the program is considered to have performance defects(Zhou et al. 2019).

In the automatic source code evaluation system, it is necessary to set a large number of test cases to detect whether the code has performance defects. In order to reduce the load of the system, we propose a code timeout anomaly detection system based on Tree-LSTM(Tai K S,Socher R,CD Manning 2015), hoping to realize the prediction of code timeout before the code runs.

Related work

In the information era, computer is becoming more and more important to the composition of human society. And software in the computer composition has a pivotal position. Therefore, in recent years, the representation and analysis of program code with deep learning has become the focus in the field of code performance analysis.

Tree-LSTM. Tree-LSTM is an improved algorithm based on LSTM proposed by Kai Sheng Tai et al., which is a tree-based algorithm. In LSTM(Hochreiter S and Schmidhuber J 2019), the model is input strictly according to a fixed in-

put sequence, that is, a sequence of words from beginning to end. The idea of tree-LSTM is mainly to build a tree based on syntactic analysis, grammar analysis and other operations, and then input each word into different nodes according to the relationship between words. According to the author's experiment, this method is better than the traditional LSTM.

Code Embedding. With the development of deep learning and Natural Language Processing (NLP), program as a kind of natural language has also been included in the scope of research. Various program representation learning models have been proposed to understand the semantic properties of programs for code embedding and application to different software engineering tasks. The availability of large amounts of source code from public repositories has also driven the use of deep learning techniques for code embedding learning(Han S et al. 2021) over the past few years. Ahmad(Ahmad W U et al. 2020) learned code embeddings by modeling pairwise relationships between code tokens to capture their long-term dependencies. Rabinovich(Rabinovich M, Stern M and Klein D 2017) used ADSL (Abstract Syntax Description Language) syntax tree and neural network to achieve code generation.

Deep Learning Based Code Performance Research. Deep learning has made significant breakthroughs in various fields of artificial intelligence. The advantages of deep learning include the ability to capture highly complex features, weak involvement of human engineering, etc. Mou(Peng H et al. 2015)proposed a "coding standard" for constructing program vector representations to qualitatively and quantitatively evaluate the learned vector representations. Based on their experimental results, it is shown that the coding standard is successful in constructing program representations, and the results also confirm the feasibility of deep learning to analyze programs. Many other similar papers (Lu H, Cukic B and Culp M et al. 2012)(Gupta R et al. 2017)(Jin G et al. 2012)(Tsakiltsidis S, Miranskyy A, Mazzawi E et al. 2016)(Hellendoorn V J and Devanbu P 2017) also suggest that deep learning will become a prominent method for program analysis in the near future.

In the field of code classification, Zhou et al. proposed to use the deep learning model to extract code features from the code sequence and the control flow graph of the code and predict whether it is timeout for classification. Mou(Mou et al. 2014) et al proposed the TBCNN model, which uses the idea of CNN (Convolutional Neural Networks) to extract the features of the abstract syntax tree of the code in the way of simulating images and uses this to carry out the next classification task.

Proposed solution

The compiler often runs the code based on the lowest level abstract syntax tree. Categorizing the test result of the code samples. Accept samples correspond to 0 and TLE samples correspond to 1. Because of the tree nature, We use the abstract syntax tree to abstract the code into a syntax dependency tree, and then use TreeLSTM to extract its features and send them to the classifier. Specific methods are as follows:

Framework Overview

The goal of this paper is to automatically divide a source code into two categories before running on the code test case: whether it exceeds the time limit, which can be used to help the online evaluation system to save testing workload. We regard the defect prediction of TLE problem as the learning task of building the prediction function f: X $\rightarrow$ Y,where $y_i \in$ Y=$\{0,1\}$ indicates whether a code $x_i \in$ X will lead to exceeding the time limit (y_i=1) or passing all test cases within the time limit (y_i=0). Code is not only raw text data, but also structured data. Therefore, in our model, the prediction function f can be learned by neural network classifier based on the learned code features converted from the abstract syntax tree of the code. Then we input the learned vector into the network classifier, and finally get the classification results.

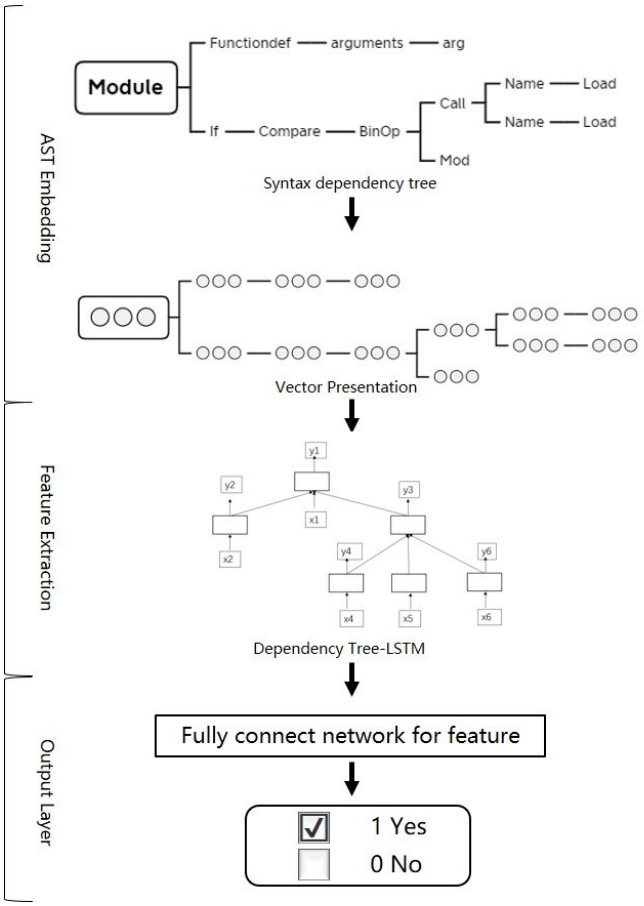

Figure 1: the framework of our method

Abstract Syntax Tree

The program analysis based on deep learning needs to consider how to vectorize the code. Now, the granularity commonly used in program code analysis is mostly at the token level, but there is a serious problem that programmers can name identifiers in source code by themselves, so the number of tokens will be very large, and we will also be trou-

bled by the problem of sparse data. In the abstract syntax tree, the relationship between program codes can be clearly expressed in a more compressed form, and there are only a limited number of types of nodes in the abstract syntax tree, which makes learning feasible. The abstract syntax tree can also represent the syntax structure of the source code more intuitively, containing all the static information of the source code structure, and it is convenient to store. Therefore, we use the abstract syntax tree to express the syntax dependency tree of the code.
Since the names of nodes behind the syntax dependency tree only include 103 types, that is, there are only 103 types left. The dimensions after we use the unique heat vector to vectorize are not high, and the matrix is not too sparse.

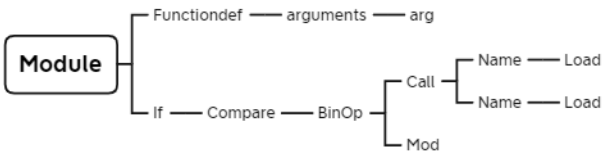

Figure 2: Syntax dependency tree converted by AST

Tree-LSTM

Tree-LSTM (Tree Structured Long Short Term Memory Networks is a LSTM structure published in 2015. Natural languages such as code do not Only the linear sequence relationship features, and the characteristics of grammar and syntax should be taken into account. The standard LSTM contains input gates and output gates, memory cells and hidden states. The difference between the standard LSTM and Tree LSTM is that the update of the gate vector and memory cell vector is based on multiple subunits. The former only needs to filter information from the previous moment, while the latter needs to filter information from multiple child nodes. Therefore, the Tree LSTM proposed in this paper can more easily combine dependency, phrase formation and other grammatical features, making semantic expression more accurate.
The author proposes two kinds of Tree LSTM structures: Child Sum Tree LSTM and N-ary Tree LSTM.This paper adopts Child Sum Tree LSTM based on dependency tree.
Take Tree LSTM module unit composed of two children as an example. With j as the index, each Tree LSTM unit, like the standard LSTM unit, contains input and output gates i_j and o_j, storage unit c_j, and hidden state h_j. The special feature of Tree LSTM unit is that the output state of sub unit can better determine the update of gating vector and memory unit state. In addition, Tree LSTM unit is not a single forgetting door, but contains a forgetting door f_{jk} for each child. This allows the hidden state of the parent node of the Tree LSTM unit to obtain and filter the information transmitted from each child node, so it can learn to retain the representation of children with rich syntax information for code classification.
On the input module of the cell, Tree LSTM is the same

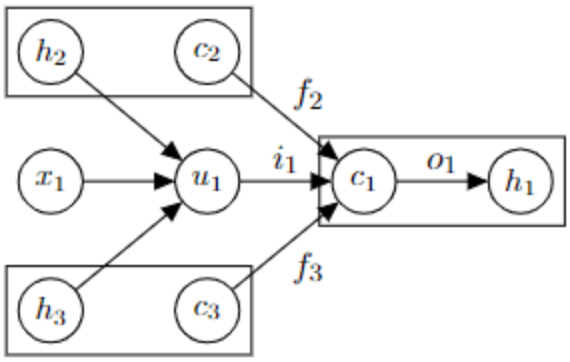

Figure 3: Module unit of Tree-LSTM

as the standard LSTM. Each Tree LSTM cell has an input vector x_j. In language processing, each x_j is the vectorized representation of each word in the sentence. In this paper, the input of x_j is the unique hot vector converted from the node type name in the syntax dependency tree. C(j) represents the calculation formula of the sub node set of node j as follows:

$$\widetilde{h} = \Sigma_{k \in C(j)} h_k \tag{1}$$

$$i_j = \sigma(W^{(i)} x_j + U^{(i)} \widetilde{h}_j + b^{(i)}) \tag{2}$$

$$f_{jk} = \sigma(W^{(f)} x_j + U^{(f)} \widetilde{h}_k + b^{(f)}) \tag{3}$$

$$o_j = \sigma(W^{(o)} x_j + U^{(o)} \widetilde{h}_j + b^{(o)}) \tag{4}$$

$$u_j = \tanh(W^{(u)} x_j + U^{(u)} \widetilde{h}_j + b^{(u)}) \tag{5}$$

$$c_j = i_j \odot u_j + \Sigma_{K \in C(j)} f_{ik} \odot C_j \tag{6}$$

$$h_j = o_j \odot \tanh c_j \tag{7}$$

where k∈ C(j).
Since all cell parameters are 128 dimensions, we need to expand the input 103 dimensions to 128 dimensions before entering the model. The expansion formula is as follows:

$$x = xA^T + b \tag{8}$$

Where the dimension of transpose matrix of A is [103,128].
In the classifier part, we want to predict the label $\hat{y}$ of a node subset in the tree from two discrete classes $Y \in \{0, 1\}$.
The label of the node in the tree obtained by the classifier in the process of parsing the tree may correspond to some static attribute characteristics of the code that the node spans. After generating the overall hidden state $\tilde{h}$ of the whole tree, the $\tilde{h}$ is sent to the hidden layer, activated by the sigmoid function, and finally passed through the log_softmax function generates the final two-dimensional result, and takes the larger one as the prediction tag result. The calculation formula is as follows:

$$out_\theta = \sigma(W^{(\theta)} \widetilde{h}_\theta B^T + b^{(i)}) \tag{9}$$

$$\hat{p}_\theta = log\_softmax(W^{(\theta)} out_\theta C^T + b^{(\theta)}) \tag{10}$$

$$\hat{y}_\theta = argmax \hat{p}_\theta(y | \{0, 1\}) \tag{11}$$

The loss function of the network is a binary cross entropy loss function. This loss function is often used to solve the binary classification problem, so we use it to measure the loss between the prediction result and the real sample result y. The formula of the binary cross entropy loss function is as follows:

$$L(\theta) = -((y_\theta \log \hat{y}_\theta) + (1 - y_\theta) \log(1 - \hat{y}_\theta)) \quad (12)$$

Finally, the value of the loss function is calculated by feedback, and the weight W is automatically adjusted by the gradient descent algorithm. Each code in each epoch participates in calculating the gradient until the optimal solution parameters for the current epoch are determined and the trained model is stored.

Experiments

Dataset

We test the efficacy of our proposed method on our own datasets, we call it CFDataset, which contains more than 54k labeled codes including 2 categories(AC, TLE) for training and testing. All of them are crawled from Codeforces.

Dataset	
Contest	1513
Problem	3323
Total programs	54848
TLE programs	27424
AC programs	27424
Max nodes	5617
Min nodes	6
Avg nodes	148.6
Avg Runtime	959.83(ms)

Table 1:Dataset description

Data preprocessing

The data preprocessing consists of three main parts: analysis, conversion and normalization operations. We use Python's AST package to parse the code.Because the various attributes of the abstract syntax tree class are too complex, we only reserve the node name as the semantic attribute of the abstract syntax tree, which becomes the syntax dependency tree.Since the traversal function of the parsed syntax dependency tree is traversed by the Breath first search (BFS) method to obtain child nodes, we use the structure of the queue and mark the serial number of each node. First, we transform the Breath first search traversal into a sequence traversal and mark the serial number of the parent node for each node, and then obtain the information of each node in order, so that it can be converted into the input form required by Tree LSTM.

Dataset protocol

We randomly sampled 25% or 50% programes for the training set, and the rest of the 50% samples was used for validation.

Evaluation Protocol

In order to evaluate the prediction results, this paper uses the accuracy, precision, recall and F1 score that are widely used in other papers(Tsakiltsidis S, Miranskyy A and Mazzawi E 2019)(Mou L et al. 2014)(Sandoval Alcocer J P, Bergel A and Valente M T 2016)(Yang X et al. 2015)(Wang S, Liu T and Tan L 2016)in the secondary classification task as the evaluation indicators of experiment.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (13)$$

$$Precision = \frac{TP}{TP + FP} \quad (14)$$

$$Recall = \frac{TP}{TP + FN} \quad (15)$$

$$F_1 = \frac{2PR}{P + R} = \frac{2TP}{2TP + FP + FN} \quad (16)$$

Implementation Details

We select a few of the more important hyper-parameters shown in Table 2.

Hyperparameter	Value
Hidden layers	1
Hidden dimension	64
Learning rate	0.01
Batch size	32
Weight decay	0.0001
Optimizer	adagrad

Table 2:Hyperparameter

Result

This section will introduce the experimental results. We ran the model on a NVIDIA GeForce RTX 2070 super, each experiment trained for 10 epochs. There are two cases to explore whether the size of training set will affect the effect of model training.Table 3 shows the comparison result. It can be seen that this model can achieve the same effect as the large sample training by using a small number of samples to train the model.

Module	Accuracy	Precision	Recall	F1 Score
TreeLSTM-25%	0.702	0.688	0.778	0.730
TreeLSTM-50%	0.718	0.691	0.788	0.737

Table 3:Training effect of 10 epochs with different proportion training sets

Due to the limitation of training equipment, this paper can only compare the three models roughly. The epoch of each model training is set to 10, and the learning rate is set to 0.01 to expect faster convergence. The data set is 54848 codes extracted in this paper. The training set is 25% of the dataset,including 6856 TLE codes and 6856 AC codes, totaling 13712 codes; The testing set was in the rest of dataset. It mainly compares the advantages and

disadvantages of the prediction of the model and the time spent.

Module	Accuracy	Recall	F1 Score
Tree-LSTM	**0.70**	**0.78**	**0.73**
Att-Bi-LSTM	0.65	0.73	0.68

Table 4:Comparison results of different modules

Module	Avg Runtime(ms)
Avg Runtime	959.83
Tree-LSTM	75
Att-Bi-LSTM	56

Table 5:Module Avg Runtime

Our model finally saved 92% of running time on the premise of having nice prediction results.

Conclusion

From the results of the above two models, it can be seen that the accuracy of Tree-LSTM model will be higher, but the prediction time will be about 30% slower than that of Att-Bi-LSTM model. This may be because Tree-LSTM model is much larger than the linear LSTM model. But compared with the original running time, it is still a very novel result.

In this paper, we propose an interesting method to predict whether the code runs overtime. By introducing abstract syntax trees and Tree LSTM, we can focus on the static syntax features at the bottom of the code to extract features that are not available in the linear model. We demonstrated the effectiveness of our model design and achieves satisfactory performance.In this experiment, we explored code embedding and feature extraction, which is a direction worthy of research.

References

Zhou, M. , Chen, J. , Hu, H. , Yu, J. , & Hu, H. . (2019). DeepTLE: Learning Code-Level Features to Predict Code Performance before It Runs. 2019 26th Asia-Pacific Software Engineering Conference (APSEC). IEEE.

Tai K S , Socher R , CD Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks[J]. Computer Science, 2015, 5(1): 36.

Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735-1780.

Han S, Wang D X, Li W, et al. A Comparison of Code Embeddings and Beyond[J]. arXiv preprint arXiv:2109.07173, 2021.

Ahmad W U, Chakraborty S, Ray B, et al. A transformer-based approach for source code summarization[J]. arXiv preprint arXiv:2005.00653, 2020.

Rabinovich M, Stern M, Klein D. Abstract syntax networks for code generation and semantic parsing[J]. arXiv preprint arXiv:1704.07535, 2017.

Peng H, Mou L, Li G, et al. Building program vector representations for deep learning[C]International conference on knowledge science, engineering and management. Springer, Cham, 2015: 547-553.

Lu H, Cukic B, Culp M. Software defect prediction using semi-supervised learning with dimension reduction[C]2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2012: 314-317. Gupta R, Pal S, Kanade A, et al. Deepfix: Fixing common c language errors by deep learning[C]Thirty-First AAAI Conference on Artificial Intelligence. 2017.

Jin G, Song L, Shi X, et al. Understanding and detecting real-world performance bugs[J]. ACM SIGPLAN Notices, 2012, 47(6): 77-88.

Tsakiltsidis S, Miranskyy A, Mazzawi E. On automatic detection of performance bugs[C]2016 IEEE international symposium on software reliability engineering workshops (ISSREW). IEEE, 2016: 132-139.

Hellendoorn V J, Devanbu P. Are deep neural networks the best choice for modeling source code?[C]Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 763-773.

Mou L, Li G, Jin Z, et al. TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing. CoRR abs/1409.5718 (2014)[J]. arXiv preprint arXiv:1409.5718, 2014.

Tsakiltsidis S, Miranskyy A, Mazzawi E. On automatic detection of performance bugs[C]2016 IEEE international symposium on software reliability engineering workshops (ISSREW). IEEE, 2016: 132-139.

Mou L, Li G, Jin Z, et al. TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing. CoRR abs/1409.5718 (2014)[J]. arXiv preprint arXiv:1409.5718, 2014.

Sandoval Alcocer J P, Bergel A, Valente M T. Learning from source code history to identify performance failures[C]Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering. 2016: 37-48.

Yang X, Lo D, Xia X, et al. Deep learning for just-in-time defect prediction[C]2015 IEEE International Conference on Software Quality, Reliability and Security. IEEE, 2015: 17-26.

Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction[C]2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016: 297-308.