Applying Transformer Models for Automatic Build Errors Classification of Java-Based Open Source Projects

Jonathan Lee
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
jlee@csie.ntu.edu.tw

Mason Li
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
r09922133@csie.ntu.edu.tw

Kuo-Hsun Hsu
Department of Computer Science
National Taichung University of Education
Taichung, Taiwan
glenn@mail.ntcu.edu.tw

ABSTRACT

In open-source development, encountering build failures is a common challenge. Addressing these issues requires analyzing the causes of errors and developing solutions for fixing them. In this work, we fine-tuned Google's BERT, a well-known language model excellent in transfer learning, to address build issues in Gradle Java projects. Our strategy utilizes this model to classify error logs and identify fixing solutions. This approach extends our previous work, Gradle ACFix, an automated build error fixing system, to explore the potential of using machine learning to classify error types and identify appropriate fixing strategies for software projects. We gathered a dataset of 11,483 open-source Gradle Java projects from GitHub for this research. The model's evaluation on the error logs of these projects demonstrated a high accuracy rate exceeding 98%.

KEYWORDS

build error, build fixing, Gradle, open source, deep learning model

ACM Reference Format:
Jonathan Lee, Mason Li, and Kuo-Hsun Hsu. 2018. Applying Transformer Models for Automatic Build Errors Classification of Java-Based Open Source Projects. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 2 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Java, ranking as the third most used programming language on GitHub in 2022, experiences widespread use of build tools, like Ant, Maven, Bazel, and Gradle. These tools aid in dependency management, build automation, and ensuring consistency, enhancing efficiency and reducing errors. However, frequent changes in source code, tests, configurations, dependencies, and scripts often lead to build failures. A previous study showed that 29% to 59% of Java projects ended up with build failure [3, 7], which frustrates developers and postpones the progress of projects. Each failure type might need a unique solution, demanding extensive knowledge of the build tool. Previous research, like Macho et al.'s work [6], explored rule-based methods to tackle this problem, but creating rules for various error types is a challenging, knowledge-intensive task.

In our study, we analyzed 11,483 open-source Gradle Java projects from GitHub, using Gradle ACFix [2], our previous work, to gather error logs and their fixes. We then fine-tuned BERT for sequence classification to interpret these logs. The model inputs an error log and suggests one of Gradle ACFix's 31 fixing strategies. Our evaluations showed that the model accurately predicts the correct strategies for different build errors.

2 METHODOLOGY

We proposed an architecture utilizing BERT, as shown in Figure 1. For our model, we employed the "BertForSequenceClassification" architecture from the "transformers" package by Hugging Face [1], using the pre-trained weights of BERT provided by Google (bert-base-cased). The BERT-based model comprises 12 layers of encoders and 12 attention heads, with a hidden layer size of 768.

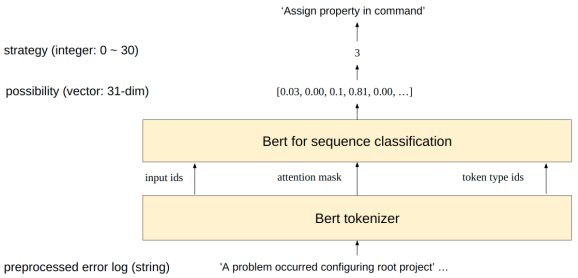

Figure 1: Proposed BERT architecture

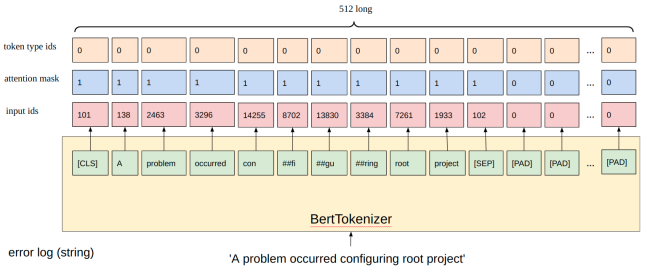

Figure 2: BERT tokenizer

The input error log is first processed by the BERT tokenizer, which is also provided by the "transformers" package (Figure 2). The

ICSE 2024, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

tokenizer divides the input sequence (error log) into tokens based on BERT's predefined vocabulary dictionary. For example, the sentence "A problem occurred configuring root project" would be divided into individual tokens as follows: "A", "problem", "##occurred", "##con", "##fi", "##gu", "##ring", "root", "project".

In addition to tokenization, BERT adds several unique tokens for various purposes. The [CLS] token, signifying "class", represents the entire meaning of a sentence and is placed at the beginning of the sequence. The [SEP] token, standing for "separate", is used to distinguish between two sentences in a dual-sentence input, being inserted between them; in cases of single-sentence inputs, [SEP] is added at the end. To ensure uniform sentence length, the [PAD] token, meaning "padding", fills any extra space. When a subword is not in BERT's dictionary, the [MASK] token, which means "mask", is used exclusively during pretraining to represent a word masked in the model's training process.

The BERT tokenizer creates two additional vectors for learning: an "attention mask" and "token type ids." The attention mask, a boolean vector, marks [PAD] tokens with 0s and others with 1s. The token type ids vector, also boolean, uses 0s for the first sentence and padding and 1s for the second sentence; it's all 0s for single sentences. These vectors are fed into BERT for sequence classification, which combines BERT's output with a linear layer from the [CLS] token. The model produces a 31-dimensional vector, each dimension ranging from 0 to 1, representing the probability of each fixing strategy for an error log.

3 RESULTS

We collected 11,483 open-source Gradle Java projects from GitHub, with 4,800 providing 16,085 error log-fixing strategy pairs. The data was divided into training, validation, and testing sets, using a balanced splitting strategy to avoid bias. For strategies with over ten error logs, we allocated 80% to the training set, 10% to the validation set, and 10% to the testing set. Single-log strategies went to testing, while two-log strategies were split between testing and validation. For 3-9 logs, one went to testing, one to validation, and the rest to training, ensuring fair representation and addressing strategy imbalances.

We utilized AdamW [5] as our optimizer, an adaptation of the Adam [4] optimizer. The learning rate was set to 1e−4 for our experiments. Despite truncation issues in many error logs, the model achieved an impressive testing accuracy of 0.9863 for 11,483 projects with a batch size of 256 and 5 epochs (Table 1).

Table 1: Results

Batch size	Epochs	Validation accuracy	Testing accuracy
8	1	0.4619	0.4613
16	1	0.9357	0.9308
32	1	0.9806	0.9800
64	3	0.9881	0.9875
128	3	0.9881	0.9776
256	6	0.9875	0.9863

The model exhibited excellent overall accuracy and demonstrated favorable precision and recall for each strategy. However, the performance was inevitably suboptimal for strategies with very few error logs, particularly for those without any error logs in the training set (Table 2). The limited availability of training data for these strategies impacted the model's ability to generalize effectively, resulting in lower performance in these specific cases.

Table 2: Performance for Each Strategy

Fixing strategy	Precision	Recall	F1-score	Count
-	0.9879	0.9892	0.9885	740
Rewrap Gradle	0.9972	0.9861	0.9916	360
Search artifact	0.9618	0.9844	0.9730	128
Switch Java version	1.0000	0.9907	0.9953	108
Replace repository	0.9759	0.9878	0.9818	82
Assign property in command	0.9667	1.0000	0.9831	58
Switch Java version by class version	1.0000	1.0000	1.0000	55
Adjust Java and Gradle version	1.0000	1.0000	1.0000	19
Exclude tasks with same name	0.8947	0.9444	0.9189	18
Exclude task	0.9286	0.9286	0.9286	14
Full clone	1.0000	1.0000	1.0000	7
Dump file	1.0000	1.0000	1.0000	6
Include subproject	1.0000	1.0000	1.0000	3
Assign property in property file	1.0000	1.0000	1.0000	2
Add repository	1.0000	1.0000	1.0000	1
Remove IDEA plugin	0.0000	0.0000	0.0000	1
Dump file to dump directory	0.0000	0.0000	0.0000	1
Add buildscript dependency	0.0000	0.0000	0.0000	1

4 CONCLUSION

This work extends our previous work, Gradle ACFix, by using machine learning to categorize build failures into specific fix strategies. It shows that transformer-based models are adequate for classifying error logs. However, identifying the right fix strategy is the first step in repairing a project build failure. Using machine learning to determine the parameters needed for these strategies is still challenging. Currently, this method only works with Gradle Java project error logs. There's a lot of potential to expand this to different build frameworks and programming languages like C++ and Python. Future research could explore these areas to broaden the use of machine learning in various software projects.

ACKNOWLEDGMENTS

This research was sponsored by the National Science and Technology Council, Taiwan, under grant number MOST 109-2221-E-002-140-MY3.

REFERENCES

[1] 2023. *Hugging Face.* https://huggingface.co/
[2] Hong-Wun Chen. 2020. *Auto-Classifying-Fixing Build Errors in Java-Based Open Source Projects.* Master's thesis. National Taiwan University.
[3] Foyzul Hassan and Xiaoyin Wang. 2018. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th international conference on software engineering.* 1078–1089.
[4] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[5] Ilya Loshchilov and Frank Hutter. 2017. Fixing weight decay regularization in adam. (2017).
[6] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner).* IEEE, 106–117.
[7] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering.* 724–734.